

Resumen Orga2 Final

Paralelismo a nivel de instrucción

Pipeline

Arquitectura que permite crear el efecto de superponer en el tiempo, la ejecución de varias instrucciones a la vez para que todos los bloques trabajen en paralelo cada uno en una instrucción diferente.

=> **Instruction Level Parallelism (ILP)**

El pipeline tarda en llegar a esa condición de régimen tantos ciclos de clock como etapas tenga (teóricamente).

Por que aumentar etapas? Cuantas mas etapas podamos definir para ejecutar, al ponerlas a trabajar a todas en paralelo en un pipeline, el tiempo de ejecución de la instrucción se reducirá proporcionalmente con la cantidad de etapas.

Su eficiencia se mide en TPI:

$$TPI = \frac{\text{Tiempo por instruccion en la CPU no-pipeline}}{\text{Cantidad de etapas}}$$

En la practica existen overheads introducidos por el pipeline que suman pequeñas demoras.

El pipeline no reduce el tiempo de ejecución de cada instrucción individual, sino que al aplicarse en paralelo al flujo de instrucciones, incrementa el número de instrucciones completadas por unidad de tiempo.

Hazards (obstaculos)

Es lo que reduce la eficiencia de un pipeline => causan **pipeline stall** => reduce la performance del procesadores

<i>Tipo de obstaculo</i>	<i>Causas</i>	<i>Posibles soluciones</i>
Estructural	<ul style="list-style-type: none">• Etapa no suficientemente atomizada, requiere mas de un ciclo de clock• Instrucciones que estan mas proximas que el tiempo que necesita esa etapa para procesar => conflicto de recursos => CPI += 1	Siempre con hardware. Para accesos a memoria: <ul style="list-style-type: none">• separar cache en datos e instrucciones• buffers de instrucciones como colas fifo• ensanchar buses hacer un deeper pipeline.
De datos	Cuando por efecto del pipeline, una instrucción requiere de un dato antes de que este esté disponible por efecto de la secuencia lógica prevista en el programa. => CPI += n n = dist entre etapas del pipeline que requieren el dato	<u>Forwarding</u> Se extrae el resultado directamente de la salida de la unidad de Ejecución cuando está disponible y se lo envía a la entrada de la etapa que lo requiere en el mismo ciclo de clock en que se escribe en el operando destino (SOLO para etapas posteriores que quedarían en stall y operaciones de ALU)
De control	Un branch es la peor situacion en perdida de performance porque se pierde la secuencia. Hace que todo lo que estaba preprocesado se descarte => n-1 ciclos de clock hasta proximo resultado para n etapas de pipeline => Branch penalty	Se soluciona con.....

Superscalar

Permite tener dos o mas unidades de ejecucion para cada stage. (fetch, decode, etc)

Consecuencias de mas ILP / Superscalar

- Los obstaculos estructurales quedan mas expuestos
- Mas probabilidad de accesos simultaneos a memoria
- Se pueden ejecutar en dos ALUs dos instrucciones. Pero si una depende del resultado de la otra, esto no es posible.
- Una falla en el branch prediction limpia ambos pipelines!

Branch prediction

Manejo de control dependence

- Control dependence: la ejecución de la instrucción siguiente depende de la actual.
- Cual debe ser el PC del fetch en el próximo ciclo? Como determinamos que va a pasar?

El control dependence es critico para mantener el pipeline lleno con la secuencia correcta de instrucciones dinámicas. Esto tiene posibles soluciones:

- Elimina instrucciones de control de flujo (**predicated execution**)
- Usar delayed branching (**branch delay slot**)
- Hacer otra cosa (**fine-grained multithreading**)
- **Adivinar la próxima fetch address (branch prediction)**
- Hace fetch de los dos caminos posibles (si sabes las direcciones de ambos) (**multipath execution**)
- Esperar hasta conocer la próxima fetch address

1. Predicated execution

Idea: Convertir control dependence a data dependence => Elimina branches

CMOV condition, R1 ← R2

R1 = (condition == true) ? R2 : R1

-*Cada instrucción tiene un bit de predicado basado en como computa el predicado.

-*Solo se hacen las instrucciones que tienen predicado T (las otras se vuelven NOPs)

(normal branch code)

```
if (cond) {
    b = 0;
}
else {
    b = 1;
}
```

```
graph TD
    A -- T --> C
    A -- N --> B
    C --> D
    B --> D
```

Assembly for normal branch code:

A	p1 = (cond)
B	branch p1, TARGET
C	mov b, 1
D	jmp JOIN
TARGET:	mov b, 0
D	add x, b, 1

(predicated code)

```
graph TD
    A --> B
    A --> C
    B --> D
    C --> D
```

Assembly for predicated code:

A	p1 = (cond)
B	(!p1) mov b, 1
C	(p1) mov b, 0
D	add x, b, 1

Ventajas	Desventajas
<ul style="list-style-type: none">+Funciona mejor cuando nunca se toma la predicción.+El compilador tiene mas libertad para optimizar código.+Elimina las predicciones erradas en los branches	<ul style="list-style-type: none">Trabajo innecesario:<ul style="list-style-type: none">-* algunas instrucciones hacen fetch/execute pero descartadas (especialmente branches facilmente predecidas)-* algunas branches se predicen facilmente

que no son fáciles de predecir => bueno si costo de error predicción > trabajo en vano por hacer predication <ul style="list-style-type: none"> +Permite optimizacion de codigo limitada por control dependency 	(disminuye performance si costo de error prediccion < trabajo innecesario) <ul style="list-style-type: none"> Necesita soporte de ISA adicional. No puede eliminar todas las branches dificiles de predecir (loops)
--	---

Predicate combining

- Transformar predicados complejos en multiples branches

Problema: Aumenta el # de control dependencies

Idea: Combinar predicados para alimentar a una sola instruccion de branch

- Los predicados se guardan y operan usando registros de condicion
- Una sola branch chequea el valor de el predicado combinado

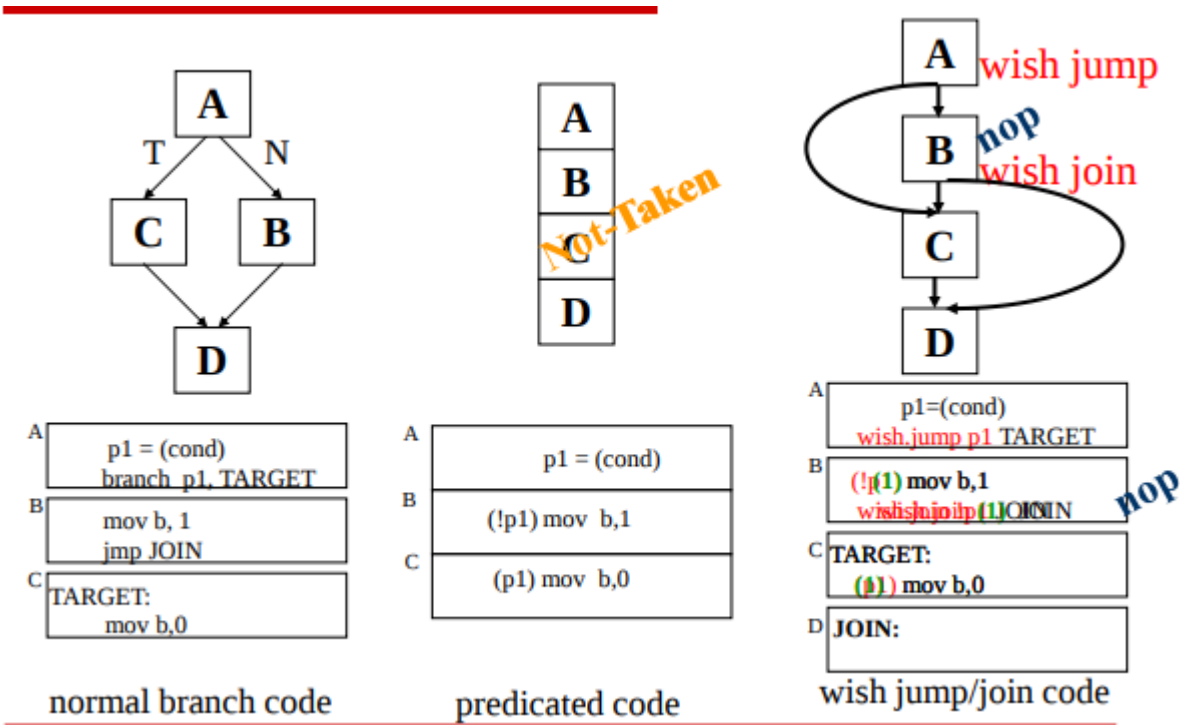
+ Menos branches => menos errores

-- Posible trabajo innecesario => si el primer predicado es falso no haria falta computar los demas

Wish Branches

Es una manera de mejorar la adaptabilidad a comportamiento de branches y aplicabilidad a Control Flow Graphs (CFG) complejos de Predicated execution (loops).

- El compilador genera codigo que puede ejecutarse como codigo predicado o no-predicado (branch normal)
- El hardware decide ejecutar uno u otro a la hora de correr en base a la confidence del branch prediction
- Facilmente predecible => branch normal // Dificil de predecir => codigo predicado



Wish Branches vs. Predicated Execution

Ventajas de Wish Branches	Desventajas de Wish Branches
---------------------------	------------------------------

<ul style="list-style-type: none"> • Reduce el costo de predicacion • Aumenta los beneficios de codigo predicado permitiendo al compilador generar codigo agresivamente-predicado • Hace que el codigo predicado dependa menos de la configuracion de maquina 	<ul style="list-style-type: none"> • Mas instrucciones de branch usan recursos de la maquina y compiten por posiciones en entradas de la tabla predictora • Reduce las posibilidades que el compilador optimice codigo
--	--

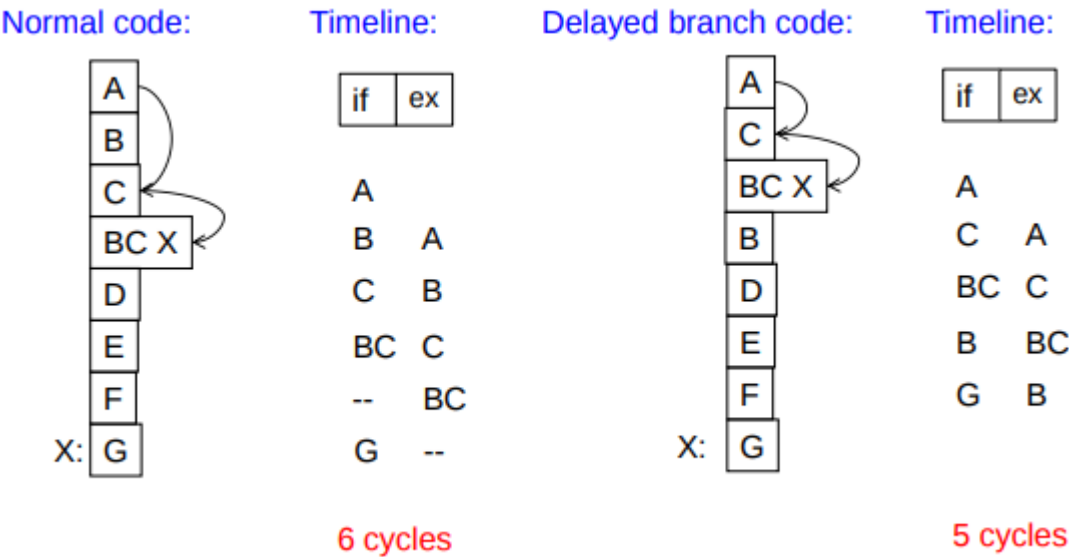
2. Branch delay slot

Cambiar la semantica de una instruccion de branch:

- Branch despues de N instrucciones.
- Branch despues de N ciclos.

Idea: Retrasar la ejecucion de un branch. N instrucciones (delay slots) que vienen despues del branch son ejecutadas siempre sin importar la direccion del branch.

Como encontramos instrucciones para llenar los delay slots? La branch debe ser independiente las instrucciones de los delay slots.



Ventajas	Desventajas
+Mantiene el pipeline lleno con instrucciones utiles asumidas de manera simple. *#delay slots = #instrucciones para mantener lleno el pipeline hasta resolver el branch *Todos los delay slots se pueden llenar con instrucciones utiles.	--No es facil llenar los delay slots --Ata la semnatica de la ISA a la implementacion de hardware.

3. Fine-Grained Multithreading

Idea: El hardware tiene multiples contextos de thread. Cada ciclo, el fetch fetchea de un thread distinto.

- Para cuando se resuelve la branch/instruccion, ninguna instruccion fue fetcheadada del mismo thread.
- La latencia de resolucion de branch/instruccion se solapa con la ejecucion de las instrucciones de otros threads.

Ventajas	Desventajas
+No requiere logica para manejar control y data dependences en un mismo hilo y prediccion de branches.	--Requiere mas complejidad de hardware. --La performance de thread unico disminuye --Mas logica para mantener los contextos de thread --No cubre latencia si no hay suficientes threads para cubrir todo el pipeline. --Hay pelea por recursos de cache y memoria entre threads --Queda logica para chequear dependencias entre threads.

4. Branch Prediction

Idea: Adivinar la siguiente instruccion cuando se fetchea una branch. Requiere adivinar la direccion y target de una branch.

Esperar en una branch vs. Adivinar una branch	Misprediction Penalty

	<p>Cuando una branch ($inst_h$) se resuelve:</p> <ul style="list-style-type: none"> -*Se fetchea el branch target ($inst_k$) -*Todas las instrucciones fetcheadas desde $inst_h$ (camino incorrecto) deben ser flusheadas.
	<p>Pipeline flush cuando hacemos una misprediction</p>

Analisis de performance

- Adivinamos => no penalty ~86% de probabilidad
- No adivinamos => 2 bubbles

Asumiendo:

<ul style="list-style-type: none"> No esperamos por data dependency 20% de instrucciones de control flow Se toman 70% de las instrucciones de control flow 	$\square \text{ CPI} = [1 + (0.20 \cdot 0.7) * 2] =$ $= [1 + 0.14 * 2] = 1.28$ <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;"> <p>probability of a wrong guess</p> </div> <div style="text-align: center;"> <p>penalty for a wrong guess</p> </div> </div>
---	--

Se pueden reducir los dos terminos de penalizacion?

