Enrique López García

# Segmentación de Cauce

Estructura de Computadores. Tema 4.

## Concepto de Segmentación

### Segmentación en un Procesador

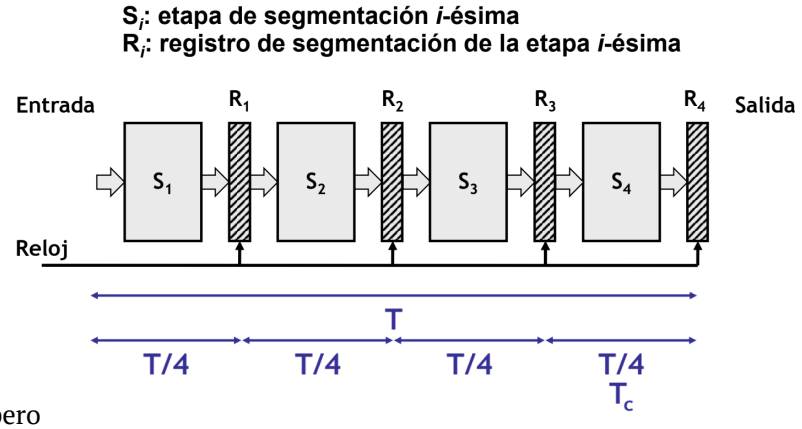
Consiste en **subdividir** el procesador en **n etapas**, permitiendo el solapamiento en la ejecución de instrucciones.

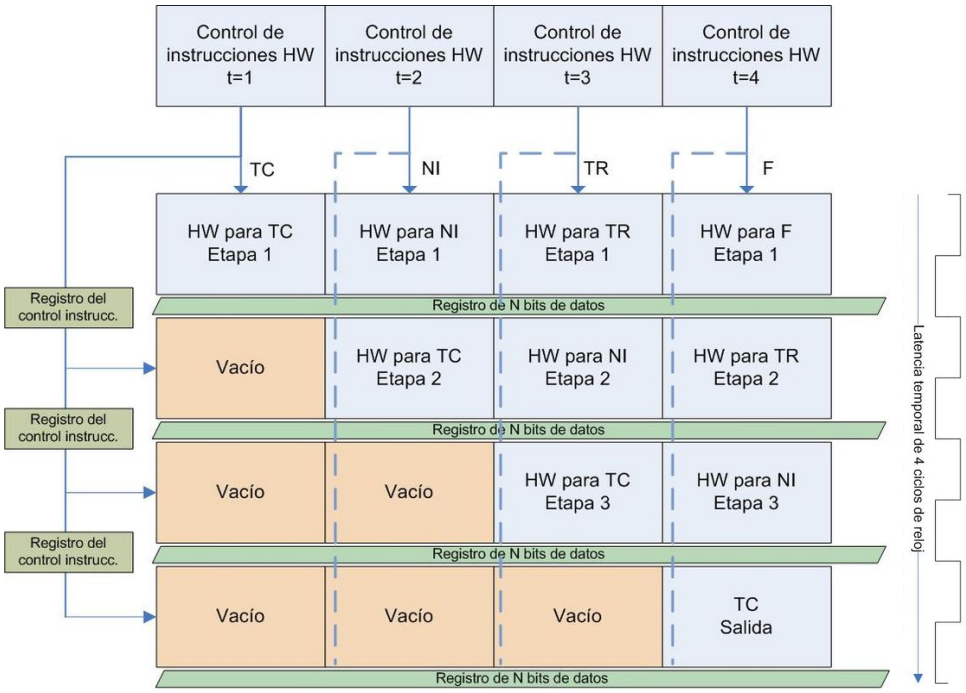
Las instrucciones **entran por un extremo** del cauce, **son procesadas** en distintas etapas y **salen por el otro** extremo.

Cada instrucción sigue ejecutándose en un tiempo T, pero hay **varias instrucciones ejecutándose simultáneamente**.

Cada etapa del cauce debe complementarse en **un ciclo de reloj**, pasando sus resultados a la etapa siguiente y recibiéndolos de la anterior.

