

# Aviso

*“Con motivo de la suspensión temporal de la actividad docente presencial en la UGR, se informa de las condiciones de uso de la aplicación de videoconferencia que a continuación se va a utilizar:*

- 1. La sesión va a ser grabada con el objeto de facilitar al estudiantado, con posterioridad, el contenido de la sesión docente.*
- 2. Se recomienda a los asistentes que desactiven e inhabiliten la cámara de su dispositivo si no desean ser visualizados por el resto de participantes.*
- 3. Queda prohibida la captación y/o grabación de la sesión, así como su reproducción o difusión, en todo o en parte, sea cual sea el medio o dispositivo utilizado. Cualquier actuación indebida comportará una vulneración de la normativa vigente, pudiendo derivarse las pertinentes responsabilidades legales.”*

2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores

## Tema 4. Arquitecturas con Paralelismo a nivel de Instrucción (ILP)

Material elaborado por los profesores responsables de la asignatura:

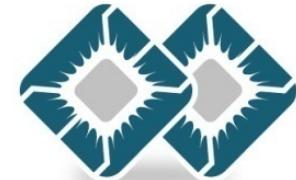
Julio Ortega – Mancia Anguita

Licencia Creative Commons



ugr

Universidad  
de Granada



# Lecciones

- Lección 11. Microarquitecturas ILP. Cauces Superescalares
- Lección 12. Consistencia del procesador y Procesamiento de Saltos
- Lección 13. Procesamiento VLIW

2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores

## Tema 4

# Lección 11. Microarquitecturas ILP. Caucos Superescalares

Material elaborado por los profesores responsables de la asignatura:  
Julio Ortega – Mancia Anguita

*Licencia Creative Commons*



*ugr*

Universidad  
de Granada

# Lecciones

- Lección 11. Microarquitecturas ILP. Cauces Superescalares
  - Introducción: Motivación y Nota Histórica
  - Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización
  - Cauces superescalares
- Lección 12. Consistencia del procesador y Procesamiento de Saltos
- Lección 13. Procesamiento VLIW

# Bibliografía

## ➤ Fundamental

- M. Anguita, J. Ortega: "Fundamentos y Problemas de Arquitectura de Computadores". Ed. Avicam, 2016.
- Capítulo 3. Secc. 3.1, 3.3.1, 3.3.2, 3.3.3. J. Ortega, M. Anguita, A. Prieto. *Arquitectura de Computadores*. Thomson, 2005. ESIIT/C.1 ORT arq

## ➤ Complementaria

- Sima and T. Fountain, and P. Kacsuk.  
*Advanced Computer Architectures: A Design Space Approach*. Addison Wesley, 1997. ESIIT/C.1 SIM adv

# Contenido de la Lección 11

- Introducción: Motivación y Nota Histórica
- Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización
- Cauces superescalares

# Arquitecturas con DLP, ILP y TLP (thread=flujo de control)

Arq. con **DLP**  
(*Data Level Parallelism*)

Tema 5

Ejecutan las operaciones de una instrucción **concurr.** o en **paralelo**

**Unidades funcionales** vectoriales o SIMD

Arq. con **ILP**  
(*Instruction Level Parallelism*)

Tema 4

Ejecutan múltiples instrucciones **concurr.** o en **paralelo**

**Cores escalares** segmentados, superescalares o VLIW/EPIC

Arq. con **TLP** (*Thread Level Parallelism*) explícito y **una** instancia de SO

Temas 3, 5

Ejecutan múltiples flujos de control **concurr.** o en **paralelo**

**Cores** que modifican la arquit. escalar segmentada, superescalar o VLIW/EPIC para ejecutar threads **concurr.** o en **paralelo**

**Multi-procesadores:** ejecutan threads en paralelo en un computador con múltiples cores (incluye **multicore**)

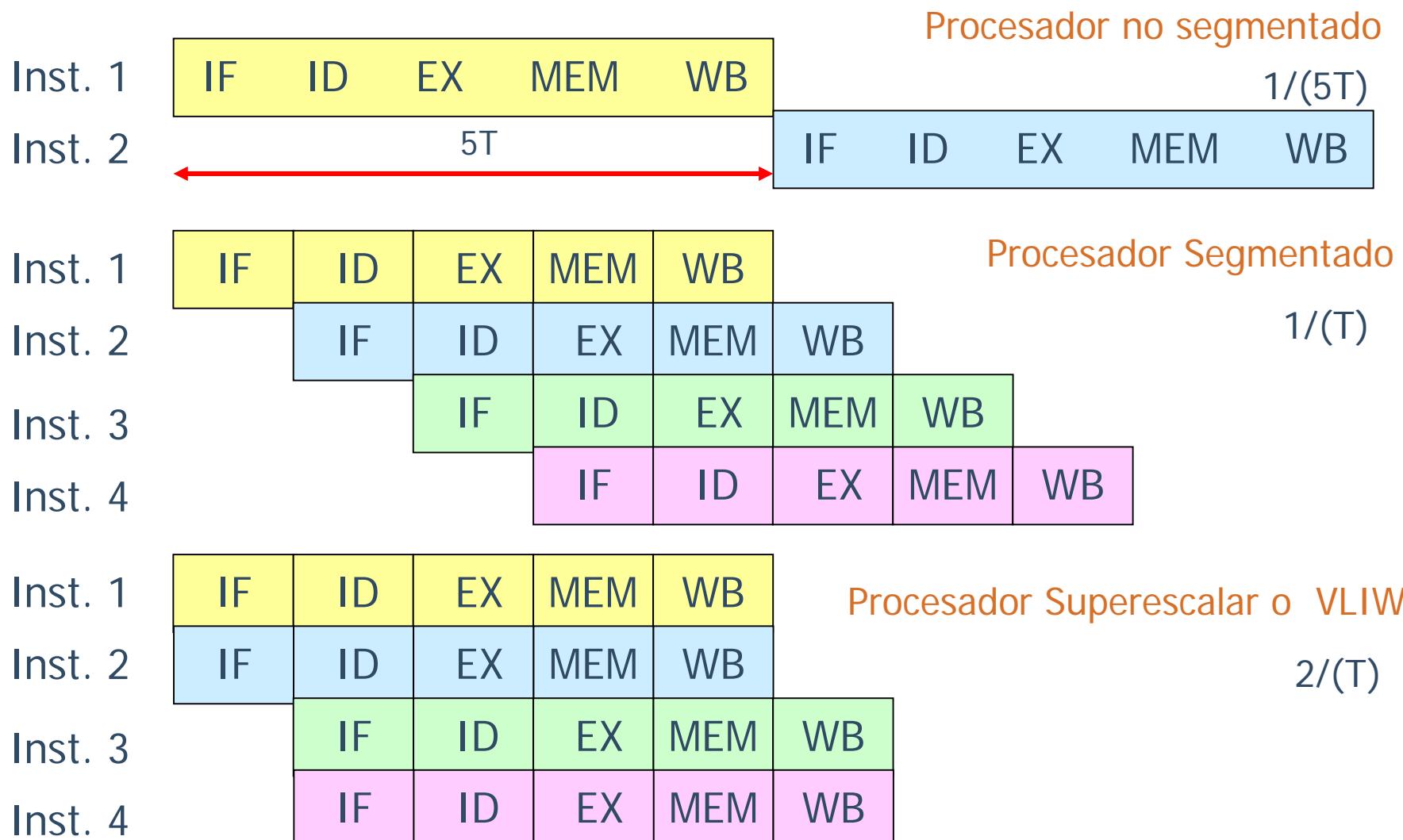
Arq. con **TLP** explícito y **múltiples** instancias SO

IC.SCAP

Ejec. múltiples flujos de control en **paralelo**

**Multi-computadores:** ejecutan threads en paralelo en un sistema con múltiples computadores

# Paralelismo entre Instrucciones



# Mejora de las Prestaciones de los Procesadores

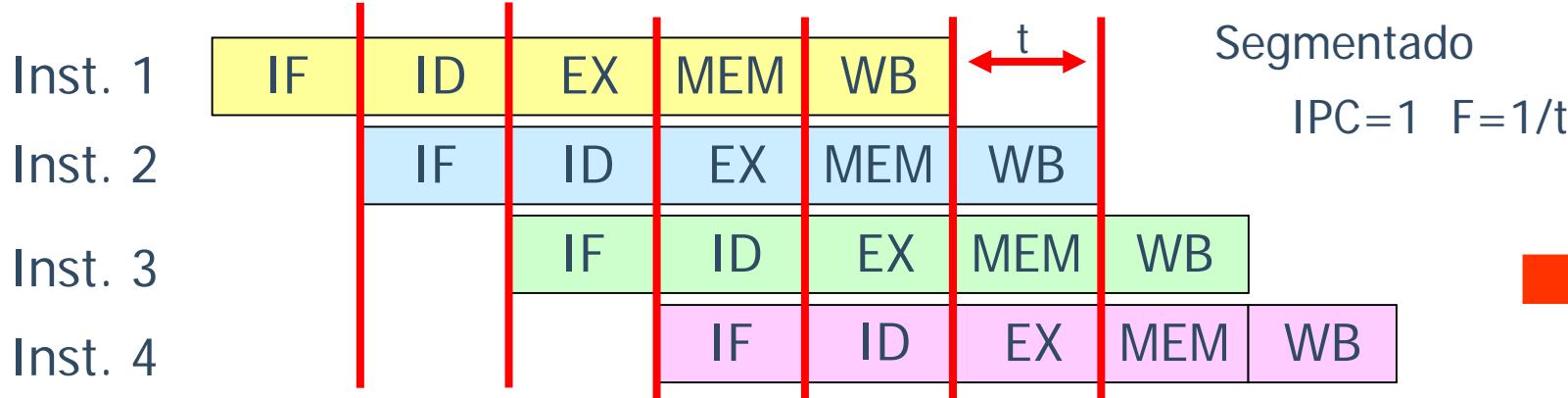
Más transistores por circuito integrado →  
**Microarquitecturas más complejas en un solo CI:**  
**Paralelismo entre Instrucciones (Procesadores Superescalares)**

Mejora de la Tecnología de Fabricación de CI basada en el Silicio → **Reducción del tamaño de los transistores + Aumento del tamaño del dato**

$$V_{CPU} = IPC \times F$$

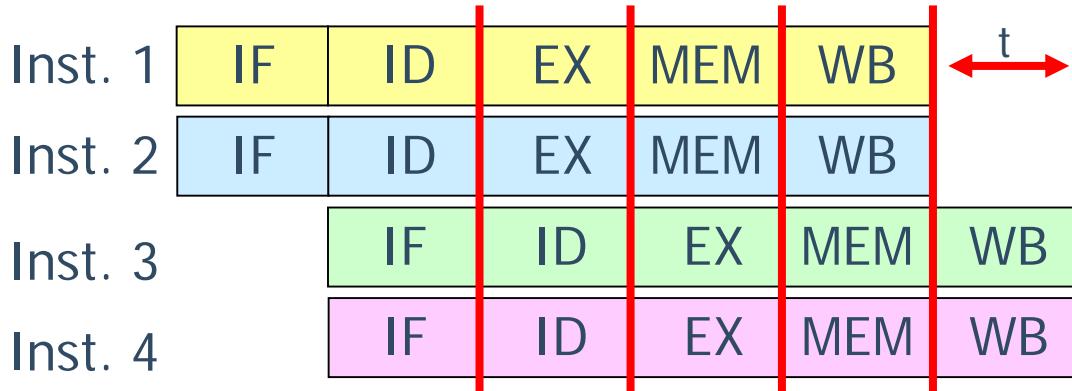
Se reduce la longitud de puerta del transistor y con ello el tiempo de conmutación → **Mayores frecuencias de funcionamiento**

