T4Segmentacindecauce.pdf

Lorenax_Caceres

Estructura de Computadores

2º Doble Grado en Ingeniería Informática y Matemáticas

Facultad de Ciencias
Universidad de Granada

Apuntes T4: Segmentación de cauce

Property	
Subject	EC
Subtipo	Segmentación de cauce
Tags	Apuntes
Tema	4

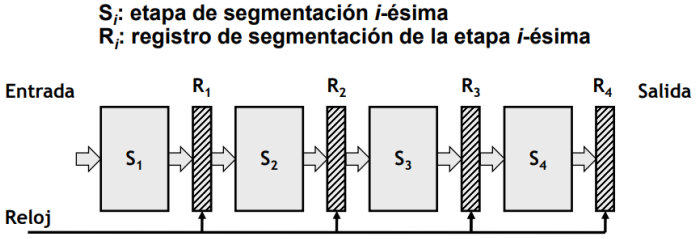
TEMA 4 SEGMENTACIÓN:

Segmentación o Pipelining:

▼ **¿En que consiste?**

En subdividir el proceso en varias etapas permitiendo el solapamiento y la mejor organización.

Surge para **mejorar la velocidad de procesamiento** sin hacer cambios físicos. Esto es, ya que cada etapa sigue requiriendo su tiempo correspondiente, peo **puede haber varias etapas ejecutándose simultáneamente.**

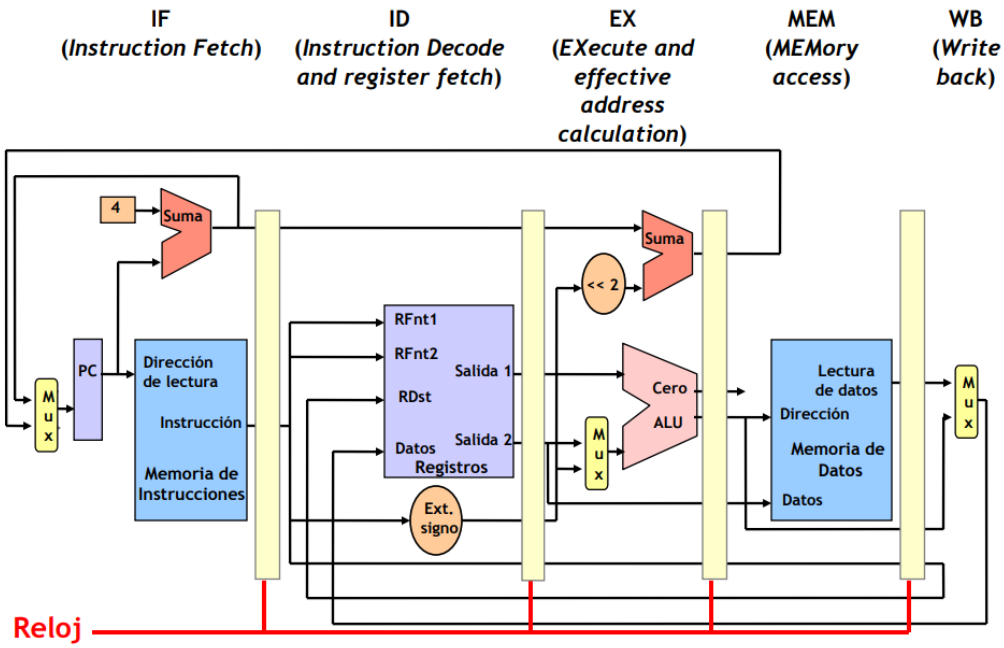


▼ **Ejemplos**

▼ **Primer ejemplo:**

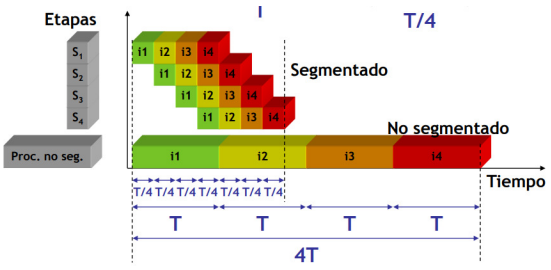
Vemos que en este ejemplo, se podría estar captando una instrucción a la vez que se escriben los resultados, pues hemos permitido el solapamiento. Es importante tener en cuenta que:

- Cada etapa debe poder completarse en un ciclo de reloj. **1 etapa ≤ 1 ciclo**
- Fases de captación y de memoria
  - Si acceden a **memoria principal**, el acceso es varias veces más **lento**
  - La **caché permite acceso en un único ciclo de reloj**
- **El periodo del reloj se toma como el de la etapa más lenta**



Aceleración

▼ **Definición**



Con la segmentación ganamos tiempo, pues llegamos a lograr solapamiento entre las partes de un proceso. Veamos como de buena puede ser esa ganancia.