- Penalty for a wrong guess:** Resolviendo el branch condition y target address antes (cambiando hardware). Puede o no ser una buena idea.
- Probability of a wrong guess:** Predecir la siguiente fetch address. Requiere predecir:
 - Si la instruccion fetchada es una branch:** Se puede lograr con un BTB: si este provee una target address para el PC, entonces debe ser una branch. Si no podemos guardar bits de “branch metadata” en el cache de instrucciones/memoria (instruccion parcialmente decodificada)
 - (Condicional) Direccion de la branch:** Como predecimos la direccion?
 - Al compilar (estatico)
 - Al momento de correr (dinamico)
 - El target address de la branch (si se toma):** Este se mantiene entre instancias dinamicas.

Idea: Guardar el target address de la instancia previa y accederlo con el PC. Esto se hace usando el BTB (Branch Target Buffer).

Branch Prediction Estatico

Tipo	Caracteristicas
Always not taken	-*Simple de implementar, no BTB ni prediccion de direccion -*Baja accuracy para branches condicionales
Always taken	-*No prediccion de direccion -*Mejor accuracy para branches condicionales: los backward branches (ej: loops) se toman.
Backward taken, forward not taken (BTFN)	-*Los backwards (loop) se toman y los demas no.
Profile-based	-*El compilador determina la direccion mas probable para cada branch usando una profile run. +Predicciones mas acertadas que las anteriores si el profile es representativo --Necesita hint bits en el formato de instruccion de branch --Accuracy depende del comportamiento dinamico del branch y representatividad del profile.
Program-based	-*Usa heurísticas basadas en analisis del programa para determinar la direccion predecida estadisticamente. +No requiere profiling --Heurísticas pueden no ser representativas o buenas --Requiere analisis del compilador y soporte de la ISA
Programmer-based	-*El programador provee la direccion predecida estadisticamente a traves de <i>pragmas</i> (Palabras clave que permiten al programador indicar pistas de si es probable que se tome o no la condicion). +No requiere profiling o analisis de programa +El programador conoce los braches y el programa mejor que otras tecnicas de analisis --Requiere soporte de lenguaje de programacion, compilador y soporte de ISA

--Molesta al programador.

Todas estas tecnicas previas pueden ser combinadas, pero tienen una desventaja en comun: no se pueden adaptar a cambios dinamicos en el comportamiento de branch.

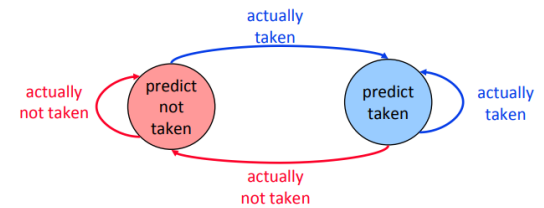
Branch Prediction Dinamico

Idea: Predecir branches basado en informacion dinamica (tomada al correr el programa)

Ventajas	Desventajas
+Prediccion basada en el historial de ejecucion de los branches (se adapta a cambios dinamicos) +No necesita profiling estatico	--Mas complejo, requiere hardware adicional

Last time predictor:

- Un bit por branch que se guarda en el BTB indica que direccion el branch fue la ultima vez.
- Siempre predice mal la ultima y primer iteracion de un loop (bueno en loops de muchas iteraciones, malo en uno de pocas).



Maquina de estados

Como se puede mejorar? Si cambia muy rapido de prediccion se puede agregar un bit de historial (hysteresis: que dependa de su historial) para que no cambie basado en solo un outcome distinto.

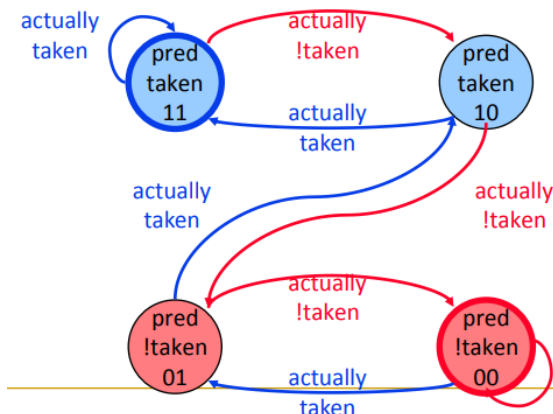
Two-Bit Counter Based Prediction

- Cada branch esta asociado con un two-bit counter.
- Un bit mas provee hysteresis.
- Una prediccion fuerte no cambia con un solo outcome distinto

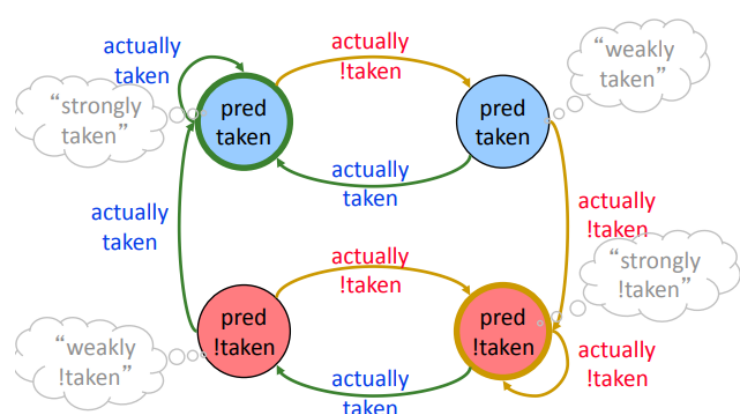
+Mejor accuracy de prediccion: $CPI = [1 + (0.20 \cdot 0.10) \cdot 2] = 1.04$ (90% accuracy)

--Mas costo de hardware (contador puede ser parte de entry de BTB)

Si cambia de prediccion despues de dos errores consecutivos...



Maquina de estados



Hysteresis

Es lo suficientemente bueno? Tiene ~85%-90% de accuracy para muchos programas, pero analicemos los ciclos que hacemos de mas cuando fallamos en la prediccion...

N = la siguiente fetch address despues de una instruccion con control flow no se determina hasta despues de N ciclos en un procesador con pipelining

W = se fetchean W instrucciones por ciclo, el pipeline tiene W de ancho

Si fallamos en la prediccion malgastamos **N x W** instruction slots

Con un accuracy de 95%, N = 20, W = 5, 1/5 de instrucciones son branch estamos fetcheando 100% de instrucciones extra.

Los predictores Last-time y 2BC se basan en lo que se puede predecir a partir de la ultima vez que se paso por esa branch.

Global branch correlation

- El resultado de una branch se relaciona con los resultados de otras.

if (cond 1) ... if (cond 1 and cond 2)	Si se toma la primera, no se toma la segunda
--	--

Idea: Asociar resultados de branches con el historial de T/NT global de todas las branches.

- Hacer una prediccion basada en el resultado de la branch la ultima vez que se encontro el mismo historial global.

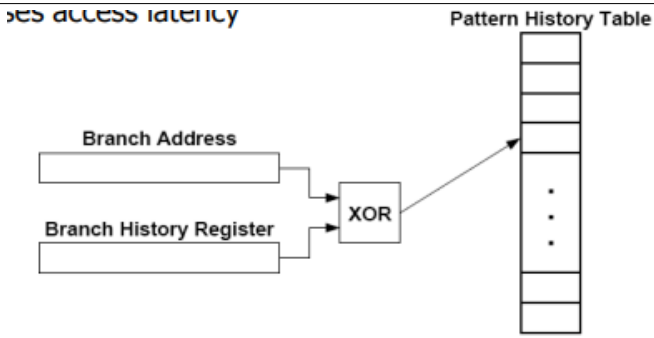
Implementacion:

- Guardar el historial **Global History Register (GHR)** => 10110:1 si se toma el branch, 0 si no
- Usarlo para indexar en la **Pattern History Table** (tabla de 2BC) el resultado guardado que se vio para cada valor de GHR del pasado reciente

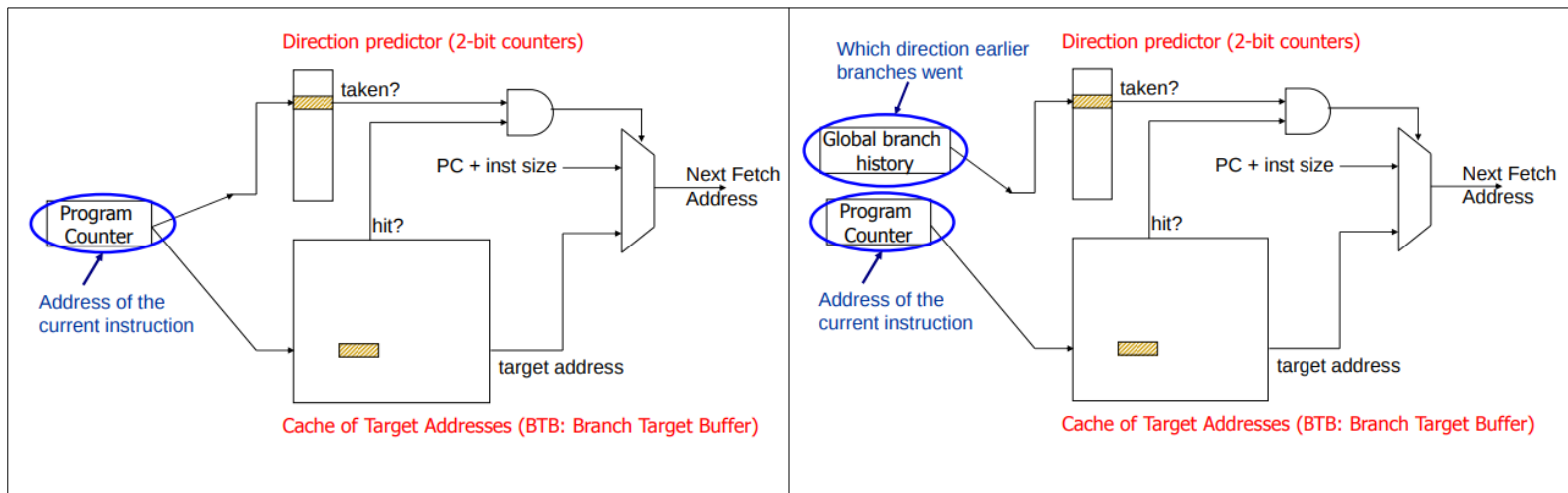
Global branch prediction en dos niveles

- Primer nivel:* GHR (N bits) => la direccion de los ultimos N branches
- Segundo nivel:* PHT (tabla de contadores con saturacion para cada entrada del historial) => la direccion que tomo el branch la ultima vez que se vio el mismo historial.

Mejorando el accuracy...

<p><u>Idea:</u> Agregar mas informacion de contexto al predictor global para tener en cuenta que branch se esta prediciendo.</p> <ul style="list-style-type: none">Gshare predictor: GHR hasheados con el PC del branch +mas info del contexto +mejor uso de PHT (prefix hash tree) --aumenta la latencia de acceso	
---	--

Global branch prediction en un nivel	Global branch prediction en dos niveles
--------------------------------------	---



Local branch correlation

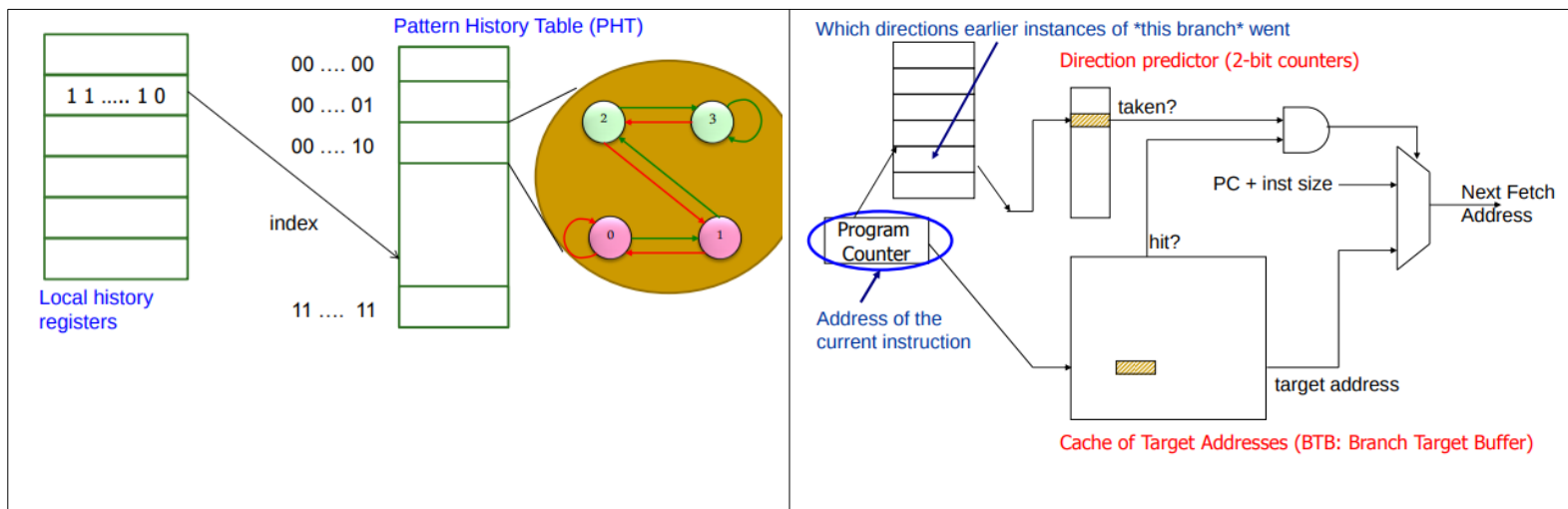
El resultado de una branch se relaciona con resultados anteriores de la misma branch, además de la última vez que se ejecuto.

Idea: Tener un registro de historial por branch

- Asociar el outcome predecido de una branch con el historial T/NT del mismo branch
- Basar la predicción en el resultado de la branch la última vez que se encontro el mismo branch history.