Para pasar los datos se usan los **registros intermedios**.



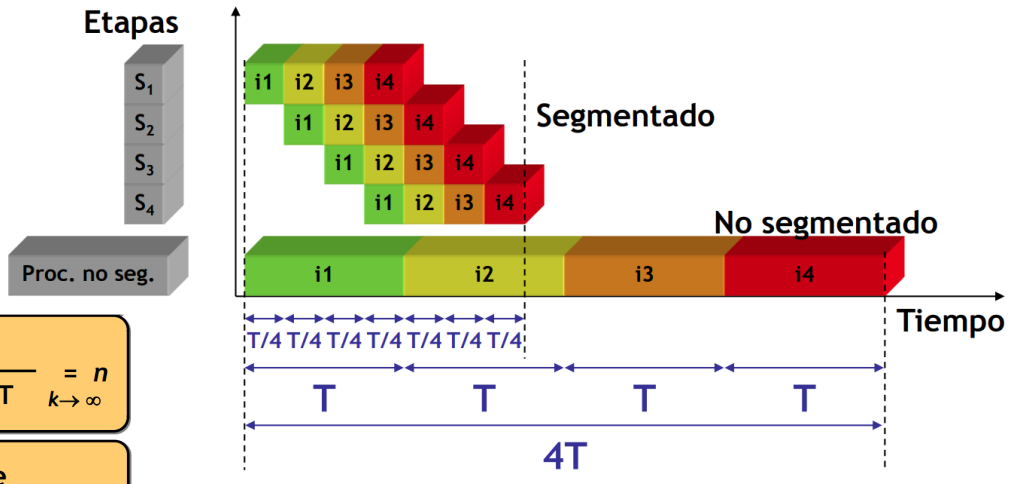




### Ejemplos de Segmentación

Fases de captación y de memoria:

- Si acceden a **memoria principal**, el acceso es varias veces más lento.
- La **caché** permite el acceso en un único ciclo de reloj.

El **periodo de reloj** se escoge de acuerdo a la etapa más larga del cauce. Pues los **periodos** de todas las **etapas** han de ser **iguales**.




## Aceleración

La aceleración ideal se calcula de la siguiente forma:

$$\text{Aceleración ideal} = \frac{kT}{(n-1+k)T/n} = \frac{nkT}{(n+k-1)T} \underset{k\to\infty}{=} n$$

($k$ instrucciones; $n$ etapas)

**La aceleración ideal coincide con el número de etapas de segmentación**

## Ejemplo

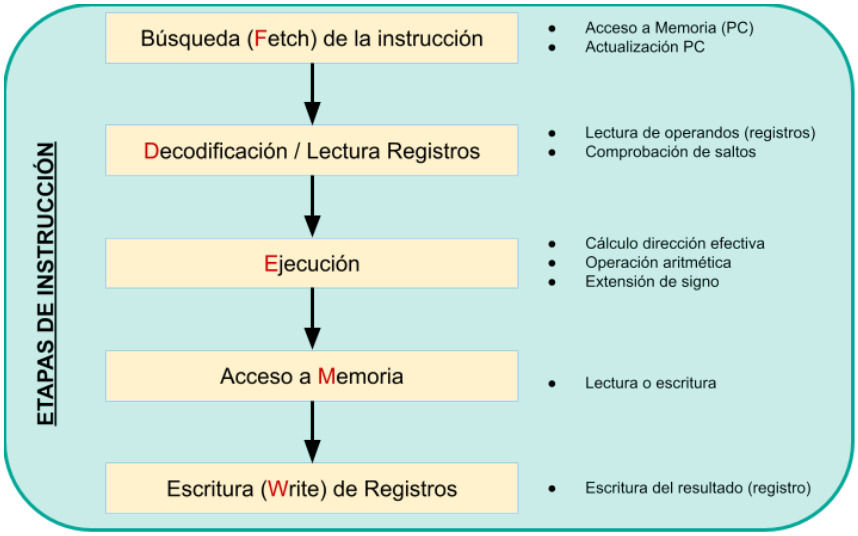
**Tiempo**$_{\text{Máquina No Segmentada}}$ = 4T