Siendo k el número de instrucciones y n las etapas:

- **Aceleración**:

$$A = \frac{t_{\text{sin sengmantación}}}{t_{\text{con sengmantación}}}$$

  - Es **la mejor** que podemos conseguir.
  - $\lim_{k\to\infty} A = n$, luego acaba coincidiendo con el **número de etapas de segmentación**

- **Aceleración ideal**:

$$A = \frac{nfT}{(n-1+k)T}$$

  - Es **la mejor** que podemos conseguir.
  - $\lim_{k\to\infty} A = n$, luego acaba coincidiendo con el **número de etapas de segmentación**

▼ **Causas que la disminuyen**

- Todo lo que haga que $\mathbf{T_c > \frac{T}{n}}$
  - **Coste** de la segmentación
  - **Duración del ciclo** de reloj impuesto por etapa más lenta
- **Riesgos** que bloquean el avance de las instrucciones.

▼ **Riesgos**

Son situaciones que impiden la ejecución de la siguiente instrucción en su ciclo correspondiente. Por tanto, modifican el curso de las instrucciones en etapas posteriores. Esto causa una **reducción de las prestaciones de la segmentación.**

Podemos encotnrarnos:

- **Riesgos estructurales**
  - Conflicto por el empleo de **recursos**, dos instrucciones necesitan un mismo recurso.
  - *Para **evitar los de tipo fallo de caché**, se suelen **captar las instrucciones antes de que sean necesarias para llevarlas a una cola de instrucciones.***
- **Riesgos (por dependencias) de datos**
  - Acceso a **datos** cuyo **valor actualizado depende** de la ejecución de **instrucciones precedentes.**
  - *Estas pueden ser **detectadas por el hardware al decodificar las instrucciones**. También pueden ser **resueltas por el compilador.***
    - *En caso de hacerlo el compilador, nos queda un **hardware más simple**, y tenemos posibilidad de **aprovechar de forma inteligente ese tiempo**, pero nos **aumenta el tamaño del código.***
- **Riesgos de control**
  - Son **consecuencia** de la ejecución de instrucciones **de salto.**
    - Podemos ver la pérdida por saltos gracias al **CPI (Cicles per instruction):**
      - $CPI = 1 + p_e b$
        - b: nº de ciclos desperdiciados cuando se produce el salto.
        - p_e: probabilidad efectiva de que haya salto, que se calcula como $p_b p_t$, la probabilidad de que se ejecute un salto por la probabilidad de que se salte una vez llegado a la instrucción.
      - Como **CPI=1 sin saltos**, podemos definir $\mathbf{F_b = \frac{1}{1+p_e b}}$ **la fracción de máximas prestaciones** que relaciona el CPI con y sin saltos.
    - *Para no perder ciclos, se puede intentar **tener la dirección del salto lo antes posible,** en la etapa de captación*
    - *Otros métodos son el **salto retardado, Annulling branch y la predicción de saltos***

Veamos esos tres métodos para mejorar los riesgos de control en más detalle:

Salto retardado delayed branch:

La idea es **no desperdiciar las etapas posteriores a la de salto.** Tenemos dos formas de lograrlo:

- La primera consiste en **buscar instrucciones anteriores al salto que se puedan colocar tras el.** Si es condicional, nos cuidamos de que no afecten.
- La segunda es **colocar las instrucciones destino tras el salto.** Esto no sirve en condicionales.

Annulling branch

Un salto de este tipo **sólo ejecuta las instrucciones si se da el salto, ya que en caso contrario las ignora.** En estos casos, los condicionales si pueden tener instrucciones tras ellos.

Predicción de saltos

Consiste en **tratar de predecir si se va a tomar o no** para tomar decisiones acordes y tratar de no perder esos ciclos de reloj.

