Organización del Computador II - Apuntes para final

Gianfranco Zamboni

14 de agosto de 2019

Índice

1. Introducción

La **arquitectura de una computadora** es el conjunto de recursos accesibles para el programador que, por lo general, se mantienen a lo largo de los diferentes modelos de procesadores de esa arquitectura (puede evolucionar pero la tendencia es mantener compatibilidad hacia los modelos anteriores).

Diseñar una arquitectura implica diseñar las interfaces hardware/software para crear un sistema computacional que posea los requerimientos funcionales, de perfomance, consumo (de energía) y de costo (económico) adecuados para realizar determinadas tareas.

Estas tareas son problemas que pasaron por varias transformaciones (desde su descripción en un lenguaje natural hasta convertirse en un programa) y deben ser ejecutadas por una computadora. La tarea del arquitecto consiste en diseñar el **Instruction Set Architecture** (ISA), un conjunto de instrucciones que usarán los programas compilados para decir al microprocesador que hacer. El ISA es implementado por un conjunto de estructuras de hardware conocidas como **microarquitectura**.

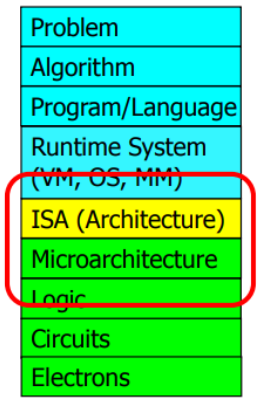

Figura 1: La computadora definida en niveles de abstracción

El ISA y la microarquitectura sientan las bases para que el diseño del procesador consiga el balance adecuado de los factores mencionados y pueda llevar a cabo ciertas tareas de la manera más optima posible. Habrá casos en los que daremos prioridad a un subconjunto de ellos en detrimento de otros (por ejemplo, podríamos elegir mejorar performance y aumentar el costo, o quitar performance para mejorar el consumo energético).

1.1. Componentes del ISA

El ISA es la especificación completa de la interfaz entre los programas y el hardware que debe llevar a cabo las operaciones. Entre otras cosas, especifica:

1.1.1. Regitros

- **Registros:** Celdas de memoria dentro del cpu que son usadas para almacenar los datos necesarios para ejecutar una instrucción. Éstos son visibles al programa y se clasifican según su uso: Acumuladores, De dirección ó De Proposito General.
- **Instrucciones**: Tareas que pueden ser llevadas a cabo por la computadora. Cada una de ellas está compuesta por su **opcode** (que se espera que la computadora haga) y sus

operandos (a que datos debe hacerlo). En una ISA podremos encontrar tres tipos de instrucciones:

- De **Operacion**: Procesan datos.
- De **transporte**: Transportan información entre la memoria, los registros y los dispositivos de entrada salida.
- De **control (Branching)**: Modifican la secuencia de instrucciones a ser ejecutada, es decir, permiten ejecutar instrucciones que no están almacenadas secuencialmente.

Dependiendo que valores puedan modificar las instrucciones de operación, podremos clasificar las arquitecturas en: **Arquitecturas Load/Store** (solo pueden operar en registros) o **Arquitecturas memory/memory** (se pueden modificar los valores directamente en memoria)

- **Tipos de datos**: Representación que deben tener ciertos valores para poder ser interpretados por la microarquitectura.
- **Espacio de memoria**: La cantidad de bloques unívocamente distinguibles en memoria y el tamaño de cada uno de estos bloques
- **Direccionamiento**: Los mecanismos usados por la computadora para saber donde están almacenados los datos. Puede ser:
  - **Implicito:** El operando se puede deducir del código de operación de la instrucción.
  - **Inmediato:** El operando está incluido en la instrucción.
  - **Directo o absoluto:** El operando es la dirección de memoria donde se encuentra el valor a ser utilizado.
  - **Registro:** Todos los operandos involucrados son Registros del procesador.
  - **Indirecto:** El operando es una dirección de memoria donde se encuentra la dirección de memoria en la que está almacenado el valor deseado.
  - **De desplazamiento:** La instrucción toma como operandos una dirección de memoria que se toma como **base** y un **offset**. La base es una dirección de memoria y el offset, un número que indica cuanto hay que desplazar la base para encontrar el valor deseado, es decir $dir = base + offset$
  - **Indexado:** Lo mismo que el anterior, pero con el *offset* guardado en un registro.
  - **De Memoria Indirecta:** El operando es un registro en el que se encuentra guardada la dirección de memoria indirecta.
- **I/O Interface**: Como comunicarse con los dispositivos de entrada/salida. Puede ser por medio de instrucciones especiales o mapeo de ciertas regiones memoria para uso de esos dispositivos.
- **Modos de privilegio:** Quien puede y quien no puede ejecutar ciertas instrucciones

- **Manejo de excepciones e interrupciones:** Qué debe suceder si una instrucción falla o cuando un dispositivo necesita usar el microprocesador.
- **Soporte de memoria virtual:** Si soporta o no el uso de **memoria virtual**, es decir, si cada programa tiene la ilusión de estar un espacio de memoria secuencial cuando en realidad el sistema operativo realiza el manejo de la memoria principal.

1.1.2. Arquitectura de Von Neumann

Como se vio en Organización del computador I, las mayoría de las ISA usadas actualmente usan el modelo de Von Neumann. Éste es un ciclo de cuatro etapas:

1. **Fecth:** Se utiliza un **program counter** para saber donde está almacenada la próxima instrucción a ser ejecutada. Y se carga desde la memoria.
2. **Decode:** Se decodifica la instrucción fetcheada y se consiguen los operandos (literales y registros) correspondientes.
3. **Execute:** En esta etapa se busca en memoria los datos requeridos (si es necesario) y se procesa los datos acorde a la instrucción.
4. **Write Back:** Se almacenan los resultados obtenidos en el lugar indicado.

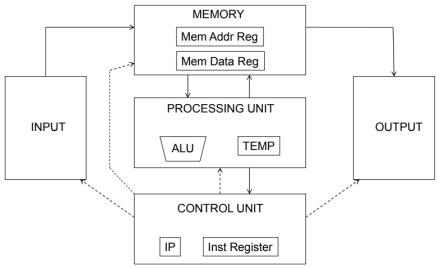

Figura 2: Arquitectura de Von Newmman

Cada instrucción es extraída de la memoria usando la dirección indicada por el **Instruction Pointer**. La unidad de control se encarga de indicar a la memoria si son necesarios otros valores para poder llevar a cabo su ejecución y, luego, pasa todo los datos a la unidad de procesamiento.

Si bien este es el ciclo que "ve" un programador, la implementación de las ISA (micro-arquitectura) usa estructuras de hardware más complejas que permiten acelerar la ejecución de cada fase.

1.2. Micro arquitectura

La micoarquitectura es la implementación a nivel hardware de la ISA, es decir, es un conjunto de componentes electrónicos organizados de cierta manera para que respeten esas especificaciones.

Desde el punto de vista de la implementación (hardware), el ciclo es realizado por unidades de procesamiento que operan sobre los datos de acuerdo a ciertas señales.

Cada instrucción es una señal que usa el procesador de instrucciones para decidir que conjunto de componentes electrónicos deben ser activados para poder llevar a cabo la operación deseada. Específicamente, las instrucciones indican:

- **Datapath:** Que elementos deben manejar y transformar los datos (unidades de procesamiento, de almacenamiento y estructuras de hardware que permiten el flujo de datos)
- **Control Logic:** Que elementos de hardware determinan las señales de control que indican al datapath lo que debe hacer con los datos.

En otras palabras, la micro-arquitectura comprende la tecnología utilizada para construir el hardware, la organización e interconexión de la memoria, el diseño de los bloques de CPU y la implementación de distintos mecanismos de procesamiento que no son visibles para el programador.

Parte I
Subsistema de memoria

2. El sistema de memoria

2.1. Principio de vencidad o localidad

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

Spatial Locality: Si actualmente se está accediendo a una dirección determinada de memoria, la probabilidad de que ésta y sus adyacentes sean accedidas en el futuro inmediato es muy alta.

La localidad de los programas surge naturalmente de su estructura. La mayoría contiene loops, osea que las instrucciones y los datos usados en ellos será accedido repetidamente lo que genera localidad temporal. Además, como son accedidas secuencialmente se tiene localidad espacial.

2.2. Jerarquías de memoria

Para aprovechar estos principios, se implementa la memoria de una computadora como una **jerarquía** que consiste en múltiples niveles de memoria con diferentes tamaños y velocidades. Mientras más rápida sean, más caras por bit son. El objetivo es presentar al usuario con tanta memoria como sea posible con la tecnología más barata proveyendo, al mismo tiempo, la velocidad de acceso provista por las memorias más rápidas.

En la actualidad hay tres tipos de tecnologías usadas para construir las distintas jerarquías. La memoria principal es implementada con DRAM (Dynamic Random Access Memory) mientras que los niveles más cercanos al procesador (caché) usan SRAM (Static Random Access Memory) [Ver sección 3].

Sin importar el tamaño de la jerarquía, los datos son copiados solo entre dos niveles adyacentes. El nivel más alto - el más cercano al procesador - es más pequeño y rápido y usa tecnología más cara que el nivel más bajo.

Bloque: La mínima unidad de información que puede estar presente en la jerarquía de dos niveles (la que está compuesta por las dos memorias que intercambian información).

Hit: Se produce cuando la información pedida por el procesador se encuentra en algún bloque de la memoria que se está utilizando.

Miss: Cuando la información debe ser buscada en el nivel inferior de la jerarquía.

Hit Rate: La fracción de accesos a memoria encontrados en cada nivel de la jerarquía. A menudo es usado como medida de rendimiento de la misma.

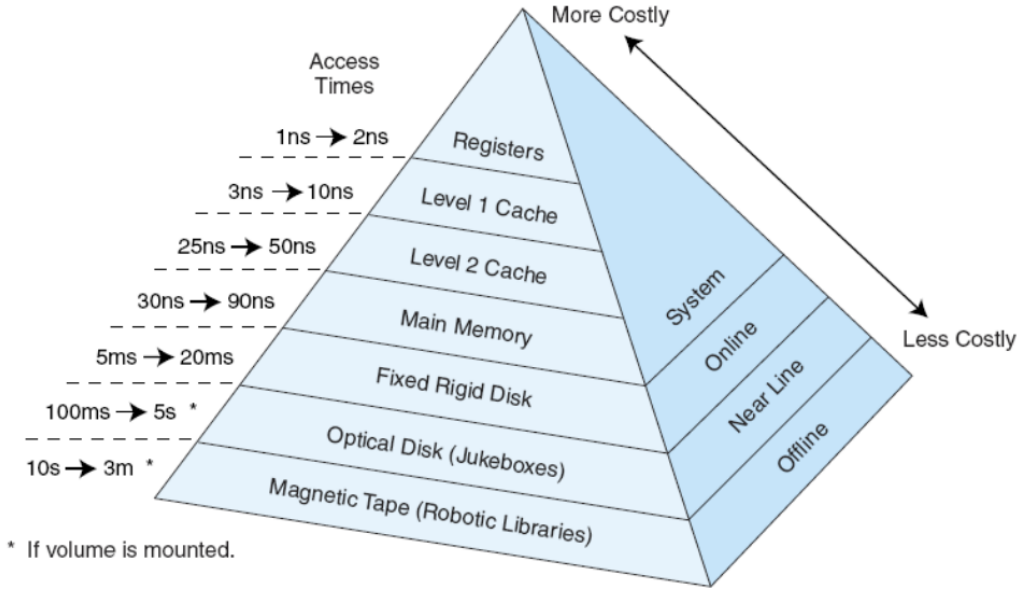

Figura 3: Memory Hierarchy

Miss Rate: 1 - hit rate (para cada nivel)

Hit time: El tiempo que se tarda en acceder a un bloque en cada nivel de la jerarquía.

Miss penalty: El tiempo requerido para fetchear un bloque desde el nivel inferior de la jerarquía (incluyendo tiempo de acceso, transmisión y copiado del bloque).

3. Tipos de memoria

Hay dos tipos de memoria:

- **No volátiles:** Son memorias capaces de retener la información almacenada cuando se les desconecta la alimentación. Son la tercera digievolución de las memorias **ROM** (Read Only Memory) que debían ser grabadas por el fabricante del chip y no eran modificables.

  De ROM pasaron a ser componentes programables que podían ser borrados con luz ultravioleta de una determinada longitud de onda. Y, luego, se convirtieron en las actuales memorias flash que pueden ser grabadas por algoritmos de escritura *on the fly* por el usuario. El ejemplo más habitual son los discos de estado sólido de los equipos portátiles modernos.

  Se usan fundamentalmente para almacenar el programa de arranque de cualquier sistema.

- **Volátiles:** Conocidas como **RAM**, se caracterizan por que una vez interrumpida la alimentación eléctrica, la información que almacenaban se pierde.

  Estas memorias pueden almacenar mayor cantidad de información y modificarla en tiempo real a gran velocidad a comparación de las No Volátiles.

  Se clasifican de acuerdo con la tecnología y su diseño interno en:

  - **Dinámicas (DRAM):** Almacenan la información en forma de una carga en un capacitor y la sostiene durante un breve lapso de tiempo con la ayuda de un transistor.

    Para guardar la información se activa el transistor que aplica el voltaje apropiado a la linea del bit. Una vez cargado el capacitor, se desactiva.

    Después de un tiempo, la carga del capacitor (que se empieza a descargar) llega a un determinado threshold y en necesario volver a activar el transistor para cargar la información otra vez (si había un uno).

    Durante una operación de lectura, los capacitores de la celda seleccionada son activados y descargados completamente. En este caso, un amplificador detecta los valores leídos y aplica voltaje a las lineas de bits necesarias para volver a cargar los capacitores necesarios. Esto aumenta el tiempo de acceso a la celda ya que no se puede liberar la operación hasta no haber repuesto el estado de carga de cada uno de ellos.