**Tiempo**$_{\text{Máquina Segmentada}}$ = 7T/4

$$\textbf{Aceleración} = \frac{\text{Tiempo}_{\text{Máquina No Segmentada}}}{\text{Tiempo}_{\text{Máquina Segmentada}}} = \frac{4T}{(7T/4)}$$

## Causas que Disminuyen la Aceleración

- **Coste** de la segmentación.
- Duración del **ciclo de reloj impuesto** por la etapa más lenta.
- **Riesgos** (hazards) ➡ Bloqueo del avance de instrucciones.




# RIESGOS

## Concepto

**Situación que impide la ejecución de la siguiente instrucción** del flujo del programa durante el ciclo de reloj designado.

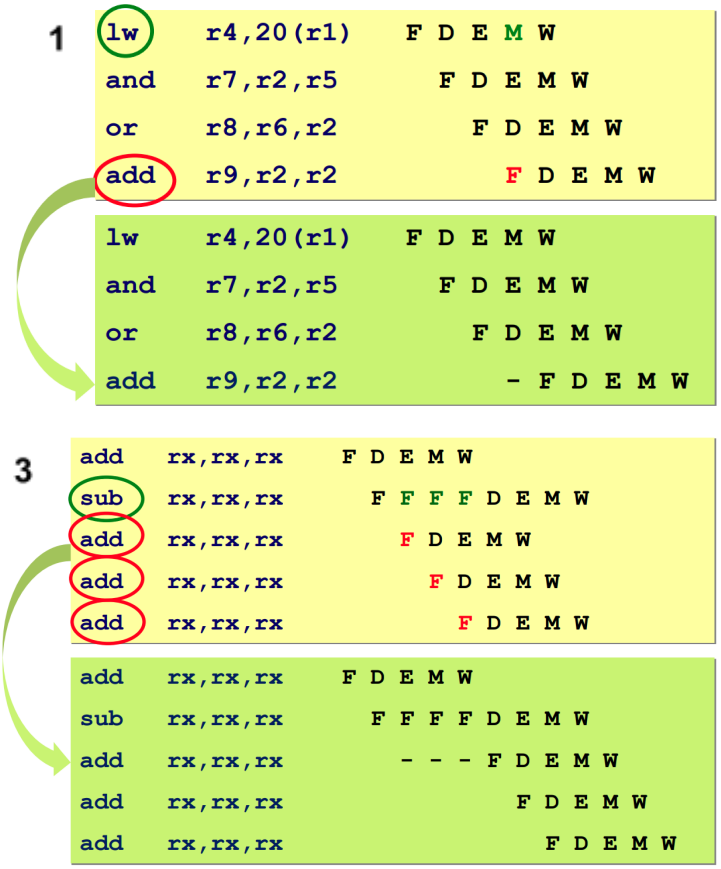
- Obliga a modificar la forma en la que avanzan las instrucciones hacia las etapas siguientes.
- Reducción de las prestaciones logradas por la segmentación

## Riesgos estructurales

**Conflicto por el empleo de recursos**, dos instrucciones necesitan un mismo recurso.

### *Ejemplos*

1. Lectura de dato + captación suponiendo **una única memoria** para datos e instrucciones.
2. Ejecución de una operación con **más de un ciclo** en E.
3. **Fallo de caché** al captar una instrucción.

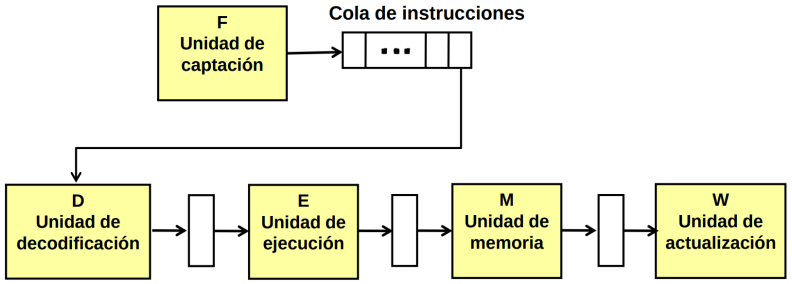




### *Observaciones*

- **Ordenamos** instrucciones según **tiempo** y **frecuencia** de aparición: en el 2 mul aparece menos que add, además mul tarda más ciclos.