# Evolución de Microarquitecturas I



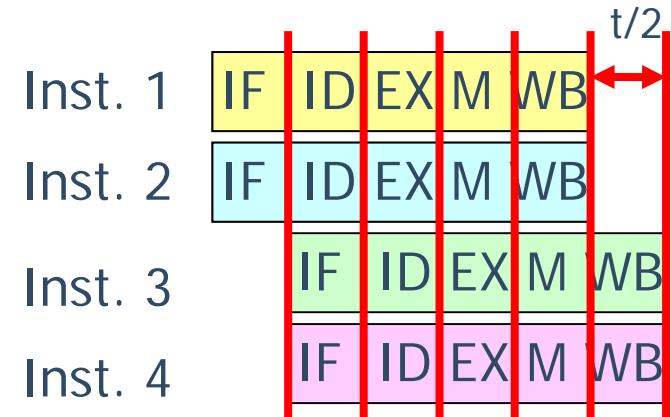
## Aumento de IPC

Superescalar o VLIW  $IPC=2$   $F=1/t$

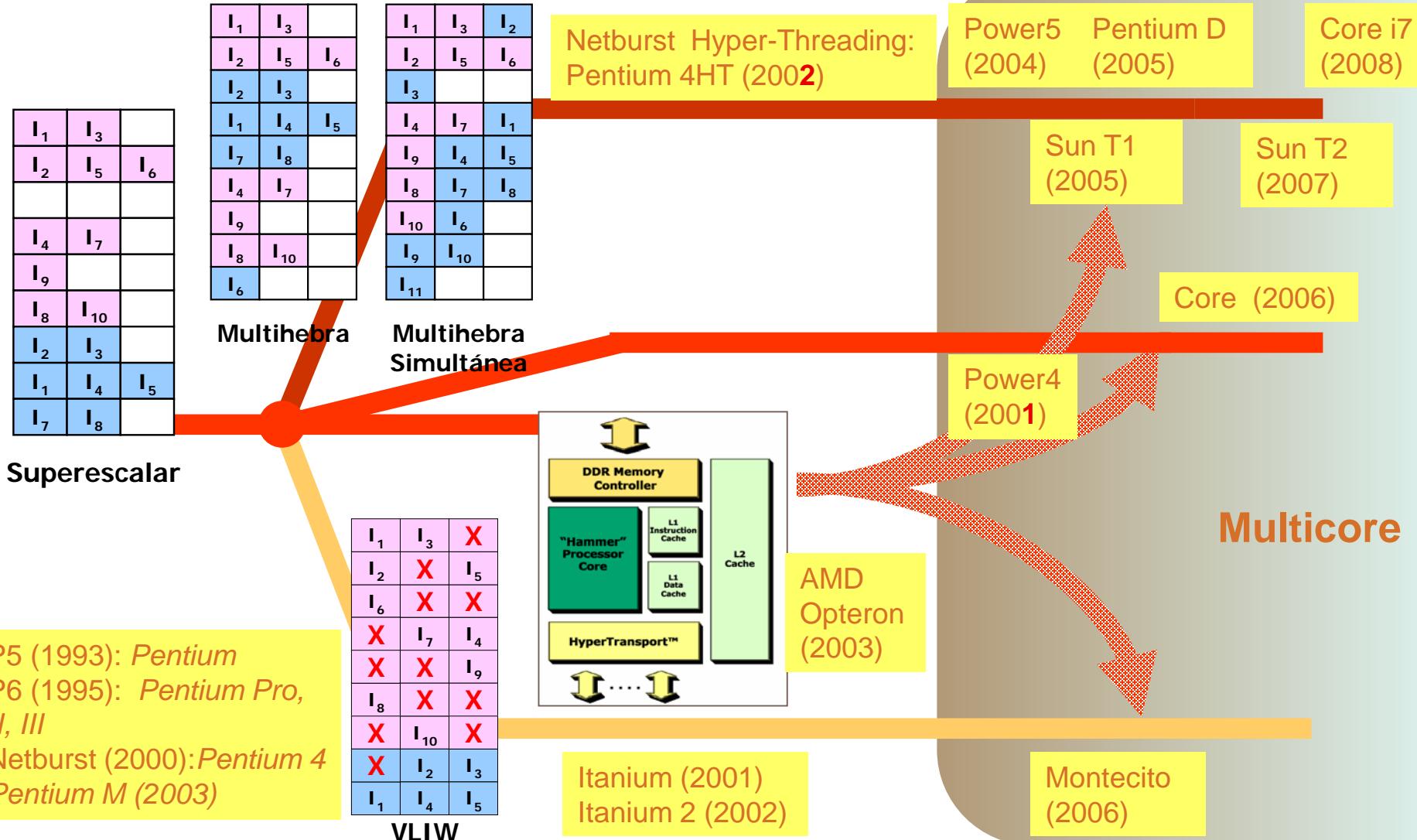


## Aumento de IPC y F

$IPC=2$   $F=2/t$



# Evolución de Microarquitecturas II



P5 (1993): Pentium  
P6 (1995): Pentium Pro,  
II, III  
Netburst (2000): Pentium 4  
Pentium M (2003)

# Nota histórica. DLP y ILP (*Instruction Level Parallelism*)

- DLP (Data Level Parallelism)
  - Unidades funcionales (o de ejecución) SIMD (o multimedia)
    - 1989 (Intel i860). 1991 (motorola M88110). 1993 (repertorio MAX en HP PA : PA7100LC). 1995 (repertorio VIS en Sun Sparc: Ultra I). 1997 (repertorio MMX en Intel x86: Pentium MMX). 1999 (repertorio SSE en Intel x86: Pentium III; repertorio Altivec en IBM Power: PowerPC 8000)
- ILP (*Instruction Level Parallelism*) :
  - Procesadores/cores segmentados
    - 1961 (IBM 7030). 1982 (chip Intel i286, Motorola 68020). 1986 (chip MIPS R2000). 1987 (chip AMD Am29000). 1988 (chip Sun Sparc)...
  - Procesadores con múltiples unidades funcionales
    - 1967 (IBM 360/91) ...
  - Procesadores/cores superescalares
    - 1989 (chip Intel 960CA (3)). 1990: (chip IBM Power1 (4)). 1992: (chips DEC α21064 (2/4), HP PA 7100 (2/2), Sun SuperSparc (3/5)) ...
  - Procesadores/cores VLIW
    - 1990 (chip DSP Intel i860 (2)). 1997 (chip DSP TMS320C6x (8)). 2001 (chip Intel Itanium)...

TEMA 4

# Contenido de la Lección 11

- Introducción: Motivación y Nota Histórica
- Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización
- Cauces superescalares

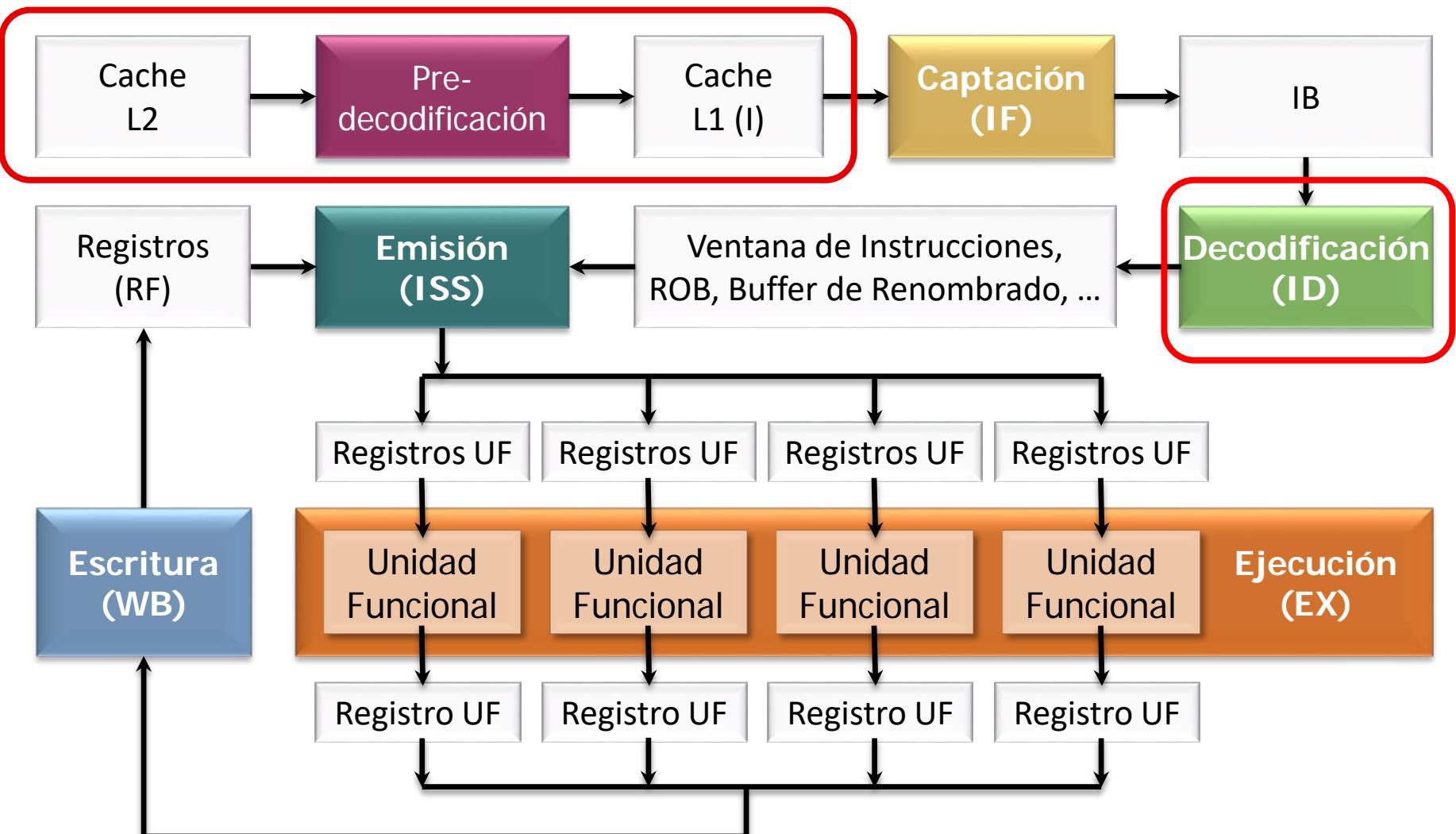
# Ordenaciones en una secuencia de instrucciones

- En una secuencia de instrucciones se pueden distinguir tres tipos de ordenaciones:
  - El orden en que se captan las instrucciones (el orden de las instrucciones en el código)
  - El orden en que las instrucciones se ejecutan
  - El orden en que las instrucciones cambian los registros y la memoria.
- El procesador superescalar debe ser capaz de identificar el paralelismo entre instrucciones (ILP) que exista en el programa y organizar la captación, decodificación y ejecución de instrucciones en paralelo, utilizando eficazmente los recursos existentes (el paralelismo de la máquina).
- Cuanto más sofisticado sea un procesador superescalar, menos tiene que ajustarse a la ordenación de las instrucciones según se captan, para la ejecución y modificación de los registros, de cara a mejorar los tiempos de ejecución. La única restricción es que el resultado del programa sea correcto.

# Contenido de la Lección 11

- Introducción: Motivación y Nota Histórica
- Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización
- Cauces superescalares
  - Decodificación Paralela y Predecodificación
  - Emisión Paralela de Instrucciones. Estaciones de Reserva
  - Renombramiento de registros

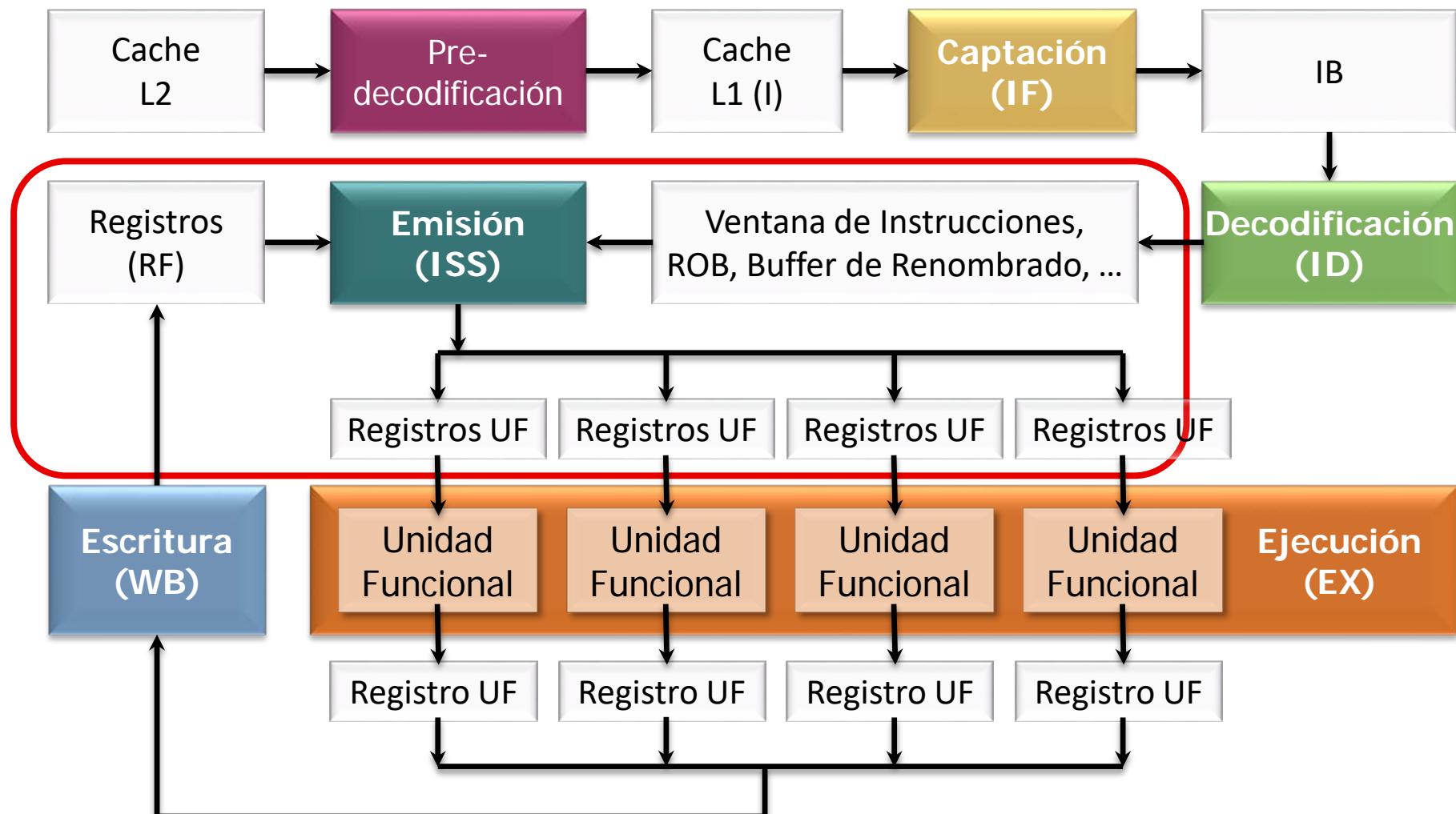
# Etapas de un Procesador Superescalar: Predecodificación I