Usa dos niveles de historial: registro de historial del branch + historial con ese valor de registro de historial

- *Primer nivel:* Registros de historial local (N bits c/u) => eleccion del registro a partir del PC del branch
- *Segundo nivel:* Tabla de contadores con saturacion para cada entrada del historial => la direccion que tomo el branch la última vez que se vio el mismo historial



Branch predictors hibridos

Idea: Uso de mas de un tipo de predictor (múltiples algoritmos) y elegir la mejor predicción

Ventajas	Desventajas
+ Mejor accuracy: predictores distintos son mejores para branches distintos + Menos tiempo de “warmup”: predictores que tardan menos se usan hasta que esta listo el otro	-- Necesita un “meta-predictor” o selector -- Mas latencia de acceso

Biased Branches

Problema: Muchos branches tienden a ir en una direccion, y estos “contaminan” las estructuras de branch prediction: hacen que la prediccion de otras branches sea dificil porque causan “interferencia” en las tablas y registros de historial.

Solucion: Detectarlas y predecirlas con un predictor simple.

Multi-Path Execution

Idea: Ejecutar ambos caminos despues de un branch condicional

<i>Ventajas</i>	<i>Desventajas</i>
<ul style="list-style-type: none">Mejora performance si costo de error al predecir > trabajo en vanoNo hace falta modificar la ISA	<ul style="list-style-type: none">Que pasa si hay otro branch dificil de predecir?Cada camino requiere su propio contextoTrabajo en vano y reduccion de performance si los caminos se mergean

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)

Call and Return Prediction

- Calls directos son faciles de predecir: siempre se toman, un solo target guardado en el BTB
- Los returns son branches indirectos:
 - Una funcion puede ser llamada desde varios puntos del codigo => return puede tener muchos target addresses
 - Idea: Usar un stach para predecir direcciones de return (Return Address Stack)
 - Fetchear un call pushea el return address (siguiente instruccion) en el stack
 - Fetchear un return popea el stach y usa la direccion como su target predecido
 - Es accurate la mayoria de las veces

Prediccion de branches indirectos

Branches de registro indirecto tienen multiples targets => No requiere predecir la direccion

Idea 1: Predecir el ultimo target tomado como proxima fetch address

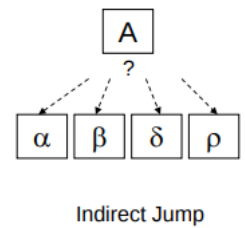
+Simple: usa BTB

--Poco accuracy: 50%. Muchos switchean target

Idea 2: Predecir en base al historial

+Mas accurate

--Un branch indirecto mapea demasiadas entries en BTB => miss por conflicto con otros branches y uso ineficiente del espacio si el branch tiene pocos target addresses



Problemas en Branch Prediction

- Identificar una branch antes del fetch.

Como hacer?

- Buscarlo en en BTB . Si hay Hit => Es una branch y nos dice el tipo de branch
- Si no hay BTB => hacer bubble en el pipeline hasta que se compute en el target address

- Latencia en prediccion

Necesitamos generar la siguiente fetch address para el proximo ciclo

Los predictores mas complejos tienen mas accuracy pero son mas lentos

Pipelining y Excepciones Precisas: Preservando Semantica Secuencial

Multi-Cycle Execution

No todas las instrucciones tardan el mismo tiempo en execute.

Idea: Tener multiples diferentes unidades funcionales que tardan un # de ciclos distinto.

- Se pueden pipelinear o no
- Pueden dejar que instrucciones independientes comiencen la ejecucion en una unidad funcional distinta mientras una instruccion de latencia larga termina de ejecutarse

Problema: No se preserva la semantica secuencial de la ISA! Que pasa si hay una excepcion?

FMUL R4 ← R1, R2
ADD R3 ← R1, R2

F D E E E E E E E E W

F D E W

F D E W

F D E W

F D E E E E E E E E W

F D E W

F D E W

FMUL R2 ← R5, R6
ADD R7 ← R5, R6

El write de la siguiente instruccion puede hacerse antes de que termine el execute de una anterior

Excepciones vs. Interrupts

	Excepciones	Interrupts
Causa	Interno al thread que se esta corriendo	Externo al thread que se esta corriendo
Cuando hacer handling	Cuando se detecta	Cuando es conveniente (excepto los de alta prioridad)
Prioridad	Proceso	Depende
Contexto de handling	Proceso	Sistema

Excepciones/Interrupts Precisos

El estado de la arquitectura debe ser consistente cuando la excepcion/interrupt esta listo para handling (instrucciones previas completamente retiradas y no instrucciones posteriores retiradas).

Retirada: ejecucion finalizada y estado de arquitectura actualizado.

Por que los queremos?

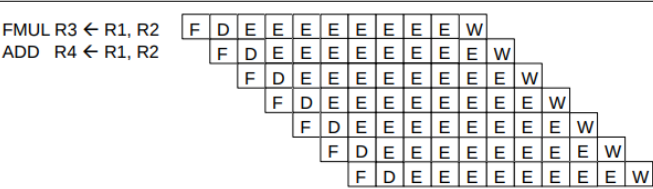
- Lo especifica el modelo de ISA de von Neumann
- Ayuda a debugear software
- Permite recuperarse facil de excepciones
- Permite que los procesos se reinicien facilmente
- Permite traps en el software

Asegurando excepciones precisas en el pipelining

Idea: Hacer que cada operacion tome la misma cantidad de tiempo

Desventajas:

- La instruccion de mas latencia determina la latencia de todas las otras
- Que pasa con las instrucciones de memoria?
- Cada unidad funcional toma el numero de ciclos del peor caso?



Posibles soluciones...

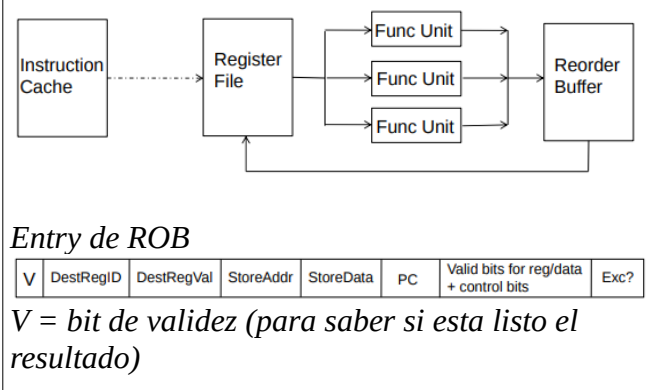
1. Reorder Buffer (ROB)

Idea: Completar instrucciones en desorden pero reordenarlas antes de hacer que los resultados sean visibles a la arquitectura

Decode de instruccion => entry en ROB

Instruccion completada => escribir el resultado en ROB

Cuando la instruccion mas vieja en el ROB se completa sin excepciones se mueve el resultado en reg o memoria.



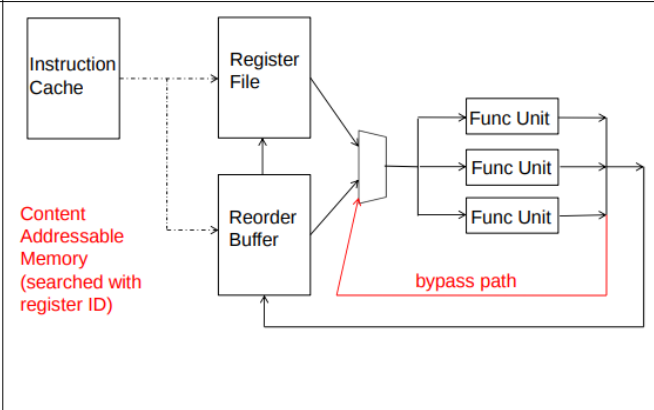
Que pasa si una operacion necesita un valor del reorder buffer? => se lee el ROB en paralelo con el register file

Como simplificar el acceso?

Idea: Usar indireccion

1. Acceder a register file

Si no es valido guarda el ID del ROB entry que tiene el valor del registro



Se mapea el register file a un ROB entry si hay una instruccion escribiendo en el registro

2. Acceder al ROB (que ya no necesita que el contenido sea addressable)

Cambio de nombre de registro con un ROB

Las output y anti dependencies no son dependencies reales: el mismo registro se refiere a valores que no tienen que ver con cada uno.

El ID de registro es renombrado como el ROB entry guardara el valor de registro.

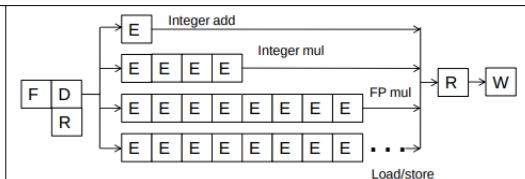
- ID de registro => ID de ROB entry
- ID de registro de arquitectura => ID de registro fisico
- Despues de renombrarlo, el ID de ROB entry se usa para referirse al registro

Esto elimina anti y output dependencies: da la ilusion de que hay un gran numero de registros.

Costo de guardado en ROB

Idea: Reducir guardado en ROB a traves de especializar por tipo de instruccion

- *Decode (D)*: Acceder regfile/ROB, guardar entry ahi, chequear si se puede ejecutar, si se puede hacer **dispatch**.
- *Execute (E)*: Instrucciones se pueden completar en desorden.
- *Completion (R)*: Escribir resultado en ROB
- *Retirement/Commit (W)*: Chequear por excepciones
 - *si no hay: escribir resultado a register file de arquitectura o memoria
 - *si hay: flush de pipeline y comenzar exception handler



Dispatch/execution => In-order
Completion => out-of-order
Retirement => in-order

Ventajas

- Conceptualmente simple para excepciones precisas
- Puede eliminar falsas dependencias

Desventajas

- Hay que acceder al ROB para obtener resultados que todavia no se escribieron al regfile.

2. History Buffer

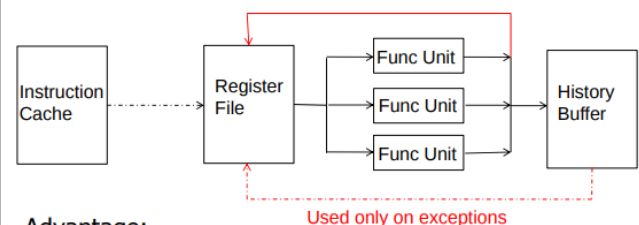
Idea: Actualizar el registro cuando se completa la instruccion pero deshacer actualizaciones cuando ocurre una excepcion.

Decode de instruccion => HB entry

Instruccion completada => guardar valor viejo de su dest. en HB

Cuando la instruccion es la mas vieja y no hay excep/interrupts => descartar HB entry

Cuando la instruccion es la mas vieja y hay excep/interrupts => se escriben los valores viejos en el estado de de la arquitectura.



Advantages:

<i>Ventajas</i>	<i>Desventajas</i>
<ul style="list-style-type: none"> El regfile tiene datos actualizados para las instrucciones siguientes 	<ul style="list-style-type: none"> Necesita leer el antiguo valor del dest. reg Necesita pasar por HB cuando hay una excepcion => mas latencia en el handling

ROB vs. HB

<i>ROB</i>	<i>HB</i>
<ul style="list-style-type: none"> Actualizacion de regfile pesimista Actualiza solo con valores no especulativos (en orden de programa) Complejidad en acceder a los nuevos valores 	<ul style="list-style-type: none"> Actualizacion de regfile pesimista Actualizacion inmediata pero se guarda el ultimo valor para recuperacion Complejidad en guardar valores viejos

Idea: tener ambos tipos de regfile

3. Future register file (FF + ROB)

Idea: tener dos regfiles (especulativo y arquitectura)

- Architectural regfile (arch regfile): actualizado en orden de programa para excepciones precisas (con un reorder buffer)

=> para recuperar el estado en excepciones (backend)
- Future regfile: actualizado apenas termina una instruccion (si la instruccion es la mas ultima en escribir a un registro)

=> para acceso rapido a los ultimos valores de registro (frontend)

<i>Ventajas</i>	<i>Desventajas</i>	
+No necesita leer los nuevos valores del ROB o antiguo valor del registro dest.	--Varios regfiles --Necesita copiar arch regfile a future file cuando hay excepcion	

<ul style="list-style-type: none"> <i>Decode (D)</i>: Acceder FF, encontrar ROB entry, chequear si se puede ejecutar y de ser asi hacer dispatch <i>Execute (E)</i>: Instrucciones se pueden completar en desorden. <i>Completion (R)</i>: Escribir resultado en ROB y FF <i>Retirement/Commit (W)</i>: Chequear por excepciones <ul style="list-style-type: none"> -*si no hay: escribir resultado a register file de arquitectura o memoria -*si hay: flush de pipeline, copiar arch. file a FF y comenzar exception handler 	<p><i>Dispatch/execution => In-order</i> <i>Completion => out-of-order</i> <i>Retirement => in-order</i></p>
--	---

Podemos reducir el costo de dos regfiles?

Idea: Usar indireccion (punteros a data en frontend y retirement)

- Un solo almacenamiento que guarda valores de datos de registros
- Tener dos mapas de registro (especulativo y arquitectura) => Register Alias Tables (RATs)

- Future Map => Acceso rapido a valores de registro (especulativo, frontend)
- Architectural Map => Para recuperar estado en excepciones (arquitectura, backend)

Si hay una excepcion...

Cuando se detecta que la instruccion mas vieja lista para retirarse causo una excepcion, la logica de control:

- Recupera estado de arquitectura
- Hace flush de todas las instrucciones siguientes del pipeline
- Guarda IP y registros (segun ISA)
- Redirecciona el fetch a la rutina de manejo de excepciones

Si se predice mal una branch...

- Si se predice mal es similar a una excepcion pero no lo ve el software (nivel microarquitectura)

Cuanto tardan en recuperar estado? Que pasos hay que hacer?

<i>Reorder Buffer</i>	<i>History Buffer</i>	<i>Future File</i>
<ul style="list-style-type: none"> • Hacer flush de instrucciones en el pipeline posteriores a la branch • terminar todas las instrucciones en el reorder buffer 	<ul style="list-style-type: none"> • Hacer flush de instrucciones en el pipeline posteriores a la branch • Deshacer todas las instrucciones posteriores a la branch desde el final del history buffer hasta el branch y restaurar uno por uno los valores de regfile 	<ul style="list-style-type: none"> • Esperar que la branch sea la instruccion mas vieja en la maquina • Copiar arch. regfile a future file • Hacer flush de todo el pipeline

Se puede mejorar? => Restaurar el frontend (FF) para que la siguiente instruccion correcta despues del branch se pueda ejecutar directamente despues de resolver el branch misprediction

4. Checkpointing

Idea: Hacer un Checkpoint del estado/mapa de registros del frontend en el momento de hacer decode de una branch y mantener ese estado actualizado con valores de instruccion anteriores a la branch.

