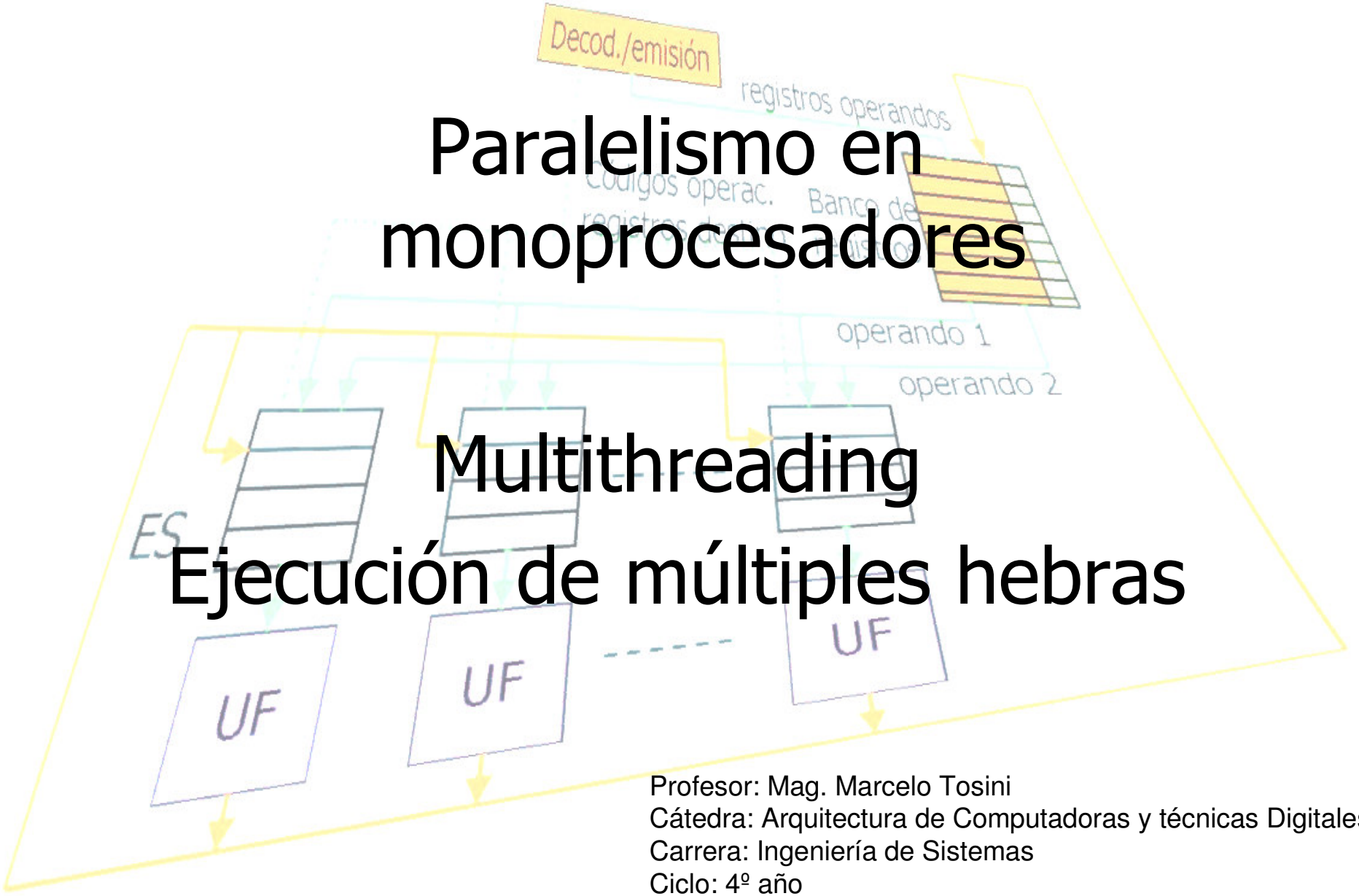


Paralelismo en monoprocesadores

Multithreading
Ejecución de múltiples hebras

Profesor: Mag. Marcelo Tosini
Cátedra: Arquitectura de Computadoras y técnicas Digitales
Carrera: Ingeniería de Sistemas
Ciclo: 4º año