# Contenido de la Lección 11

- Introducción: Motivación y Nota Histórica
- Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización
- Cauces superescalares
  - Decodificación Paralela y Predecodificación
  - Emisión Paralela de Instrucciones. Estaciones de Reserva
  - Renombramiento de registros

# Etapas de un Procesador Superescalar: Emisión Paralela de Instrucciones



# Ventana de Instrucciones

- La ventana de instrucciones almacena las instrucciones pendientes
  - todas, si la ventana es centralizada, o las de un tipo determinado, si es distribuida
- Las instrucciones se cargan en la ventana una vez decodificadas
  - Se utiliza un bit para indicar si un operando está disponible (se almacena el valor o se indica el registro desde donde se lee) o no (se almacena la unidad funcional desde donde llegará el operando)
- Una instrucción puede ser emitida cuando tiene todos sus operandos disponibles y la unidad funcional donde se procesará.
  - Hay diversas posibilidades para el caso en el que varias instrucciones estén disponibles (características de los buses, etc.)

## Ejemplo de Ventana de Instrucciones

#	opcode	address	rb_entry	operand1	ok1	operand2	ok2
2	MULTD	loop + 0x4	2	1	0	0	0
1	LD	loop	1	0	0	0	1

Lugar donde se almacenará el resultado

Dato no válido  
(indica desde dónde se recibirá el dato)

Dato válido  
(igual a 0)

# Emisión Paralela de Instrucciones: Ordenada

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -
mult r5 [r1] 1 - [r5] 1 -
add r4 [r1] 1 - [r2] 1 -

↓ Se han emitido [1] y [2]

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -

↓ Ha terminado [1]

sub r5 [r4] 1 - [r3] 1 -
sub r6 r5 0 mult [r2] 1 -



Ha terminado [2]: pueden emitirse [3] y [4]

add/sub: 2

mult: 1

[1] add r4,r1,r2 (2)

[2] mult r5,r1,r5 (5)

[3] sub r6,r5,r2 (2)

[4] sub r5,r4,r3 (2)

Instrucc.	ISS	EXE
add	(1)	(2)-(3)
mult	(1)	(2)-(6)
sub	(7)	(8)-(9)
sub	(7)	(8)-(9)

sub r5 [r4] 1 - [r3] 1 -
sub r6 [r5] 1 - [r2] 1 -

# Emisión Paralela de Instrucciones: Desordenada

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -
mult r5 [r1] 1 - [r5] 1 -
add r4 [r1] 1 - [r2] 1 -

↓ Se han emitido [1] y [2]

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -

↓ Ha terminado [1]

sub r5 [r4] 1 - [r3] 1 -
sub r6 r5 0 mult [r2] 1 -

Ventana de Instrucciones

add/sub: 2

mult: 1

[1] add r4,r1,r2 (2)

[2] mult r5,r1,r5 (5)

[3] sub r6,r5,r2 (2)

[4] sub r5,r4,r3 (2)

Instrucc.	ISS	EXE
add	(1)	(2)-(3)
mult	(1)	(2)-(6)
sub	(7)	(8)-(9)
sub	(4)	(5)-(6)

No afecta al tiempo

Se ha emitido [4] y ha terminado [2]: puede emitirse [3]

sub r6 [r5] 1 - [r2] 1 -
--------------------------

# Ejemplo de Emisión Paralela de Instrucciones Ordenada

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -
mult r5 [r1] 1 - [r5] 1 -
add r4 [r1] 1 - [r2] 1 -

↓ Se han emitido [1] y [2]

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -

↓ Ha terminado [1]

sub r5 [r4] 1 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -

Ventana de Instrucciones

add/sub: 1

mult: 1

[1] add r4,r1,r2 (2)

[2] mult r5,r1,r5 (5)

[3] sub r6,r5,r2 (2)

[4] sub r5,r4,r3 (2)

Instrucc.	ISS	EXE
<b>add</b>	(1)	(2)-(3)
<b>mult</b>	(1)	(2)-(6)
<b>sub</b>	(7)	(8)-(9)
<b>sub</b>	(10)	(11)-(12)

Ha terminado [2] y se ha emitido [3].  
Cuando la unidad quede libre se emitirá [4]

sub r5 [r4] 1 - [r3] 1 -

# Ejemplo de Emisión Paralela de Instrucciones Desordenada

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -
mult r5 [r1] 1 - [r5] 1 -
add r4 [r1] 1 - [r2] 1 -

↓ Se han emitido [1] y [2]

sub r5 r4 0 add [r3] 1 -
sub r6 r5 0 mult [r2] 1 -

↓ Ha terminado [1]

sub r5 [r4] 1 - [r3] 1 -
sub r6 r5 0 mult [r2] 1 -

add/sub: 1

mult: 1

[1] add r4,r1,r2 (2)

[2] mult r5,r1,r5 (5)

[3] sub r6,r5,r2 (2)

[4] sub r5,r4,r3 (2)

Ventana de  
Instrucciones



Instrucc.	ISS	EXE
add	(1)	(2)-(3)
mult	(1)	(2)-(6)
sub	(7)	(8)-(9)
sub	(4)	(5)-(6)

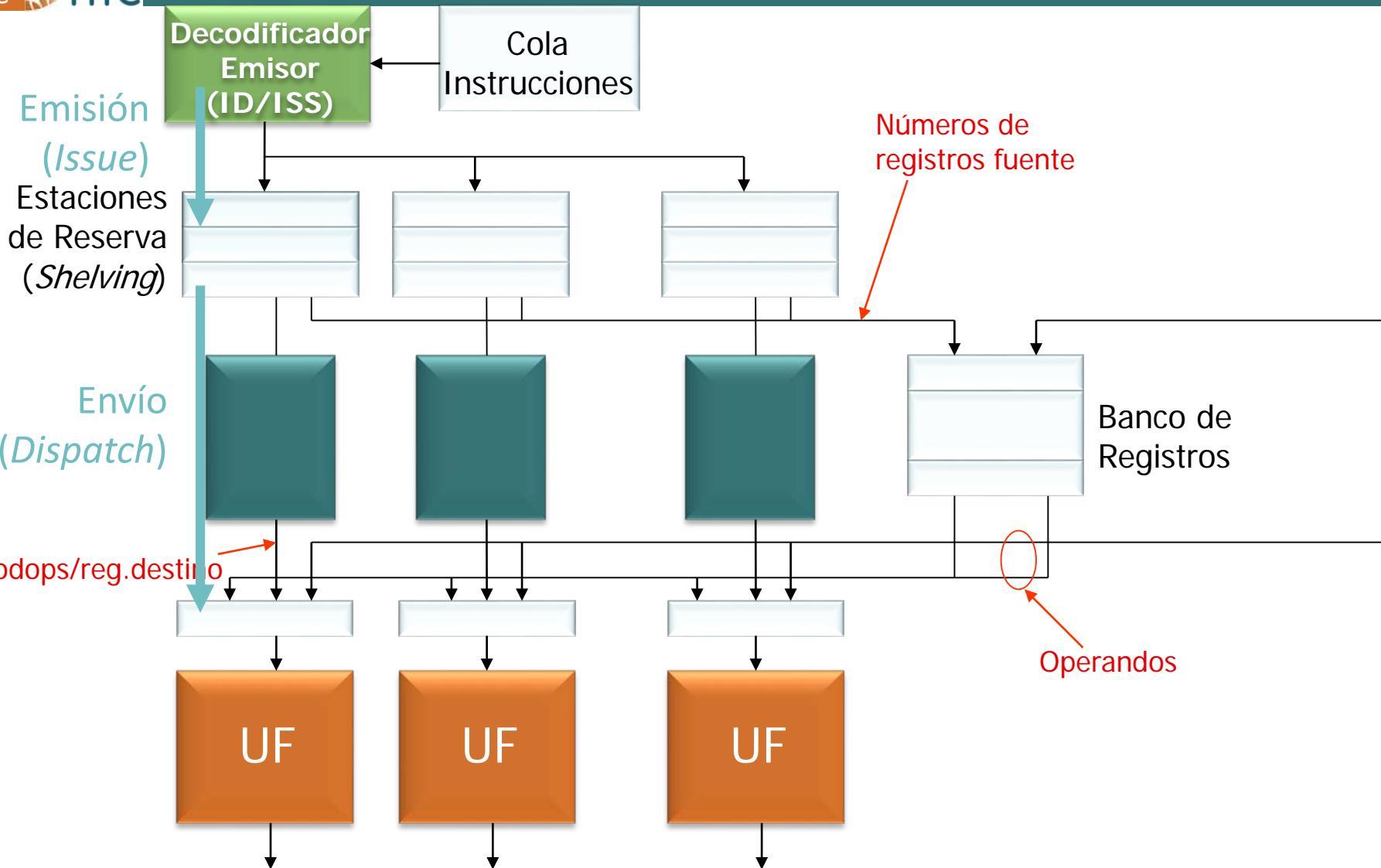
Se ha emitido [4] y ha terminado [2]: puede emitirse [3]