Decode de branch => Hacer copia del F F/M y asociarlo a la branch.

Instruccion produce valor de registro => Todos los checkpoints de F F/M posteriores a la instruccion son actualizados con el valor.

Se detecta un branch misprediction =>

- Restaurar el checkpoint de F F/M de la branch cuando se resuelve el misprediction
- Hacer flush en el pipeline de las instrucciones posteriores a la branch
- Borrar checkpoints posteriores a la branch.

<i>Ventajas</i>	<i>Desventajas</i>
<ul style="list-style-type: none"> • Estado de registros frontend correcto disponible apenas se hace el checkpoint restoration => Poca latencia para recuperar estado 	<ul style="list-style-type: none"> • Costo de almacenamiento • Complejidad al manejar checkpoints

5. Readings

Registros vs. Memoria

	<i>Registros</i>	<i>Memoria</i>
<i>Dependencias</i>	Se conocen estaticamente	Se determina dinamicamente
<i>Estado</i>	Pequeña	Grande
<i>Visibilidad de estado</i>	No visible a otros threads/procesadores	Compartida entre threads/procesadores

Manteniendo el estado especulativo de memoria

Al manejar el completado de operaciones de memoria out-of-order...

- Deshacer una escritura en memoria es mas dificil que una en registro => usar un ROB
- **Store/write buffer** => similar a ROB pero solo para instrucciones de guardado
 - Operaciones de guardado no commiteadas en orden de programa
 - Decode de store => hago un Store Buffer Entry
 - Cuando la direccion y data de guardado estan disponibles los guardo en SB Entry
 - Cuando el store es la instruccion mas vieja en el pipeline => actualizo la direccion de memoria con la data del SB

Scheduling estatico vs. Dinamico

Rol de hardware/software en el orden de ejecucion en pipeline:

- Scheduling basado en software => estatico
- Scheduling basado en hardware => dinamico

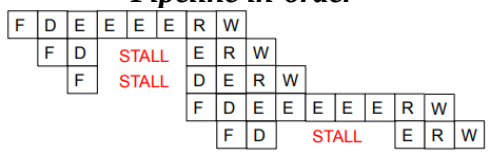
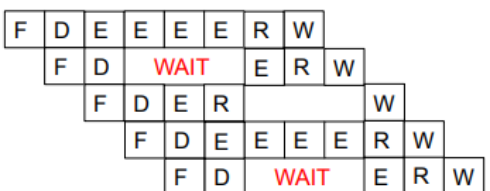
Scheduling Estatico

Problema: El compilador no sabe nada de las cosas que se determinan al correr (variaciones de latencia, direccion de branch, etc)

Scheduling Dinamico

El hardware sabe sobre eventos dinamicos para cada instruccion (cache miss, branch misprediction, etc)

Ejecucion out-of-order

<p><i>Pipeline in-order</i></p>  <p><i>Pipeline out-of-order</i></p> 	<p><u>Problema de ejecucion in-order:</u> Dependencia de data real ralentiza el despacho de instrucciones a unidades de ejecucion. Las maneras de prevenir dependencia tienen desventajas.</p> <p>=> hago dispatch (scheduling y ejecucion) out-of-order</p> <p><u>Idea:</u> Mover instrucciones dependientes a un area de descanso para poder ejecutar las independientes.</p> <ul style="list-style-type: none">• Monitorear los sources de los valores de instrucciones de area de descanso
---	--

16 ciclos vs. 12 ciclos

- Cuando estan todos listos los de una instruccion => hago dispatch

Beneficio: Tolerancia de latencia

Pasos para hacer ejecucion OoO

1. Linkear el consumidor de un valor con el productor => **cambio de nombre de registros**: asociar un tag con cada valor
2. Hacer buffer de instrucciones hasta que estan listas para ejecutarse => ponerlas en **estaciones de reserva**
3. Tener registro de valores source => **emitir el tag** cuando se produce el valor y **compararlo** con los de las otras instrucciones (si hay match marco como listo el valor)
4. Cuando todos los valores source de una instruccion estan listos, hago dispatch

Riesgos de OoO

- *Write After Read (WAR)*
- *Write After Write (WAW)*
- *Read After Write (RAW)*
- *Excepciones imprecisas*: son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden. Posibles motivos:
 - El pipeline ha completado la ejecución de una o mas instrucciones posteriores a la que produce la excepción.
 - El pipeline no ha completado aún al menos una instrucción previa a la que genera la excepción.

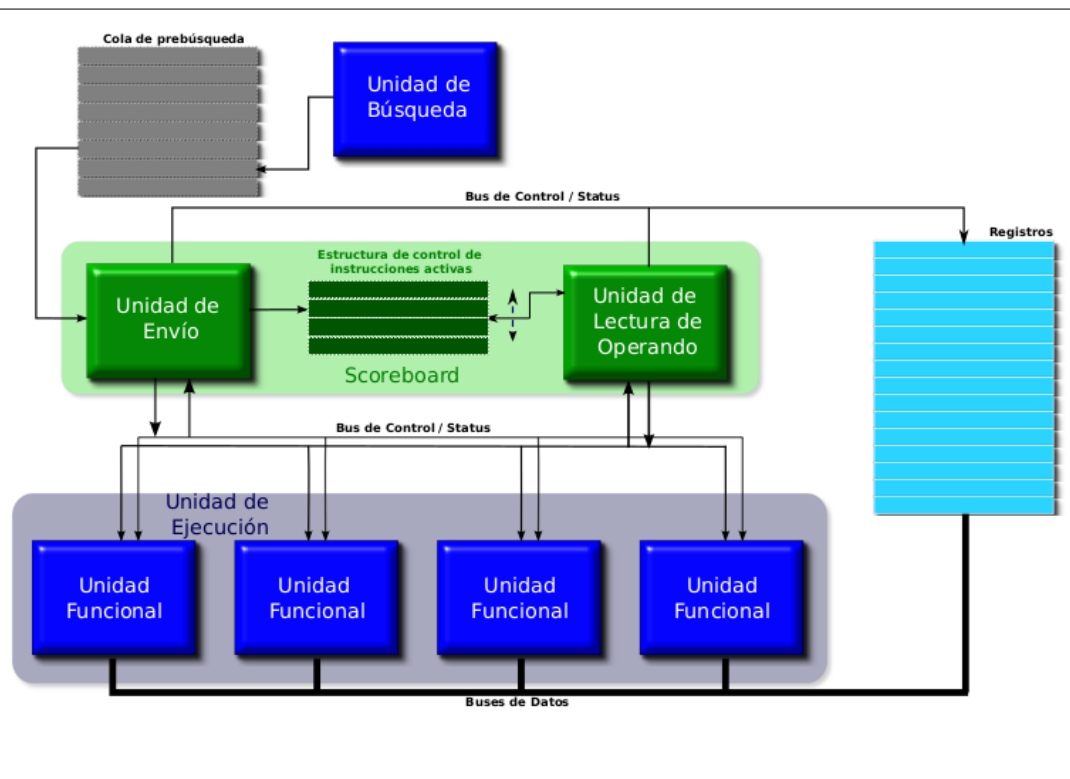
Scoreboarding

La manera mas sencilla de implementar OoO sin WAR y WAW.

Limitaciones:

-Nuevos Obstáculos estructurales debido a que la cantidad de buses es limitada para paralelizar las transferencias entre el scoreboard y los registros.

- Los operandos se leen directamente en los registros => no Forwarding



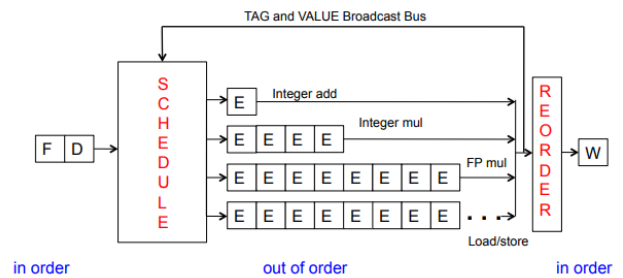
Algoritmo de Tomasulo

La solución actual a los problemas de OoO. Minimiza RAW e implementa register renaming en WAR y WAW

Schedule/ Reservation Station (RS): Implementa las funciones:

- Mantiene instrucciones hasta que estén listas para ejecución
 - Indica cuando una instrucción esperando operando(op) tiene los mismos en ready
 - Despacha la instrucción a la unidad funcional cuando sus op están ready
- Recibe cuando los op están ready a través de broadcasts de la unidad de ejecución.

De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo



Pipeline Moderno

Reorder/ROB: Almacena los resultados de las instrucciones ejecutadas desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.

Contiene: tipo de instrucción, destino, valor, ready

Tabla de renombre de registros

	tag	value	valid?
R0			1
D1			1

Es el link entre el productor de dato y su consumidor. Asocia un "tag" con cada operando.

- **If (RS tiene recursos disponibles antes de renaming)**
 - insertar en **RS** instr + op renombrados (valor + **tag**)
 - se renombra \Leftrightarrow **RS** tiene recursos disponibles
- **Else stall**
- **While** (esté en la **RS**, cada instrucción debe:)
 - Mirar el **Common Data Bus (CDB)** en busca de **tags** que correspondan a sus op 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

Ejecución Out-of-Order con Excepciones Precisas

Idea: Usar un ROB para reordenar instrucciones antes de committearlas al estado de la arquitectura

- Una instr. actualiza la tabla de alias del registro (como un FF) cuando completa su ejecución
- Una instr. actualiza el regfile de arquitectura cuando es el más antiguo en la máquina y completa la ejecución

En procesadores modernos, se usa...

- Un ROB para mantener el retiro de instrucciones in-order

- Un regfile para guardar registros (especulativos y de arquitectura)
- Future Register Map para renombrar
- Architectural Register Map => para recuperar estados

Ejecucion OoO: Dataflow Restringido

Una maquina OoO construye el grafico de dataflow de un programa dinamicamente

=> esta limitado a la **Ventana de Instruccion** (las instrucciones que se hizo decode pero no se terminaron), es decir, por la microarquitectura

Manejo de dependencias de memoria

Hay que mantener dependencias en una maquina OoO con una buena performance y la direccion de memoria no se conoce hasta que se ejecuta un load/store =>

- Renombrar direcciones de memoria es dificil
- Determinar dependencia o independencia de load/store se debe hacer despues de la ejecucion parcial
- Cuando un load/store tiene su direccion lista, puede haber loads/stores menores/mayores de los que todavia no se determino la direccion

Donde pongo un load en el schedule de una OoO?

La dependencia no se conoce hasta que todos los stores anteriores estan disponibles. Como se determina la dependencia?

- *Opcion 1:* Esperar a que se commiten todos los stores anteriores (no hace falta chequear)
- *Opcion 2:* Tener todos los stores pendientes en un buffer y chequear si las direcciones coinciden

Como trata la OoO la instruccion load en el schedule?

- *Conservador:* Asumir que es dependiente de todos los stores anteriores. Retrasar el load hasta que todos los store anteriores hayan computado su direccion.
 - + No necesita recovery
 - Retrasa los loads independientes en vano
- *Agresivo:* Asumir que es independiente de todos los stores anteriores y ponerlo directamente.
 - + Simple y puede ser el caso comun
 - Requiere recovery y re-ejecucion de load y dependientes si hay mispredict
- *Inteligente:* Predecir la dependencia (basado en el historial es bueno)
 - + Mas accuracy. Las dependencias de load/store permanecen en el tiempo
 - Requiere recovery y re-ejecucion de load y dependientes si hay mispredict

Data forwarding entre stores y loads

No se puede actualizar la memoria de un programa OoO => usamos una Load Queue (LQ) y Store Queue (SQ)

- Un load busca en SQ despues de computar direccion

- Un store busca en LQ despues de computar direccion

Data flow: Exploiting Irregular Paralellism

En una maquina de data flow, un programa consiste de nodos que disparan cuando todos los inputs estan listos

Representacion en la ISA

*	R	ARG1	R	ARG2	Dest. Of Result
---	---	------	---	------	-----------------

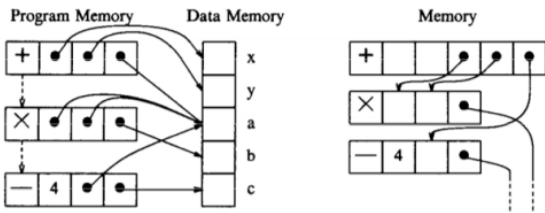
Los valores se representan como token:

token $\langle ip, p, v \rangle$

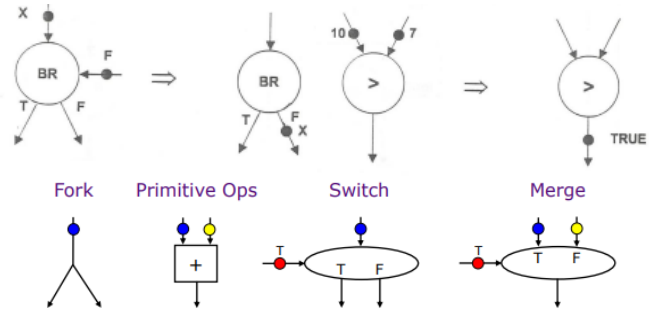
instruction ptr port data

Control flow vs. Data flow

a := x + y
b := a × a
c := 4 - a



Nodos de dataflow condicional y relacional



El de la izquierda es un programa de **control flow** y las flechas son las direcciones de memoria usadas o creadas. Las flechas punteadas son el control flow.

El de la derecha es de **data flow**, en el que solo se involucra una memoria. Cada instruccion tiene punteros a todas las instrucciones que consumen sus resultados

Caracteristicas de Data Flow

- La ejecucion a nivel de instruccion de codigo grafico la regula la *data*, no el control
- Solo las dependencias reales restringen el procesamiento
- No hay stream de instrucciones secuenciales (no PC)
- Ejecucion triggereada por la presencia de data
- Las operaciones se ejecutan asincronicamente

Que pasa con los loops y llamados de funcion?