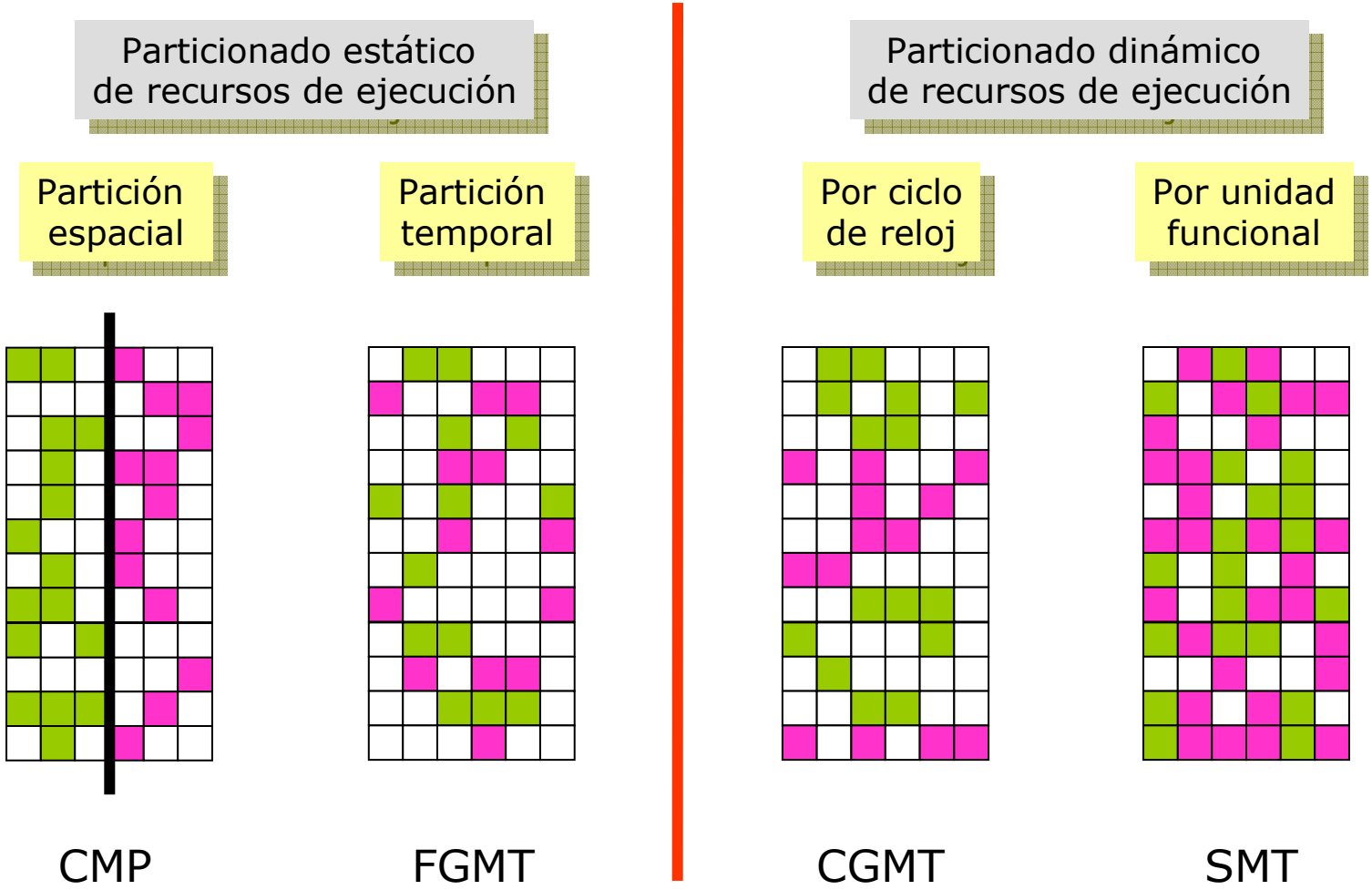
Introducción

- Puesto que los programas usualmente ejecutan mas de un thread, un paso mas de desarrollo de procesadores es la incorporación de capacidad para ejecutar varias hebras
- Desarrollo motivado por aumentar la utilización de los recursos de cálculo sobre todo cuando el thread actual por algún evento de E/S de mucha latencia
- Se pueden mantener los recursos del procesador ocupados aún cuando un thread se frene ante un fallo de caché o una falla de predicción de salto
- Aproximaciones:
    - CMP (Chip Multi Processor)
    - FGMT (Fine Grained Multi Threading)
    - CGMT (Coarse Grained Multi Threading)
    - SMT (Simultaneous Multi Threading)

Aproximaciones multithreading

Aproximación MT	Recursos compartidos entre threads	mecanismo de cambio de contexto
Ninguna	Todos	Cambio de contexto explícito del SO
FGMT	Todos excepto banco de registros y lógica de control	Cambio de contexto en cada ciclo de reloj
CGMT	Todos excepto buffers de fecthing, banco de registros y lógica de control	Cambio de contexto ante frenado del pipeline
SMT	Todos excepto buffers de fecthing, pila de hardware, banco de registros, lógica de control, ROB y restore queue	No hay cambio de contexto. Todos los contextos están activos simultáneamente
CMP	Caché secundaria y sistema de interconexión con el exterior	No hay cambio de contexto. Todos los contextos están activos simultáneamente

Aproximaciones multithreading



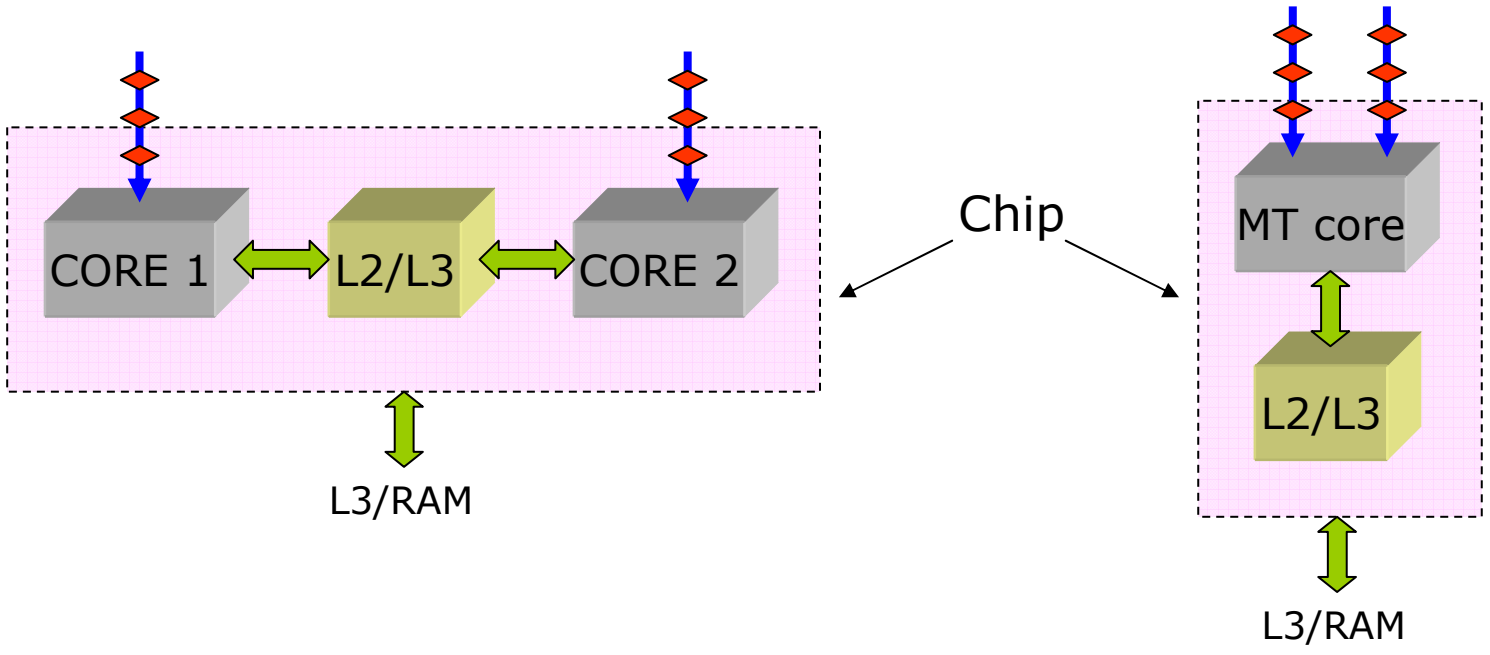
Aproximaciones multithreading

CMP: Chip Multiprocessing
(SMP: Symetric multiprocessing)