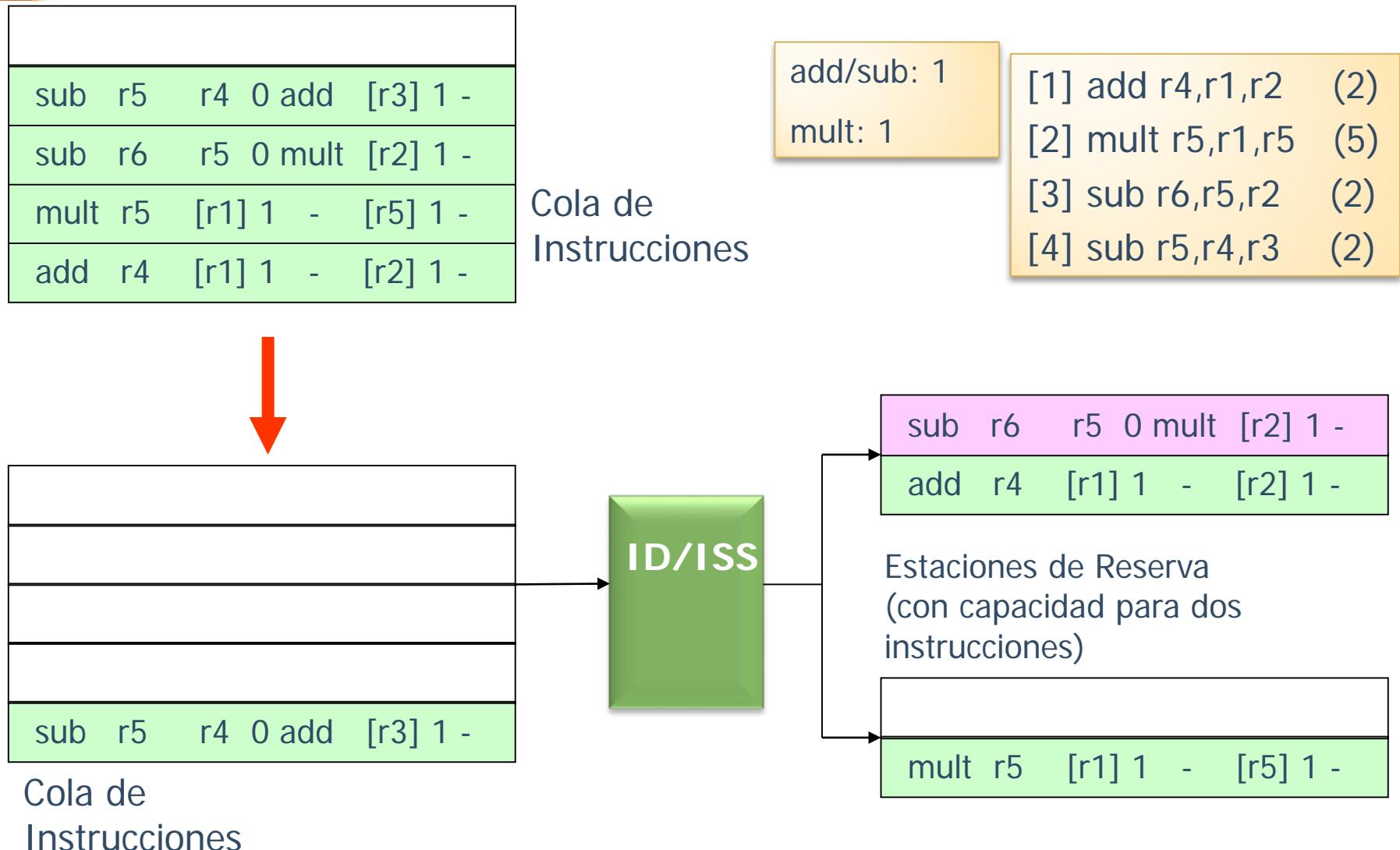
sub r6 [r5] 1 - [r2] 1 -

# Estaciones de Reserva I

## (Ventana de Instrucciones Distribuida)



# Ejemplo de Emisión con Estaciones de Reserva



# Alternativas para el Envío a las Unidades Funcionales

**Reglas de Selección:** Se determina las instrucciones que pueden enviarse  
Las instrucciones ejecutables

**Reglas de Arbitraje:** Instrucción que se envía si hay varias ejecutables  
La más antigua entre las ejecutables

**Orden de Envío:** Ordenadas, Desordenadas, o Parcialmente ordenadas (ciertas instrucciones no ejecutables bloquean instrucciones de un tipo, pero no de otros)

Prest  
Tend

**Ordenada:** Power 1 (90), PowerPC 603 (93), Am29000 (95)

**Parcialmente ordenada:** Power 2 (93), PowerPC 604 (95), PowerPC 620(96)

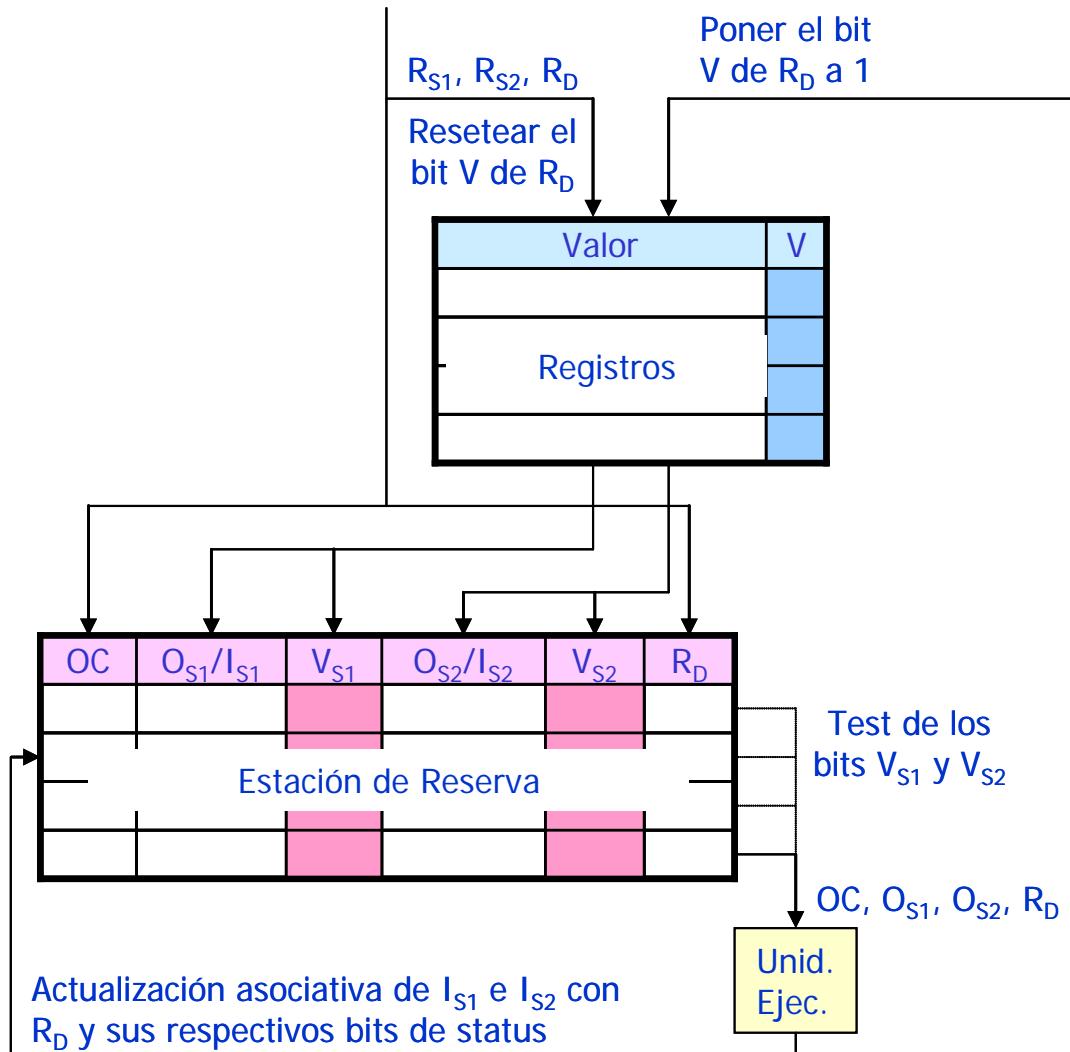
**Desordenada:** ES/9000(92), PentiumPro (95), R10000(96), PA8000 (96)

**Velocidad de Envío:** Número de instrucciones que se envían por ciclo

**Una por ciclo:** PowerPC 603 (93), PowerPC 604 (94), PowerPC 620 (95)

**Varias por Ciclo:** Sparc64(95), R10000 (96), PA8000 (96), PentiumPro (95)

# Comprobación de los Operandos



Comprobación de los bits de validez en la ventana o estación de reserva

(Captación de los operandos en la emisión)

- OC: Código de operación
- $R_{S1}$ ,  $R_{S2}$ : Registros fuente
- $R_D$ : Registro de destino
- $O_{S1}$ ,  $O_{S2}$ : Operandos fuente
- $I_{S1}$ ,  $I_{S2}$ : Identificadores de los operandos fuente
- $V_{S1}$ ,  $V_{S2}$ : Bits válidos de los operandos fuente

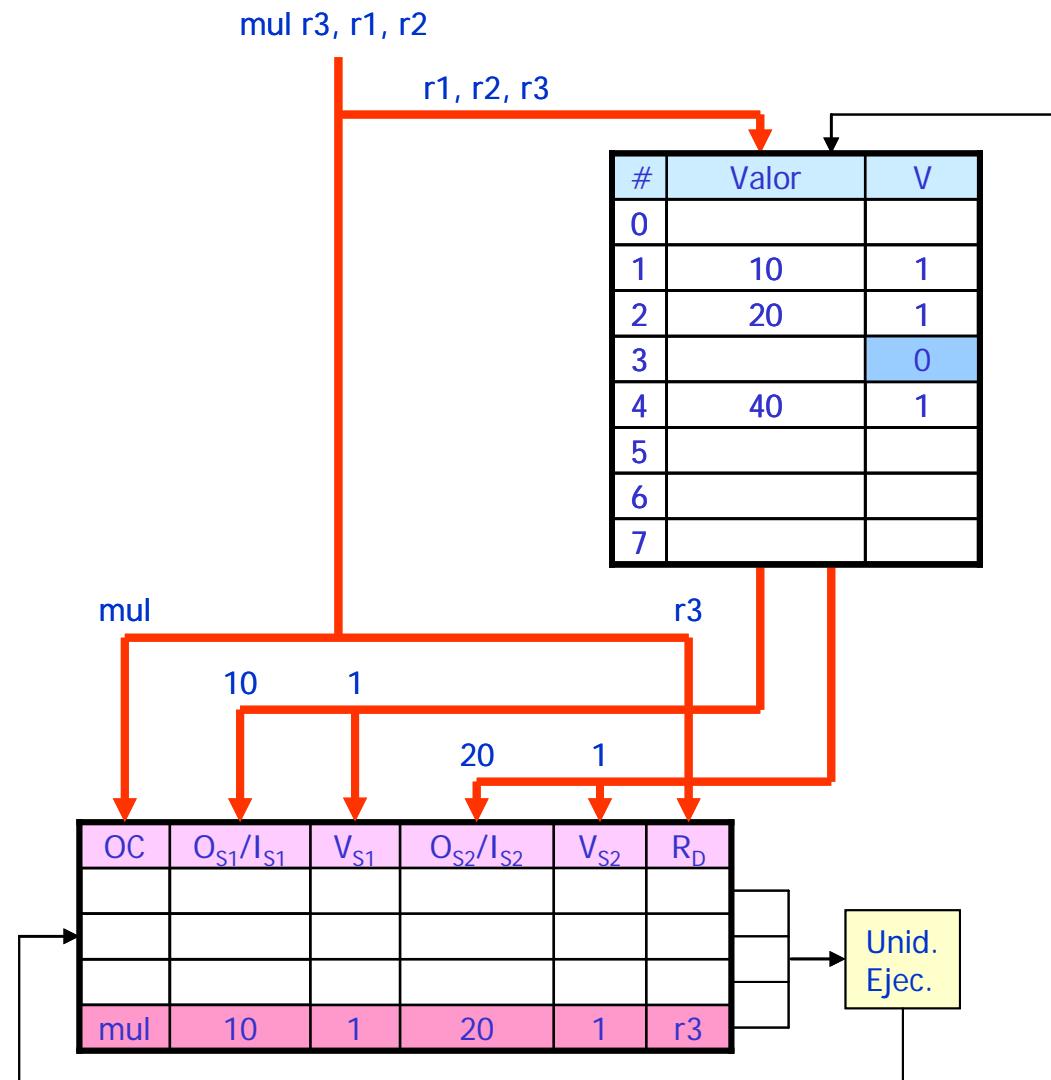
# Ejemplo de uso de Estaciones de Reserva I

Ciclo i: mul r3, r1, r2

Ciclo i+1: add r5, r2, r3  
add r6, r3, r4

Ciclo i:

- Se emite la instrucción de multiplicación, ya decodificada, a la estación de reserva
- Se anula el valor de r3 en el banco de registros
- Se copian los valores de r1 y r2 (disponibles) en la estación de reserva