Problema: Multiples instancias dinamicas pueden estar instancias activas de la misma instruccion. No alcanza el IP para distinguirlas.

Solucion: Distinguir entre instancias creando nuevos tags/frames

a tagged token $\langle fp, ip, port, data \rangle$

frame pointer (tag or context ID) instruction pointer

Ventajas y desventajas de data flow

Ventajas	Desventajas
<ul style="list-style-type: none"> • Bueno explotando paralelismo irregular • Solo dependencias reales restringen procesamiento 	<ul style="list-style-type: none"> • Debuggear es dificil (no hay un estado preciso, dificil handling de excepcion/interrupcion) • La implementacion de estructuras de datos dinamicas en modelos de data flow puros es dificil • Demasiado paralelismo? Se requiere control • Mucho costo de bookkeeping (tag matching, data storage) • Ciclo de instruccion ineficiente, no se explota localidad de memoria

Combinando Data y Control Flow

- *Modelo 1:* Control flow a nivel de ISA, dataflow debajo, preservar semanticas secuenciales
- *Modelo 2:* Mantener dataflow, pero incorporar control flow al nivel de ISA para mayor eficiencia, explotar localidad y facilitar el manejo de recursos.

Incorporar hilos en el dataflow: Instrucciones ordenadas estaticamente, cuando la primera instruccion dispara, las demas se ejecutan sin interrupcion

Jerarquia de memoria

Memoria Virtual vs. Fisica

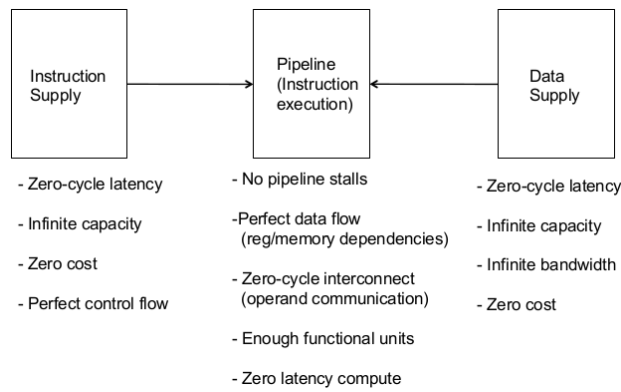
El programador ve **memoria virtual**, se puede asumir que es “infinita”

La **memoria fisica** es mucho mas chica de lo que asume el programador. El sistema mapea direcciones de memoria virtual a memoria fisica y la maneja transparentemente para el programador (el costo lo tiene el software y arquitectura).

Sistema de memoria fisica

Se necesita un nivel de almacenamiento mayor para manejar una pequena cantidad de memoria fisica automaticamente.

Idealmente...



Tecnologias de memoria

No volatiles (ROM)	Volatiles (RAM)
<ul style="list-style-type: none">• Retienen informacion cuando se desconecta la alimentacion• Modificables en tiempo real	<ul style="list-style-type: none">• Una vez interrumpida la alimentacion, pierden la informacion almacenada• Pueden almacenar mayores cantidades de informacion y modificarla en tiempo real a gran velocidad en comparacion con las no volatiles (menor tiempo de acceso)• Se clasifican en dinamicas (DRAM) y estaticas (SRAM)

Memoria Ideal vs. Problemas

- Cero tiempo de acceso (latencia) => mas rapido es mas caro
- Capacidad infinita => mas grande es mas lento
- Cero costo
- Ancho de banda infinito (accesos en paralelo) => mas ancho es mas caro

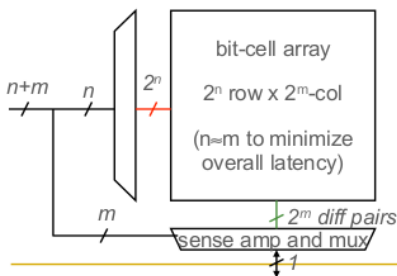
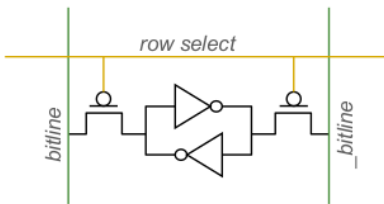
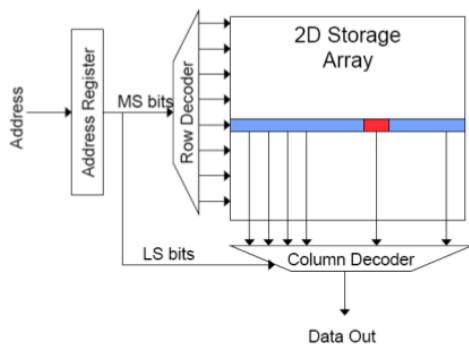
Tecnologias de memoria

- DRAM (Dynamic Random Access Memory)
 - El estado de carga de capacitor indica el valor guardado (1 capacitor + 1 transistor de acceso)

- El capacitor pierde a través del camino RC (pierde carga y debe ser refrescado)
- Generalmente esta en estado de corte. Consume minima energia.
- **SRAM** (Static Random Access Memory)
 - Dos inversores cruzados guardan un bit (4 transistores para guardado + 2 transistores para acceso)
 - El feedback permite que el valor guardado permanezca en la celula

DRAM	SRAM
<ul style="list-style-type: none"> • Acceso mas lento (Requiere refresh) • Alta densidad (1T 1C) • Costo menor • Consumo minimo • Alta capacidad de almacenamiento • Requiere poner un capacitor y logica juntas 	<ul style="list-style-type: none"> • Acceso mas rapido (No necesita refresh) • Menor densidad (6 T) • Costo mayor • Consumo alto • Baja capacidad de almacenamiento • Manufactura compatible con el proceso logico (no capacitor)

Organizacion y operacion del banco de memoria



Acceso de read:

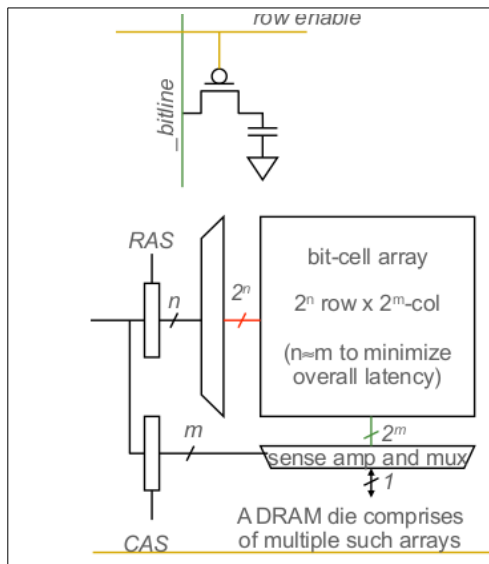
1. Decodificar la address de row y llevar los word-lines
2. Los bits seleccionados llevan los bit-lines (lectura de row completa)
3. Amplificar la data de row
4. Decodificar la direccion de column y seleccionar un subset de la row y enviarlo a output
5. Precargar bit-lines para siguiente acceso

Acceso de read de SRAM

1. Decodificar la address
2. Llevar row select
3. Los bit-cells seleccionados llevan bitlines (se lee la row entera)
4. Sensado diferencial y column select (la data esta lista)
5. Precargar todos los bitlines (para el siguiente read o write)

Latencia principal en datos 2 y 3

Tiempo de ciclo dominado por pasos 2, 3 y 5



DRAM

Bits guardados como cargas en el capacitor del nodo
=> La celula pierde carga con el read y a traves del tiempo

Acceso de read de DRAM

Pasos 1~3 igual a SRAM

4. Un sense amp que flip-flopea amplifica y regenera el bitline
5. Precargar todos los bitlines

Para mantenerlos cargados, el controlador debe leer cada row en el tiempo de refresco para recargarlos.

Por que jerarquia de memoria?

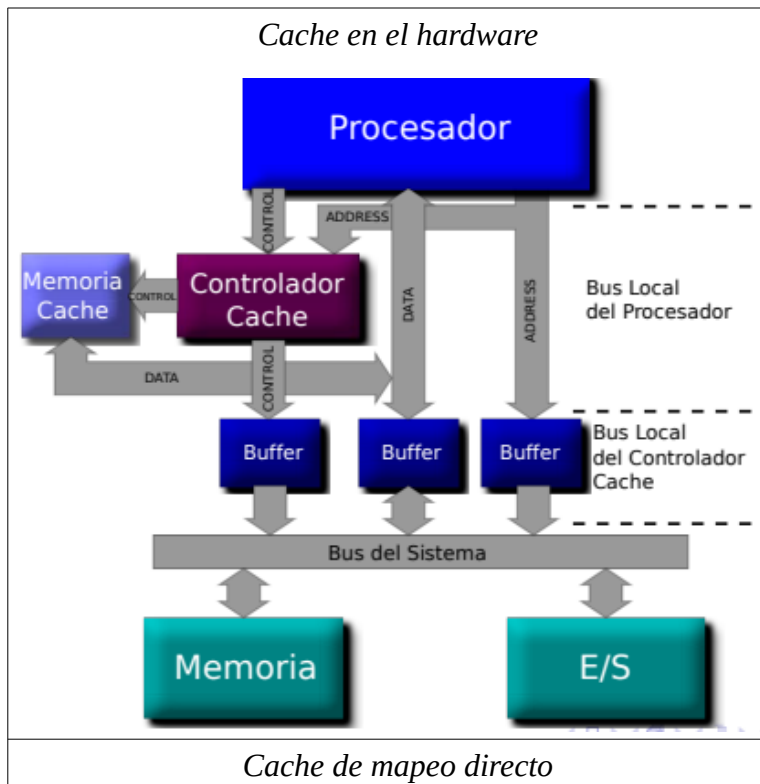
Idea: Queremos que la memoria sea grande y rapida, entonces usamos niveles de memoria (progresivamente mayor y mas lento al alejarse del procesador) y hacemos que la mayor cantidad de data que el procesador necesita este en los niveles mas rapidos

Cache

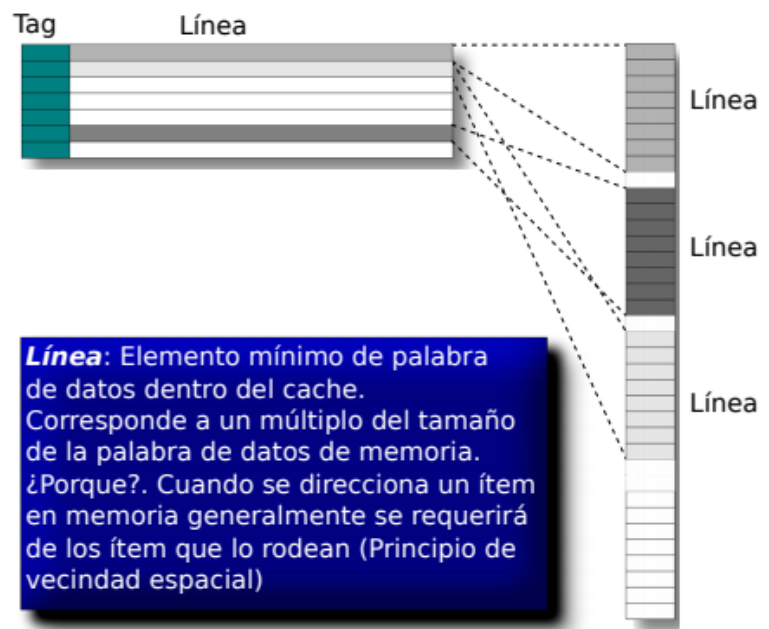
Idea: Guardar la data accedida en el cache (SRAM de alta velocidad) ya que la data accedida recientemente sera accedida en el futuro cercano. Requiere hardware

El tamaño de la cache debe ser:

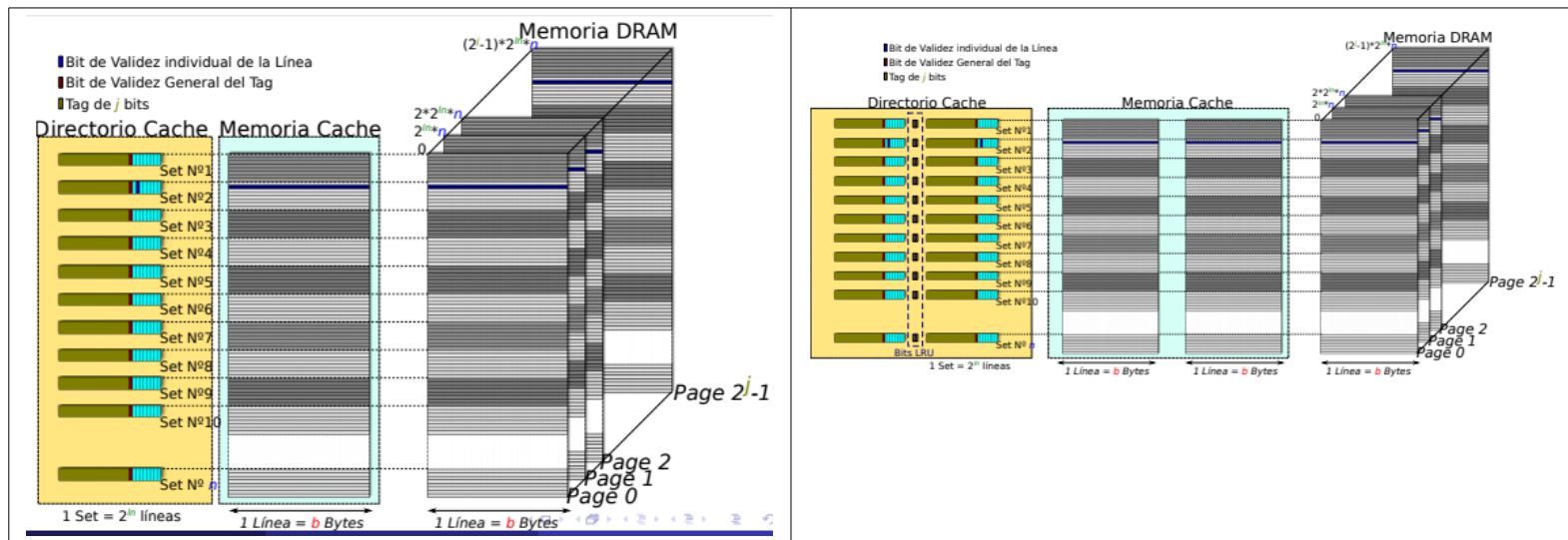
- Lo suficientemente grande para que el procesador resuelva la mayor cantidad posible de busquedas de codigo y datos en esta memoria asegurando una alta performance
- Suficientemente pequeña para no afectar el consumo ni el costo del sistema



Organizacion de cache: lineas



Cache asociativo a 2 vias



Explotando localidad espacial

Idea: Guardar en cache direcciones adyacentes a la accedida recientemente

- Dividir la memoria lógicamente en bloques de igual tamaño
- Guardar en cache el bloque entero

Cache en el diseño de pipeline

Necesitamos integrar el cache al pipeline para, idealmente, acceder en un ciclo.