  - **Estáticas (SRAM):** Almacenan la información en un bi-estable Una celda se compone de seis transistores (por lo que tienen menos capacidad por componente que las dinámicas).

    Tres de los seis transistores están saturados (conducen la máxima corriente posible de forma permanente) y los otros tres están al corte (conducen una corriente prácticamente insignificante pero no nula). Esto genera un mayor consumo de energía por celda.

    La lectura es directa y no destructiva lo que se traduce en un tiempo de acceso bajo en comparación con las memorias dinámicas.

4. Memoria Caché

Los niveles de caché son bancos de SRAM de muy alta velocidad que contienen una copia de los datos e instrucciones que están en memoria principal. Deben ser lo suficientemente grandes para que el procesador resuelva la mayor cantidad posible de búsquedas de código y datos en memoria asegurando una alta performance y lo suficientemente pequeñas para no afectar el consumo ni el costo del sistema.

Para implementar estas memorias se debe agregar un controlador que se encarga de mantener la caché actualizada y de indicarle que datos debe mandar al procesador.

4.1. Operación de lectura

- El procesador inicia un ciclo de lectura de memoria, envía la dirección necesaria al controlador de caché.
- El controlador busca la dirección en el directorio de la caché.
  - Si hay un **hit**, busca el ítem en la memoria caché y lo envía al procesador.
  - Si hay un **miss**, busca el ítem en el sistema de memoria y lo copia en la caché. Actualiza el directorio de la misma y después manda la data al procesador.

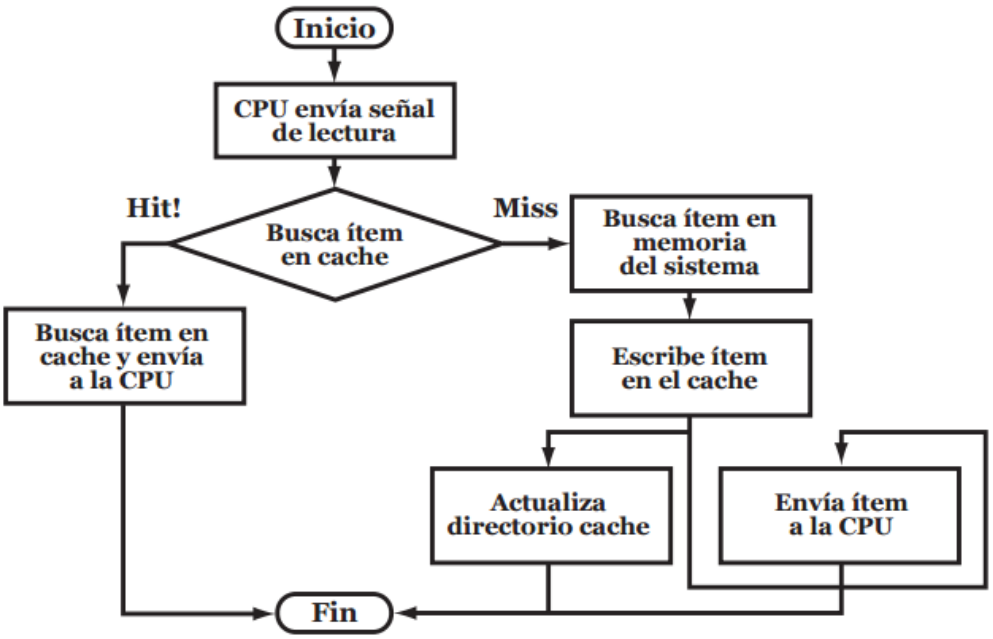

Figura 4: Operación de lectura

4.2. Organización de la caché

Línea: Elemento mínimo de palabra de datos dentro del caché. Corresponde a un múltiplo del tamaño de la palabra de datos de memoria. Esto nos permite copiar, en memoria, el ítem requerido y aquellos que lo rodean (principio de vecindad espacial).

Set: Un conjunto que contiene 2^{ln} lineas de la caché.

4.2.1. Mapeo directo

Se divide, a la memoria en $2^j - 1$ paginas. Y cada página en n bloques.

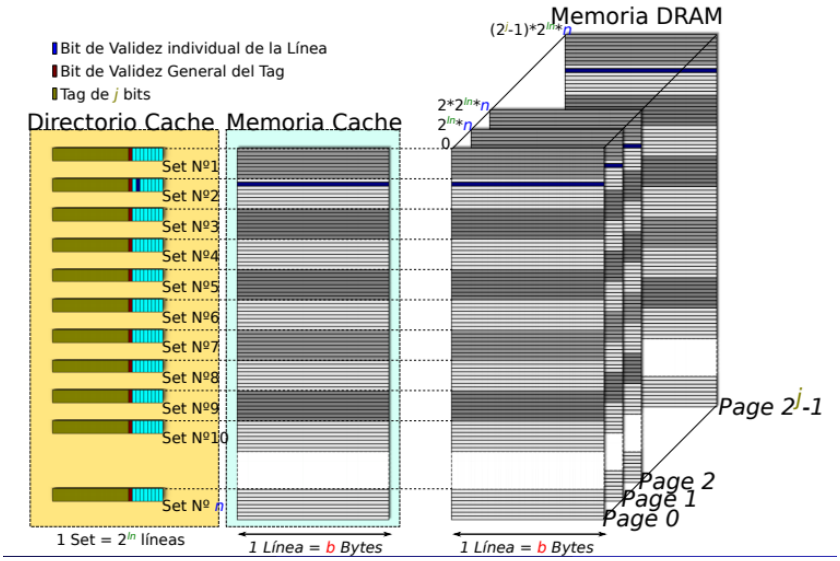

Figura 5: Mapeo directo

- Los primeros j bits de la dirección identifican la página a la que pertenece la línea.
- Los siguientes n, el set que les corresponde en la caché.
- Los ln bits identifican la línea que le corresponde dentro del set
- y los últimos b bits indican el índice de la palabra buscada dentro de la línea.

Para buscar una dirección de memoria, en este tipo de caché:

- Se busca el set en el que se encontraría esa dirección.
- Si el set tiene información válida, se corrobora que sea de la página de memoria a la que pertenece esa dirección.

- Si la página es correcta, entonces se accede al set en busca de la línea correspondiente y se vuelve el dato pedido
- Sino, se trae de memoria un bloque de tamaño del set y remplaza el set actual por ese.

4.2.2. Totalmente asociativa

Un bloque de memoria puede ser cargado en cualquier línea de la cache. Para determinar si un bloque se encuentra en la cache, se debe examinar el tag de todas las líneas para encontrar un match. Cuando la caché esta llena, se utiliza un algoritmo (generalmente least recently used - LRU)) para decidir que bloque remplazar.

Otros algoritmos de remplazo son FIFO (Firt In- First out, remplaza el bloque mas viejo) o LFU (Least frequently used, remplaza el bloque menos referenciado).

4.2.3. Asociativa de dos vías

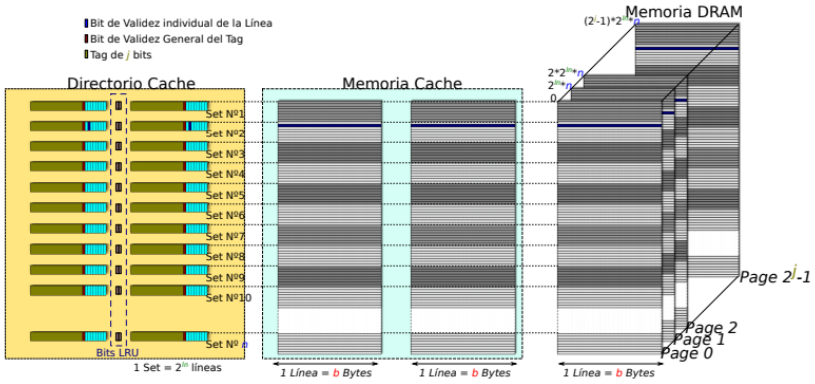

Figura 6: Asociativo por 2 vías

Es una combinación de las dos anteriores. La cache se divide en dos y a un bloque le corresponde un set especifico. Sin embargo, dentro de ese set puede ser copiado en cualquiera de las dos mitades. Cuando hay que remplazar un bloque, se utiliza un LRU para decidir cual mitad usar.

4.3. Operación de escritura (Coherencia de caché)

Cuando modificamos un valor, sería ideal que el cambio se vea reflejado tanto en la caché como en memoria principal. Si esto no pasa, entonces diremos que las memorias son inconsistentes/incoherentes.

Dependiendo del sistema (si hay una sola CPU o más de una) se utilizan distintas **políticas de escritura**. Decidir cuál de ellas usar constituye una de las decisiones más importantes en el diseño del sistema de memoria:

- **Write through:** Cada vez que hay que hay que modificar una dirección, el procesador manda el dato tanto a memoria principal como al controlador de la caché y se realiza la escritura en ambas memorias.

  Esto garantiza coherencia entre ambos datos de manera absoluta pero penaliza cada escritura con el tiempo de acceso a DRAM. Osea que la performance se degrada durante estas operaciones.

- **Write through buffered:** El procesador manda la data al controlador de caché que la actualiza y sigue ejecutando instrucciones y usando datos de la misma.

  El controlador dispone de un buffer de escritura que va almacenando estas modificaciones mientras se espera a que sean escritas a memoria. Cuando la escritura finaliza, se desencola y sigue con la próxima modificación.

  Si el buffer está lleno cuando el procesador pide una escritura, entonces se debe parar la ejecución y esperar a que tenga una entrada libre. Si bien la ocurrencia de los stalls es reducida, estos siguen pasando.

- **Copy back / Write back:** Se modifican solo las líneas de la caché y se marcan como **dirty** (modificadas). El controlador escribe el bloque en memoria cuando debe ser remplazada por otro.

  Este método mejora el performance cuando el procesador puede generar escrituras tan o más rápido de lo que puede escribir en memoria. Sin embargo, es mucho más difícil de implementar.

Si el procesador realiza un miss mientras el controlador caché está accediendo a la DRAM, deberá esperar a que la escritura termine para que su request sea atendido.

4.3.1. Coherencia en sistemas multi-procesador (Snoop Bus)

El bus es un conjunto de cables que conecta varios dispositivos, cada uno de los cuales puede observar cada transacción del mismo. Cuando un procesador emite un pedido a su caché, el controlador examina el estado de la misma y realiza las acciones adecuadas para completarlo. Esto puede generar transacciones para acceder a memoria.

SMP (Symetric Multiprocessors:) Cada procesador posee una caché de primer nivel y todas ellas están conectadas al subsistema de memoria principal y dispositivos de E/S a través un único bus compartido.

Cuando un procesador modifica un bloque de memoria presente en su caché, se produce una incoherencia en las caches de los otros procesadores que tenían su propia copia del bloque. Esto

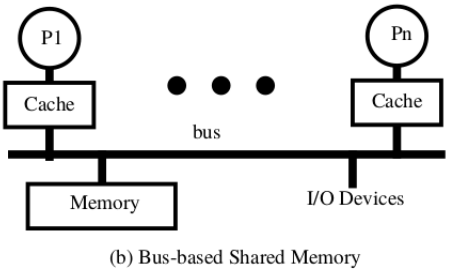

Figura 7: Bus de memoria compartido

sucede porque contienen un valor obsoleto en la dirección de memoria modificada por el primer procesador.

La coherencia se mantiene haciendo que cada controlador de caché espíe (snoop) el bus y monitoreé las transacciones. El controlador del bus funciona como arbitro para decidir el orden en el que éstas son ejecutadas. Cuando una transacción es enviada por el bus, se envía la dirección a la que se está accediendo y el tipo de operación que se está realizando sobre ella.

Cada controlador de cache toma la dirección enviada por el bus y chequea si tiene una copia del bloque referenciado por la misma.

- **Write-through:** Todas las escrituras se realizan directamente en la memoria principal. Todas las caches que tengan una copia del bloque modificado invalidan esa entrada. De esta forma, cuando el procesador necesite leer esa dirección, se tendrá que volver a cargar el bloque desde la memoria principal.

  El problema con este método es que se accede a memoria por cada operación de almacenamiento.

- **Write-back:** Al momento es el método más utilizado ya que reduce drásticamente estos accesos. Sin embargo, no puede ser usado directamente en estos sistemas ya que no sería posible identificar donde está el último valor válido de una dirección. Por esta razón, se desarrollaron protocolos de coherencias (el más popular de ellos M.E.S.I) que nos permiten identificar la validez de nuestros datos.

4.3.2. Protocolo MESI

En este protocolo, un bloque de memoria en la caché puede estar en cuatro estados:

- **Modified (M):** Es la única copia valida. La memoria principal está desactualizada y ninguna otra caché tiene una copia valida del mismo bloque.
- **Exclusive (E):** Esa caché es la única que contiene una copia de ese bloque. Además, esa copia no está modificada.

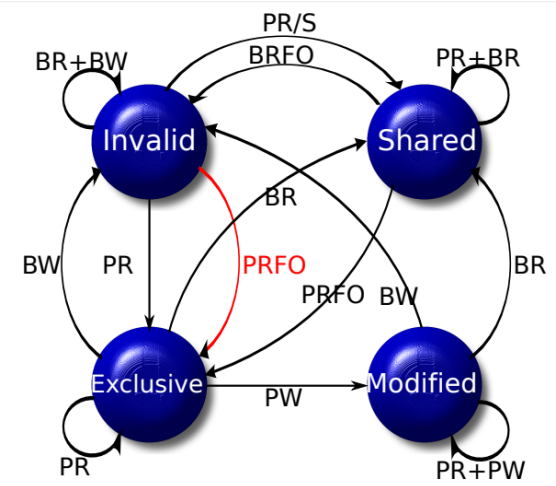

Figura 8: Diagrama de estados de MESI

- **Shared (S):** Está presente en esa caché sin ninguna modificación, la memoria principal está actualizada y puede haber otra caché que también lo tenga.
- **Invalid (I):** No está presente en la caché o contiene información desactualizada.