Se pueden hacer **dos tipos de predicción:**

- **Estática**: se toma la misma decisión para cada tipo de instrucción
- **Dinámica:** cambia según la historia de ejecución de un programa

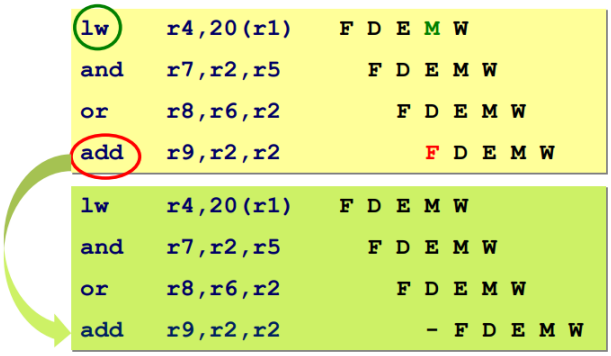
▼ **Ejemplos**

Supongamos un set de etapas:

- F: búsqueda (fetch) de instrucción.
- D: decodificación de instrucción / lectura de registros.
- E: ejecución / cálculo de direcciones
- M: acceso a memoria.
- W: escritura (write) de resultados.

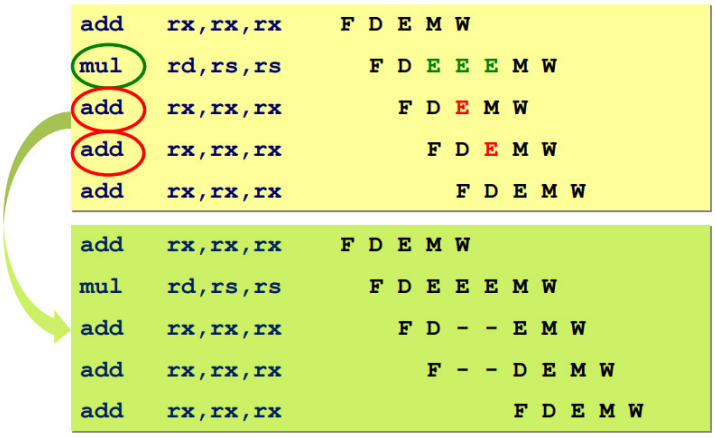
▼ **Ejemplos estructurales:**

- **Lectura de dato + captación** suponiendo una única memoria para datos e instrucciones.



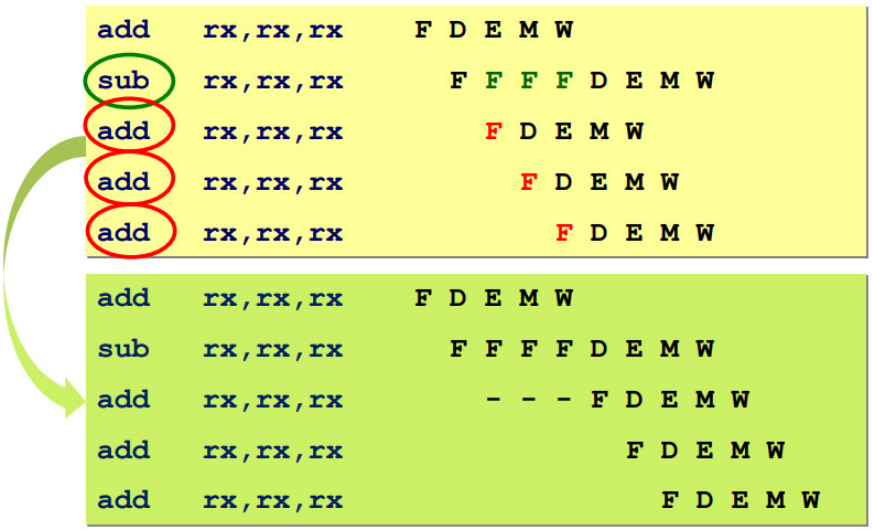
Es un problema porque se quiere sacar algo de memoria (en F) a la vez que ya se está leyendo un dato (M)

- **Ejecución de una operación con más de un ciclo en E.**



Cuando una se excede y tarda más, el siguiente ya quiere usar ese recurso pero sigue ocupado.