Si el pipeline es de alta frecuencia no se puede hacer muy grande el cache => **jerarquía de cache**

Manejo Manual vs. Automatico

Manual	Automatico
El programador maneja el movimiento de data entre niveles -- Malo para el programador	El hardware maneja la data entre niveles transparentemente para el programador + Facilita el trabajo del programador
Decisiones de diseño de cache <ul style="list-style-type: none"> • <i>Ubicacion:</i> donde y como ubicar un bloque en cache • <i>Reemplazo:</i> que data borro para hacer lugar? • <i>Granularidad:</i> bloques grande, chicos, uniformes? • <i>Politica de escritura</i> • <i>Instrucciones/data:</i> se tratan por separado? 	Jerarquia de memoria moderna
Metricas de cache <ul style="list-style-type: none"> • $\text{Hit Rate} = \frac{\# \text{hits}}{\# \text{accesos}}$ • $\text{AMAT (Average Memory Access Time)} = (\text{hit-rate} * \text{hit-latency}) + (\text{miss-rate} * \text{miss-latency})$ 	

Políticas de escritura

Las variables en la cache tambien estan en alguna direccion de la DRAM, idealmente ambas manteniendo el mismo valor.

Si esto deja de pasar, hay diferentes maneras de actuar: **políticas de escritura.**

- **Write-through:** El procesador escribe en la DRAM y el controlador cache refresca la cache con el dato actualizado
=> garantiza coherencia pero penaliza escrituras con tiempo de acceso a DRAM
=> refresco inmediato, alto costo de performance
+ Pueden haber multiples writes en el mismo bloque antes de echarlo
-- Requiere dirty bit en tag store
- **Write-through buffered:** El procesador actualiza la SRAM cache, y el controlador cache luego actualiza la copia en memoria DRAM mientras el procesador continua ejecutando instrucciones y usando datos de la cache. Para eso, el controlador cache tiene un buffer de escrituras para encolar las operaciones de escritura a memoria.
=> refresco lo antes posible sin afectar performance
- **Copy-back:** Se marcan como modificadas (o dirty) las lineas de la memoria cache que contienen variables en cuya direccion el procesador escribio. Solo actualiza la copia de memoria cuando elimina la linea del cache.
=> refresco solo al borrarlo de cache
+ Mas simple
+ Todos los niveles estan actualizados. **Consistencia:** Coherencia de cache simple porque no necesita chequear niveles de abajo
-- Requiere mas ancho de banda, no se unen los writes

Asignamos un bloque de cache en un write miss?

- **Allocate on write miss:** Si
+ Consolida writes en vez de escribir cada uno individualmente en el siguiente nivel
+ Simple, write y read miss se tratan igual
-- Requiere transferir el bloque de cache entero
- **No-allocate on write miss**
+ Conserva espacio de cache si la localidad de write es baja (posible mejor hit rate)

Diseño basico de cache en hardware

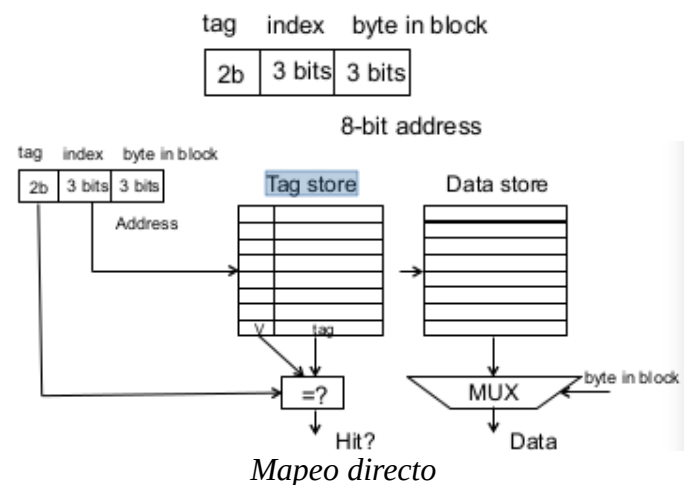
Bloques y direccionamiento

Se divide la memoria en bloques de tamaño fijo: cada uno mapea a una ubicacion en el cache determinada por los bits de indice en la direccion.

Para acceder:

1. Indexar en el tag y data store con los bits index
2. Chequear bit de validez en el tag store
3. Comparar los bits de direccion con los del tag store

Tag store tiene: Bit de validez, tag, bits de replacement policy, y si es necesario, dirty bit



Tipos de mapeo

- *Mapeo directo*
- *Asociatividad de set:* en vez de tener una columna de N bloques, tengo M de N/2M bloques para tener memoria asociativa
 - + Mejor adaptacion a conflictos (dos bloques muy demandados que compiten por la misma posicion)
 - Mas complejo, acceso mas lento, tag store mas grande
- *Asociatividad completa:* Un bloque puede ser colocado en cualquier posicion de cache

Grado de asociatividad: cuantos bloques mapean al mismo indice (o set)?

Tradeoffs de mas asociatividad:

- + Mayor hit rate
 - Mas latencia de acceso
 - Hardware mas caro (mas comparadores)
- Diminishing returns de asociatividades mayores

Problemas en caches con asociatividad de set

Si cada bloque en un set tiene una prioridad que nos dice que tan importante es que se quede en el cache, como determinamos esa prioridad?

- *Insercion:* Se coloca en cache? Donde?
- *Promocion:* Como y cuando cambio la prioridad de bloque?
- *Reemplazo:* Cual bloque sacar? Como ajusto prioridades?

Primero reemplazo un bloque invalido, y luego...

- *Random*
- *FIFO (First In First Out)*
- *LRU (Least Recently Used)*

Se necesita saber el orden de acceso de los bloques. La mayoría de los procesadores no usan “LRU perfecto” en cache altamente asociativo: es muy complejo y de todos modos es una aproximacion.

- *Not MRU (Not Most Recently Used) o LRU Jerarquico*

Dividir el set en grupos, solo tener registro del grupo MRU y el bloque MRU en cada grupo. Para reemplazar elegir uno random que no sea del bloque o grupo guardado.

- *Least Frequently Used*
- *Least Costly to Re-Fetch*
- *Victim/Next-Victim (V/NV)*

Solo guardar el status de 2 regs: V y NV, los demas son O (ordinario).

miss => reemplazo V, NV ahora es V, elijo NV random

hit a V => NV ahora es V, elijo NV random, V ahora es O

hit a NV => elijo NV random, NV ahora es O

- *Hibrido*

Set thrashing: Cuando el “set de trabajo del programa” es mas grande que la asociatividad del set. Causa que haya que reemplazar constantemente lo que esta en cache, al punto de que no nos sirva tenerlo porque desborda.

=> conviene usar random

Lo ideal es un hibrido de LRU y random y elegir a traves de **Set Sampling**

Politica de reemplazo optima? Belady's OPT

- Reemplazar el bloque que va a ser referenciado mas lejos en el futuro por el programa.
Es optimo para minimizar miss rate y tiempo de ejecucion? NO, eso varia de bloque a bloque porque se solapan caches remotos vs. Locales y los miss

Cache vs. Reemplazo de Pagina

La memoria fisica (DRAM) es una cache para el disco, se maneja por software como memoria virtual.

El reemplazo de pagina es similar al de cache; el page table es un “tag store” para el guardado de memoria fisica. La diferencia entre cada uno es...

- Tiempo de acceso
- #bloques
- Tiempo para encontrar un candidato de reemplazo
- Rol de hardware vs software

Caches sectorizados

Idea: Dividir el bloque en subbloques, tener bits de validez y dirty de cada uno.

- + No necesita transferir el bloque entero de cache en la cache (un write solo valida y actualiza el subbloque)
- + Mas libertad para transferir subbloques a la cache (un bloque no tiene que estar entero en la cache)
- Diseño mas complejo
- No aprovecha la localidad espacial completamente cuando se usa para reads



Cache de instruccion vs data: Unidos o separados?

Unidos:

- + Espacio de cache compartido dinamicamente: no se aloca mas espacio que el estrictamente necesario
- Instrucciones y data pueden thrashearse entre si (no hay espacio garantizado para ninguna)
- I y D se acceden se acceden en diferentes lugares del pipeline, donde ponemos el cache unificado?
 - El cache de 1er nivel esta casi siempre **separado**
 - las decisiones son muy afectadas por el tiempo de ciclo
 - hay asociatividad baja
 - tag store y data accedidos en paralelo.

- Los demas, estan casi siempre **unificados**
 - balancea hit rate y latencia de acceso
 - son grandes y altamente asociativos, porque la latencia no es tan importante
 - tag store y data accedidos serialmente

Accesos de niveles seriales vs paralelo

- Serial: El segundo nivel de cache se accede solo si en el primer nivel hay miss
=> el primer nivel es un filtro, las politicas de manejo son distintas

Performance de cache: Parametros vs Hit/Miss Rate

Tamaño de cache: Capacidad de data (sin tag)

Si es muy grande usa mejor la localidad temporal, pero no siempre es mejor

- *Cache muy grande* afecta negativamente el la latencia de hit y miss
- *Cache muy chico* no aprovecha la localidad temporal bien, y reemplaza data util muy seguido

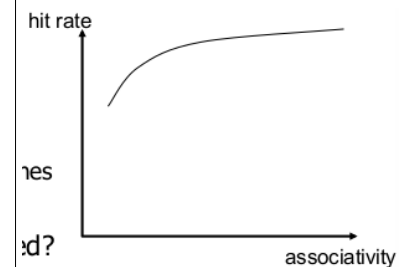
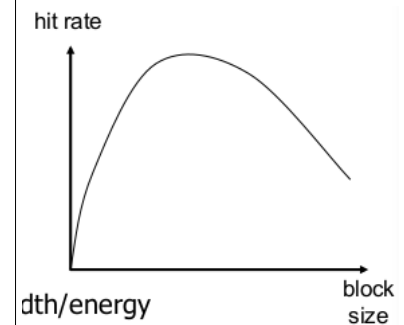
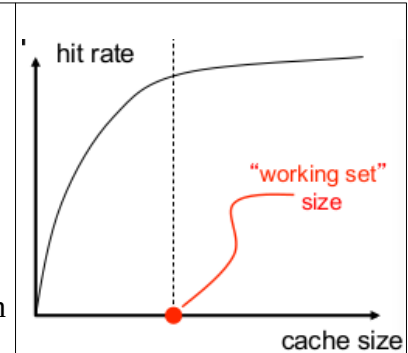
Working set: El conjunto entero de data que referencia la ejecucion de una aplicacion en un intervalo de tiempo

Tamaño de bloque: La data asociada con un address tag, no necesariamente la unidad de transferencia entre niveles.

- *Bloques muy chicos* no aprovechan bien la localidad y tienen un costo de tag mas grande
- *Bloques muy grandes*:
 - * tienen menor # de bloques => menor aprovechamiento de localidad temporal
 - * desperdician espacio de cache y ancho de banda si no hay mucha localidad espacial => conviene usar subbloques
 - * pueden tomar mucho tiempo en guardarse en cache => se guarda la palabra critica primero y reinicia el acceso a cache antes de terminar de llenar

Asociatividad: Cuantos bloques pueden mapear al mismo indice o set?

- *Asociatividad mayor*: Menos miss rate, menos variacion entre programas, mayor latencia de hit
- *Asociatividad menor*: menor costo, menor latencia de hit (especialmente L1)



Clasificacion de miss y como reducirlos

- Compulsory miss: Primer referencia a un bloque y siguientes solo si se lo echo
=> precaching
- Capacity miss: El cache es demasiado chico como para guardar todo lo necesario. Definido como los misses que ocurrirían hasta en una cache completamente asociativa con reemplazo optimo de la misma capacidad

=> Usar mejor el espacio de cache (quedarse con bloques que van a ser referenciados) y manejo de software (dividir el working set para que cada frase entre en el cache)

- Conflict miss: Miss que no es compulsory o de capacidad
=> mas asociatividad o alternativas (victim cache, hashing, pistas de software)

Mejorando performance de cache

- Reducir miss rate:
 - Mas asociatividad
 - mejores politicas de reemplazo
 - alternativas a asociatividad
 - manejo por software
- Reducir latencia de miss:
 - cache en varios niveles
 - non-blocking caches (varios miss en paralelo)
 - palabra critica primero
 - varios accesos por ciclo
 - subbloques
 - manejo por software
 - mejores politicas de reemplazo
- Reducir latencia de hit

Coherencia de cache

Si hay varios procesadores modificando variables, si un procesador modifica una variable, el otro debe enterarse cuanto antes del cambio por si tiene esa misma variable en su cache.

Como nos aseguramos que los caches esten actualizados?

- **Snoop bus**:

Idea: Todos los caches espian los read/write requests de los otros y mantienen la coherencia del bloque
Es facil de implementar si todos los caches comparten un bus. Es bueno para multiprocesadores de escala pequeña

El **controlador de cache (CC)** marca en su directorio cache interno las direcciones de memoria que tiene almacenadas en el cache.