Lectura de un bloque: Supongamos que el procesador necesita leer un bloque de memoria, entonces produce un processor read (**PR**):

- Si el bloque esta en caché (**Modified**, **Exclusive** o **Shared**), se resuelve el pedido.
- Si el bloque es **Invalid** entonces se genera un busRead (**BR**) y todas las cachés responden con una señal indicando si tienen o no una copia del valor pedido:
  - Si no está en ninguna otra caché, se lo carga en modo **Exclusive** desde memoria principal.
  - Si lo tiene alguna otra, se copia el bloque en modo **shared** después de haber tomado las medidas necesarias para mantener la consistencia. Si la otra caché lo tiene:
    - En modo **Exclusive**, se lo pasa a estado **Shared**.
    - En modo **Modified**, se copian a memoria principal los cambios realizados y se lo pasa a **Shared**
    - En modo **Shared**, no se hace nada.

Escritura de un bloque: Para escribir en un bloque, el procesador primero debe asegurarse que es el único con permisos de escritura, entonces:

- Si el dato está en caché en modo **Modified** o **Exclusive**, genera un processor write (**PW**) y se modifica la caché. En el segundo caso, se pasa al estado **Modified**.

- Si está en **Invalid** ó **Shared**:
  - Genera un Processor Request For Ownership (**PRFO**), pasa el bloque a estado **Exlusive** y todas las otras cachés invaliden sus copias.
  - Luego, genera el proccesor write (**PW**), realiza las modificaciones y pasa el bloque a **Modified**

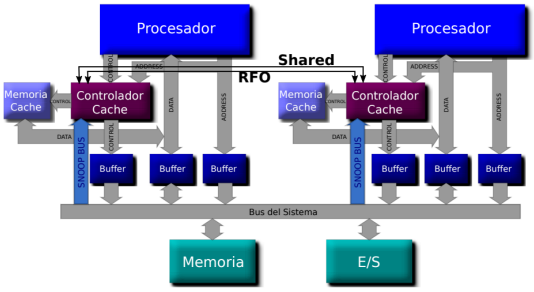

Figura 9: Implementación de MESI en un SMP

Parte II
Instruction Level Parallelism

5. Pipelining

El Pipeline es una técnica que permite superponer el procesamiento de múltiples instrucciones en una ejecución. Una vez que la primera instrucción fue fetcheada y pasa a la etapa de decodificación, se comienza a fetchear la siguiente instrucción.

Bajo condiciones ideales y con un gran número de instrucciones, la mejora en velocidad de ejecución es directamente proporcional a la cantidad de etapas en el pipe. Es decir, un pipeline de 5 etapas, es aproximadamente 5 veces más rápido que el procesamiento secuencial.

Si bien el pipelining reduce el tiempo de ejecución de un programa, debemos notar que no lo hace modificando el tiempo que se tarda en procesar una instrucción (**latency / latencia**) sino que aumenta la cantidad de instrucciones que se procesan por unidad de tiempo (**throughput**).

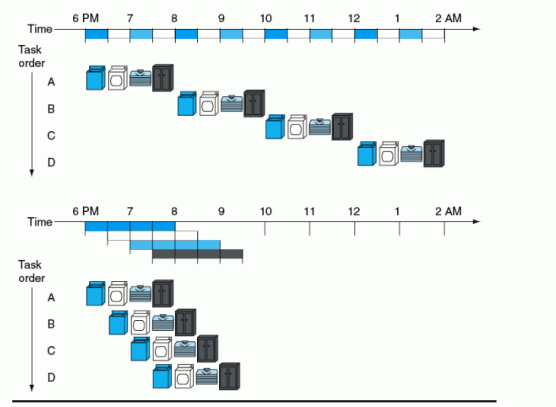

Figura 10: Analogía de la lavandería: 4 personas tienen que lavar, secar, doblar y guardar su ropa sucia. Solo disponemos de una lavadora, una secadora, un "doblador" y un ropero. Si cada parte del proceso toma 30 minutos, y lo realizamos de manera secuencial, completarlo para las 4 personas tomaría ocho horas, mientras que si lo hacemos con un pipeline el tiempo se reduce a tres horas y media.

Para facilitar una implementación efectiva del pipeline, el set de instrucciones propuesto debe permitir que todas las etapas tarden el mismo tiempo en ser ejecutadas. Esto implica que hay que tener ciertas consideraciones al momento de diseñarlo:

- Todas las instrucciones deben tener la misma longitud (en la arquitectura IA-32, donde las instrucciones tienen longitud variada, esto se logra descomponiendo cada instrucción en micro-operaciones de longitud fija y a estas micro-operaciones se les aplica el pipelining).

- Todas las instrucciones deben tener la misma estructura, es decir, la cantidad de parámetros y la ubicación dentro de la instrucción que los explicita deben ser las mismas (o lo más parecidas posibles).
- Los operandos deben estar alineados en memoria. Esto es, las direcciones de memoria que ocupan deben ser múltiplo del tamaño de las palabras usadas para que los datos puedan ser transferidos en una única etapa del pipeline.

5.1. Pipeline Hazards

Hay situaciones en las que una instrucción no puede ser ejecutada en el siguiente ciclo de reloj debido a algún obstáculo (**hazards**). Estos eventos pueden llegar a detener el flujo del pipeline (**pipeline stall**) y generar una demora en el procesamiento de las instrucciones. A continuación veremos los tres tipos de obstáculos que se pueden dar:

- **Estructurales (Structural Hazards):** El hardware no soporta la combinación de instrucciones que queremos ejecutar en el mismo ciclo de reloj. Por ejemplo, si una instrucción debe acceder a memoria mientras otra debe realizar un fetch en la misma memoria. En este caso, las dos instrucciones deben utilizar los mismos recursos y debería dársele prioridad a la primera instrucción dejando a la segunda en espera.

  Para resolver esto debería agregarse hardware (como buffers, caches, etc).

- **De datos (Data Hazards):** Hay una instrucción en el pipeline que depende de los resultados de otra instrucción (también en el pipeline) y debe esperar a que ésta se complete para poder terminar. Esto puede bloquear el pipe durante varios ciclos de reloj ya que se debe procesar completamente la primer instrucción.

  Para minimizar el impacto de este obstáculo, por lo general, se agrega extra hardware que permite conseguir el valor deseado apenas sea calculado (cuando termina la etapa de ejecución) directamente de los componentes internos para no tener que esperar a que sea guardado en memoria (esta técnica se llama **forwarding** o **bypassing**).

- **De control (Control or Branch Hazards):** La instrucción que se fetcheó no es la que debe ejecutarse en este ciclo de reloj. Cuando aparece una instrucción de control es fetcheada, el pipeline no puede saber cual es la próxima instrucción que debe ser ejecutada ya que esto depende del resultado de la instrucción actual.

  Una posible solución es parar apenas se haga el fetch del condicional y esperar hasta obtener sus resultado. Otra, es realizar predicciones (**branch prediction**) y ejecutar las instrucciones con más probabilidad de ser ejecutadas. En este último caso, el pipeline procede sin demoras si la predicción fue correcta sino debe eliminar las instrucciones procesadas y volver a comenzar desde el lugar correcto.

6. Branch Prediction

Branching: Cuando se fetchea una instrucción condicional, se genera una ramificación en el código. Hay dos caminos posibles por el que puede seguir la ejecución y el camino que se debe ejecutar lo decide el resultado de la instrucción fetcheada.

Untaken Branch: Se dice cuando el camino tomado es el que ejecuta la instrucción que le sigue al condicional dentro del programa.

Taken Branch: Cuando se ejecuta la instrucción que se encuentra en la dirección del salto de la instrucción condicional.

Branch penalty: Cuando se fetchea la instrucción incorrecta, debe descartarse todo lo que estaba pre-procesado hasta ese momento. El pipeline se vacía y deben transcurrir $n-1$ ciclos de clock hasta el próximo resultado (n = cantidad de etapas del pipeline).

6.1. Predicciones estáticas

- **Assume Branch Not Taken:** Se fetchea la siguiente instrucción secuencial del programa. Si el salto no se realiza entonces la ejecución del pipe continúa sin problemas. Si el salto se realiza, se aplica el branch penalty.
- **Assume Branch Taken**: Análogo al anterior pero se fetchea la instrucción en la dirección de memoria apuntada por el salto.
- **Predict by Opcode**: Se asume que el salto va a ser tomado o no dependiendo de la instrucción a ser ejecutada.

Este tipo de predicciones funcionan bien para piplines simples. Sin embargo, la penalidad de descartar instrucciones y el tiempo que se tarda en restaurar el sistema aumenta acorde a la complejidad del mismo. Las predicciones dinámicas intentan disminuir este problema manteniendo estadísticas de uso y modificando la decisiones tomadas a medida que se realiza la ejecución.

6.2. Predicciones dinámicas

- **Branch Prediction Buffer**: Se mantiene una tabla indexada por la dirección de memoria de la instrucción del salto y 2 bits que indican si los últimos dos saltos hacia esa instrucción fueron tomados o no.

  Las primeras implementaciones de esta técnica hacían uso de un único bit que se remplazaba cada vez que la predicción fallaba. En ciertos casos, la eficiencia de este método no era satisfactoria llegando a fallar completamente en otros (por ejemplo, si los saltos termina formando una secuencia intercalada de Taken y Not taken).

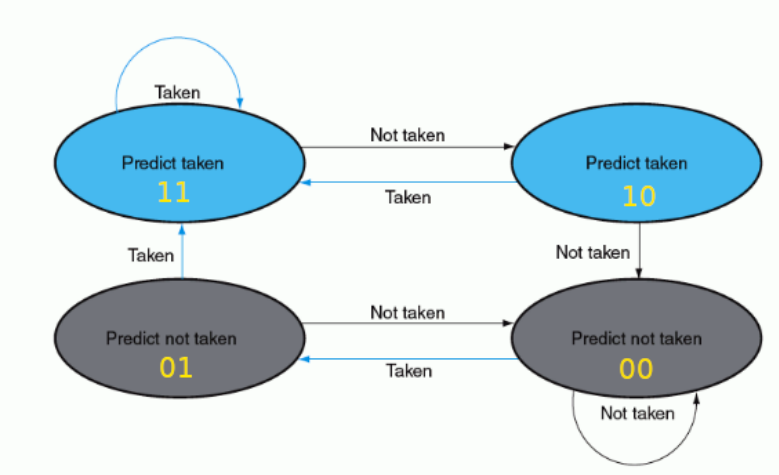

Figura 11: Maquinas de estados de predicción para un buffer de 2 bits

- **Branch Target Buffer (BTB)**: Es una caché en la que se almacena, en cada entrada, la dirección de la instrucción de control y la dirección de la instrucción que fue ejecutada después de resolver el branching.

  Cuando se fetchea una instrucción de control, se accede a esta caché usando el valor del program counter.

  - Si el valor no está en el buffer, entonces se asume *taken*.
    - Si el resultado es *Non-Taken* se acepta el delay en el pipeline y no se almacena nada en el BTB.
    - Si el resultado es *taken*, se ingresa el valor al BTB.
  - Si el valor se encuentra en el BTB, se aplica el campo de dirección de target almacenado.
    - si el resultado es *taken* no hay penalidad. No se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos sirve.
    - Si el resultado es *Non-taken*, guarda el nuevo valor en el BTB, luego de la penalidad correspondiente en el pipeline.

7. Procesadores Superscalares

Son procesadores que explotan la ejecución paralela de instrucciones para mejorar el rendimiento. Agrega más pipelines a la ejecución, lo que permite incrementar aún más la cantidad de instrucciones procesadas por ciclo de reloj.

Un problema es que los obstáculos estructurales quedan más expuestos. Cada etapa, ademas de lidiar con los data hazards de su propio pipeline, debe lidiar también con las mismas etapas del otro pipeline.

Este tipo de procesadores analizan los binarios secuenciales de los programas y lo paralelizan eliminando esencialidad innecesaria. Por esta razón, los programas binarios deben ser vistos más como una especificación de lo que debe hacerse y no como lo que realmente sucede.

Más precisamente, un procesador superescalar implementa:

1. Estrategias de fetch que permiten fetchear múltiples instrucciones mediante la predicción de resultados y saltos.

2. Métodos para determinar dependencias de registros y mecanismos para comunicar esos valores cuando sea necesario durante la ejecución.

3. Métodos para iniciar, o resolver, múltiples instrucciones en paralelo.

4. Recursos para la ejecución en paralelo de varias instrucciones, incluyendo múltiples unidades funcionales de pipelines y jerarquías de memorias capaces de atender simultáneamente múltiples referencias a memoria.

5. Métodos para manejar datos a través de instrucciones de lectura/escritura e interfaces de memoria que tengan en cuenta el comportamiento dinámico (y muchas veces impredecibles) de las jerarquías diseñadas.

6. Métodos para actualizar los estados del proceso en el orden correcto (para mantener la apariencia de la ejecución en orden secuencial).

7.1. Dependencias de instrucciones

Por lo general, se interpreta el binario de un programa como un conjunto de bloques compuestos de instrucciones contiguas. Una vez que se fetchea un bloque, se sabe que todas sus instrucciones van a ser ejecutadas eventualmente. Cuando esto suceda, diremos que el bloque es iniciado en una **ventana de ejecución**.

Una vez que las instrucciones entran en esta ventana, son ejecutadas en paralelo teniendo en cuenta sus dependencias que pueden ser de control o datos. Las de control son generadas por condicionales y pueden ser resueltas/optimizadas con predicciones (Sección 6).

Las **dependencias de datos** se da entre instrucciones que referencian el mismo espacio de memoria. En estos casos, si las instrucciones no se ejecutan en el orden correcto, puede haber errores en las operaciones. Pueden ser Verdaderas o Artificiales

- **Dependencias verdaderas:** Cuando una instrucción debe leer un valor que todavía no fue generado por una instrucción previa (**Read after write hazard**).
- **Dependecias artificiales:** Resultan de instrucciones que deben escribir un nuevo valor en una posición de memoria pero debe esperar a que las instrucciones previas utilicen el valor actual (**Write after Read hazard**) o cuando varias instrucciones deben escribir la misma posición de memoria (**Write after Write hazard**).