Notación Intel ➡ (Dest, Src)
*Notación nuestra ➡ (Src, Dest)*

- Para **reducir** el efecto de los **fallos de caché** se suelen captar instrucciones antes de que sean necesarias (**precaptación**) y se almacenan en una cola de instrucciones.



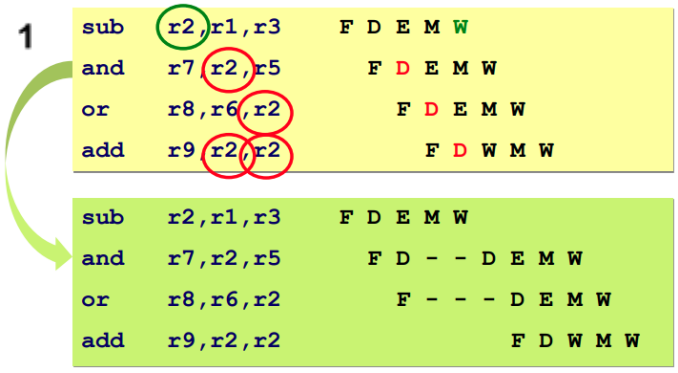

## Riesgos (por dependencias) de Datos

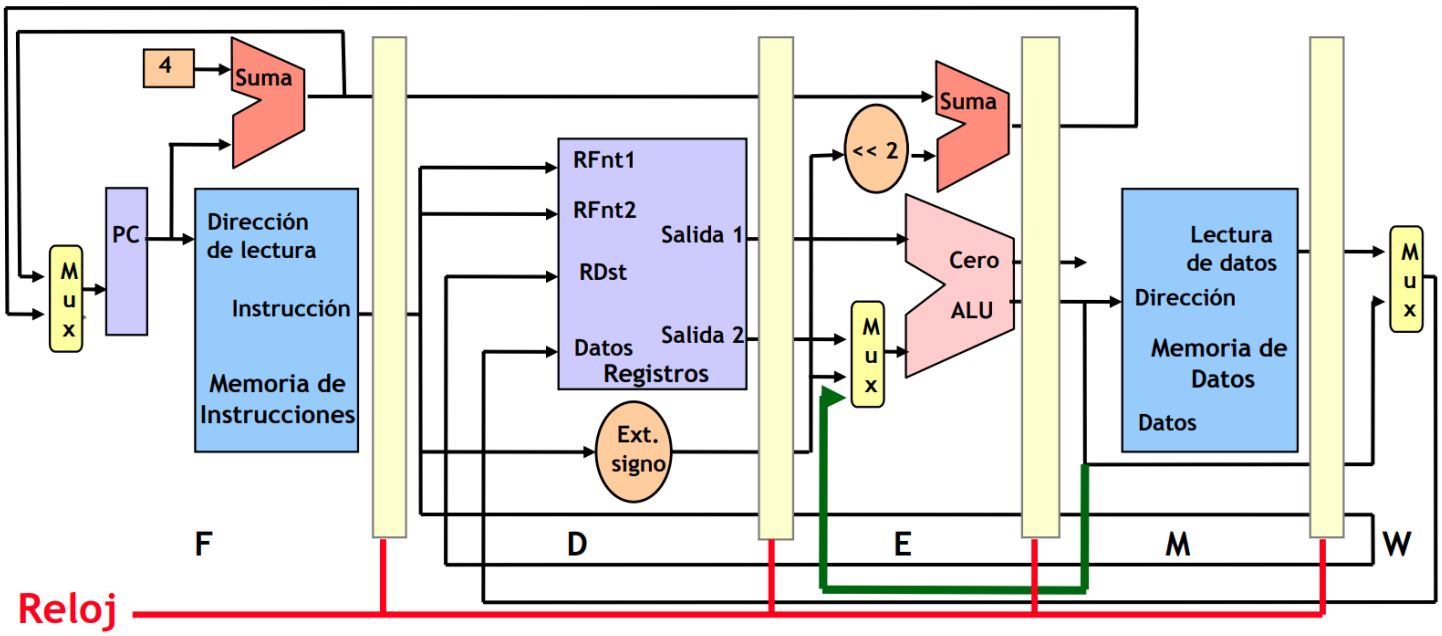
**Acceso a datos** cuyo **valor** actualizado **depende** de la ejecución de **instrucciones anteriores**.