2 o mas cores en el mismo chip

SMT: Simultaneous multithreading
(HT: Hiper Threading de Intel)

un único core en el chip

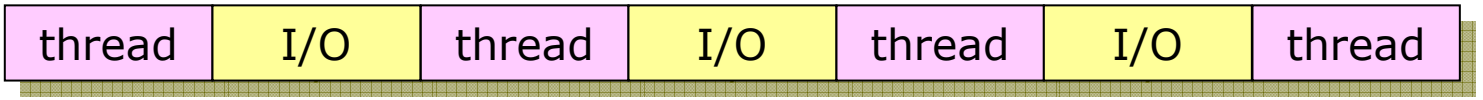


Paralelismo a nivel de hebra (Thread Level Paralelism - TLP)

- El paralelismo a nivel de instrucciones (ILP) se basa en operaciones "independientes" que pueden resolverse cuando se ejecuta un programa
- Por otro lado, un procesador superescalar ejecuta "al mismo tiempo" varios programas (procesos, threads, etc) pero multiplexados en el tiempo
- Las instrucciones de diferentes threads son paralelizables
- **Objetivo:** Explotar este paralelismo a nivel de hebra (thread level) mediante ejecución concurrente, para mejorar el rendimiento del procesador
- La mejora anterior NO mejora los tiempos de ejecución de cada thread
- La idea básica: Cuando se ejecuta un solo thread quedan recursos sin ejecutar en el procesador. Entonces se pueden utilizar ejecutando mas threads

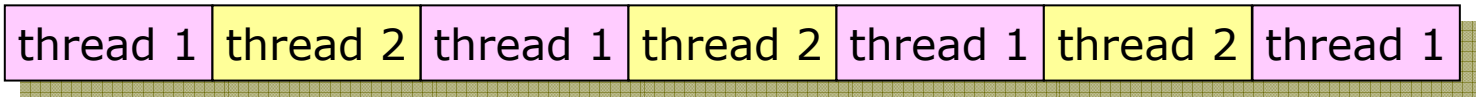
Ejecución de threads

Un solo thread



- Habiendo un solo thread el procesador queda inactivo durante los eventos de I/O
- La perdida de rendimiento puede ser de millones de ciclos de reloj

Múltiples threads



- Cuando un thread entra en espera por I/O se debe liberar el procesador para permitir a otro thread el uso de los recursos
- El proceso de cambio entre threads se realiza mediante un cambio de contexto

Cambio de contexto tradicional

Contexto: Estado del procesador asociado a un proceso particular

- Contador de programa
- Registros
- Datos de memoria
- Registros de estado y control, punteros de paginas y segmentos
- Registros de sombra
- ¿Contenidos de caché, entradas de BTB y TLB?

Cambio de contexto tradicional

1. Una interrupción (precisa) del timer detiene un programa en ejecución
2. El SO almacena el contexto del thread parado
3. El SO recupera el contecto de un thread detenido con anterioridad (a excepción del PC)
4. El SO utiliza un "retorno de excepción - IRET" para saltar al "PC" reiniciando el thread.

El thread nunca se entera que fue detenido, dezplazado, recuperado y reiniciado en el procesador

Cambio de contexto rápido

Un procesador queda inactivo (idle) cuando un thread entra en espera por un fallo de caché

- Normalmente un fallo de caché necesita del orden de 16 a 32 ciclos de penalización de fallo y
- Un cambio completo de contexto puede llevar del orden de cientos de ciclos, entonces…

No siempre conviene un cambio de contexto por el SO

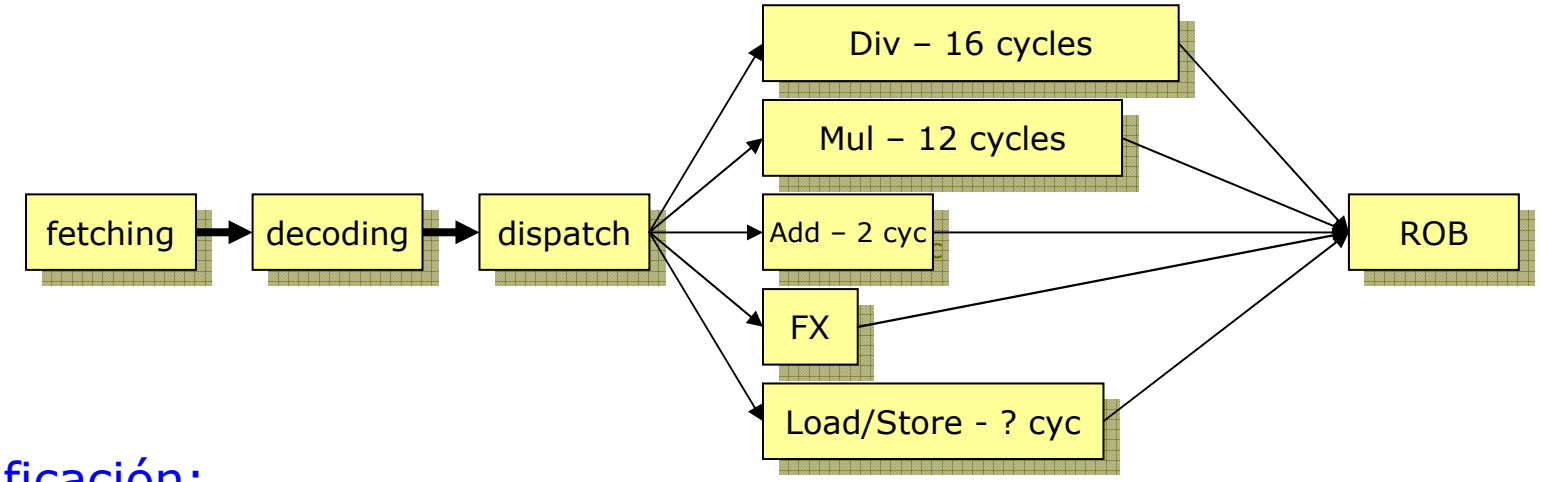
Solución: cambio de contexto por hardware

- Replicar registros de contexto (PC, GPRs, flags, pointers, etc) eliminando el copiado a y desde memoria RAM
- Varios contextos pueden compartir recursos comunes (caché, BTB, TLB) incluyendo un campo "process ID" en sus entradas… Se elimina la necesidad de cambio de contexto (Se eliminan fallos forzosos por cambio de contexto)
- El cambio de contexto por HW consume pocos ciclos de reloj:
  - PID se carga con el siguiente identificador de proceso
  - Seleccionar el conjunto de registros de contexto a activar

Optimización del cambio de contexto rápido

¿Es posible el cambio de contexto cuando un thread se detiene por dependencias RAW?