  Estas dependencias son producidas por código no optimizado, por escasez de registros disponibles, por el deseo de economizar el uso de la memoria principal o por ciclos donde una instrucción puede colisionar consigo misma.

7.2. Fetching and Decode

En los procesadores superescalares, una caché de instrucciones se usa para reducir la latencia y incrementar el ancho de banda del fetching. Esta caché está organizada en bloques o lineas que contienen varias instrucciones consecutivas y almacena el tipo de cada una de ellas (si es de control, de operación, de lectura de memoria, etc).

El program counter se utiliza para determinar la posición de una instrucción en la caché.

- Si se produce un hit, se fetchea el bloque de instrucciones y se suma al program counter el tamaño del bloque fetcheado.
- Si hay un miss, el caché pide la instrucción buscada en memoria y la copia.
- Se identifica el tipo de cada instrucción. Si alguna es de control, se realizan las predicciones usando alguno de los métodos mencionado en en la sección 6.
- Las instrucciones son decodificadas en **tuplas de ejecución** que contienen la operación a ser ejecutada, las identidades de los elementos donde se encuentran los parámetros de entrada y donde deben guardase los resultados.

  En el programa estático, las instrucciones utilizan los registros **lógicos** (los de la arquitectura). Por esta razón, cuando son decodificadas, se mapean (o renombran) a un registro físico y las dependencias artificiales se resuelven indicando a las instrucciones involucradas que usen distintos registros físicos (**Algoritmo de tomasulo**, sección 7.4).
- Una vez que todas las instrucciones asociadas a un registro físico son completadas (modifican el estado visible), se libera el registro para que pueda ser usado por otro bloque de instrucciones.

7.3. Scoreboarding

Cuando una instrucción es decodificada, se loguea en un archivo de registros llamado **scoreboard**. Éste se encarga de commitear, mandar a ejecutar y detectar dependencias. Cuando detecta que las instrucciones pueden leer sus operandos las manda a ejecutar. Si decide que no

puede hacer esto, monitorea cada cambio del hardware hasta que los operandos necesarios estén disponibles. Además, controla cuando una instrucción puede escribir sus resultados en el registro correspondiente. De esta manera, la detección de hazards y su resolución es centralizada en el scoreboard.

Durante la ejecución, cada instrucción pasa por cuatro etapas:

1. **Issue:** Si hay una unidad funcional libre para la instrucción y ninguna otra instrucción activa necesita escribir en el registro destino, el scoreboard la envía a esta unidad y actualiza su estado interno. Si hubiese otra instrucción que necesita del mismo registro, entonces se bloquea (*stall*) hasta que todos los obstáculos hayan sido resueltos.

   Cuando esta etapa se bloquea *stall*, el buffer de Fetching también entra en este estado cuando se llena.

2. **Lectura de operandos:** Un operando se encuentra disponible si ninguna instrucción activa emitida anteriormente necesita escribirlo. Cuando los operandos fuentes están habilitados, el scoreboard le dice a la unidad funcional que proceda a leerlos desde los registros y comience la ejecución. De esta forma se evitan los RAW hazards que, como se resuelven dinámicamente, permiten que las instrucciones sean ejecutadas fuera de orden.

3. **Ejecución:** Las unidades funcionales comienzan la ejecución después de haber recibidos todos los operandos. Cuando el resultado está listo, lo notifica al scoreboard.

4. **Escritura de resultado:** Una vez que el scoreboard recibe la notificación, chequea por WAR hazards. Si los encuentra, bloquea la escritura de la instrucción hasta que estos son resueltos.

Desventajas de este método

- No puede aprovechar el forwarding de datos porque considera que un operando solo está listo cuando fue escrito en registro.
- El scoreboard controla el progreso de una instrucción comunicándose con las unidades funcionales. Al haber un numero limitado de buses de operandos y resultados, debe asegurarse que el número de unidades funcionales habilitadas para proceder no excedan estos recursos.

7.4. Tomasulo

El algoritmo de Tomasulo crea un scheduling dinámico que permite ejecutar instrucciones fuera de orden y habilita el uso de múltiples unidades de ejecución. Para lograr esto necesita:

- Mantener un “link” entre el productor de un dato con su(s) consumidor(es).
- Mantener las instrucciones en espera hasta que estén listas para ejecución.
- Saber cuando sus operandos están listos (“Ready”).

- Despachar ("dispatch") cada una de ellas a su Unidad Funcional correspondiente ni bien todos sus operandos estén listos.

Register Renaming: El primer ítem se logra asignando a cada registro un alias. Para eliminar los WAR y WAW hazards, se utiliza una tabla llamada **Register Alias Table** que asocia un tag con cada operando de una operación.

Register Station (RS): Una vez realizado el renombre se le asigna, a la instrucción, un RS. Un RS es un subsistema de hardware compuesto de bancos de registros internos que se encarga de mantener las instrucciones en espera hasta que estén listas para ser ejecutadas. Éste chequea constantemente por la disponibilidad de los operandos de la instrucción. Para cada uno de ellos cuyo valor no esté disponible, el RS, guarda el *tag* que se le asignó en el paso anterior.

Cada vez que una unidad de ejecución pone disponible un operando, transmite el **tag** asociado al mismo junto con su valor a todas las RS. Cuando una instrucción tiene todos sus operandos disponible, el RS espera a que la unidad funcional asociada a ella esté libre y luego la despacha.

Common Data Bus : Es un datapath que cruza la salida de las unidades funcionales y atraviesa las RS, los Floating Point Buffers, los Floating Points Registers y el Floating Point Operations Stack. Se usa para realizar el broadcast de los resultados al finalizar cada operación.

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

if RS tiene recursos disponibles antes del renaming **then**
Se inserta en la RS la instrucción y los opernados renombrados (valor fuente / tag)
Se renombra si y solo si la RS tiene recursos disponibles.
else
stall
end if
while esté en la RS, cada instrucción debe **do**
Mirar el tráfico por el **Common Data Bus (CDB)** en busca de **tags** que correspondan a sus Operandos fuente.
Cuando se detecta un **tag**, se graba el valor de la fuente y se mantiene en RS.
Cuando ambos operandos están disponibles, la instrucciónn se marca **Ready** para ser despachada
if Unidad Funcional disponible **then**
Se depscacha la instrucción a esa unidad funcional
end if
if Finalizada la ejecución de la instrucción **then**
La unidad funcional pone el valor correspondiente al **tag** en el **CD** (**tag broadcast**)
if El archivo de registros está conectado al CDB **then**
if tag del Archivo de Registro == tag broadcast **then**
Registro = valor broadcast

```
                bit de validez = '1'
                Recupera tag renombrado
                No queda copia válida del tag en el sistema
            end if
        end if
    end if
end while
```

7.5. Ejecución de instrucciones

7.5.1. Interrupciones

Existen dos tipos de interrupciones:

- **Excepciones** (o trampas): Se generan cuando se produce un error durante la ejecución o el fetching de ciertas instrucciones. Por ejemplo, opcodes ilegales, errores numéricos o page faults.
- **Interrupciones externas**: Son causadas por instrucciones específicas y dispositivos externos qué están ejecutando algún proceso. Por ejemplo las interrupciones generadas por los dispositivos de entrada/salida (mouse, teclados, pantallas), timers, etc.

Cuando ocurre una interrupción, el software o el hardware (o una combinación de ambos) guardan el estado del proceso interrumpido. Este estado consiste, generalmente, del program counter, los registros y la memoria.

Interrupción precisa: Una interrupción es precisa si el estado guardado es consistente con la arquitectura secuencial del modelo. Osea se debe cumplir que:

1. Todas las instrucciones previas a la indicada por el program counter deben haber sido ejecutadas y deben haber modificado el estado del proceso correctamente.
2. Ninguna de las instrucciones siguientes a la indicada por el program counter debe haber sido ejecutadas ni deben haber modificado el estado del proceso.
3. Si la interrupción es causada por una excepción en una instrucción del programa, entonces el program counter guardado debe apuntar a la instrucción interrumpida.

Si el estado guardado es inconsistente con el modelo de arquitectura secuencial y no satisface estas condiciones entonces la interrupción es **imprecisa**.

7.5.2. In-Order Instruction Completion

Las instrucciones modifican el estado del proceso solo cuando se sabe que todas las instrucciones previas están libres de excepciones. Para asegurar esto se utiliza un registro llamado "result shift register" que contiene una tabla de n entradas (n la longitud del pipeline más largo). Una

instrucción que toma i ciclos de reloj, reserva la i-ésima entrada de la tabla y en cada ciclo se la desplaza una posición hacia abajo.

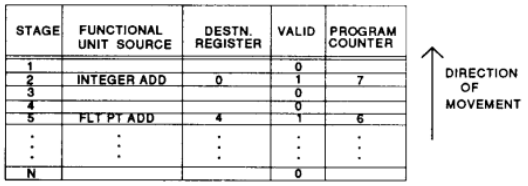

Fig. 2. Result shift register.

Figura 12: Result Shift Register

Si la i-ésima entrada contiene información valida entonces se pausa la instrucción. En el próximo ciclo, se re chequea si la entrada está libre y se realiza el movimiento.

Para evitar que una instrucción más corta se complete antes que otra de mayor longitud (cuando este es el orden deseado) se rellenan con información invalida todas las entradas anteriores que no fueron reservadas. De esta forma, la nueva instrucción es pausada hasta el próximo ciclo de reloj.

7.5.3. Reorder-Buffer

La principal desventaja del método anterior es que instrucciones rápidas serán retenidas a pesar de no tener dependencias.

Para resolver este problema se agrega un **reorder buffer** que indica el orden en el que las instrucciones deben modificar el estado visible. Entonces se permite que las instrucciones terminen en cualquier orden pero se les asigna un tag que cumple con el objetivo mencionado. Cuando son completadas, se las almacena en la entrada indicada (por el tag) del buffer.

El buffer contiene un puntero a la entrada que corresponde a la próxima instrucción a ser ejecutada. Cuando dicha entrada contenga información válida, se chequea si hubo excepciones. Sí no las hubo, se modifican los registros/memoria necesarios y se mueve el puntero a la próxima entrada. Sino, se emite la excepción y se invalidan todas las entradas posteriores.

En algunos casos, la dependencia entre instrucciones requiere el uso de valores que todavía no fueron escritos en los registros. Para conseguirlos es necesario acceder al buffer de reordenamiento, lo que agrega latencia y complejidad a la ejecución.

History Buffer: Una solución posible es dejar que las instrucciones modifiquen los registros cuando son completadas, pero que se guarden los valores previos en un buffer que permita recuperar estos datos en el caso de una excepción.

En este caso, si ocurre una excepción, se debe recorrer el history buffer para poder recuperar un estado preciso.

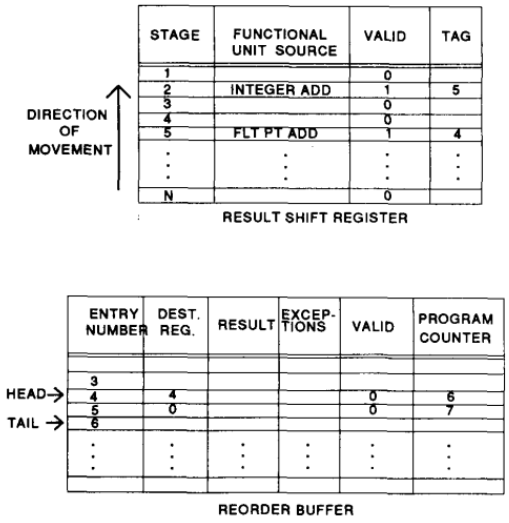

Figura 13: Reorder buffer

Future File: Otra solución es mantener dos archivos de registros. Uno que se actualice en el orden especificado por el programa (**Architectural File**). Y otro que se actualiza apenas se completa una instrucción (**Future File**).

Si ocurre una excepción simplemente se debe mostear el Architecture File para conseguir un estado preciso. Sin embargo, hay que copiarlo al future file para restaurar el estado del sistema.

Checkpointing: Cuando se predice el resultado de una instrucción de control, se guarda el estado del future file (**checkpoint**). Si al ejecutarla, la predicción resulta ser errónea entonces solo hay que copiar la información del checkpoint en el future file para recuperar el estado del sistema y comenzar de nuevo en la instrucción correcta.

Parte III

Procesadores Intel

1A partir de acá resumenes de los papers. Faltan los que hablan del Intel Core 2.

8. Netburst microarchitecture (Pentium 4 [1])

La microarquitectura de este procesador se puede separar en 4 grandes bloques: Front End, Out-of-order engine, Integer and Floating Point Execution Units and the memory subsytem.

