Unidad 6 - Microarquitectura Superescalar

1. Ejecución Paralela

Los compiladores y microarquitecturas modernas aprovechan las oportunidades en las que se puede solapar la ejecución de instrucción de instrucciones, haciendo más eficiente la ejecución de un programa.
El **paralelismo a nivel de instrucción** (Instruction Level Paralelism - ILP) es la medida de cuántas operaciones de un programa se pueden ejecutar simultáneamente, mientras que el **paralelismo a nivel de máquina** (Machine Level Paralelism - MLP) es la capacidad del procesador de aprovechar el ILP de un programa, es decir, los recursos necesarios para resolver los problemas generados por las dependencias. Para lograr un alto desempeño **necesitamos tanto ILP como MLP**.

>
> Ejecución Segmentada

Una de las técnicas para incrementar el ILP ya la vimos, la segmentación. Dividimos la ejecución de las instrucciones en tareas pequeñas y desacopladas.

1.1. Dependencias

Las dependencias son propias de los programas y es crítico determinarlas para alcanzar el mayor paralelismo posible, es decir que imponen un límite al mismo. La presencia de una dependencia indica la *posible* aparición de un **riesgo**, aunque su aparición y la consiguiente parada (stall) dependa de la microarquitectura.

- Dependencias de **Datos**: las Read-After-Write, llamadas tambien dependencias *verdaderas*.
- Dependencias de **Nombre**: pueden ser *antiverdaderas* (Write-After-Read) o *de salida* (Write-After-Write). Estas dependencias son fáciles de resolver ya que se pueden eliminar simplemente **usando otros nombre**.

Estas dependencias son fáciles de determinar cuando se dan entre registros, pero no tanto en el caso de direcciones de memoria, ya que se debe conocer las dependencias entre load's y store's para determinar el reordenamiento.

Otras dependencias son

- Dependencias de **Control**: cada instrucción depende de un conjunto de saltos, y esta dependencia debe preservarse para preservar el **orden del programa**. La excepción a la regla es cuando el resultado del programa no se ve afectado. Pero lo importante es preservar el comportamiento de las excepciones y el flujo de datos.

1.2. Políticas de Ejecución

Para mejorar el ILP se deben paralelizar las actividades sin afectar el resultado final. Una forma de hacer esto es **reduciendo** las dependencias de datos y control, y para esto podemos **cambiar el orden** en que las instrucciones se leen (Fetch), se ejecutan (Execute) y se actualizan (Write-back) los resultados, sin cambiar los resultados mismos. Podemos cambiar el orden de las instrucciones bajo las siguientes **condiciones**:

- Preservar la verdadera dependencia (modo de flujo de datos).
- Preservar el comportamiento de las excepciones.
- Encontrar el ILP dentro de una ventana de instrucciones a partir de un predictor de saltos preciso.

Las políticas de ejecución que se pueden implementar son

- **In-Order Issue - In-Order Completion** (lectura en orden, ejecución en orden): las instrucciones se leen, ejecutan y sus resultados se actualizan en el orden en que se ejecutan en el programa
    - No es eficiente bajo dependencias.
    - Propensa a detenciones (stall) y vaciado del pipeline (flush).
- **In-Order Issue - Out-of-Order Completion** (lectura en orden, ejecución fuera de orden): las instrucciones se leen en el orden en que se ejecutan en el programa, pero se ejecutan se actualizan los resultados de acuerdo a la disponibilidad de recursos y de acuerdo a las dependencias existentes.
    - Evita las dependencias de datos.
    - Las instrucciones solo son detenidas (stall).
- **Out-of-Order Issue - Out-of-Order Completion** (lectura y ejecución fuera de orden): las instrucciones son leídas, ejecutadas y los resultados se actualizan según los recursos disponibles y las dependencias existentes. Esta política permite rellenar los "huecos" de tiempo ejecutando instrucciones que ya estén listas (*orden de datos*) y luego **reordena** los resultados de forma que aparenten haber sido ejecutadas de forma normal (*orden de programa*).
    - Desacopla la decodificación de la ejecución.
    - Lee y decodifica hasta que la ventana (?) esté llena.
    - Ejecuta la instrucción cuando la unidad funcional está disponible.

Para implementar estas ideas se utilizan la **planificación estática** y **planificación dinámica**

1.2.1. Planificación Estática

Es en la que el orden de ejecución es determinado por el compilador al analizar el código.

1.2.2. Planificación Dinámica

Es un método de ejecución implementado por hardware que determina qué instrucción ejecutar. El procesador ejecuta las instrucciones fuera de orden de acuerdo a la disponibilidad de los operandos y los recursos.
Con este método también se aprovecha el paralelismo que no es "visible" desde el compilador y son más versátiles ya que no es necesario volver a compilar el código.

- Implementa la política de **Out-of-Order Issue - Out-of-order Completion**.
- Verifica dependencias estructurales y de datos en la etapa de decodificación.