Cambio de contexto a nivel de unidades funcionales



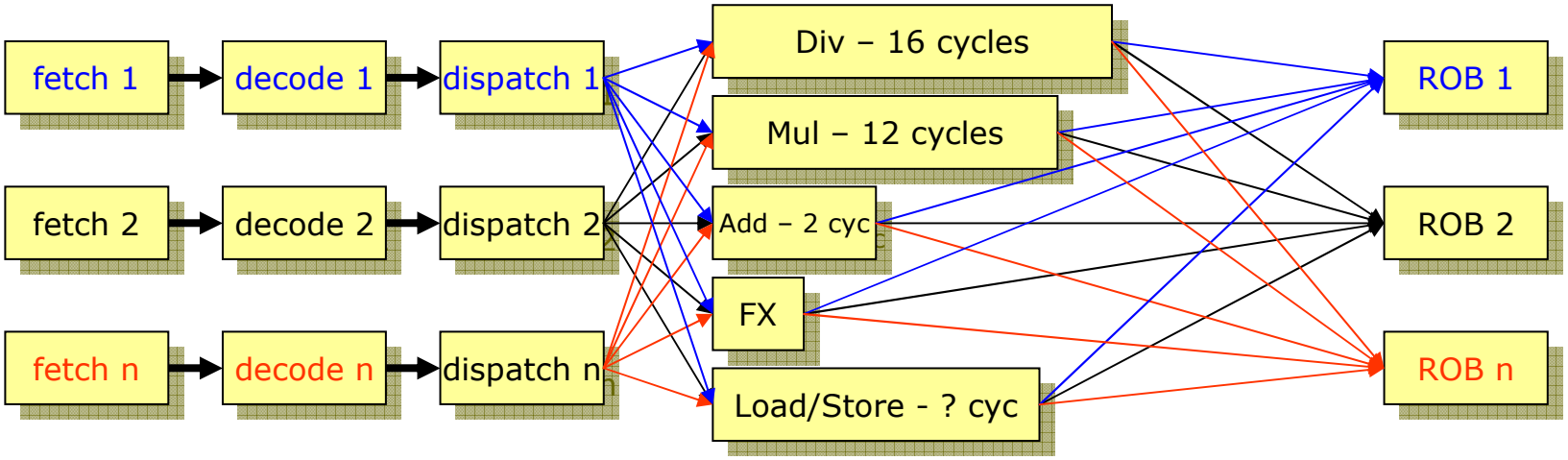
Justificación:

- El grado de un superescalar es usualmente mayor que el ILP logrado
- En la actualidad, procesadores de grado 8 alcanzan ILP de 2 ó 3

¿Por qué no usar las unidades funcionales ociosas en otro thread?

Implementación para varios threads

- Se puede suponer varios pipelines que repliquen la parte "in order" de un superescalar.
- Cada pipeline tiene su propio PC y su banco de registros
- Mayor lógica de administración de los recursos compartidos (UF)



- El rendimiento de cada thread es peor que en un superescalar debido a degradaciones asociadas al aumento de complejidad
- Mayor utilización de los elementos compartidos

Fine Grained Multi-Threading (FGMT)

- Un procesador de granularidad fina provee dos o más contextos en un mismo chip
- Los contextos son intercambiados en un tiempo fijo y breve, usualmente cada ciclo de reloj
- Precursores: Cray CDC 6600 (1960) y Denelcor HEP (1970)
- Justificación: Intercalar instrucciones de diferentes threads a fin de
  - Enmascarar latencias de memoria
  - Evitar detectar y resolver dependencias RAW entre instrucciones
- La Tera MTA (Tera Computer Company, 1998) maximiza el uso del pipeline de acceso a memoria intercalando peticiones de diversos threads en esa etapa

  Tera MTA posee 128 registros de contexto que permiten la ejecución de 128 threads, con lo que enmascara totalmente las latencias de memoria a tal punto que no se necesita usar caché
- Principal desventaja de esta máquina: El compilador debe esforzarse para encontrar varios (hasta 128) threads independientes, sino se degrada el rendimiento

Fine Grained Multi-Threading (FGMT)

Desventajas:

- El rendimiento de un thread en particular decae fuertemente respecto de su ejecución en un procesador superescalar clásico
- Las ganancias en uso del procesador generadas por las latencias de I/O del thread no son suficientes para compensar las demoras impuestas por la ejecución de los otros contextos (otros threads)

Ejemplo:

En un entorno con varios threads intentando acceder a una zona de memoria compartida, las variables compartidas permanecen mucho tiempo bloqueadas por threads inactivos

Coarse Grained Multi-Threading (CGMT)

Características generales:

Aproximación intermedia de multi-threading que tiene los beneficios de FGMT pero sin la severa restricción de tiempo de contexto de esta

Primer procesador comercial: Power PC (NorthStar y Pulsar) de IBM en 1996 y 1998

El **cambio de contexto** entre threads se realiza sólo cuando el thread actual se frena por algún **evento de gran latencia** (pe: fallo de caché)

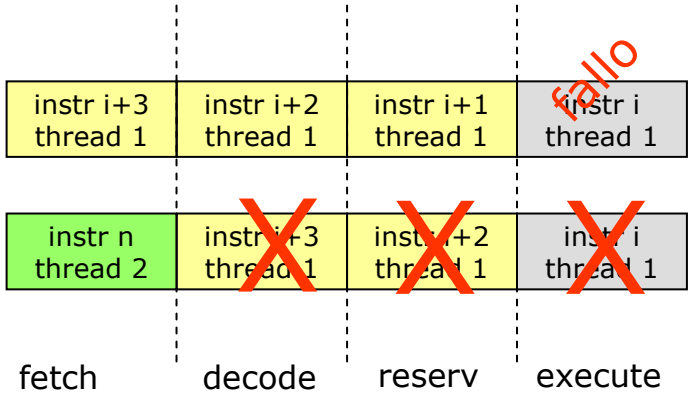
En lugar de frenar el pipeline, sus etapas se cargan con instrucciones ejecutables de otro thread