El **snoop bus** es un conjunto de lineas entrantes al controlador de cache provenientes del bus de address del sistema para ver si cada operacion de lectura y escritura sobre direcciones de memoria del sistema y chequea si estan en su cache. De ser asi y es una escritura, la invalida.
- **Directory**:

Un punto de serializacion *por bloque*.
Procesadores hacen pedidos especificos para bloques
El directorio lleva registro de que cache tiene cada bloque y coordina la invalidacion y actualizaciones

Idea: un directorio logicamente central tiene registro de donde estan las copias de cada bloque de cache, los caches consultan este directorio para asegurarse coherencia

Ejemplo: Cada bloque de cache en la memoria guarda P+1 bits en el directorio (1 bit por cada cache indicando si esta en cache; 1 bit exclusivo indicando que un cache tiene la unica copia del bloques y puede actualizarlo sin notificar a otros)

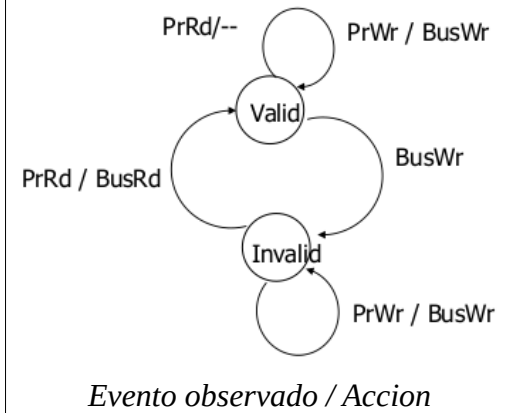
- En un read: setear el bit de cache y organizar el supply de data
- En un write: invalidar todos los caches que tienen el bloque y resetear sus bits

De todos modos, esto se puede optimizar considerando las politicas de escritura (en particular, copyback) usando protocolos de coherencia.

Ejemplos de protocolos de coherencia por directorio

VI (Valid Invalid)

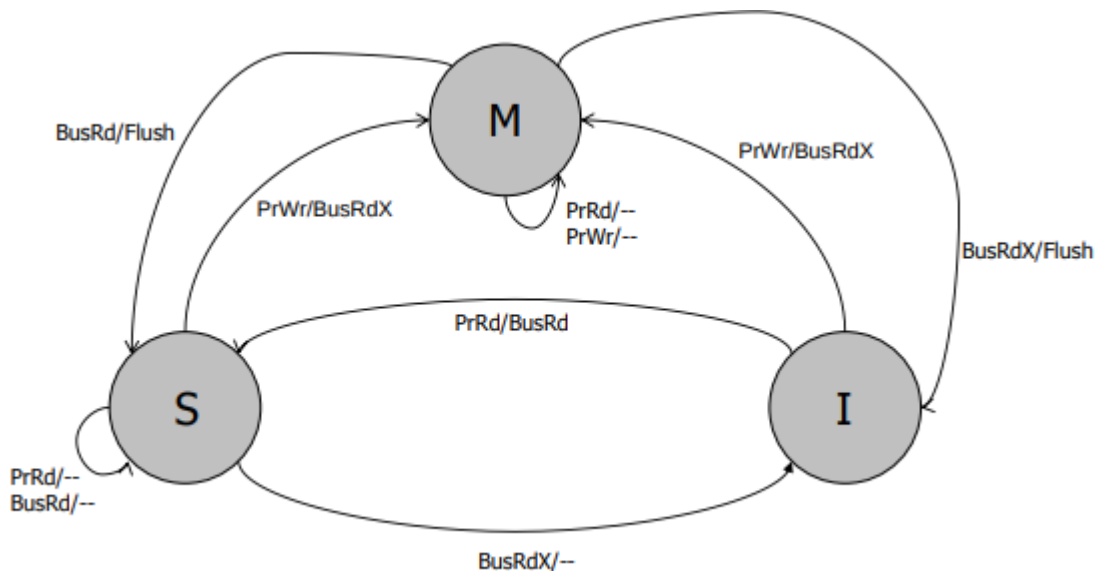
- Write-through, no-write-allocate cache
- Acciones del procesador local en el bloque de cache: PrRd, PrWr
- Acciones que se transmiten en el bus para el bloque: BusRd, BusWr



MSI (Modified Shared Invalid)

Extiende la metadata por bloque para tener 3 estados:

- **Modified:** La linea de cache es la unica copia de cache y es dirty. Requiere *Write-Back* a la memoria antes que otro procesador lea desde alli el dato (xq ya no es valido).
- **Shared:** La linea de cache es potencialmente una de varias copias
- **Invalid:** La linea no esta presente en este cache



Read miss => *Read* request en bus y transiciona a **S**

Write miss => *ReadEx* request y transiciona a **M**

Procesador espia *ReadEx* de otro writer => invalida su copia (si tiene)

Actualizacion $S \rightarrow M \Rightarrow$ sin releer data de la memoria (por invalidaciones)

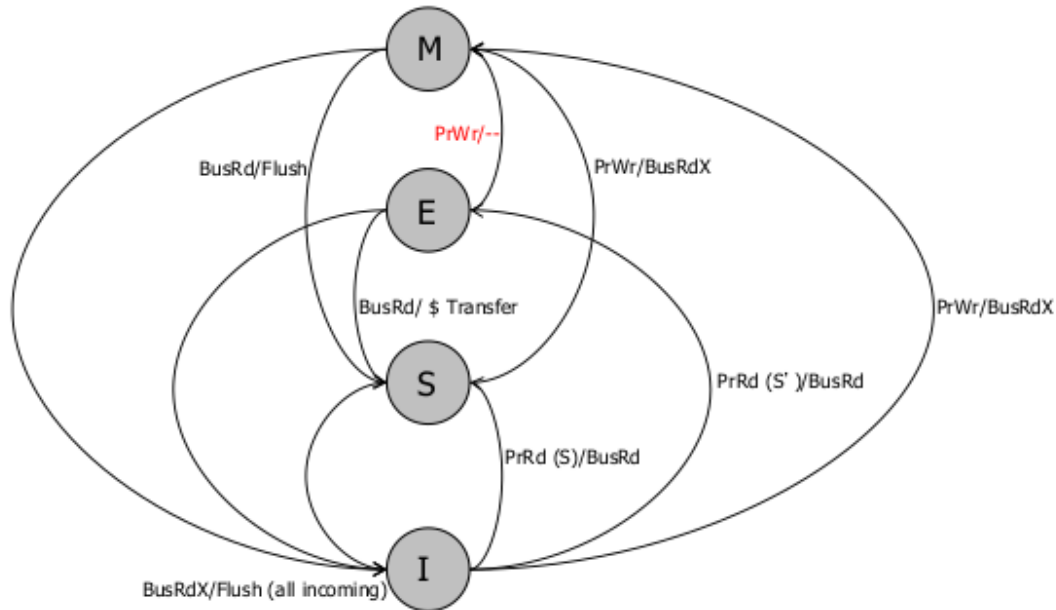
Problema: Un bloque no esta en ningun cache al principio. En un read, el bloque va directamente a *Shared* a pesar de que puede ser la unica copia para cachear

\Rightarrow si el cache que leyo el bloque lo quiere escribir en algun punto, necesita transmitir *Invalidate* a pesar de ser la unica copia cacheada \Rightarrow innecesario

MESI (Modified, Exclusive, Shared, Invalid)

Agrega otro estado indicando si es la unica copia cacheada y esta limpia (*Exclusive*).

Un bloque entra a este estado si durante un *BusRd* no lo tenia ningun otro cache.



La transicion silenciosa $E \rightarrow M$ es posible en write.

El estado compartido requiere que la data este limpia (todas las caches que tienen el bloque y la memoria tienen la copia actualizada)

Problema: Se necesita escribir el bloque en la memoria cuando *BusRd* ocurre cuando el bloque esta en *M*

\Rightarrow memoria se puede actualizar innecesariamente \Rightarrow otro procesador puede querer escribir en el bloque de nuevo

Como se puede mejorar?

- **Idea 1:** No transicionar $M \rightarrow S$ en un *BusRd*. Invalidar la copia y dar el bloque modificado al procesador que lo pidio directamente sin actualizar memoria
- **Idea 2:** Transicionar $M \rightarrow S$, pero designar un cache como **Owner**, que va a writebackear el bloque cuando lo eche.
 \Rightarrow *Shared* ahora es *Shared and potentially dirty* ~ MOESI

Funcionamiento detallado

- Todas las lecturas enviadas por el procesador se resuelven desde el cache, excepto si el estado de esa linea es **I**
- Las lineas invalidas se buscan en DRAM, o las provee otro CC \Rightarrow pasa a **S** o **E**
- El CC con **E** monitorea en el snoop bus. Si hay acceso a una linea almacenada en su cache la cambia a **S** y hace broadcast al resto (los que los reciben de ser necesario marcan la linea como **S**)

- Una línea **S** o **E** puede ser descartada y pasar a **I** en cualquier momento. **M** también, pero requiere actualizar previamente la DRAM
- Una línea **M** o **E** puede escribirse desde su CPU en cualquier momento. En el caso **E** pasa a **M**
- Si se necesita escribir una línea de estado **S**, el protocolo indica que todas las demás caches que tienen esa línea la hagan **I** a través de **Request For Ownership**
- Un cache que tiene una línea en estado **M** y detecta por snoop una lectura de esa línea debe “insertar” ese dato mantenido en su línea (**M** y **E** usan copyback):
 - Activa RFO para indicar que el dato está incoherente
 - Escribe en DRAM el valor actual de la línea. El lector copia ese valor correcto a su cache cuando aparece en el bus de datos y ambos pasan a **S**
- Una línea en **S** pasa **I** cuando se recibe un **RFO**
- **M** y **E** siempre son precisos: la línea solo está en ese cache (ownership)
- **S** es impreciso, nunca puede pasar a **E** porque no sabe cuántas otras caches tienen la línea
- **E** es el más apto para optimizar el mínimo de transacciones en el bus ya que al ser escrito cambia a **M** pero no informa al resto

Ready For Ownership

Es una combinación de escritura de una línea con el broadcast de una invalidación al resto de los controladores cuando el CC intenta escribir una línea que está en **S** o **I**. => el resto de los caches con esa línea la invalidan.

Compromisos de Invalidación Snoopy

- Un downgrade de **M** debería ir a **S** o **I**?
 - **S**: Si es probable que la data se reutilice (antes de que sea escrita por otro procesador)
 - **I**: Si es probable que la data no se reutilice (antes de que sea escrita por otro procesador)
- En un *BusRd*, la data debería venir de otro cache o memoria?
 - *Otro cache*: Puede ser más rápido si la memoria es lenta o muy competitiva
 - *Memoria*: Más simple (no necesitan esperar a ver si otro cache tiene la data primero), menos conflictos en los otros caches, requiere writeback en downgrade de **M**.
- Writeback en $M \rightarrow S$ necesario?
 - *Estado Owner* (MOESI): una cache es dueña de la data más nueva, writeback ocurre cuando todos los caches echan copias

Compromisos en protocolos de coherencia sofisticados

+ El protocolo puede ser optimizado con más estado y mecanismos de predicción para reducir invalidaciones innecesarias y transfers de bloques

Pero más estados y optimizaciones...

-- Son más difíciles de diseñar y verificar

-- Dan diminishing returns

Coherencia snoopy cache vs por directorio

<i>Snoopy cache</i>	<i>Directory</i>
+ Latencia de miss es corta: request => transaccion de bus a memoria + Serializacion global es facil: ya lo provee el bus + Simple: puede adaptar uniprosesadores facilmente -- Necesita que los mensajes emitidos los vean todos los caches en el mismo orden => un solo punto de serializacion (bus): no escalable => necesita un bus virtual (o conexion en orden)	-- Agrega indireccion al miss latency (camino critico) request → dir → mem -- Requiere almacenamiento extra para registrar sets que comparten. -- Los protocolos son mas complejos (para alta performance) + No requiere emitir a todos los caches + Es mucho mas escalable que el bus

Protocolos basados en directorios (cont)

Necesarios al escalar mas alla de la capacidad de un bus, pero

- Coherencia sigue requiriendo un solo punto de serializacion (para write)
- El lugar de serializacion puede ser diferente para cada bloque

Analicemos el protocolo para un solo bloque: un servidor (*directory node*) y muchos clientes (*caches privados*)

Directorio: estructuras de datos

- Recibe *Read* y *ReadEx* requests, y envia *Invl* requests: la invalidacion es explicita (a diferencia que los buses snoopy)
- La operacion clave a incluir es *set inclusion test*: los falsos positivos no estan mal, pero determinan la performance
- El de mas accuracy (y caro) => Full bit-vector
- Se puede usar representacion comprimida o lista enlazada

0x00	Shared: {P0, P1, P2}
0x04	---
0x08	Exclusive: P2
0x0C	---
...	---

Directorio: operaciones basicas

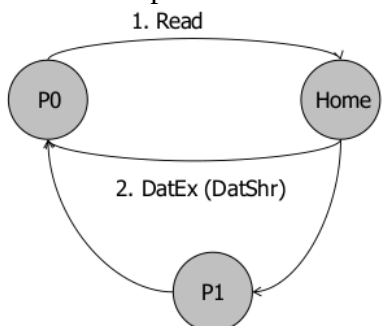
Sigue las semanticas de snoop pero con mensajes explicitos de request y reply. El diseño de protocolo es flexible.