1.2.2.1. Algoritmo de Scoreboard

Este algoritmo es una implementación de la planificación dinámica utilizando *tablas* (scoreboards) para

- Mantener registro de las instrucciones que se **recuperan, emiten y ejecutan**.
- Determinar los recursos (unidades funcionales y operandos) que se **utilizan y necesitan**
- Hacer seguimiento de las instrucciones que **modifican cada registro**.

Con esta información se determina dinámicamente **cuándo y dónde** comienza y termina la ejecución de una instrucción, pasando por cuatro etapas

- **Issue**: comprueba que haya una unidad funcional disponible y no haya riesgo WAW, y en caso de detectar alguno de los dos, detiene la instrucción.
- **Operands Fetch**: verifica la disponibilidad de los operandos de la instrucción, es decir que no lo va a escribir ninguna instrucción emitida en ese momento ni que se está escribiendo en el registro. En ese caso, indica a la unidad funcional que lea los operandos de los registros y comience a ejecutarse.

- **Execute**: la unidad funcional notifica al algoritmo cuando finaliza la ejecución.
- **Writeback**: comprueba si existe peligro de WAR y en tal caso detiene la unidad funcional hasta que desaparezca, por lo que está no está disponible para otras instrucciones.

Para este control, mantiene tres tablas de estado:

- **Estado de Instrucción**: para cada instrucción que se está ejecutando, indica en qué etapa se encuentra.
- **Estado de Unidad Funcional**: por cada unidad funcional, indica su estado actual a partir de nueve campos:
  - *Busy*: indica si se esta utilizando.
  - *Op*: indica la operación a realizar.
  - *Fi*: indica el registro destino.
  - *Fj, Fk*: indican los registros fuente.
  - *Qj, Qk*: indican las unidades funcionales que actualizaran *Fj* y *Fk*.
  - *Rj, Rk*: indican cuándo estarán listo *Fj* y *Fk*.
- **Estado de Registro**: para cada registro, indica qué unidad funcional escribirá el resultado.

Algunas limitaciones de este algoritmo son:

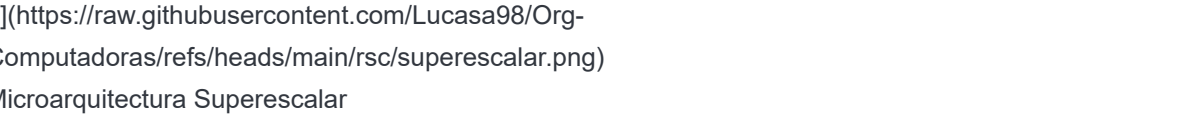
- No permite adelantar datos, primero escribe el registro y luego lo lee.
- Limitado a instrucciones en el bloque básico (ventana pequeña).
- Detiene al procesador ante WAW y WAR.

1.2.3. Algoritmo de Tomasulo

Este es otro método de planificación dinámica (en tiempo de ejecución). Un resumen de su funcionamiento:

- **Estaciones de Reserva**: cada instrucción es primero colocada en una estación de reserva en lugar de asignarse a una unidad funcional. De esta forma las instrucciones no bloquean la ejecución mientras se espera que los datos estén disponibles.
- **Renombrado de Registros**: cada vez que una instrucción emite un resultado, se asocia con un tag único para que las demás instrucciones puedan apuntar a esta en lugar de a un registro específico. De esta forma se evitan conflictos por dependencia de nombre (WAR y WAW).
- **Escritura en Común**: cada vez que una instrucción completa su operación, escribe el resultado en el Common Data Bus (CDB) y de esa forma las instrucción que esperan ese dato lo pueden "capturar" de forma simultanea.
- **Control de Dependencias**: Tomasulo hace uso de estas etiquetas y estaciones de reserva para resolver **dependencias de datos**:
  - RAW: asegura que las instrucciones que dependen de resultados previos no se ejecuten hasta que estén disponibles los datos.
  - WAR y WAW: se resuelven por medio del renombrado de registros.

2. Implementación Superescalar


Microarquitectura Superescalar

Es una microarquitectura que incrementa los recursos (unidades funcionales y registros) disponibles de manera independiente, implementando paralelismo a nivel de máquina (MLP). Así, obtenemos múltiples pipelines con múltiples etapas que puede ejecutar múltiples instrucciones al mismo tiempo.

Esto permite la ejecución de múltiples grupos (streams) de instrucciones en simultaneo.
Este tipo de procesador lee múltiples instrucciones al mismo tiempo, tratando de encontrar aquellas que puedan ejecutarse de manera independiente. El ILP aumenta al combinar:

- Optimización del Compilador: el código del programa generado se optimiza teniendo en cuenta las características del procesador.
- Técnicas de Hardware: los procesadores se diseñan para minimizar los efectos de las dependencias que no pudo eliminar el compilador, aunque está limitada por:
    - Dependencias de lectura-escritura: no se puede ejecutar una instrucción que depende del resultado de la anterior.
    - Dependencias de procedimiento: no se pueden paralelizar instrucciones antes y después de un salto.
    - Conflictos de recursos: dos instrucciones no pueden acceder al mismo recurso al mismo tiempo.