### *Ejemplo 1*

Se está leyendo (D) **r2** en las instrucciones ***and***'s y ***or*** cuando en ***sub*** no se ha escrito (W) todavía el resultado en **r2.**

Otra posible **solución** sería actualizar la arquitectura, conectando la salida de la ALU con la entrada a la misma para que la salida de la ejecución (E) de ***sub*** se pueda usar como decodificación (D) en la siguiente instrucción.




Las dependencias de datos **las descubre el hardware** al decodificar las instrucciones (**ejemplo 1**).

Aunque **alternativamente** las puede resolver **el compilador** (imagen de la derecha). Esto tiene las siguientes **ventajas**:

- Requiere un hardware más simple
- Reorganizando las instrucciones se puede hacer un trabajo más útil (en vez de *nop*).

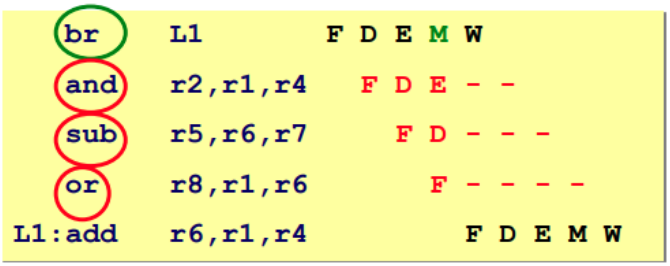
```
sub   r2,r1,r3    F D E M W
nop               F D E M W
nop                 F D E M W
nop                   F D E M W
and   r7,r2,r5          F D E M W
```

## Riesgos de Control

Estos son **consecuencia** de la ejecución de **instrucciones de salto**.

## *Salto incondicional*

En este ejemplo se pierden 3 ciclos (*huecos de retardo de salto*), ya que **hasta que no se obtenga el resultado** de la instrucción ***br*** (W), **no se conoce la dirección de salto** y por tanto se están tratando instrucciones que no van a ser "efectivas".




## *Salto condicional*

Pasaría lo mismo que el caso anterior solo que solo se perderían los ciclos si la condición de salto se verifica. En caso contrario no se pierde ningún ciclo.

## *Degradación de prestaciones debida a los saltos.*

Supongamos algún mecanismo hardware que permita descartar la ejecución de las instrucciones siguientes si se produce el salto.

- b: nº de ciclos desperdiciados cuando se produce el salto.
- $p_b$: probabilidad de que se ejecute una instrucción de salto (entre 0,15 y 0,30 normalmente)
- $p_t$ : probabilidad de que realmente se produzca el salto cuando se ejecuta una instrucción de salto
- $p_e = p_b \cdot p_t$ : probabilidad efectiva de que se produzca un salto
- CPI: nº de ciclos de reloj por instrucción (suponer CPI = 1 sin saltos)
- F**b**: fracción de máximas prestaciones (relación entre el nº de ciclos CPI si no hubiera saltos y el nº de ciclos CPI con saltos)

$$CPI = (1 - p_b)\,(1) + p_b\,[\,p_t\,(1 + b) + (1 - p_t)\,(1)\,] = 1 + p_b\,p_t\,b = 1 + p_e\,b$$

$$F_b = \frac{1}{1 + p_e\,b}$$

**CONCLUSIÓN:** la degradación crece rápidamente al crecer $p_e$ (más rápidamente a medida que b es mayor)

## *Salto retardado (delayed branch)*

**OBJETIVO**: En lugar de desperdiciar las etapas posteriores a la de salto, una o más instrucciones parcialmente completadas se completarán antes de que el salto tenga efecto.