El directorio:

- Recibe *Read*, *ReadEx*, *Upgrade* requests de los nodos
- Envia mensajes de *Inval/Downgrade* a los sharers si es necesario
- Reenvia request a la memoria de ser necesario
- Le responde al que hizo el request y actualiza el sharing state

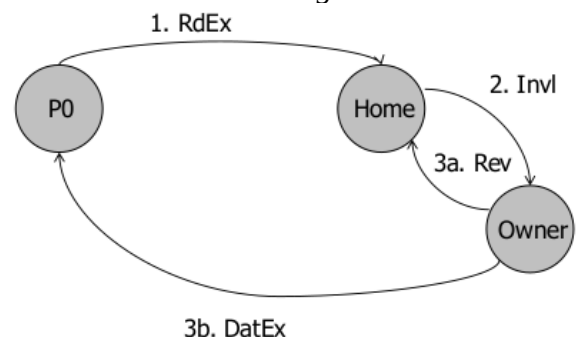
MESI Directory Transaction: read

P0 adquiere una direccion para leer



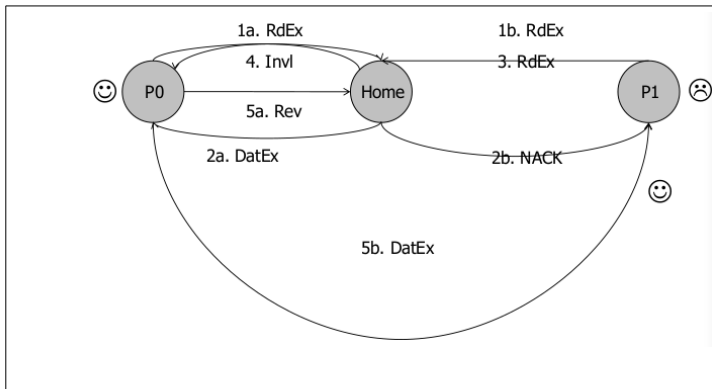
Resolucion de conflictos (de write)

RdEx con antiguo dueño



Problemas con la resolucion de conflictos

Necesita escapar las **race conditions** (situaciones en las



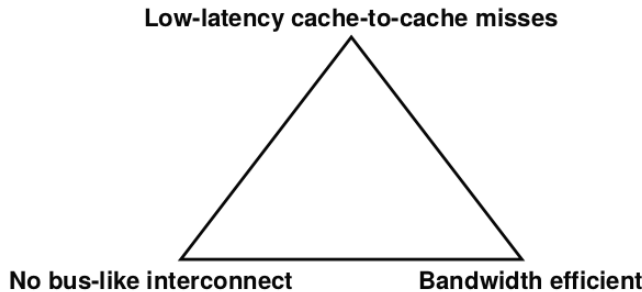
que el sistema intenta hacer dos operaciones al mismo tiempo, pero deben ser hechas en una determinada secuencia para ser correcto) a través de:

- NACK a los requests a cosas busy (el requestor intenta de nuevo)
- ponerlos en queue

Que requestor debería ser preferido en un conflicto?

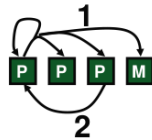
Advancing coherence

Motivacion: 3 atributos deseables



Tendencias de workload => snooping protocols

- *Comerciales:* muchos miss de cache a cache, clusters de multiprocesadores
- *Objetivos:* Misses de cache a cache directos (2 saltos, no 3), mas escalabilidad



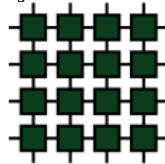
Low-latency cache-to-cache misses
(Yes: direct request/response)

No bus-like interconnect
(No: requires a "virtual bus")

Bandwidth efficient
(No: broadcast always)

Tendencias de tecnologia => Interconexiones desordenadas (directory protocols)

- *Links de alta velocidad de punto a punto:* No (multi-drop busses)
- *Mas integracion de diseño:* Mejora costo y latencia
- *Deseo: baja latencia de interconexion:* Evitar ordenar por virtual bus, permitido por protocolos de directorio



Low-latency cache-to-cache misses
(No: indirection through directory)

No bus-like interconnect
(Yes: no ordering required)

Bandwidth efficient
(Yes: avoids broadcast)

Coherencia de tokens

- *Objetivo de coherencia basada in invalidaciones*
 - Invariante: **muchos readers o single writer**
 - Lo imponen acciones **globalmente** coordinadas
- *Imponer este invariante directamente usando tokens*
 - **Numero fijo de tokens** por bloque
 - **Un token para read, todos los tokens para write**
- *Garantiza seguridad en todos los casos*
 - Invariante global impuesto solamente con reglas **locales**
 - Independiente de races, orden de request, etc.

Objetivo: Los 3 atributos

Low-latency cache-to-cache misses

Step#1

Step#2

No bus-like interconnect

Bandwidth efficient

Microarquitectura

Scaling

- El proceso de un circuito integrado se caracteriza por su tamaño: los transistores se estan achicando.
- Otro parametro importante en un transistor es su rendimiento: al ser mas pequeños hay que reducir su tension de alimentacion, lo que hace que al reducir el tamaño, estos ganen linealmente rendimiento.
- Los “alambres” son los caminos de señal que conectan los diferentes componentes (buses internos)
=> generan delays y consumo de energia

Delays

- Los buses se acortan al reducir dimensiones, pero sus efectos no se reducen.
- Cobran relevancia las capacidades e infuctancias mutuas entre cada bus con el vecino
=> la señal inyectada se mueve con determinada demora que depende de cada bus (posible destiempo)

Consumo/Energia

- La energia que disipa cada alambre es despreciable, pero hay muuuchos.
- La tension de alimentacion se redujo mucho en los ultimos años
- La capacidad de carga depende del # de dispositivos que se conectan a la salida de un transistor y la tecnologia de integracion empleada
- Para una tarea fija, reducir la frecuencia de clock disminuye la potencia disipada pero no tiene efecto con la energia consumida

El consumo es un **problema** porque el incremento del # de transistores por mm² de sup tiene mas efecto sobre la energia que los ahorros de enegia de cada transistor por el cambio de tecnologia

=> cada vez es mas critico el manejo del consumo

=> cada vez existen mas limitaciones en la distribucion de alimentacion y ahorro de potencia y energia

Para **reducir el consumo** la mayoría de los procesadores actuales contiene bloques de hardware para control de consumo (solo alimenta los bloques que se necesitan en el momento)

A pesar de que un transistor este al corte, circula una pequeña corriente de todos modos que se llama **corrientes de fuga (leakage)**, que aumenta con la cantidad de transistores.

Arquitectura vs. Microarquitectura

<i>Arquitectura</i>	<i>Microarquitectura</i>
Es el conjunto de recursos accesibles para el programador, que por lo general se mantienen a lo largo de los diferentes modelos de los procesadores de esa arquitectura. <ul style="list-style-type: none">• Registros• Set de instrucciones• Estructuras de memoria	Es la implementacion en el silicio de la arquitectura. Lo que esta detras del set de registros y el modelo de programacion. Puede ser muy simple o sumamente robusta y poderosa. Cambia de un modelo a otro dentro de una misma familia.

Definicion de arquitectura de un computador

- En este paso se define la arquitectura para maximizar el rendimiento teniendo en cuenta las limitaciones impuestas por los usuarios, el costo economico y consumo de energia moderado.
- Incluye: set de instrucciones, manejo de memoria y su direccionamiento, los demas bloques funcionales de CPU, diseño logico e implementacion (diseño de circuito integrado, encapsulado, montaje, alimentacion y refrigeracion)

ISA

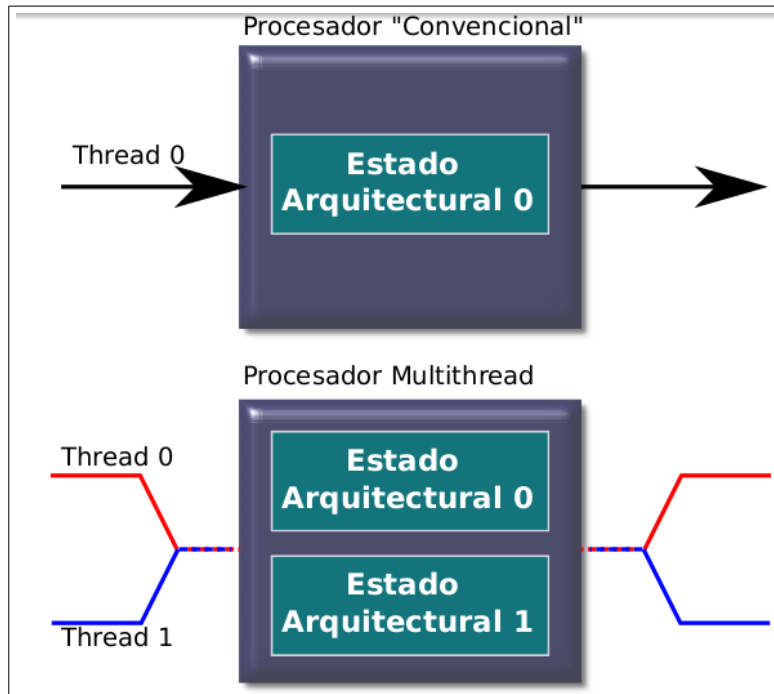
Es el set de instrucciones visibles por el programador. El limite entre software y hardware.

- **Clases de ISA:** Registros de proposito general vs. Registros dedicados. Registro-memoria vs. Load store
- **Direccionamiento de Memoria:** Alineacion obligatoria de datos vs. Administracion de a bytes
- **Modos de Direccionamiento:** Como se especifican los operandos
- **Tipos y tamaños de operandos:** Enteros, floating point, punto fijo. Distintos tamaños y precisiones
- **Operaciones:** Pocas simples (RISC) o muchas complejas (CISC)
- **Instrucciones de control de flujo:** Saltos condicionales, calls
- **Longitud del codigo:** Instrucciones de tamaño fijo vs. variable

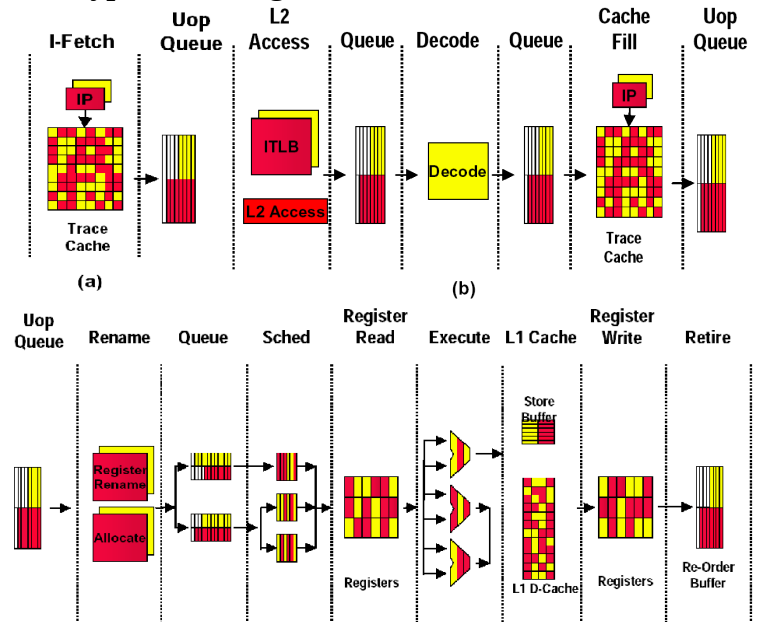
Microarquitectura = Organizacion + Hardware

<i>Organizacion</i>	<i>Hardware</i>
Son los detalles de implementacion de la ISA: <ul style="list-style-type: none">• Organizacion e interconexion de la memoria• Diseño de los diferentes bloques de la CPU y para implementar el set de instrucciones• Implementacion de paralelismo a nivel de instrucciones y/o datos => Hay procesadores que poseen la misma ISA pero organizacion muy diferente	Son los detalles de diseño logico y tecnologia de fabricacion => Hay procesadores con la misma ISA y la misma organizacion pero absolutamente distintos a nivel de hardware y diseño logico detallado

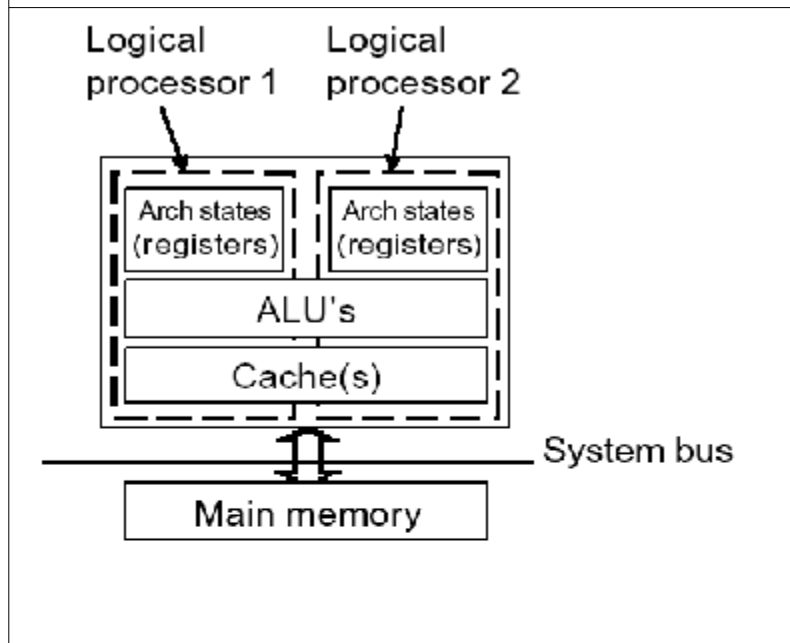
Procesadores Multithread



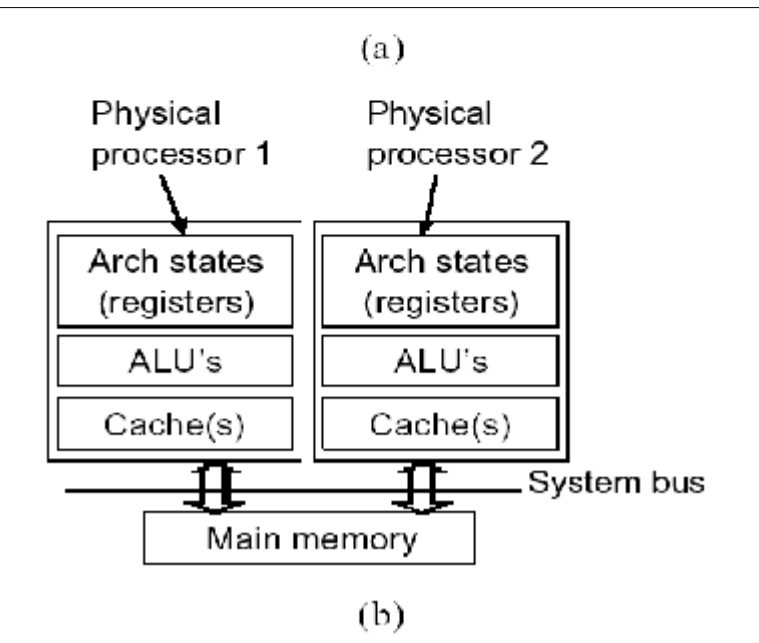
Intel Hyperthreading



Hyperthreading



Multicore



1 supercore o muchos sencillos?

- *Varios cores*: = performance, < consumo
- *Problema*: Mas cores => mas overhead de protocolo de coherencia
- *Solucion*: Red de multicores con un router para comunicarse