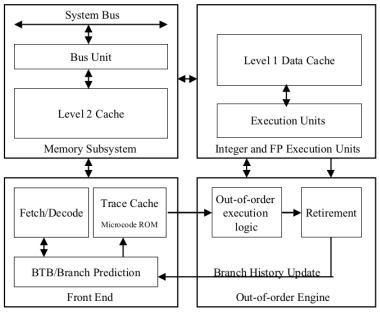

Figura 14: Diagrama básico

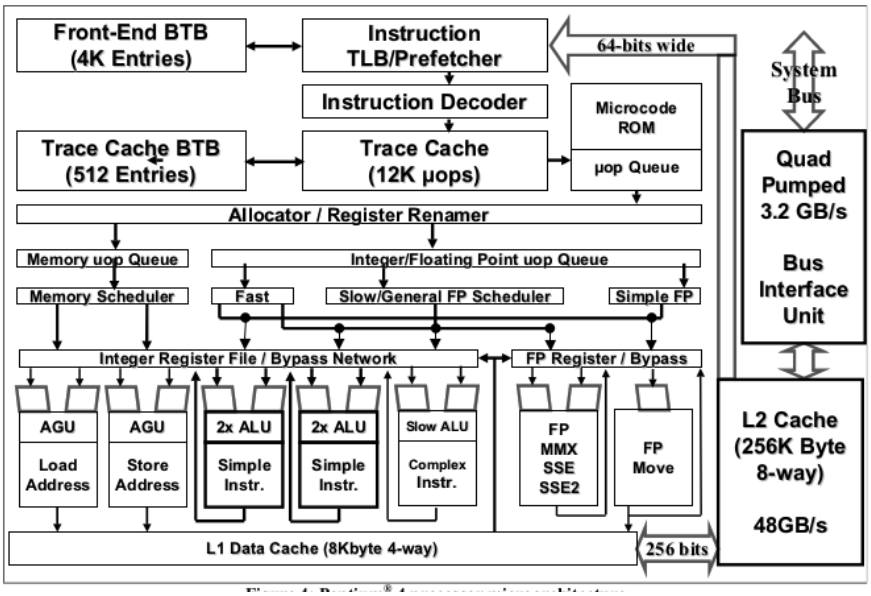

Figura 15: Architectura del procesador

8.1. Front end

Se dedica al fetcheo y decodificación de instrucciones y la predicción de saltos. Esta compuesto por:

Instruction Translation Lookaside Buffer (ITLB)

Traduce las direcciones lineales del puntero de instrucciones en las direcciones físicas necesarias para acceder a la caché L2. Además, realiza los chequeos de permisos de paginación.

Front end Branch Target Buffer

Fetchea las instrucciones IA-32 que deben ser ejecutadas de la caché L2. El uso de Branch Prediction permite al procesador comenzar a fetchear y ejecutar instrucciones mucho antes de que el outcome de las ramificaciones sea averiguado.

Si una rama no se encuentra en el BTB, el hardware de predicciones decide que hacer dependiendo el desplazamiennto de la rama (**forward** o **backward**). Backwards branches se asumen Taken y Forward Branches se asumen Not-Taken.

IA-32 Instruction Decoder

Recibe las instrucciones IA-32 desde la caché L2 y las decodifica en primitivas (llamadas micro-operaciones). Una vez decodificadas, si la instrucción consiste en menos de 4 micro-operaciones, entonces es almacenada en la **Trace Caché**. Sino es enviada a la **microcode ROM**.

Trace Caché

Es la caché de instrucciones primaria o de Nivel 1 (**L1**) donde se guardan ya decodificadas en forma de micro-operaciones. Una vez almacenadas, cada micro-op puede ser usada repetidamene cono si fuese una caché de instrucciones convencional.

Las instrucciones IA-32 tienen un número variable de bits y muchas opciones. La lógica de decodificación necesita tener en cuenta esto y convertirlas en micro-operaciones que la máquina sepa como ejecutar. Esto se hace especialmente difícil cuando se tratan de decodificar varias instrucciones en un único ciclo de reloj de alta frecuencia. Además, cuando la predicción de un branch es errónea, el tiempo de recuperación es mucho más corto si la máquina no tiene que re-decodificarlas para reanudar la ejecución en el lugar correcto.

El cacheo de las micro-operaciones permite, a la microarquitectura, evitar el uso decodificador de instrucciones IA-32 durante gran parte de la ejecución.

La caché ensambla las micro-ops en secuencias llamadas trazas (**traces**) que respetan el orden de ejecución predecido del programa. Esto permite que el target de los branch sean incluidos en la misma linea de traza incluso si estan separados por miles de bytes en el programa. Así, tanto el target como el branch, pueden ser enviados al motor fuera-de-orden al mismo tiempo.

La mayoría de las instrucciones son fetcheadas y ejecutadas desde esta caché. Solo cuando hay un caché miss, la microarquitectura fetchea y decodifica la instrucciones desde la caché **L2**.

La caché tiene su propio BTB (TBTB) para predecir cual es la próxima micro-operación a ser ejecutada. Esta es más chica que la BTB principal ya que solo debe realizar predicciones sobre el subset de instrucciones presentes en la caché.

Microcode ROM

Esta ROM es usada para instrucciones IA-32 complejas como mover strings y para manejar faults e interrupciones.

Cuando se encuentra una instrucción IA-32 compleja, la Trace Chache la llama y, ésta, emite las micro-operaciones necesarias para ejecutarla. Cuando termina de hacer esto, el control vuelve al Trace Caché.

Las micro-operaciones que provienen de la caché y la microcode ROM son encoladas en un buffer ordenado (**uop queue**), lo que ayuda a suavizar el flujo de micro-operaciones al Out-Of-Order Engine.

8.2. Out-of-order engine

Reordena las instrucciones para que sean ejecutadas apenas todos sus operandos estén listos y se encarga de presentar el estado correcto al front end cuando es necesario modificarlo. Su principal tarea es extraer paralelismo del flujo de instrucciones preservando la semántica de ejecución correcta del programa. Esto lo logra reordenando las micro-operaciones de acuerdo a sus dependencias y las disponibilidad de sus operandos.

Allocator (Asignador de recursos)

El allocator se encarga de distribuir los recursos (entradas del archivo de registros, buffers, ALUs, etc) del sistema entre las micro operaciones que deben ser ejecutadas. Si un recurso no está disponible para alguna de ellas, se pausa la asignación por un ciclo de reloj. Cuando todos los recursos están disponibles, entonces los asigna a las instrucciones correspondientes y las despacha al pipeline para que sean ejecutadas.

A todas las instrucciones les asigna una entrada en el reorder buffer (para mantener un seguimiento del estado en el que se encuentran), el registro donde se guardarán sus y una entrada en alguna de las colas para scheduling.

Register Renaming

El proceso de renombre remueve falsos conflictos causados por múltiples instrucciones independientes que necesitan de un mismo registro. Esto se logra creando varias instancias del mismo registro que indiquen quien las escribió por última vez y el valor adecuado. Ver sección 7.2

Scheduling de micro-operaciones

Hay dos colas FIFO, una para operaciones de memoria y otra para el resto. Las colas son atendidas en cualquier orden, lo que permite que la ventana de ejecución fuera de orden sea más

grande.

Hay varios schedulers, que se encargan de procesar distintos tipos de instrucciónes. Cada uno de ellos puede determinar cuando una micro-operación está lista para ser ejecutadas (cuando todos sus operandos fueron generados) trackeando sus registros de entrada. Cuando esto sucede, el scheduler la agrega a la cola de ejecución.

El scheduler detecta los flujos de ejecución independientes en una secuencia de micro-operaciones y permite que sean ejecutados paralelamente sin importar su orden general.

Los scedulers, están conectados a distintos puertos de despachos que envían las instrucciones a sus respectivas unidades de ejecución.

8.3. Integer and Floating Point Excution Units

Es donde se ejecutan las instrucciones. Los registros de enteros y los de puntos flotantes se encuentran separados. Cada archivo de registros está conectado a una red que permite enviar resultados completos que todavía no fueron escritos en los registros a las nuevas micro-operaciones.

Low Latency Integer ALU

Está diseñada para optimizar performance manejando los casos más comunes tan rápido como sea posible. Es capaz de completar operaciones en medio ciclo del reloj principal. Aproximadamente el 60-70 % de las micro-operaciones de enteros pasan por ella. Ejecutandolas a la mitad de la latencia del clock principal ayuda a acelerar la ejecución de la mayoría de los programas.

Complex Integer Operation

Los shift, rotaciones multiplicaciones y divisiones tienen una larga latencia (ocupan varios ciclos de reloj), por esta razón son mandados a otros componentes más complejos.

Low Latency Level 1 (L1) Data Cache

La latencia de las operaciones de lectura es un aspecto clave en la performance del procesador. Esto es especialmente verdad para los programas del IA-32 porque hay un limitado numero de registros en el ISA. Por esta razón, se agrega una pequeña caché de baja latencia que consigue su información desde una caché de segundo nivel. La L2 es más grande, tiene un gran ancho de banda y latencia media.

Con la alta frecuencia y la profundidad del pipeline del pentium 4, el scheduler de micro-operaciones despacha instrucciones más rapido de lo que el load puede ser ejecutado. En la mayoría de los casos, el scheduler asume que hubo en hit en la caché L1. Si la lectura genera un miss entonces habrá, en el pipeline, operaciones dependientes con información errónea que deben volver a ser ejecutadas. Para esto se utiliza un mecanismo conocido como **replay** que trackea las instrucciones durante su ejecución y, si ocurre el caso mencionado, las detiene e intenta ejecutar nuevamente.

Store-To-Load Forwarding

Los datos son escritos en la caché L1 en el orden programático y solo cuando se puede garantizar que el valor es no-especulativo. Es decir, todas las operaciones anteriores a la isntrucción de almacenamiento deben haber sido completadas antes de que sea commiteado. Debido a la profunidad del pipeline, esto puede tomar varios ciclos de reloj.

A menudo, nuevas instrucciones deben hacer del dato que se está reteniendo y hacerlas esperar hasta que estén en la caché deterioraría el performance del pipeline. Por esta razón, se utiliza un buffer de escrituras pendientes (**pending store buffer** o **store forwading buffer**, SFB) que permite que los resultados que todavía no fueron escritos en cache sean utilizados por ellas. Este proceso es llamado **store-to-load forwarding**.

Cuando una instrucción necesita un dato, en paralelo con el acceso a caché, se reliza una comparación parcial de la dirección con todas las entradas en el SFB. Si hay algun match entonces se carga la información desde el buffer en vez de usar la caché. Se utiliza una comparación parcial para que la latencia del mecanismo sea la misma que la del lookup de la caché.

Mas adelante en el pipeline, un **Memory Ordering Buffer** debe asegurar que el dato forwardeado provenga de la escritura más reciente. En caso que el forwarding sea incorrecto, entonces la lectura debe ser ejecutada otra vez después de que todos los writes de los que depende hayan sido guardados en caché.

El forwarding puede ser realizado solo cuando se cumplen las siguientes condiciones:

- El load utiliza la misma dirección que haya sido completado pero todavía no commiteado en caché (osea que se encuentre en store forwading buffer).
- El dato pedido deber ser del mismo tamaño o más chico que el del store pendiente e iniciar en la misma dirección.

Si la información pedida por el load se superpone parcialmente con la información del store o necesita hacer uso de más de dos stores pendientes, entonces el forwarding no se permite y la instrucción debe conseguir su data desde la caché luego de que las operaciones pendientes sean commiteadas.

FP/SSE Execution Units

Es donde se ejecutan las operacion de punto flotante, MMX, SSE y SSE2.

8.4. Memory Subsysten

El sistema de memoria, compuesto por una caché L2 y el bus del sistema.

Level 2 Instruction and Data Cache

Contiene las instrucciones que hacen miss en la Trace Cache y la información que produce un miss en la cache L1. Es una caché asociativa de 8 vías y usa un algoritmo write-back para escribir las modificaciones en memoria.

A esta caché se asocia un prefetcher que monitorea los patrones de acceso a memoria y recuerda la historia de los misses para detectar flujos de datos concurrentes y rellenar la caché con información que podría ser utilizada más adelante por el programa. Además, intenta minimizar el prefetching de datos que podrían causar una sobrecarga de la memoria del sistema y provocar el delay de los accesos que el programa realmente necesita.

400Mhz System Bus

Tiene un ancho de banda de 3.2 Gbytes por segundo, esto permite que las aplicaciones realizen data streaming desde la memoria. Tiene un protocolo que divide las transacciones y un pipeline profundo para permitir que el subsistema de memoria atienda varios accesos/escrituras simultáneamente.

9. Thread Level Paralelism (Procesadores Xenon [2])

Tradicionalmente, para mejorar el perfomance, los diseñadores se enfocaban en desarrollar relojes de alta velocidad, ejecución paralela de instrucciones y caches. Sin embargo, en los últimos años, con el desarrollo del internet, se ha visto la necesidad de ejecutar varias tareas simultáneamente. Por esta razón, se han comenzado a desarrollar distintas técnicas de paralelismo a nivel thread (procesos):

- **Chip multiprocessing:** Se utilizan dos procesadores. Cada uno de ellos tiene a su disposición un conjunto completo de recursos de ejecucion y arquitecturales. Pueden o no compartir una caché de primer nivel. Tiene como desventaja que el chip resultante es mas grande y consume más que un chip single-core.
- **Time-slice multithreading:** Un solo procesador ejecuta múltiples threads switcheando entre ellos. El switching se haces después de un determinado período de tiempo o cuando se produce un evento de larga latencia (operaciones que tomen demasiados ciclos de reloj, como lecturas de memoria). La primer opción genera demasiado tiempo muerto que podría ser aprovechado para ejecución de instrucciones, la segunda sirve para aplicaciones de servidores que tienen un gran número de caché misses y ambos threads ejecutan tareas similares. Ninguna de ellas logra superposición optima cuando varias fuentes usan ineficientemente los recursos (branch mispredictions, instrucciones dependientes, etc).
- **Simultaneous multi-threading:** Varios threads pueden ser ejecutados simultaneamente en un solo procesador sin necesidad del switching. Este método es el que mejor aprovecha los recursos del procesador y es el que implementa la arquitectura de Hyper-Threading de Intel.

9.1. Hyper-threading technology architecture

La tecnología de hyper-threading hace que un procesador físico aparezcan como múltiples procesadores lógicos en la arquitectura. Osea que los programas y los sistemas operativos usarán los procesadores lógicos como si fuesen físicos, luego la microarquitectura se encargará de realizar la distribución de recursos necesarias entre los procesadores físicos.

En chips normales, cada procesador tiene una copia completa del estado de las arquitecturas (registros de proposito general, de control, de interrupciones y algunos de estado de la maquina).