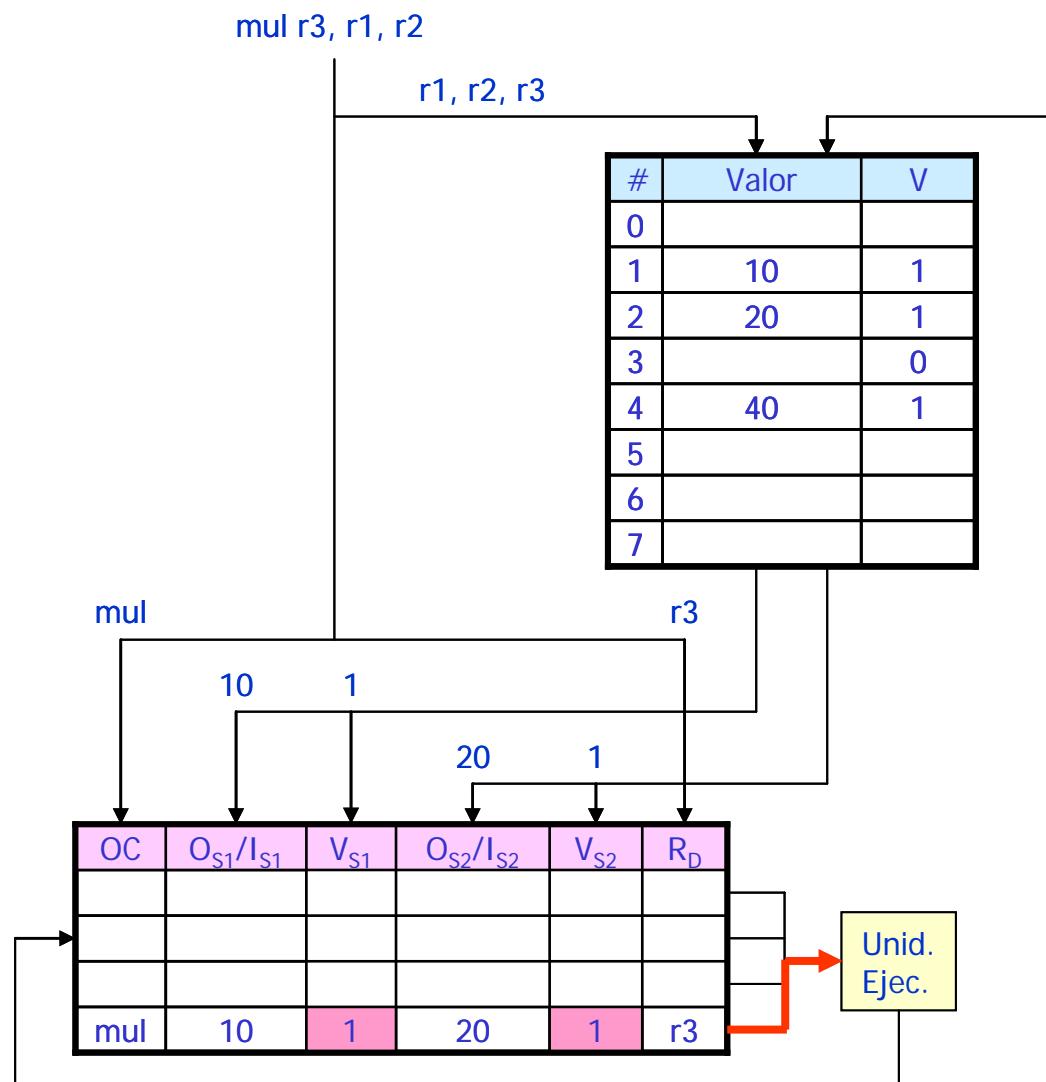


# Ejemplo de uso de Estaciones de Reserva II

Ciclo i: mul r3, r1, r2  
Ciclo i+1: add r5, r2, r3  
add r6, r3, r4

Ciclo i + 1:

- La operación de multiplicación tiene sus operadores preparados ( $V_{S1} = 1$  y  $V_{S2} = 1$ )
- Así que puede enviarse a la unidad de ejecución



# Ejemplo de uso de Estaciones de Reserva III

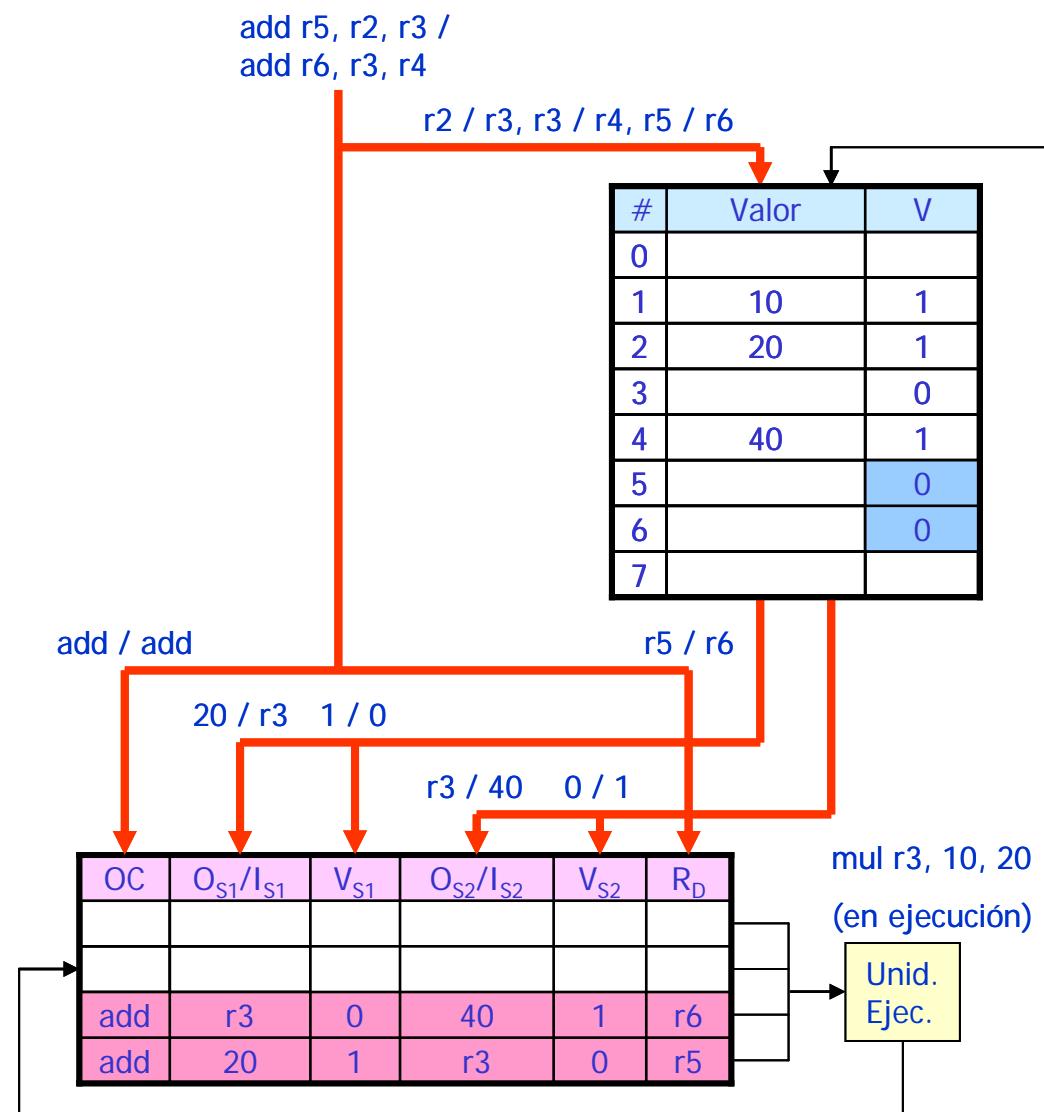
Ciclo i: mul r3, r1, r2

Ciclo i+1: add r5, r2, r3

add r6, r3, r4

Ciclo i + 1 (*cont.*):

- Se emiten las dos instrucciones de suma a la estación de reserva
- Se anulan los valores de r5 y r6 en el banco de registros
- Se copian los valores de los operandos disponibles y los identificadores de los operandos no preparados

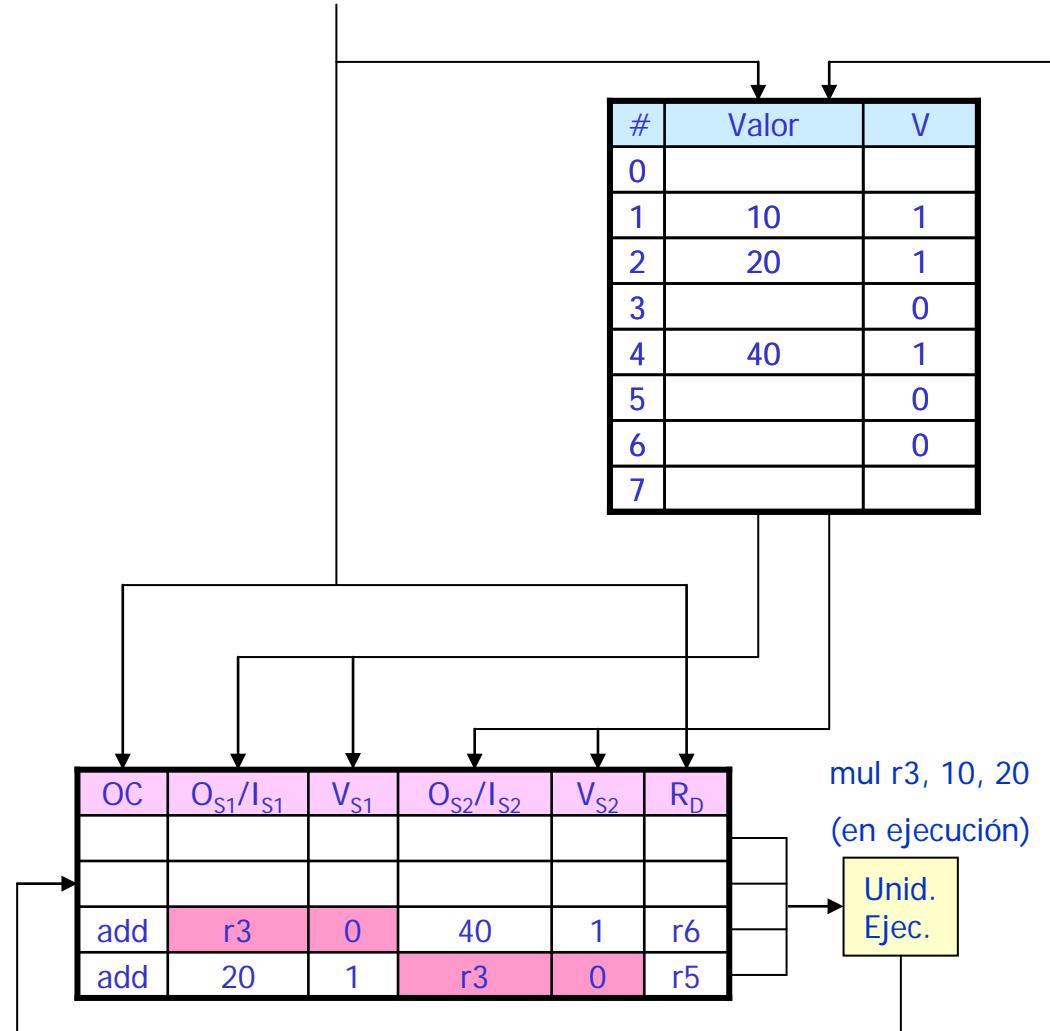


# Ejemplo de uso de Estaciones de Reserva IV

Ciclo i: mul r3, r1, r2  
Ciclo i+1: add r5, r2, r3  
add r6, r3, r4

Ciclos i + 2 .. i + 5:

- La multiplicación sigue ejecutándose
- No se puede ejecutar ninguna suma hasta que esté disponible el resultado de la multiplicación (r3)



# Ejemplo de uso de Estaciones de Reserva V

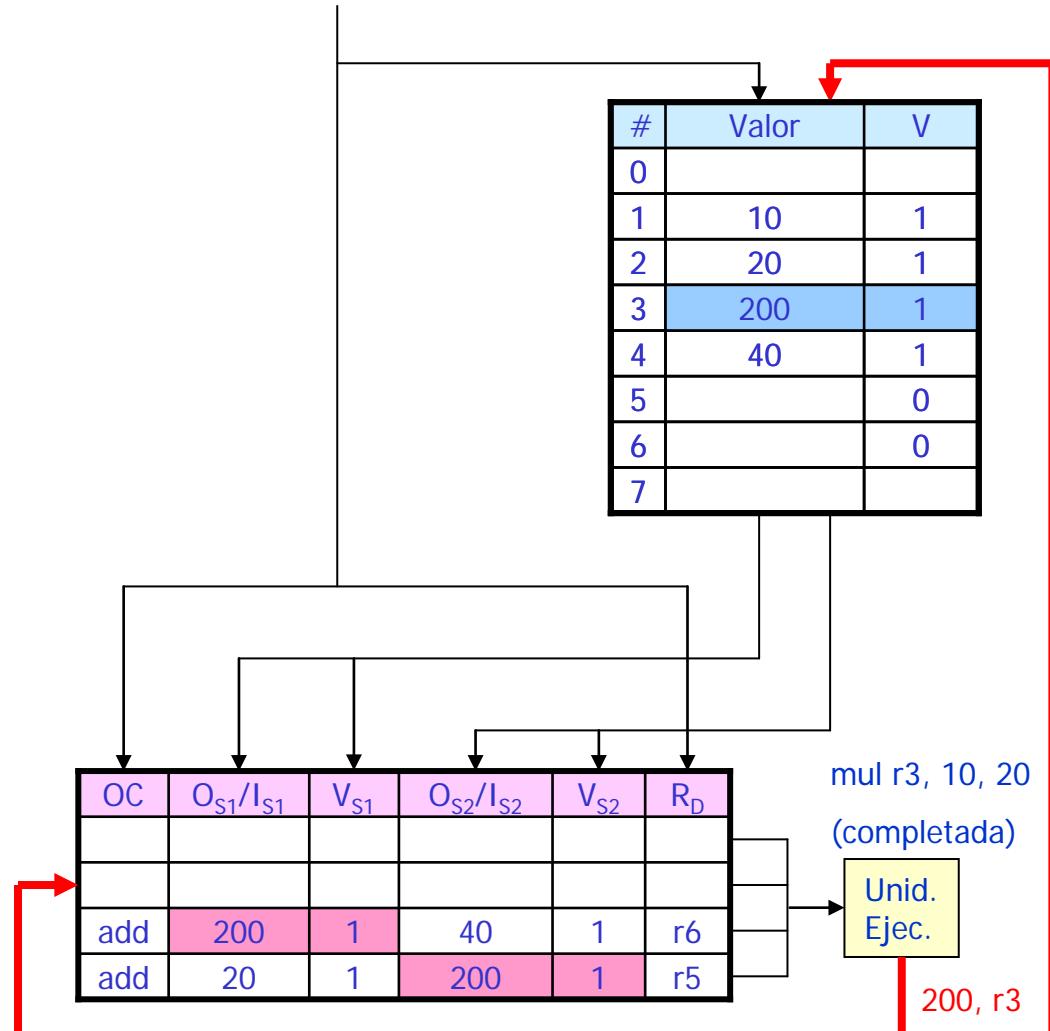
Ciclo i: mul r3, r1, r2

Ciclo i+1: add r5, r2, r3

add r6, r3, r4

Ciclo i + 6:

- Se escribe el resultado de la multiplicación en el banco de registros y en las entradas de la estación de reserva
- Se actualizan los bits de disponibilidad de r3 en el banco de registros y en la estación de reserva

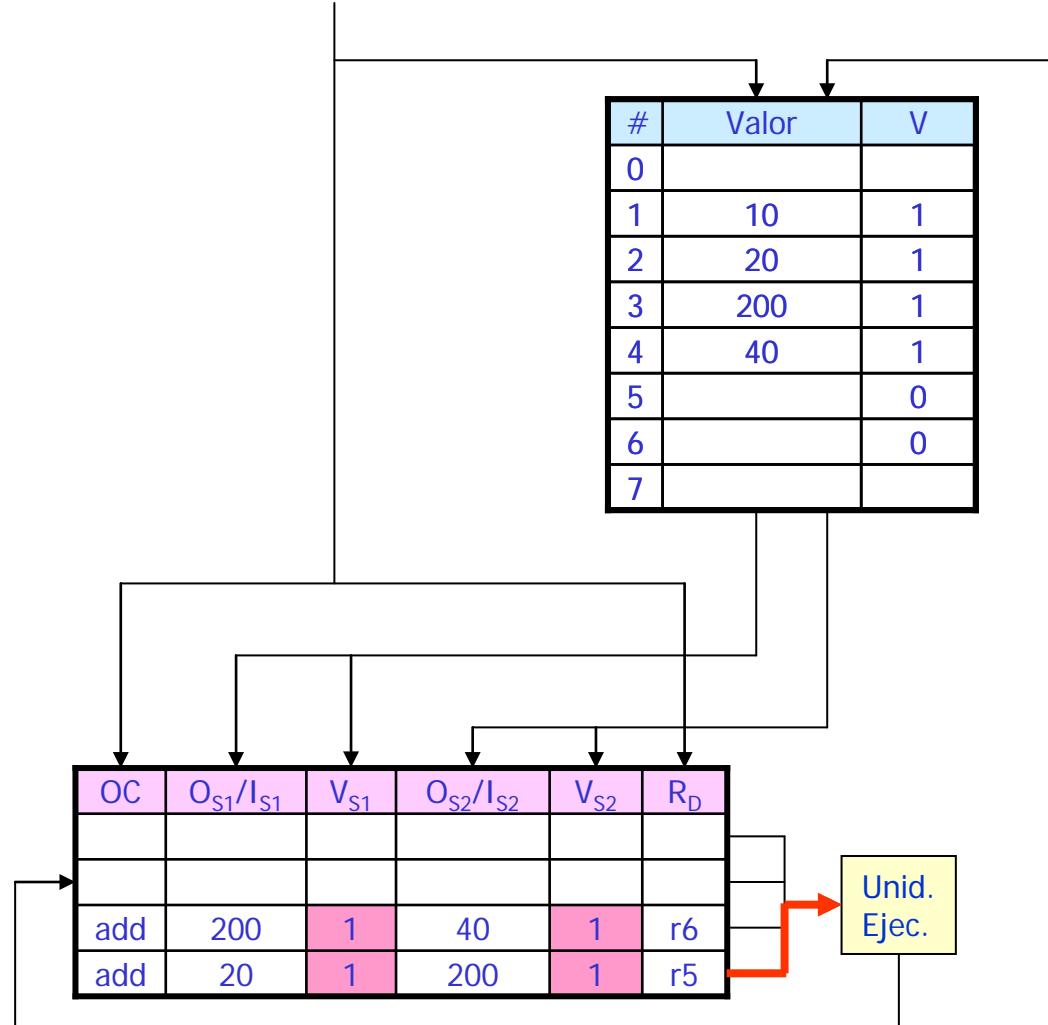


# Ejemplo de uso de Estaciones de Reserva VI

Ciclo i: mul r3, r1, r2  
Ciclo i+1: add r5, r2, r3  
add r6, r3, r4

Ciclo i + 6 (*cont.*):

- Las sumas tienen sus operadores preparados ( $VS1 = 1$  y  $VS2 = 1$ )
- Así que pueden enviarse a la unidad de ejecución



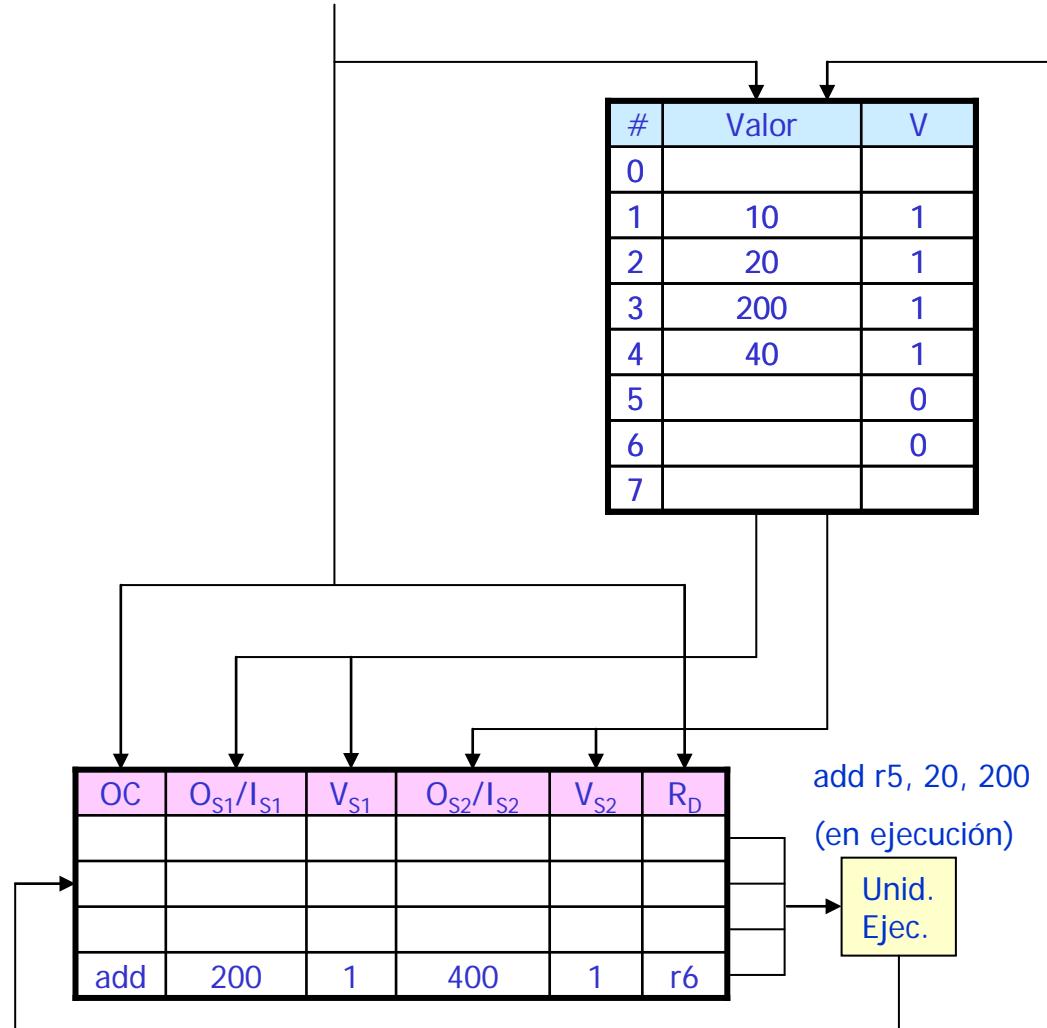
# Ejemplo de uso de Estaciones de Reserva VII

Ciclo i: mul r3, r1, r2

Ciclo i+1: add r5, r2, r3  
add r6, r3, r4

Ciclo i + 6 (*cont.*):

- Como sólo hay una unidad de ejecución, se envía la instrucción más antigua de la estación de reserva, la primera suma



# Contenido de la Lección 11

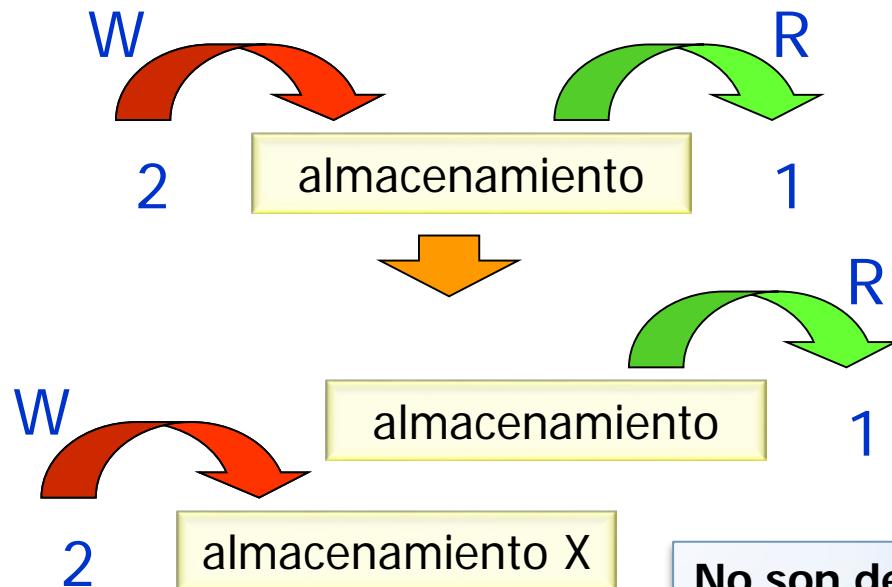
- Introducción: Motivación y Nota Histórica
- Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización
- Cauces superescalares
  - Decodificación Paralela y Predecodificación
  - Emisión Paralela de Instrucciones. Estaciones de Reserva
  - Renombramiento de registros

# Riesgos de Datos

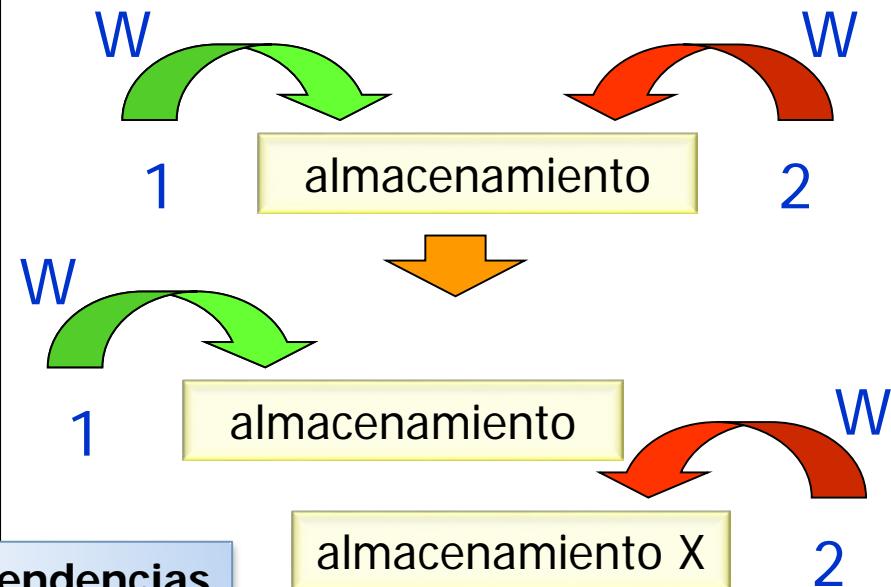
## RAW (Read After Write)



## WAR (Write After Read)



## WAW (Write After Write)



No son dependencias verdaderas

# Renombramiento de Registros

Técnica para **evitar el efecto de las dependencias WAR, o Antidependencias** (en la emisión desordenada) y **WAW, o Dependencias de Salida** (en la ejecución desordenada).

```
R3 := R3 - R5  
R4 := R3 + 1  
R3 := R5 + 1  
R7 := R3 * R4
```