Coarse Grained Multi-Threading (CGMT)

Problemas de la aproximación:

- **Penalización de cambio de contexto:** En FGMT el cambio de contexto se hace en cada ciclo, por lo tanto la etapa de fetching carga en cada nuevo ciclo la instrucción siguiente del thread siguiente. N**O hay frenado por llenado del pipe**

  En CGMT el cambio de contexto es asincrónico (depende de un evento externo como un fallo de caché) por lo que, cuando ocurre (en la etapa de ejecución) las etapas anteriores del pipeline deberian llenarse con instrucciones del nuevo thread. **HAY frenado por llenado del pipe**



- **Potencial inhanición:** En el caso de que un thread no produzca fallos de caché, nunca libera el procesador y los demás threads quedan inactivos

Penalización de cambio de contexto

Se debe realizar un cambio de contexto sin perdidas de ciclos y a un costo razonable

Solución

Replicar los registros de pipeline para cada thread y salvarlos cuando se desaloja un thread del procesador

De este modo, un nuevo juego de registros de pipeline (con un estado congelado del thread nuevo) puede estar disponible en sólo un ciclo de reloj

Desventaja:

- Aumento de área necesaria para mantener el "estado" de los threads
- Aumento de la complejidad de control para organizar el cambio de contexto

Reflexión: Esta solución sólo evita la perdida de pocos ciclos (3 a 5) a costa de un incremento de la compeljidad, entonces, no es tan efectiva

Eliminación de inhanición

Se debe asegurar equidad en la cesión de recursos del procesador a los diferentes threads

No es sencillo (para el programador o el compilador) determinar cuando ocurrirán fallos de caché en un thread

Entonces, se deben proveer mecanismos adicionales para prevenir la inhanición de threads

Solución sencilla:

Un thread con baja tasa de fallos (mucho tiempo entre fallos) será desalojado después de un periodo de tiempo

Desventaja:

PROBLEMAS!!!

Eliminación de inhanición

Metas de rendimiento:

- Proveer esquemas para minimizar las bajas tasas de ejecución
  - Un thread en un "Busy wait state" (pe: ciclando)
  - Un thread entrando en un "operating system idle loop"

- Maximizar el tiempo de ejecución en casos críticos
  - Un thread dentro de un área de memoria compartida con otros thread en el sistema esperando por el recurso compartido

  En este escenario la ejecución del thread no debe ser suspendida aún si tiene un fallo de caché porque:

  - *Los cambios de contexto degradan la performance del thread con más prioridad*

  ó

  - *Los threads de menos prioridad pueden causar conflictos adicionales de cache o memoria*

Manejo de prioridades de threads

- Esquema de prioridades con al menos 3 niveles de prioridad
  - Alta
  - Media
  - Baja

- **Las prioridades reflejan dinámicamente la importancia relativa de ejecución en la fase actual de un thread**

- Manejo de las prioridades mediante la intervención del programador

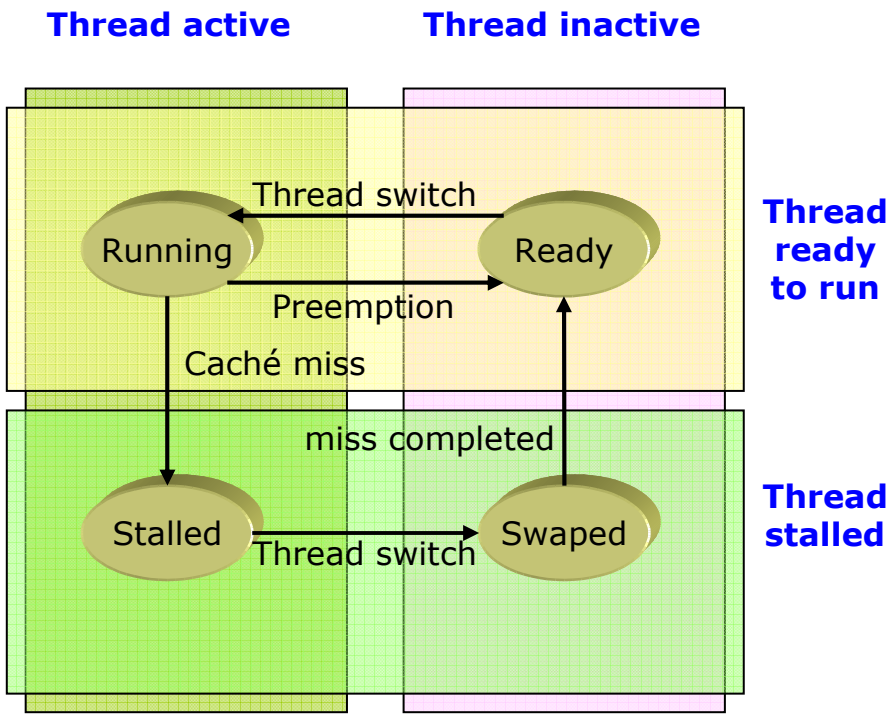
- Se agregan al código instrucciones que cambian el nivel de prioridad de un thread

  - **Bajo** si el thread entrará en un *Idle loop* o *busy wait state*
  - **Alto** si el thread entrará a una zona crítica
  - **Medio** si el thread está en una zona de ejecución normal