En hyper-threading, cada procesador físico tiene una copia completas de este estado y un controlador de instrucciones programables (APIC) por cada procesador lógico que contenga. Las cachés, unidades de ejecución, branch predictors, lógicas de control y buses son compartidas por todos ellos.

9.1.1. Front-End

Execution Trace Cache (TC): Por cada procesador lógico, la caché tiene un set de next-instruction-pointers que trackean independientemente el progreso de alguno de los dos threads

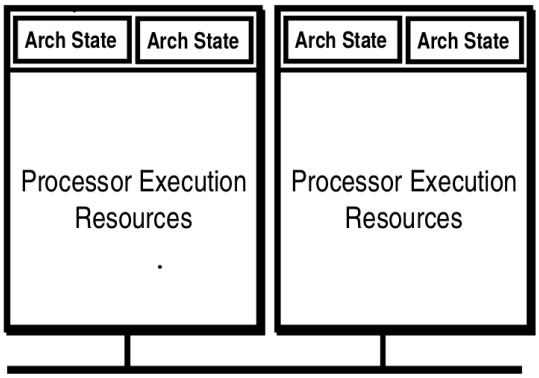

Figura 16: Procesador con tecnología de hyper-threading

ejecutandose. Si ambos necesitan accederla al mismo tiempo, el acceso se otorga a uno de ellos y luego al otro en ciclos de reloj alternados.

Si uno de los procesadores lógicos está bloqueado (*stalled*) o es incapaz de usar la TC, el otro puede usar la caché completamente (cada ciclo de reloj).

Cada entrada de la caché tiene un tag que indica el thread al que pertenece y es reservada dinámicamente según sea necesario.

Microcode ROM: Se usan dos punteros para contolar independientemente los flujos si los dos procesadores lógicos están ejecutando instrucciones complejas. Comparten las entradas de la ROM. El acceso es alternado de la misma manera que en la trace caché.

ITLB y Branch Prediction: Cada procesador lógico tiene su propia ITLB y su propio conjunto de punteros a instrucciones, esto permite trackear el progreso de fetch para cada uno de ellos por separado.

La lógica de fetch a cargo de hacer requests a las caché L2, los arbitra en orden first-come first-served manteniendo reservado un espacio para cada procesador en la cola. De esta manera, ambos procesadores lógicos pueden tener fetches pendientes simultáneamente.

Una vez fetcheadas, cada procesador lógico tiene su propio conjunto de buffers para retener las instrucciones que deben ser decodificadas.

El *return stack buffer*, que predice la dirección a la que debe retornar una instrucción, es duplicado ya que las predicciones son mejores si se hacen independientemente para cada thread.

Cada procesador lógico tiene un *branch history buffer* local y los dos comparten un buffer global en el que, a cada entrada, se agrega un tag con el ID del procesador al que corresponde.

IA-32 Instruction Decode: Se comparte este componente. Cuando los dos threads tiene que decodificar instrucciones simultáneamente, la lógica de decodificación alterna el uso de los buffers. En general, varias instrucciones de un mismo procesador son decodificadas antes de pasar al otro.

A pesar de decodificar las instrucciones de cada procesador lógico por separado, se mantiene una copia de todos los estados necesitados para decodificarlas por cada uno de ellos.

Si solo un procesador necesita usar el decodificador, se le asigna el uso completamente a éste. Las instrucciones decodificadas se escriben en la Trace Caché y son forwardeadas a la cola de micro-operaciones.

Cola de micro operaciones (uop queue): Está particionada para que cada procesador lógico tenga la mitad de las entradas.

9.1.2. Out-of-order execution engine

Allocator: Cada procesador lógico solo puede usar la mitad de entradas de los buffers principales (de reordenamiento, tracing y secuencialización).

- Si ambos procesadores tienen micro-operaciones en la uop queue, el allocator las atendera alternadamente en cada ciclo de reloj.
- Si uno de ellos ha alcanzado el límites de recursos que puede usar, le manda una señal para bloquearlo *stall* hasta que libere alguno.
- Si la uop queue solo contiene micro-operaciones de uno de los procesadores, el allocator lo atenderá en cada ciclo (manteniendo el limite de asignación de recursos).

La límitación de recursos ayuda a la máquina, a reforzar **fairness** (uso justo) y previene deadlocks.

Register Rename Dado que cada procesador lógico debe mantener y traquear su propio estado de arquitectura, hay un register alias table por cada uno. El proceso de renombre se realiza en paralelo a la asignación de recursos (mientras el allocator está trabajando) por lo que la lógica de renombre trabaja sobre las mismas micro-operaciones que el allocator.

Una vez que los dos componentes mencionados terminan de procesar una operación, ésta es ubicada en la cola correspondiente (general o de memoria) a la espera del scheduler.

Instruction Scheduling: La cola general y la de instrucciones de memoria les envian micro-operaciones tan rapido como puedan alternando las instrucciones de ambos procesadores.

Cada scheduler tiene su propia cola de scheduling. Para asegurar fairness, cada procesador lógico puede tener un limitado número de entradas activas en éstas (este limite es independiente de su tamaño). A parte de eso, al momento de evaluar las micro-operaciones, el scheduler solo se basa en la dependencia de inputs y la disponibilidad de recursos para elegirlas.

Execution units Dado que los registros lógicos fueron renombrados a registros físicos en un achivo físico de registros compartidos, para ejecutar las microoperaciones solo es necesario acceder a éste. Por lo que las unidades de ejecución funcionan igual que antes.

Después de la ejecución, las micro-operaciones son enviadas al buffer de reordenamiento que está particionado de tal manera que cada procesador lógico solo pueda la mitad de sus entradas.

Retirement: Cuando hay micro-operaciónes de ambos procesadores lógicos que deben ser retiradas, se retiran las instrucciones en orden del programa altarnedamente.

Si un procesador lógico no está listo para retirar ninguna instrucción, entonces se atiende completamente al otro procesador..

Una vez que los almacenamientos han sido retirados, deben ser escritos en la caché L1 del procesador físico. Para esto, se commitea en caché atendiendo alternadamente a cada procesador lógico.

9.1.3. Subsistema de memoria

Direction Tranlation Lookaside Buffer (DTLB): Traduce direcciones a direcciones físicas. Es compartido. Cada entrada incluye el ID del procesador lógico a la que está asignada. Además, cada uno de ellos, tienen reservado un registro para asegurar fairness y progreso al procesar los DTLB misses.

L1, L2 y L3 Data Caches: Son cachés asociativas con líneas de 128-bytes. Los dos procesadores lógicos comparten todas sus entradas.

Las caches son ajenas a la lógica de multithreading. Esto puede llegar a potenciales conflictos lo que podría resultar en un deterioro del performance. Sin embargo, tambien existe la posibilidad de que uno de los procesadores pida información que el otro quiera usar. En estos casos, el performance se ve benificiado.

Bus: Incluye al controlador de interrupciones programables (APIC) local. La cola de requests y los buffer de espacios son compartidos. Los request al APIC local y los recursos de envío de interrupciones son únicos para cada procesador lógico. Las requests de cada uno de ellos es distinguida con un tag pero no se les da ningun tipo de prioridad.

9.2. Modos Single-Task y Multi-Task

- **Multi-Task Mode (MT):** Los dos procesadores lógicos se encuentran activos y los recursos compartidos se dividen como fue descrito anteriormente.
- **Single-Task Mode (ST):** Solo un procesador lógico está activo y los recursos que habían sido particionados en modo multi task son completamentes asignados a éste.

La arquitectura IA-32 tiene una instrucción llamada HALT que para la ejecución de un procesador y le permite entra en modo de bajo consumo. Ésta es una instrucción privilegiada, es decir, solo pueden ejecutarla el sistema operativos y procesos de privilegio 0.

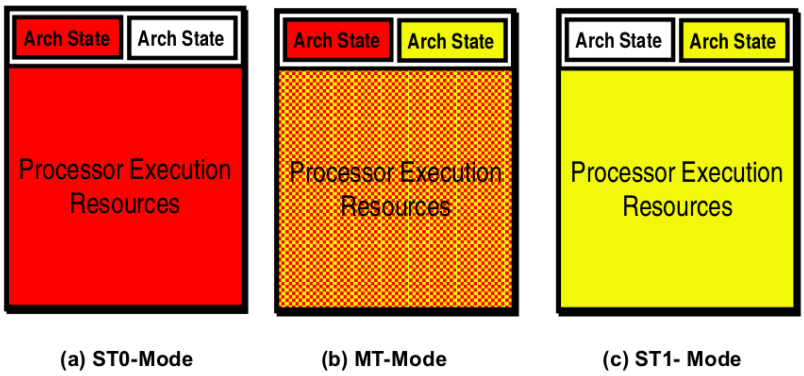

Figura 17: Asignación de recursos para los distintos modos

En un procesador con tecnología de hyper-threading, HALT transiciona de modo MT a ST0 o ST1 dependiendo de en que procesador lógico se halla ejecutado la instrucción. Si fue ejecutada por el procesador 0, entonces solo queda activo el 1 y viceversa.

Si se envía una interrupción a un procesador halteado entonces se vuelve a modo MT.

9.3. Sistemas operativos y aplicaciones

El sistema operativo se debe encargar de implementar dos optimizaciones para obtener el mejor rendimiento posible.

La primera es usar la instrucción HALT si uno de los procesadores lógicos está activo y el otro no. Un sistema operativo que no use esta optimización hará que el procesador que no se encuentra en uso entre en un idle-loop durante el cual se ejecutaran instrucciones para chequear si hay trabajo para hacer. Este loop puede llegar a consumir una cantidad significante de recursos de ejecución que podrían ser usados por el procesador lógico activo.

La segunda optimización es que, al momento de realizar el scheduling de threads, se los debería mandar a procesadores físicos distintos. Cuando esto ya no sea posible, recién ahí comenzar a usar los procesadores lógicos.

10. Intel Enhanced SpeedStep Technology (Pentium M [3])

Diseñar un procesador *Mobile* requiere considerar distintso tradeoffs de consumo/performance que no son necesarios en los procesadores tradicionales.

- **Potencia, consumo de energía y temperatura**: Para mantener los transitores en un rango de temperatura de trabajo aceptable, el calor generado tiene que ser disipado de una manera costo-efectiva. Este problema aplica tanto a Mobile como a Desktop, sin embargo, en Mobile, el tamaño y peso de los dispositivos decrementan considerablemente el pico de consumo energético del procesador.
- **Vida de la bateria:** Mientras más alta sea la potencia promedio usada menos es el tiempo durante el que puede operar una batería. Esto limita el consumo de potencia promedio del procesador.

10.1. Power-Awareness Philosophy and strategies

Se intenta:

- Conseguir un buen balance entre rendimiento y duración de la batería.
- Intercambiar rendimiento por energía y viceversa, cuando sea necesario.

Más rendimiento vs Mayor duración de la bateria: Respecto a esto no hicieron nada porque el procesador ya producía menos del 10 % del consumo de energía y no iban a lograr ninguna mejora decente.

10.1.1. Intercambio entre rendimiento y energía

Hay veces en las que implementar un feature que permita ganar performance o ahorrar energía es mejor que simplemente escalar el voltaje/frecuencia de trabajo (método que se usaba en procesadores anteriores). La lógica y componentes necesarios para implementarlas aumenta el consumo de energía pero la ganancia (en tiempo) que generan superan este costo.

Algunas de las features que se agregaron fueron:

- **Reducir el número de instrucciones por tarea:** Se utilizan mejores predictores para disminuir el número de instrucciones mal especuladas lo que reduce el número total de instrucciones procesadas.
- **Reducir el número de micro-ops por instrucción:** Atender y ejecutar cada instrucción consume energía. Eliminando micro-operaciones del flujo de ejecución o combinando varias de ellas se reduce el consumo total.
- **Reducir el número de switches entre transistores por micro-op:** Procesadores de alto rendimiento proveen un alto nivel de paralelismo de instrucciones provocando demasiados switches. En algunos casos, se pueden realizar optimizaciones locales (por ejemplo,

activar solo la parte que se va a usar de un componente). En otros, se agregan componentes que deciden si una unidad va a estar activa o no en el próximo ciclo y las apaga en caso de que ocurra lo segundo.

- **Reducir la cantidad de energía por switch de transistor:** Depende de la cantidad de transistores usados, su tipo, el voltaje de operacion y temperatura. Se utilizan dispositivos de descarga lenta (*low-leakage*) y la tecnología Intel SpeedStep para disminuir los switches y, por lo tanto, el consumo energético.

10.2. Branch Prediction avanzado

Se utiliza como base el predictor del Pentium 4. Se le agregan dos componentes adicionales: Un detector de Loops y un predictor de branchs indirectos.

Detector de Loops: Analiza branches para ver si se comportan como loops. Esto es, si se mueven en una dirección (taken o not-taken) un número fijo de veces intercalando con único movimiento en la dirección opuesta. Cuando este comportamiento es detectado, se crea un conjunto de contadores que permite predecir con toda precisión el comportamiento del programa para iteraciones largas.

Indirect Branchs: Son generadas por instrucciones de control que, en vez de especificar la dirección a la que hay que saltar, indican donde está guardada esa dirección. Lenguajes como C++ y Java los usan para los switch case.

Indirect Branch Predictor: Predice basandose en el history buffer. Los targets siempre están ubicados en la tabla de punteros a instrucciones junto con el tipo de branch que las usa. Cuando ocurre una predicción errónea en un indirect branch, el predictor almacena una nueva entrada en la tabla de punteros que corresponde a la entrada del history buffer que apunta a esta instrucción.

Cada branch indirecto puede guardar tantas targets como sea necesario para diferentes patrones de historia global. Esto permite predecir correctamente usando solo la tablas de punteros.

10.3. Fusión de micro-operaciones

Una micro-operacion consiste en una única operación y dos operandos fuente. Algunas instrucciones deben ser decodificadas en mas de una micro-operación porque tienen que usar mas de dos operandos, o porque requieren completar una secuencia de operaciones no relacionadas. Esto tiene algunas desventajas:

- El gran número de micro-operaciones crea presión en los recursos con ancho de banda o capacidad limitada.
- El proceso de decodificación es más complejo que para aquellas instrucciones que se traducen en una única micro-operación.