El **compilador** busca **instrucciones anteriores lógicamente** al salto que pueda colocar tras el salto.

- Hay que tener en cuenta que si el salto es condicional, las instrucciones colocadas detrás no deben afectar a la condición de salto.

**Otra posibilidad** es colocar la(s) instrucción(es) destino de un salto tras el salto.

- Esto no funcionaría para saltos condicionales
- No podemos "subir" una instrucción que sólo tiene que ejecutarse algunas veces (cuando el salto se produce) a una posición donde siempre se ejecuta.

Antes:

```
     call subr
     nop
     ...
subr:
     mov r3,#100
```

Después:

```
     call subr+4
     mov r3,#100
     ...
subr:
     mov r3,#100
```

## *Annulling branch*

Un salto de este tipo ejecuta la(s) instrucción(es) siguiente(s) sólo si el salto se produce, pero la(s) ignora si el salto no se produce. Con un salto de este tipo, el destino de un salto condicional sí puede colocarse tras el salto.

## *Predicción de saltos*

**Intentar predecir si** una instrucción de salto concreta **dará lugar a un salto** (branch taken) o no (branch not taken). Si los resultados de las instrucciones de salto condicional fueran aleatorios, comenzar a ejecutar las instrucciones siguientes al salto desperdiciaría ciclos en la mitad de las ocasiones. Se pueden minimizar las pérdidas de ciclos inútiles si para cada instrucción de salto se puede predecir con un acierto > 50% si el salto se producirá o no. Se pueden hacer predicciones distintas si el salto es hacia direcciones menores o mayores.

Tipos de predicción:

- **Estática**: se toma la misma decisión para cada tipo de instrucción.
- **Dinámica**: cambia según la historia de ejecución de un programa (tarda más en compilar porque hace cambios, pero luego la ejecución es mucho más rápida).

# Influencia en el repertorio de instrucciones

## Modos de direccionamiento

Es deseable que **un operando no necesite más de un acceso a memoria**, mediante diferentes métodos:

- Constante
- Registro
- Indirecto a través de registro
- Indexado (EA= reg. + desp., o EA = reg. + reg.)

Es deseable que **sólo accedan a memoria** las instrucciones de **carga** y **almacenamiento**.

## Códigos de Condición

Las dependencias que introducen los bits de condición dificultan al compilador reordenar el código, aunque es deseable para **evitar riesgos**.

En el ejemplo, la decisión de saltar se produce tras la etapa E de la instrucción *cmp*.

Es deseable elegir en cada instrucción si afecta o no a los códigos de condición, para reordenar el código.

Antes:
```
add r1,r2
cmp r3,r4
jz etiq
```

Después:
```
cmp r3,r4
add r1,r2
jz etiq
```

**Al reordenar el código, se puede tomar la decisión de salto un ciclo antes, y desperdiciar un ciclo menos tras el salto**

# Funcionamiento Superescalar

## Comparación

### *Procesamiento segmentado*

- Una instrucción tras otra
- Rendimiento ideal: **una instrucción por ciclo**

### *Procesamiento superescalar*

- 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

```
fadd   rx,rx,rx     F D E1 E2 E3 M W
add    rx,rx,rx     F D E M W
fsub   rx,rx,rx       F D E1 E2 E3 M W
sub    rx,rx,rx       F D E M W
```

## Características

- El **efecto negativo** de los riesgos es **más pronunciado**.
- El compilador puede **reordenar instrucciones para evitar riesgos** (como en segmentación pero más costoso).
- Las instrucciones **pueden** emitirse en orden y **finalizar de forma desordenada** (ejemplo superior).