Manejo de prioridades de threads

Ejemplo de FSM de manejo de prioridades en CGMT

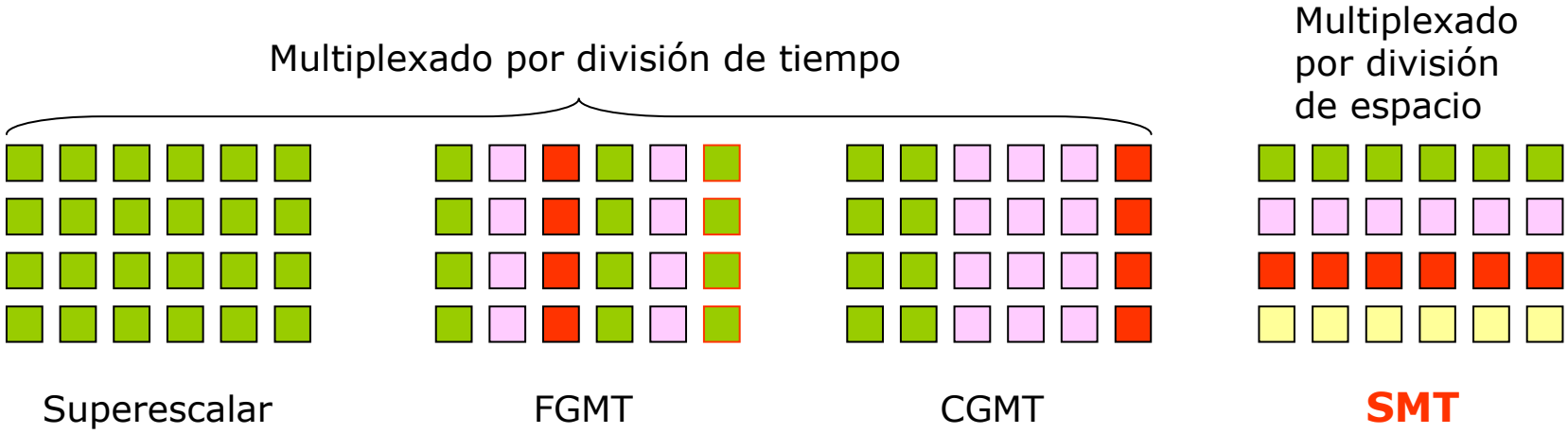
- Ocurre un fallo de caché en el thread primario y existe otro thread listo
- El thread primario entra en idle loop y hay otro thread listo en espera
- El thread primario entró en un loop de sincronización dentro del SO y hay otro thread listo y en espera
- Un thread desalojado por tiempo fue cambiado al estado de listo y este thread ahora tiene mas prioridad que el thread activo
- Un thread listo no ha ejecutado ninguna instrucción en los últimos n ciclos de reloj (prevención de inhanición)



Simultaneous Multi-threading (SMT)

Características generales:

- Permite ejecución de granularidad fina e intercambio dinámico de instrucciones entre múltiples threads
- La propuesta original argumenta que el mecanismo de intercambio de contexto (FGMT y CGMT) comparte el HW del procesador de manera ineficiente.
- El paradigma de cambio de contexto restringe el pipeline entero o, en el otro extremo, cada etapa del pipeline a contener instrucciones de un mismo thread



Simultaneous Multi-threading (SMT)

Características generales:

- Varias características de los procesadores "Out of Order" (superescalares) permiten la implementación eficiente de SMT
  - Las instrucciones atraviesan las etapas "Out of Order" del procesador sin importar el orden secuencial de programa o el orden de fetching

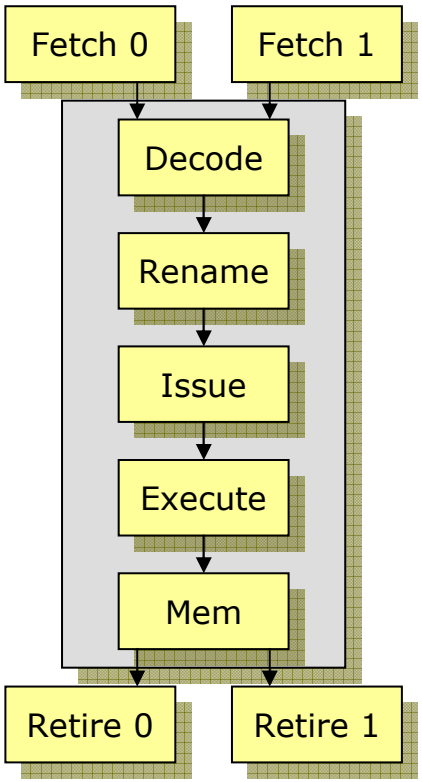
    Habilita a instrucciones de distintos threads a mezclarse
  - Dependencias de datos restringen el paralelismo a (a lo sumo) dos instrucciones

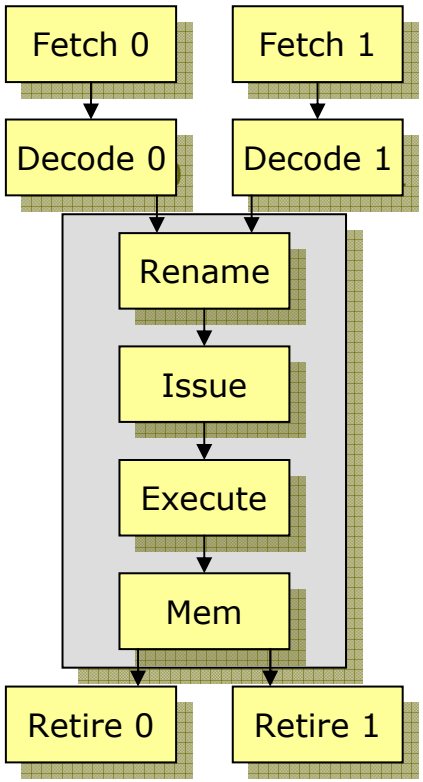
    Un thread alternativo e independiente puede ser usado para ocupar recursos no utilizados por el primero
  - El banco de registros es usualmente renombrado para poder compartir un grupo común de registros físicos. Este renombrado elimina la necesidad de rastrear threads cuando se resuelven dependencias de datos dinámicamente

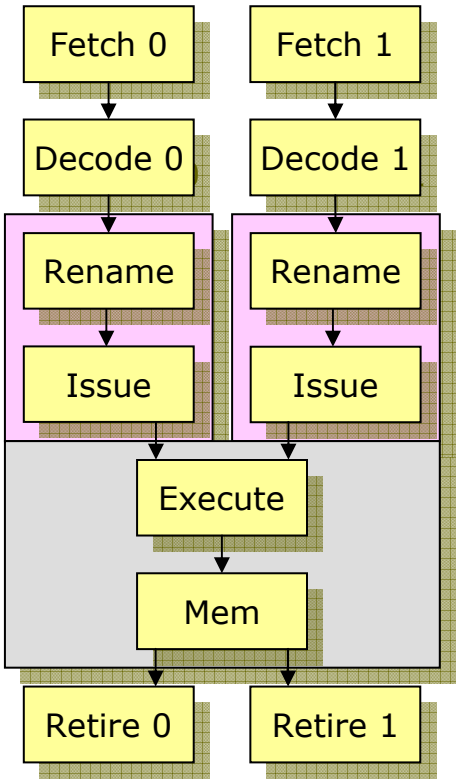
    Si el renombre sirve para "aislar" registros de igual nombre en un mismo thread igualmente aislará registros de igual nombre en distintos threads
  - Uso estensivo de buffers (ROB, Issue queue, Load/Store queue, store queue, etc) necesarios para "suavisar" la ejecución irregular de instrucciones en paralelo