- Enviar más micro-operaciones al sistema incrementa la energía requerida para completar una secuencia de instrucciones dada.

Con el mecanismo de fusión, el decoder puede fusionar dos micro-operaciones en una. A la micro-operación resultante se le asigna una única entrada en el Reorder Buffer y en el Reservation Station (RS) (cada una de sus entrada puede acomodar tres operandos fuente para soportar este nuevo tipo de operacion).

El dispatcher trata cada una de ellas como si ocupasen la entrada completa. Ambas micro-operaciones pueden ser emitidas a sus respectivas unidades de ejecución en paralelo. Recién cuando las dos se ejecutaron completamente, pueden ser retiradas como una micro-operacion fusionada.

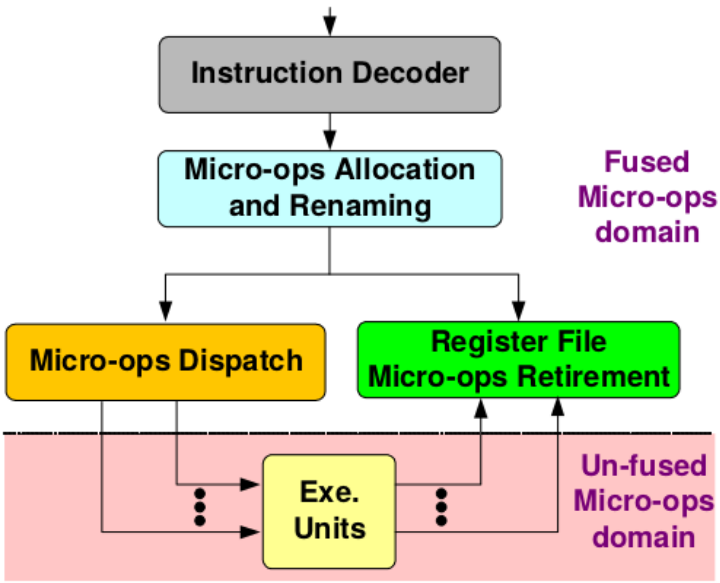

Figura 18: Dominios de fusión de micro-operaciones

10.4. Dedicated Stack Engine

La ISA provee las instrucciones PUSH, POP, CALL y RET que controlan el comportamiento del stack. Además posee el registro ESP (Stack Pointer Register) que es modificado por cada una de estas instrucciones.

En el Pentium M, este efecto es implementado usando lógica dedicada cerca de los decodificadores. La idea es representar el punto de vista del programador (ESP_P) en cualquier momento usando un ESP historico (ESP_O, que se encuentre en el motor fuera de orden) y un delta (ESP_D, offset mantenido en el front-end).

$$ESP_P = ESP_O + ESP_D$$

Cuando una secuencia de pushs o pops es encontrada en el flujo de instrucciones, el hardware dedicado al stack ejecuta las modificaciones en el decoder y actualiza el ESP_D además de modificar el parámetro de dirección para cada una de las micro-operaciones que referencien el stack. De esta manera, la unidad de generación de direcciones puede calcular la locación de memoria correcta.

Los beneficios de esto son:

- Las dependencias respecto del ESP son eliminidas dado que ESP_O, usado por el scheduler del motor fuera-de-orden, no cambia durante la secuencia de operaciones de stack. Esto nos da la oportunidad de paralizar aún más la ejecución.
- ESP_D se actualiza usando sumadores especialmente dedicados lo que libera a las unidades de ejecución general para trabajar en otras micro-ops.
- Actualizar el ESP_D en el front end elimina las micro-ops de actualización de ESP del motor fuera-de-orden. Esto permite ahorrar energía ya que hay menos micro-ops para ejecutar además de que los sumadores dejan de ser usados para operaciones pequeñas.

Dado que el motor dedicado del stack vive al principio del pipeline, todos sus calculos son especulativos. Para recuperar un estado preciso en cualquier momento, los valores de ESP_0 y ESP_D deben ser recuperados para todas las instrucciones en la maquina.

El motor fuera-de-orden mantiene el ESP_0 como cualquier otro registro de proposito general y una tabla para guardar el ESP_D por cada instrucción en el pipeline.

Cuando el valor arquitectural (ESP_P) se necesita dentro del motor, la lógica de decodificación automáticamente inserta una micro-operación que lo calcula. Cuando esto sucede, el ESP_D es reseteado.

10.5. El bus del Pentium M

Es un bus con una cola in-order de 8 transacciones. Fue diseñado para sistemas mobile y ambientes uni-procesador.

Sense amplifiers: Son componentes electrónicos que amplifican las oscilaciones del voltaje para que los valores lógicos (0 o 1) sean adecuadamente reconocibles.

Cuando el bus está en modo idle, ahorra energía agresivamente desactivando todos sus *sense amplifiers* por lo que no consumen nada. El procesador debe enviarle una señal (**DPWR#** al bus de datos y **BPRI** al de direcciones) para que éste los active y puedan ser usados. Después de atender todas las transacciones necesarias, vuelve automáticamente a modo idle.

Low Vtt: Señal para reducir el consumo del bus. Como esto puede conllevar a ciertos problemas de comunicación se implementa un método de compensación para ajustar dinámicamente la potencia de uso durante la ejecución.

PSI: Señal para reducir el consumo de toda la plataforma.

10.6. Optimizaciones de bajo nivel

Otra forma de optimizar consumo, es apagando los clocks o desactivando partes especificas de un componente.

Algunos componentes son capaces de darse cuenta cuando deben entrar en estado idle agregando un poco de lógica, otros no.

Para el segundo tipo se crearon controles que pueden identificar o predecir sus periodos idle y ordenar, a la unidad, que reduzca su consumo. La lógica de predicción usada por estos controles debe permitir que el componente se active completamente sin niguna penalidad de performance.

10.7. Tecnología Enhanced Intel Speedstep

En generaciones anteriores de procesadores, esta tecnología cambiaba el modo de operacion del procesador (Lowest Frequency Mode o Highest Frequency Mode) bloqueando su uso durante la transición. Está transición consiste en ir reduciendo o aumentando en pequeños pasos la frecuencia y el voltaje.

La versión Enhanced trata de abordar los siguientes desafíos:

- **Minimizar indisponibilidad del sistema y el procesador:** Existen ciertas limitaciones físicas del sistema de suministro de energía durante la transición del voltaje que se traduce en un delay en estos componentes (quedan inhabilitados durante este tiempo).
- Se debe prevenir la perdida de eventos del sistema como interrupciones y snoops que fueron bloqueados durante la transición.

Para resolver el problema se utilizan tres métodos:

- **Cambio por separado del voltaje y la frecuencia:** Se hace en dos etapas. El voltaje se cambia en peqeños pasos incrementales, previniendo ruido en los clocks y permitiendo que el procesador siga ejecutando durante esta transición.
- **Particionamiento del clock y recuperación:** Se para el clock del core del procesador y el Phase-Locked-Loop (lo que se encarga de transmitir la señal del clock). La lógica de interrupciones y secuenciamiento utilizan el clock del bus (que permanece activo).
- **Bloqueo de eventos:**No deben perderse interrupciones, eventos de pin ni pedidos de snoop enviados durante la etapa de transición de frecuencia.

  La lógica del SpeedStep samplea todo los eventos de pin cuando el clock del núcleo está parado y son re-enviados al procesador una vez que está disponible.

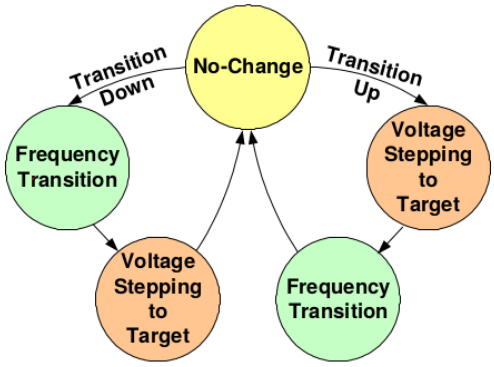

Figura 19: Secuencia de transicion de SpeedStep

Los eventos del bus (interrupciones y snoops) son bloqueados usando un protocolo nativo de estos dispotivos que los capturan durante el período de transición de frecuencia.

11. Mejoras realizadas al Pentium 4 con la técnología de 90nm [4]

11.1. Store-To-Forwarding Enhancement

Previamente, el forwarding era cancelado cuando las direcciones estaban desalineadas o se forwardeaba un dato incorrecto debido a la comparación parcial de direcicones (Sección 8.3).

False forwarding: Ocurre cuando el SFB detecta un match de dirección parcial entre un load y un store, pero sus direcciónes completas no matchean.

Force Forwarding: Permite que el Memory Ordering Buffer (MOB) tome control sobre la lógica de comparación parcial del Store Forwading Buffer (SFB).

Cuando se necesita hacer un load, el MOB determina su "verdadera" dependencia y debe decidir si la decisión del SFB es correcta o si la instrucción debe ser ejecutada otra vez. Si el chequeo de dependencia falla entonces puede corregir la logica del buffer de dos maneras:

- Forzando el forwardeo de la entrada correcta y rotando la data si es necesario (en caso de que el dato se encuentre desalineado).
- Desactivando el forwarding si la dependencia no se encuentra en el SFB.

11.2. Front End

11.2.1. Branch Predictor

Previamente, cuando un branch no se encontraba en el branch target buffer (BTB), si era una rama que saltaba hacia atras (Backward branch) se tomaba como *taken* y si realizaba un salto hacia adelante (forward branch) no (Seccion 8.1). Si bien este algoritmo nos permite predecir correctamente cuando el backwards branch es de un loop, no siempre pertenece a uno.

Algunas mejoras en el predictor estático:

- Existe un threshold para la distancia entre un backwards branch y su target: Si el BTB no tiene ninguna predicción para una rama de este tipo y la distancia a su target es menor al threshold, entonce predice *taken* sino predice *not-taken*.
- Hay branches con ciertas condiciones que, muchas veces, no son tomadas sin importar las dirección ni la distancia. Como estas condiciones no son comunes en loops, se predicen *not-taken*.

Al predictor dinámico, se le agrego el Indirect Branch Predictor implementado para el Pentium M para reducir predicciones erróneas.

11.2.2. Detección de dependencias extras

En vez de mover un cero a un registro, una técnica común para resetarlo, es usar la instrucción `xor` consigo mismo. Esta técnica es preferida porque el código escrito es mucho más pequeño.

En una máquina fuera-de-orden, esta extra dependencia puede resultar en perdida de rendimiento. Previos procesadores remueven esta dependencia en el registro fuente dado que el resultado será siempre el mismo sin importar que valor se encuentre en él. Como se agregaron nuevas instrucciones (SSE2) que pueden ser usadas de la misma manera, se agregó lógica para que también sean reconocidas.

11.2.3. Trace Caché

Se aumentó su tamaño y puede codificar mas tipos de micro-ops que en previos procesadores. Si una instrucción usa una micro-ops que no puede ser codificada, entonces debe ser secuenciada en la Microcode ROM. Esta mejora permite un mayo ancho de banda de micro-operaciones desde el front end hacia el motor de ejecución fuera de orden ya que remueve varias transciciones al ROM.

11.2.4. Unidades de ejecución

- Se agrega un shifter a una de las ALUs. Esto permite ejecutar estas operaciones sin necesidad de utilizar recursos que deberían estar destinados a instrucciones complejas de enteros (como se hacia en previos procesadores).
- Previamente, las multiplicaciones de enteros se ejecutaban en los multiplicadores de puntos flotantes. Esto generaba latencia porque se debían mover los operandos a la zona de punto flotante y después devolverlos a la de enteros. Ahora, hay un multiplicador de enteros especializados que evita esto.
- El tamaño de la caché L1 aumenta considerablemente su tamaño y el tamaño de la comparación de direcciónes parciales. Esto reduce el número de casos de falso aliasing.
- Los schedulers dedicados a instrucciones x87/SSE/SSE2/SSE3 aumentan su tamaño, lo que permite que aumente la ventana de oportunidad para encontrar paralelismo en algoritmos multimedia.
- Se aumenta el tamaño de todas las colas que alimentan a los schedulers por lo que se pueden bufferear más micro-operaciones antes de que el allocator entre en modo *stall*.
- Se agrega un predictor que indica si es muy probable que una micro-op de lectura reciba data forwardeada y, si es así, desde que entrada. Dada esta información, el scheduler de loads la retiene hasta que la instrucción de almacenamiento de la que depende sea puesta en la cola de ejecución. De esta forma, se reduce la latencia introducida cuando se debe re-ejecutar una instrucción de lectura fallida.

11.3. Sistema de memoria

Cachés: Se incrementan el tamaño de las cachés para reducir el tiempo que se gasta esperando a que la DRAM devuelva información.

Page Table Walk: Se recorre la tabla de paginación del sistema para ver si existe un mapeo de alguna dirección especifica.

Software Prefetching: El programador puede insertar instrucciones de prefetch para traer data a la cache antes de lo necesario. En las versiones previas, estas instrucciones traian datos desde la DRAM hasta la cache L2 para no contaminar la cache L1.

Ahora, se agrega un mecanimos que permite a la instrucción iniciar un page table walk y rellenar al data translation lookaside buffer en caso de que el prefetch se haga sobre una página no cacheada.

Hardware Prefetching: Un dispositivo detecta flujo de datos e intenta predecir que información va a ser usada a continuación. Este método es superior al software prefecthing ya que no requiere ningún esfuerzo por parte del programador y puede mejorar el rendimiento en código que no tiene instrucciones de prefetch.

Buses: Se ensanchan los buses y se agregan buffers lo que permite tener una mayor cantidad de lecturas y escrituras simultáneas.