Cada escritura se asigna  
a un registro físico distinto

```
R3b := R3a - R5a  
R4a := R3b + 1  
R3c := R5a + 1  
R7a := R3c * R4a
```

Sólo RAW

R.M. Tomasulo (67)

**Implementación Estática:** Durante la Compilación

**Implementación Dinámica:** Durante la Ejecución (circuitería adicional y registros extra)

## Características de los Buffers de Renombrado

- **Tipos de Buffers** (separados o mezclados con los registros de la arquitectura)
- **Número de Buffers de Renombrado**
- **Mecanismos para acceder a los Buffers** (asociativos o indexados)

## Velocidad del Renombrado

- **Máximo número de nombres asignados por ciclo** que admite el procesador

# Buffers de Renombramiento

Búsq. asoc. de r2

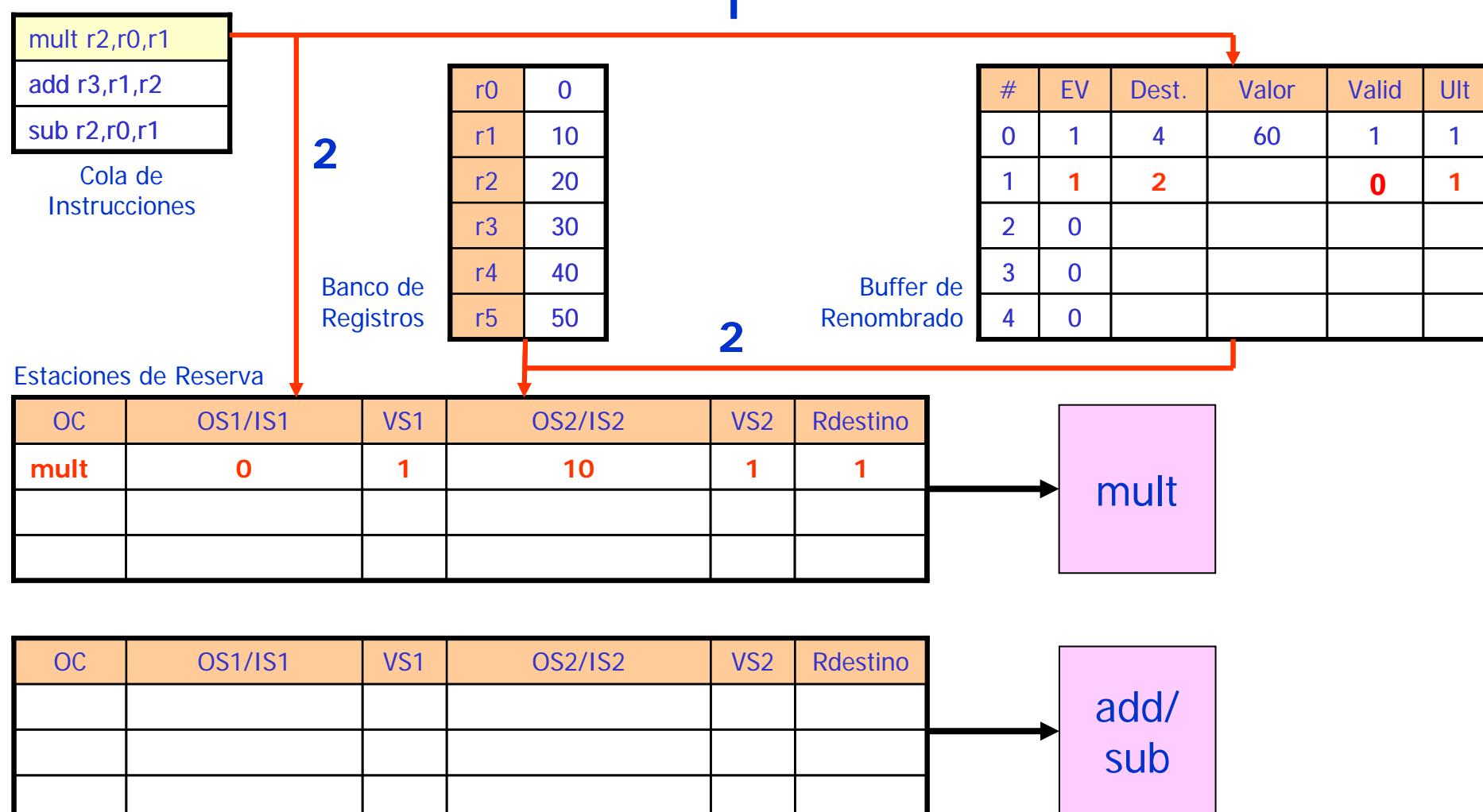
Entrada Válida	Registro Destino	Valor	Valor Válido	Último
1	5	50	1	1
1	12	1200	1	1
1	2	20	1	1
1	1	3	1	1
:	:	:	:	:

Permite varias escrituras pendientes a un mismo registro

El bit de último se utiliza para marcar cual ha sido la más reciente

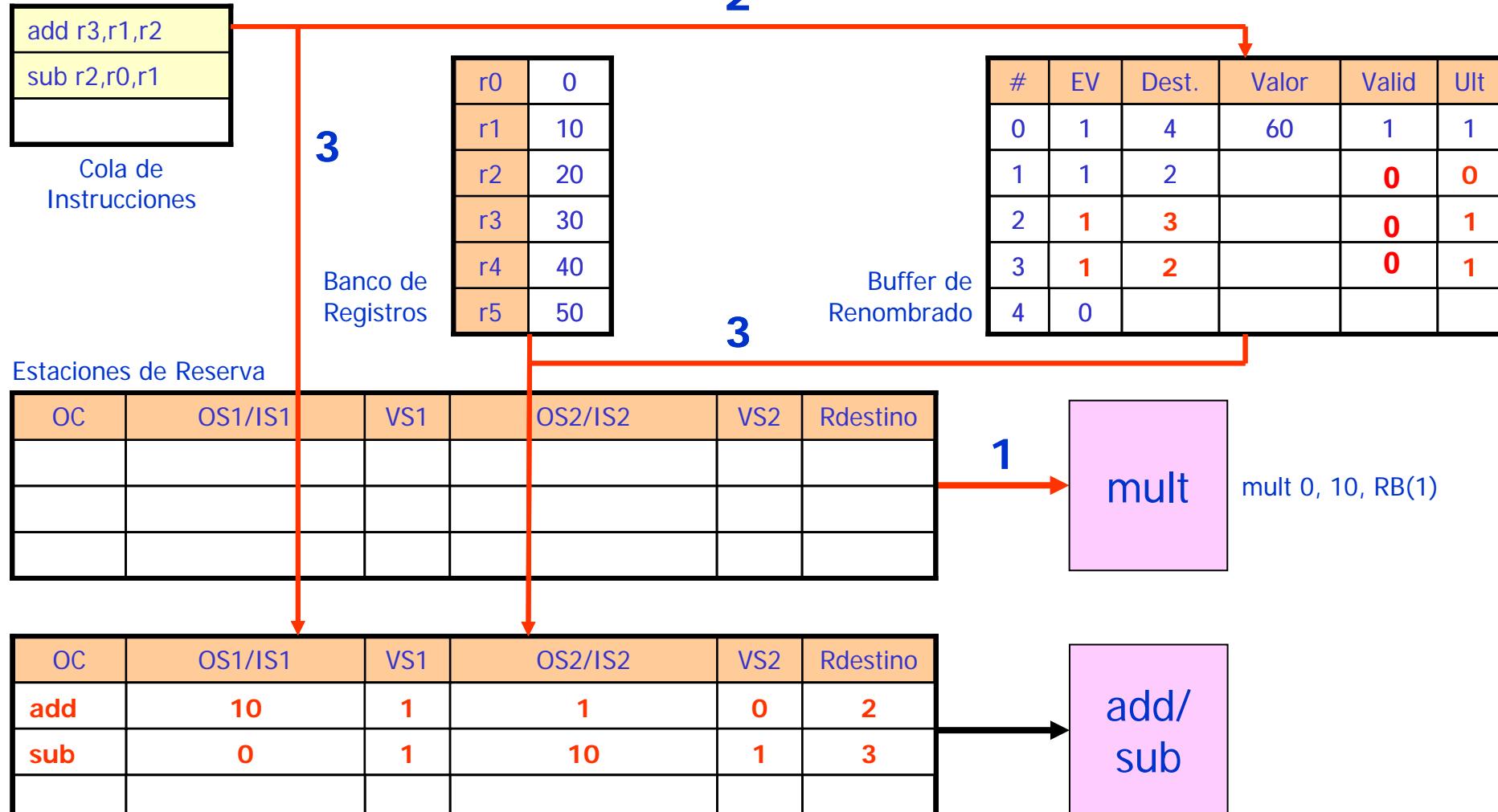
# Algoritmo de Tomasulo I

## Emisión de la multiplicación



# Algoritmo de Tomasulo II

## Envío de la multiplicación y emisión de la suma y la resta 2



# Algoritmo de Tomasulo III

## Envío de la resta


Cola de Instrucciones

r0	0
r1	10
r2	20
r3	30
r4	40
r5	50

Banco de Registros

#	EV	Dest.	Valor	Valid	Ult
0	1	4	60	1	1
1	1	2		0	0
2	1	3		0	1
3	1	2		0	1
4	0				

Buffer de Renombrado

Estaciones de Reserva

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

mult

mult 0, 10, RB(1)

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino
add	10	1	1	0	2

1

add/  
sub

sub 0, 10, RB(3)

# Algoritmo de Tomasulo IV

Termina la resta


Cola de Instrucciones

r0	0
r1	10
r2	20
r3	30
r4	40
r5	50

Banco de Registros

#	EV	Dest.	Valor	Valid	Ult
0	1	4	60	1	1
1	1	2		0	0
2	1	3		0	1
3	1	2	-10	1	1
4	0				

Buffer de Renombrado

Estaciones de Reserva

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

mult

mult 0, 10, RB(1)

1

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino
add	10	1	1	0	2

add/  
sub

sub 0, 10, RB(3)

# Algoritmo de Tomasulo V

Termina la multiplicación


Cola de Instrucciones

r0	0
r1	10
r2	20
r3	30
r4	40
r5	50

Banco de Registros

#	EV	Dest.	Valor	Valid	Ult
0	1	4	60	1	1
1	1	2	0	1	0
2	1	3	0	0	1
3	1	2	-10	1	1
4	0				

Buffer de Renombrado

Estaciones de Reserva

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

mult

mult 0, 10, RB(1)

1



OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino
add	10	1	0	1	2

add/  
sub

# Algoritmo de Tomasulo VI

## Envío de la suma


Cola de Instrucciones

r0	0
r1	10
r2	20
r3	30
r4	40
r5	50

Banco de Registros

#	EV	Dest.	Valor	Valid	Ult
0	1	4	60	1	1
1	1	2	0	1	0
2	1	3		0	1
3	1	2	-10	1	1
4	0				

Buffer de Renombrado

Estaciones de Reserva

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

mult

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

1

add/  
sub

add 10, 0, RB(2)