    Todos esos buffers pueden ser usados mas eficientemente por instrucciones de threads múltiples

Arquitecturas alternativas de SMT







Etapas compartidas en SMT

Fetch unit

Básicamente compuesta de dos partes:

- Lectura de instrucciones de la I-caché

  Acceso a la I-caché a través del puerto de instrucciones
  Dado que el puerto apunta a un bloque con instrucciones consecutivas, dos threads difícilmente pueden ser accedidos de manera eficiente

  NO se comparte acceso a instrucciones

- Unidad de predicción de saltos

  Si se comparte sus memorias internas tendrán menos tamaño para memorizar historia de cada thread

  Por otro lado, no se puede mezclar entradas en las BHT sin perder eficiencia en el funcionamiento del algoritmo

  NO se comparte predicción de saltos

Etapas compartidas en SMT

Decode unit

Función: Para instrucciones RISC, identificar operandos fuente y destino

Para instrucciones CISC, determinar la semántica de instrucciones mas complejas y (usualmente) convertirlas en una secuencia de instrucciones RISC

- Decodificar n instrucciones (identificación de operandos) tiene complejidad $O(n^2)$

- (Por definición) no hay dependencias entre instrucciones de distintos threads, entonces se puede decodificar juntas instrucciones de ellos

Ejemplo:

Dos decodificadores de 4 caminos c/u pueden decodificar hasta 4 instrucciones de 2 threads simultáneamente

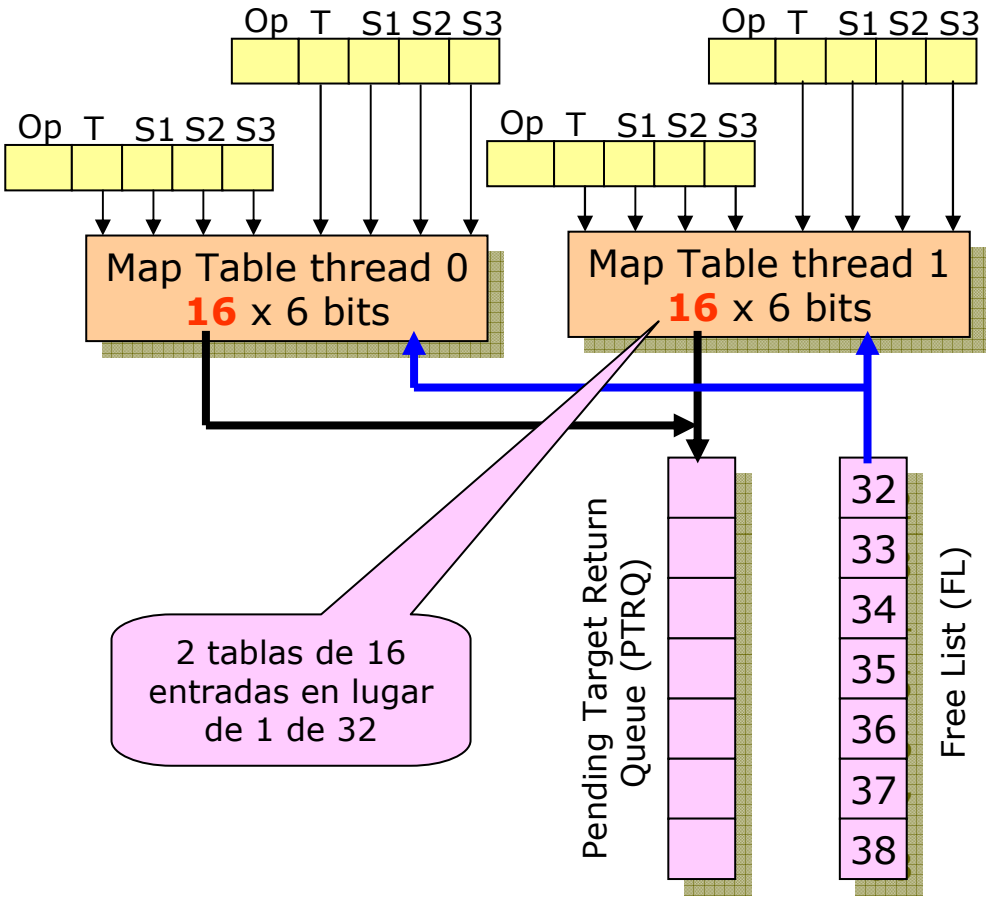
La solución anterior tiene menor complejidad que un solo decodificador de 8 caminos

Etapas compartidas en SMT

Rename unit

Función: Asignación de registros físicos y mapeo de registros de arquitectura a registros físicos

- Registros físicos elegidos de un banco común, entonces, es sencillo compartir ese banco común entre threads
- Desde el punto de vista de los registros de arquitectura es necesario identificarlos por thread
- Potencialmente se limita el rendimiento de threads simples con alto ILP



Etapas compartidas en SMT

Issue unit

Función: Distribución dinámica de instrucciones a partir de un proceso de dos fases: "Wake-up and Select"
Despertar todas las instrucciones que tienen todos sus operandos y Seleccionar una de ellas para emitir a una unidad funcional

- En SMT el proceso de Select puede abarcar instrucciones de mas de un thread
- El proceso de Wake-up es mas restrictivo pues una instrucción solo debe despertarse en función de una interdependencia de datos con una instrucción previa de su mismo thread
- Las ventanas de emisión deben separarse entre los distintos threads o debe identificarse apropiadamente cada thread

Etapas compartidas en SMT

Execute unit

Función: Realiza las operaciones contenidas en las instrucciones ejecutando cada instrucción en una unidad funcional diferente

- Compartir las unidades funcionales es sencillo para implementar SMT
- Se pueden aplicar algunas optimizaciones a la arquitectura básica:
  - Simplificar el *Common Data Bus* (red de interconexión que retroalimenta resultados desde las unidades funcionales hacia las estaciones de reserva para despertar instrucciones dependientes) dado que instrucciones de diferentes threads nunca necesitan retroalimentar resultados a otras estaciones distintas a la del propio thread
  - Completar los ciclos ociosos de las unidades funcionales con instrucciones independientes de nuevos threads