11.4. Hyper-Threading Technology

Se resuelven dos cuellos de botellas, que antes no habían sido tenidos en cuenta porque procesadores single-threaded no generaban problemas:

- Antes, el procesador solo podía realizar un único page walk o atender un acceso de memoria. En multi-threading esto puede generar un problema por lo que ahora pueden ocurrir simultáneamente una o más veces cada operación.
- El esquema de indexación parcial usado para la cache L1 genera conflictos cuando el patrón de acceso de cada procesador lógico matchean al mismo tag parcial incluso si están accediendo a regiones separadas de la memoria física. Esto causa contención en la caché, lo que reduce el hit rate.

  Para disminuir la probabilidad de que esto ocurra, por cada procesador lógico, se agrega al tag parcial un bit de contexto. Este bit es seteado o reseteado dinámicamente dependiendo de la inicilización de la estructura de la tabla de paginación de cada procesador.

  Si dos procesadores lógicos intentan compartir el mismo directorio base en la memoria física, el bit de contexto de cada uno de ellos es seteado al mismo valor. Esto les permite compartir las entradas de la caché L1.

Si las direcciones bases del directorio de páginas es diferente entonces es probable que estén trabajando en regiones físicamente separadas. En este caso, los bits de contexto se setean en valores distintos y la cache no se comparte.

12. Chip Multi-Processor (Intel Core Duo [5])

La tecnología core multi-procesor, a diferencia de predecesores, incluye dos multiprocesadores físicos dentro de un mismo núcleo.

La tecnología Intel Core Duo se implementó con el objetivo de crear un procesador que sea adecuado para todas las plataformas mobiles (celulares, notebooks, datases, impresoras, etc). se basa en dos microprocesadores Pentium M mejorados que fueron integrados y comparten un caché L2. Con el fin de satisfacer las demandas de rendimiento y consumo se tomaron las siguientes decisiones:

12.1. Mejoras de rendimiento implementadas

- Mientras que la microarquitectura SSE es de 128 bits de ancho, la del Pentium M es de 64. En estos procesadores, cada instrucción SSE se parte en pares de micro-operaciones de 64 bits lo que implica un cuello de botella en el front end del pipeline durante la etapa de decodificación.

  Una máquina más ancha produciría mas calor e impactaría significativamente en el diseño térmico del sistema y la duración de la batería. Dado que el Pentium M fue diseñado principalmente para mobilidad, se decidió mantener la arquitectura del mismo tamaño y tratar de diluir el problema. Para esto, se introdujo un nuevo mecanismo que permite *laminar*[1] pares de micro-operaciones similares. En cierto punto del pipeline, el flujo de micro-operaciones laminadas es delaminado para ejecutar las micro-operaciones originales.

  Este cambio no solo mejora el rendimiento de las operaciones vectoriales sino que ahorran energía dado que puede ser desactivado cuando el buffer se llena hasta cierto punto.

- Otro cuello de botella descubierto, fue en el Floating Point Control Word (FPCW, un componente que controla la precisión de las operaciones de punto flotante). Antes, se consideraba "constante" durante la ejecución de un programa, pero en algunos programas se le cambiaba el modo antes de realizar ciertas operaciones.

  Cada una de estas manipulaciones provoca un *stall* en el pipeline. Por esta razón, se introdujo un nuevo mecanismo de renombre para que 4 versiones distintas de este registro puedan coexistir durante la ejecución sin interrumpir el pipeline.

- Antes, la división se realizaba iterando un número máximo (fijo) de iteraciones, lo que producía largas latencias. En esta versión del procesador, se agregó lógica que calcula el número de iteraciones requeridos para lograr la operación. De esta forma, una vez ejecutada la cantidad de iteraciones requeridas, la división devuelve el resultado.

- Se agrega un prefetcher de instrucciones a la caché L2 (Ver sección 11.3)

[1] El paper no dice que significa laminar pero da a entender que es algo parecido a la fusión

12.2. Estructura general del procesador

Los procesadores Intel Core Duo utilizan un arquitectura de caché compartida para maximizar el rendimiento de las aplicaciones single y multi-threaded.

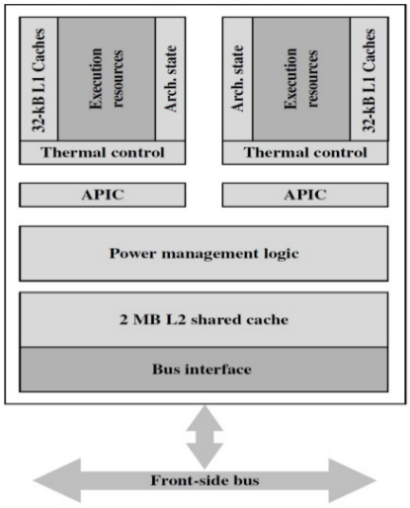

Figura 20: Estructura general del Intel Core Duo

- Cada core tiene su propio APIC. Se presentan al sistema operativo como dos procesadores lógicos separadoss.
- Cada core tiene su propia unidad de control de temperatura.
- El sistema combina control de consumo por cada core y para todo el chip en general.

12.3. Control de consumo energético

Para ahorrar perdida de energía (cosumo estático), se utilizan dos técnicas: Enhanced Sleep State Control y Dynamic Intel Smart Cache Sizing. Para controlar consumo de energía activa: Se utiliza una técnica basada en Intel SpeedStep technology (Sección 10.7).

(Bueno, estoy apurado y necesito terminar antes del martes y ponerme a repasar. La traduccion y tratar de que quede bien fue)

12.3.1. Leakage power consumption

The challenge of adding a second core on die while improving the overall power-consumption demands an improvement to the power states of the system in order to avoid power being wasted whenever a core is not active. We face two main problems: (1) since only a single power plane is used, it forces us to run all cores with the same voltage and frequency, and (2) the chip-set and

the OS see both cores as a single entity that has the same state at the same time. Thus, the Intel Core Duo processor presents two separate views on the power state of the system, internally we manage the states of each core independently (per-core power state) ande externally we view the system as having a single, synchronized power state.

There are five posible states:

- **C0 - Active (per-core):** Power state is assumed to be in running mode.
- **C1 - Auto halt (per-core):** When the core has nothing to do, the OS issues a halt command that shut off the core clock.
- **C2 - Stop Clock (per-core):** Core and bus clocks are off
- **C3 - Deep sleep (per-core):** Clock generator if off;
- **C4 - Deeper Sleep:** Whole system voltage is reduced. Since the cores are connected to the same power plane, this must be done in coordination between the two cores.
- **DC4 - Deeper C4:** Further reduce voltage.

While being in a sleep state, the system still consumes static power (leakage). In Intel Core Duo Technology, we implement an advanced algorithm that tries to anticipate the effective cache memory footprint that the system need when moving from a deep sleep state to an active mode. The new mechanism keeps only the minimum cache memory size needed active, and it uses special circuit techniques to keep de rest of the cache memory in a state that consumes only a minimal amount of leakage power.

12.3.2. Active power consumption

In order to control the active power consumption, Intel Core Duo Technology uses Intel SpeedStep technology. When a set of working points is defined, each one has a different frequency and voltage and so different power consumption. The system can define at what working point it works in order to strike balance between the performance needs and the dynamic power consumption. This is usually done via the OS, using the ACPIs.

12.4. Thermal Design Point

In the previous Pentium M processor, a single analog thermal diode was used to measure die temperature. Thermal diode cannot be located at the hottest spot of the die and therefore some offset was applied to keep the CPU within specifications. For these system it was sufficient, since the die had a single hot spot. In the Intel Core Duo processor, there are several hot spots that change position as a function of the combined workload of both cores.

The use of a digital thermometer allows tighter thermal control functions, allowing higher performance in the same form factor. The improved capability also allows us to achieve better ergonomic systems that do not get to hot, can operate more quietly, and are more reliable.

The thermal measurement function provides interfaces to power-management software such as the industry standard ACPI. Each core can be defined as an independient thermal zone, or a single thermal zone for the entire chip. The maximum temperature for each thermal zone is reported separately via dedicated registers that can be polled by the software.

Intel Core Duo technology implemented a dual-core power monitor capability. It continuously tracks the die temperature. If the temperature reaches de maximum allowed value, a throttle mechanism is initiated. Throttling starts with the more efficient dynamic voltage scaling policy and if not sufficient, the power monitor algorithm continues lowering the frequency. If an extreme cooling malfunctions occurs, an Out of Spec notification will be initiated, requesting a controlled complete shutdown.

12.5. Platfrom Power Managment

Inter Core Duo processor technology closely interacts with other components on the platform. One such component is the Voltage Regulator (VR).

The CPU has internal knowledge of the activity demand and it communicates a request to go to higher power early enough for the VR to get ready for the increased demand.

12.6. Intel Core Solo Processor

In order to fit into very limited thermal constraints and power consumption, the Intel Core Duo processor has a derivative that contains a single core only. This can be achieved:

- **by user or OS decision:** The system will keep the second core idle at CC4 state. Each time an interrupt is received or a broadcast IPI is sent, this core may need to wake up and go immediately back to sleep state, consuming small amounts of dynamic power.
- **by a BIOS option:** The system does not recognize the other core and so it is kept in CC4 state all the time, consumming no dynamic power at all.
- **Total disconnection:** The disadvantage of the two methods described above is that the core still consumes static power. In order to avoid this and reduce the power consumption of the core even further, Intel introduces the single-core version of the Inter Core Duo Technology, called Intel Core Solo Processor, which disconnects the non-active core from the power grid.

13. Chip Multi Procesor (Intel Core Duo [6]): Known Performance issues

13.1. MESI Protocol Modification

Intel Core Duo processor implements the same MESI protocol as in all other Pentum M processors.

In order to improve performance, we optimized the protocol for faster communication between the cores, particularly when the data exist in the L2 cache.

When a core issues an **RFO** (Request For Ownership), if the line is shared only by the other cache within CMP die, we can resolve the RFO internally very fast, without going to the external bus at all. Only if the line is shared with another agent on the external bus do we need to issue the RFO externally.

In the case of a multi-package system, the number of coherence messages over the external bus is smaller than in similar DP or MP systems, since much of the communication is being resolved internally.

13.2. Shared vs Split Cache

Recently, different architectures use a split last-level cache in order to achieve a fast time-to-market of a dual-core system. Clear downsides of this solution are as follows:

1. Cache coherent-related events that need to be served over the File System Buss, such as RFO or invalidation signals, greatly impact performance and power.

2. An ST application cannot take full advantage of the entire cache.

The hard partitioned cache may prevent one application from significantly reducing the amount of cache memory available to an application running on the other core.

13.2.1. False Sharing

False sharing happens when two or more threads access different address ranges on the same cache line simultaneously. This causes the cache line to be in the first level cache of the two cores.

False sharing causes a severe performance penalty if one or more threads writes to the shared cache line. This causes invalidation of the cache line at the L1 cache of the other core. As a ressult, the next time that the other core accesses the cache line in question it will have to transfer it from the core that wrote it earlier through the bus, thereby incurring a mayor latency penalty.

13.2.2. Parallelizing code thata use great amount of data

When parallelizing code sections that use data sets exceeding the L2 cache and/or bus bandwith, if one of the threads is using the L2 cache and/or bus, then it is expected to get the maximum possible speedup when the other thread running on the other core does not interrupt

its progress. However, if the two threads use the second-level cache there may be performance degradation if one of the following condition is true:

- Their combined data set is greater than the L2 cache size.
- Their combined bus usage is greater than bus capacity.
- They both have extensive access to the same set in the L2 cache, and at least one of the threads writes to this cache line.

Bibliografía

Introduccion

8. Furfaro, A. *Organización del Computador II - Clases Teóricas* 2019.
9. Mutlu, O. *Introduction to Computer Architecture - Spring Course of Carnegie Mellon University* 2015.

Instruction Level Paralellism

8. Furfaro, A. *Organización del Computador II - Clases Teóricas* 2019.
10. Patterson, D. A. y Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface* en. 5th edition. Cap. 6. Enhancing perfomance with Pipeline Sección 1 y 6 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013).
11. Smith, J. E. y Sohi, G. S. The microarchitecture of superscalar processors. *Proceedings of the IEEE* **83,** 1609-1624. ISSN: 0018-9219 (dic. de 1995).
12. Smith, J. E. y Pleszkun, A. R. Implementation of Precise Interrupts in Pipelined Processors. *SIGARCH Comput. Archit. News* **13,** 36-44. ISSN: 0163-5964 (jun. de 1985).

Jerarquías de memoria

8. Furfaro, A. *Organización del Computador II - Clases Teóricas* 2019.
13. V. Carl Hamacher Zvonko G. Vranesic, S. G. Z. *Computer Organization* en. 5. ed. Cap. 5. The Memory System, Sección 5 (McGraw-Hill, 2002).
14. Culler, D. E., Gupta, A. y Singh, J. P. *Parallel Computer Architecture: A Hardware/Software Approach* en. 1st edition. Cap. 5: Shared Memory Multiprocessors Sección 2 y 4 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997). ISBN: 1558603433.

Procesadores Intel

1. Sager, D., Group, D. P. y Corp, I. The microarchitecture of the pentium 4 processor. *Intel Technology Journal* **1** (2001).
2. Marr, D. *y col.* Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* (feb. de 2002).
3. Gochman, S. *y col.* The Intel Pentium M processor: Microarchitecture and performance (2003).
4. Boggs, D. D. *y col.* The microarchitecture of the intel pentium 4 processor on 90nm technology (2004).

5. Gochman, S., Mendelson, A., Naveh, A. y Rotem, E. Introduction to Intel Core Duo Processor Architecture. *Intel Technology Journal* **10** (ene. de 2006).

6. Mendelson, A. *y col.* CMP Implementation in Systems Based on the Intel Core Duo Processor. **10,** 99-107. ISSN: 1535-766X (mayo de 2006).

7. Baliga, H. *y col.* Improvements in the Intels Coret2 Penryn Processor Family Architecture and Microarchitecture (2008).