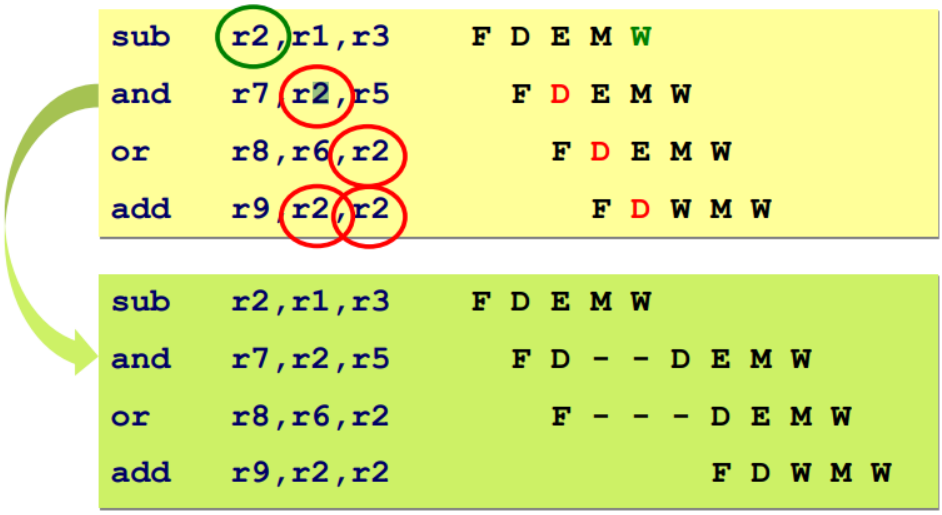
- **Fallo de caché al captar una instrucción.**



Si el caché falla y al captar la instrucción ocupamos demasiado, todo lo posterior debe retrasarse también

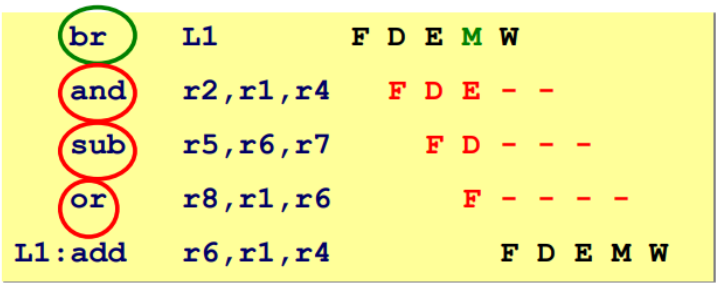
▼ **Ejemplos de datos:**

- *Se pide R2 en todas las operaciones, pero es necesario acabar la anterior y actualizar su valor antes de continuar, pues en el caso contrario usaríamos un valor desactualizado*



▼ **Ejemplos de control:**

- **Salto incondicional:** *Perdemos 3 ciclos, pues no se conoce la dirección del salto hasta ese momento*



- **Salto condicional**: Es igual, pero solo si se da

Influencia en el repertorio de instrucciones

Modos de direccionamiento

Nos interesa que **un operando no necesite más de un acceso a memoria**, es decir, que sea constante, registro, indirecto a través de registro o indexado. También es deseable que **sólo puedan acceder a memoria las instrucciones de carga y almacenamiento**

Sabiendo esto, podemos **tener un direccionamiento más complejo**, con 2 accesos a memoria **o uno sencillo** que permita solo 1. Al final van a dar códigos de **duración similar**, pero con el simple nos ahorramos complejidad del hardware.

Códigos de condición

Las dependencias que introducen los bits de condición dificultan al compilador reordenar el código (deseable para evitar riesgos). Por ello, **es deseable elegir en cada instrucción si afecta o no a los códigos de condición.**

Funcionamiento superescalar

¿Qué es? ¿En que se diferencia con el segmentado?

Tenemos el **segmentado**, que **ejecuta una instrucción tras otra**, y en un **rendimiento ideal lleva una instrucción por ciclo.**

Y luego está el **superescalar**, que tiene las siguientes carácteristicas:

- Varias instrucciones en **paralelo**
- Necesidad de **varias unidades funcionales**
- **Emisión múltiple:** se puede comenzar a ejecutar más de una instrucción por ciclo de reloj

- Rendimiento: **más de una instrucción por ciclo**
- Es **fundamental poder captar instrucciones rápidamente**: conexión ancha con caché + cola de instrucciones
- Las **instrucciones no tienen por que finalizar en el órden** en el que se emitieron

---

VENTAJAS:

- Mejora el **rendimiento** por poder hacer **más de una instrucción por ciclo.**

DESVENTAJAS:

- **Mayor efecto negativo de los riesgos**
  - Pero el compilador puede reordenar las instrucciones para evitar dichos riesgos