Etapas compartidas en SMT

Memory unit

Función: Realiza accesos a caché para satisfacer las demandas de las instrucciones de lectura y resuelve dependencias entre loads y stores

- Compartir los buffers de memoria facilita el acceso de un thread a datos escritos por otro thread sin necesidad de salir del ámbito del procesador
- El manejo del hardware que detecta y resuelve dependencias de memoria no es trivial
  Consiste de un buffer que mantiene *loads* y *stores* en el orden de programa y detecta si *loads* tardios coinciden con *stores* anteriores
- Con varios threads hay que tener en cuenta el modelo de consistencia de memoria ya que algunos modelos no permiten adelantar valores de un *store* de un thread a *loads* de otros threads
- HW avanzado debe permitir adelantamientos o frenar instrucciones load dependiendo del modelo adoptado por los programas

Etapas compartidas en SMT

Retire unit

Función: Efectiviza la escritura de resultados en el banco de registros en el orden del programa para mantener la consistencia secuencial

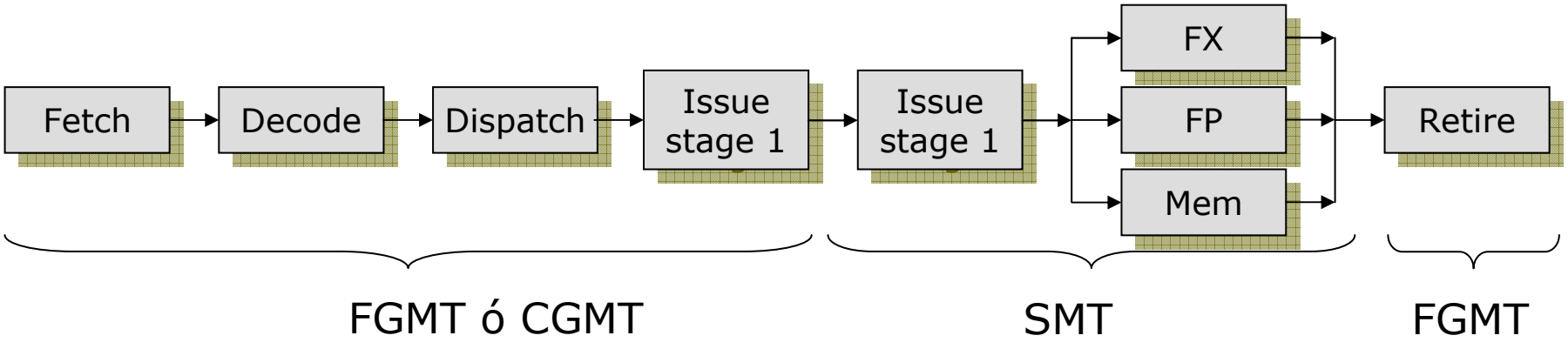
- El proceso involucra el chequeo previo de excepciones u otras anomalías antes de la actualización de registros físicos o de arquitectura según corresponda
- La escritura en orden de programa asegura que no se violarán dependencias WAW dentro de un thread
- En el caso de múltiples threads no afecta compartir la etapa ya que no puede haber dependencias WAW entre diferentes threads

Pentium IV – Multithreading híbrido

Incorpora una arquitectura multithreading híbrida que permite que dos procesadores lógicos compartan algunos de los recursos de ejecución del procesador físico

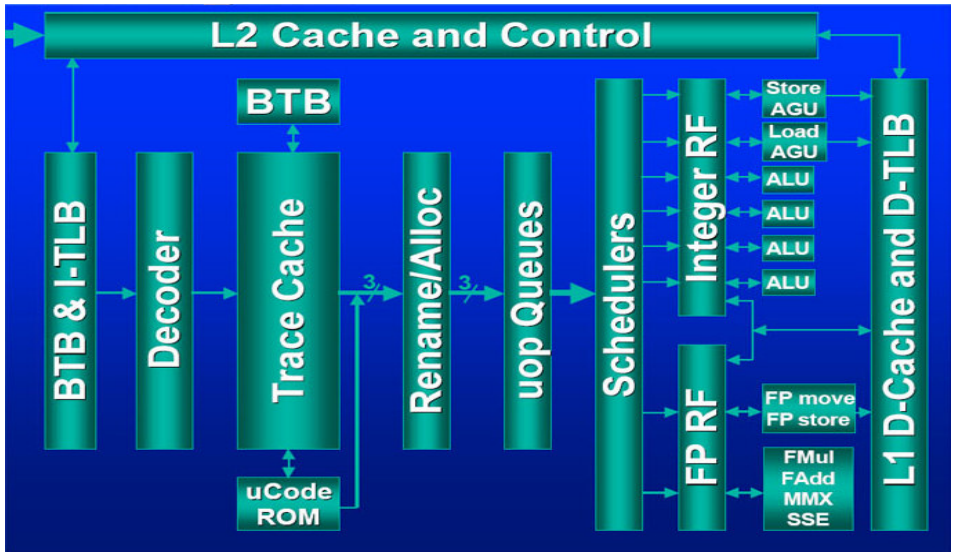
Multithreading en Intel = Hiperthreading (XT)

El pentium se paraleliza a nivel de threads de distintas maneras



Los dos threads lógicos realizan el Fetch, Decode y Retire en ciclos anternativos (FGMT), a menos que alguno de ellos se frene; con lo cual el otro thread permanece activo en todos los ciclos hasta que el primero se despierta (CGMT)

Pentium IV – Multithreading híbrido



- La primera etapa de emisión se comporta como las etapas de fetch y Decode, o sea, como CGMT.
- La segunda etapa de emisión se comporta como SMT, al igual que las etapas de Execute y Memory
- En las etapas en SMT los 2 threads pueden mezclar sus instrucciones arbitrariamente
- Para implementar el SMT todos los buffers de la sección "Out Of Order" (ROB, Load Queue, Store Queue) se dividen en dos mitades, una para cada thread
- El procesador dispone de un modo "single thread" que deshabilita el SMT con lo que los buffers se dedican completamente a un solo thread
- Dado que los buffers se dividen en sus 2 mitades en SMT, el rendimiento por thread puede decaer