# Algoritmo de Tomasulo VII

## Termina la suma


Cola de Instrucciones

r0	0
r1	10
r2	20
r3	30
r4	40
r5	50

Banco de Registros

#	EV	Dest.	Valor	Valid	Ult
0	1	4	60	1	1
1	1	2	0	1	0
2	1	3	10	1	1
3	1	2	-10	1	1
4	0				

Buffer de Renombrado

Estaciones de Reserva

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

mult

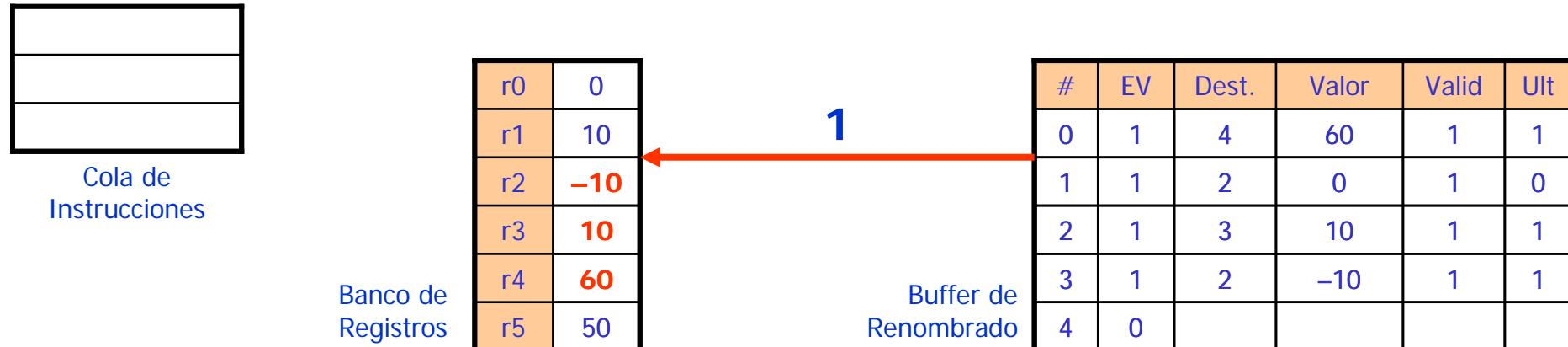
OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

add/  
sub

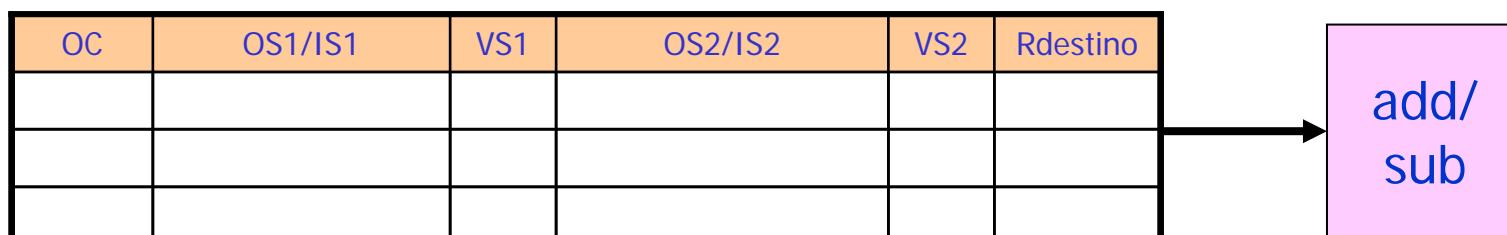
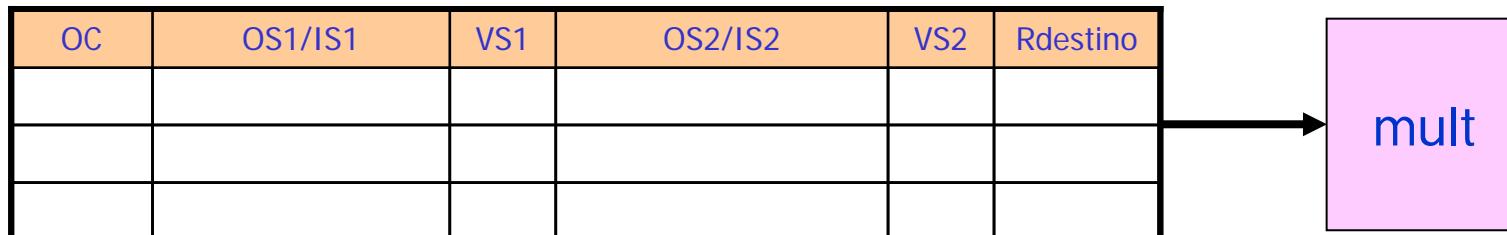
add 10, 0, RB(2)

# Algoritmo de Tomasulo VIII

Se actualizan los registros



Estaciones de Reserva



# Para ampliar .....

## ➤ Artículos de Revistas:

- Sussenguth, E.: “IBM’s ACS-1 Machine”. IEEE Computer, 22:11. Noviembre, 1999
- Anderson, D.W.; Sparacio, F.J.; Tomasulo, R.M.: “The IBM 360 Model 91: Processor phylosophy and instruction handling”. IBM J. Research and Development, 11:1, pp.8-24. Enero, 1967.
- Tomasulo, R.M.: “An efficient algorithm for exploiting multiple arithmetic units”. IBM Res. And. Dev., 11:1, pp.25-33. Enero, 1967.
- Keller, R.M.: “Look-ahead processors”. Computing Surveys, 7, pp.177-196. Diciembre, 1975. (**Se introduce el término renombramiento de registros**).

# Para ampliar .....

## ➤ Artículos de Revistas:

- Agerwala, T.; Cocke, J.: “High Performance Reduced Instruction Set Processors”. IBM Tech. Report, Marzo, 1987. (Descripción del Proyecto América y de la generación del término Superescalar).
- Wall, D.W.: “Limits of instruction-level parallelism”. Research Rep. 93/6, DEC. Noviembre, 1993.

2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores

## Tema 4

# Lección 12. Consistencia del procesador y Procesamiento de Saltos

Material elaborado por los profesores responsables de la asignatura:

Julio Ortega – Mancia Anguita

Licencia Creative Commons



ugr

Universidad  
de Granada



# Lecciones

- Lección 11. Microarquitecturas ILP. Cauces Superescalares
- Lección 12. Consistencia del procesador y Procesamiento de Saltos
  - Consistencia. Reordenamiento
  - Procesamiento especulativo de saltos
- Lección 13. Procesamiento VLIW

# Bibliografía

## ➤ Fundamental

- Capítulo 4. M. Anguita, J. Ortega: "Fundamentos y Problemas de Arquitectura de Computadores". Ed. Avicam. 2016.
- Capítulo 3, Secc. 3.3.4, 3.4, 3.5. J. Ortega, M. Anguita, A. Prieto. *Arquitectura de Computadores*. Thomson, 2005. ESIIT/C.1 ORT arq

## ➤ Complementaria

- Sima and T. Fountain, and P. Kacsuk.  
*Advanced Computer Architectures: A Design Space Approach*. Addison Wesley, 1997. ESIIT/C.1 SIM adv

# Contenido de la Lección 12

- Consistencia. Reordenamiento
- Procesamiento especulativo de saltos

# Consistencia I

En el **Procesamiento de una Instrucción** se puede distinguir entre:

- El **Final de la Ejecución de la Operación** codificada en las instrucciones  
(Se dispone de los resultados generados por las UF pero no se han modificado los registros de la arquitectura).
- El **Final del Procesamiento de la Instrucción** o momento en que se Completa la Instrucción (*Complete o Commit*)  
(Se escriben los resultados de la Operación en los Registros de la Arquitectura. Si se utiliza un Buffer de Reorden, ROB, se utiliza el término Retirar la Instrucción, Retire, en lugar de Completar)

La **Consistencia** de un Programa se refiere a:

- El **orden** en que las instrucciones se **completan**
- El **orden** en que se **accede a memoria** para leer (LOAD) o escribir (STORE)

Cuando se ejecutan instrucciones en paralelo, el orden en que termina (*finish*) esa ejecución puede variar según el orden que las correspondientes instrucciones tenían en el programa pero **debe existir consistencia entre el orden en que se completan las instrucciones y el orden secuencial que tienen en el código de programa.**

# Consistencia II

			Tendencia
<b>Consistencia de Procesador</b>	Débil: Las instrucciones se pueden completar desordenadamente siempre que no se vean afectadas las dependencias	Deben detectarse y resolverse las dependencias	Power1 (90) PowerPC 601 (93) Alpha R8000 (94) MC88110 (93)
	Fuerte: Las instrucciones deben completarse estrictamente en el orden en que están en el programa	Se consigue mediante el uso de ROB	PowerPC 620 PentiumPro (95) UltraSparc (95) K5 (95) R10000 (96)
<b>Consistencia de Memoria</b>	Débil: Los accesos a memoria (Load/Stores) pueden realizarse desordenadamente siempre que no afecten a las dependencias	Deben detectarse y resolverse las dependencias de acceso a memoria	MC88110 (93) PowerPC 620 UltraSparc (95) R10000 (96)
	Fuerte: Los accesos a memoria deben realizarse estrictamente en el orden en que están en el programa	Se consigue mediante el uso del ROB	PowerPC 601 (93) E/S 9000 (92)

Tendencia / Prestaciones

# Reordenamiento Load/Store I

Las instrucciones LOAD y STORE implican cambios en el Procesador y en Memoria

- LOAD:**
- Cálculo de Dirección en ALU o Unidad de Direcciones
  - Acceso a Cache
  - Escritura del Dato en Registro

- STORE:**
- Cálculo de Dirección en ALU o Unidad de Direcciones
  - Esperar que esté disponible el dato a almacenar (en ese momento acaba)

**La Consistencia de Memoria Débil** (reordenación de los accesos a memoria):

- **'Bypass' de Loads/Stores:**

Los Loads pueden adelantarse a los Stores pendientes y viceversa (siempre que no se violen dependencias)

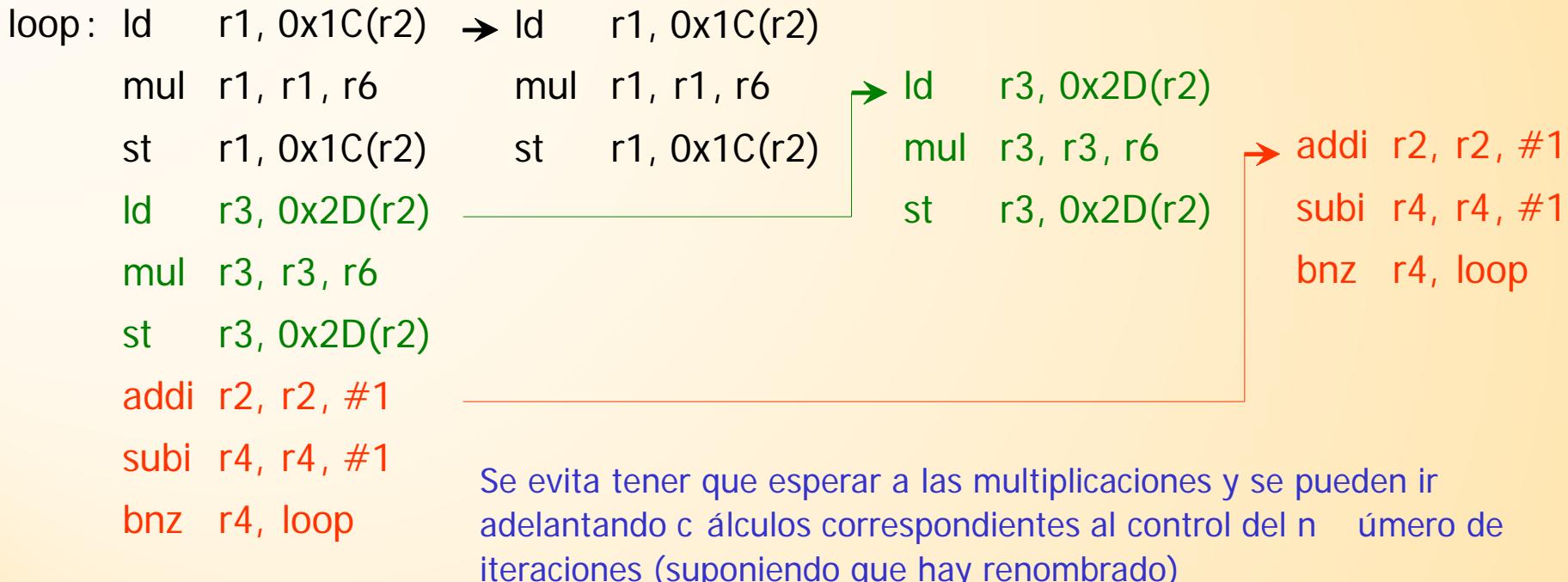
- **Permite los Loads y Stores Especulativos:**

Cuando un Load se adelanta a un Store que le precede antes de que se haya determinado la dirección se habla de Load especulativo. Igual para un Store que se adelanta a un Load o a un Store.

- **Permite ocultar las Faltas de Cache:**

Si se adelanta un acceso a memoria a otro que dio lugar a una falta de cache y accede a Memoria Principal.

# Reordenamiento Load/Store III



Si se tuviera:      st r1,0x1C(r2)

                        ld r3,0x2D(r7)

Las direcciones 0x1C(r2) y 0x2D(r7) podrían coincidir.

Se tendría un **load especulativo**

# Buffer de Reordenamiento (ROB) I

1			
2	instr(n)	f	.....
3	instr(n+1)	x	.....
4	instr(n+2)	f	.....
5	instr(n+3)	x	.....
6	instr(n+4)	x	.....
7	instr(n+5)	i	.....
8	instr(n+6)	i	.....
9			
10			

Cola

(Las instrucciones se retiran desde aquí)

Cabecera (Las instrucciones que se decodifican se introducen a partir de aquí)

*La gestión de interrupciones y la ejecución especulativa se pueden implementar fácilmente mediante el ROB*

- El puntero de cabecera apunta a la siguiente posición libre y el **puntero de cola a la siguiente instrucción a retirar**.
- Las instrucciones **se introducen en el ROB en orden de programa estricto** y pueden estar marcadas como **emitidas (issued, i)**, **en ejecución (x)**, o **finalizada su ejecución (f)**
- Las **instrucciones sólo se pueden retirar** (se produce la finalización con la escritura en los registros de la arquitectura) **si han finalizado**, y todas las que les preceden también.
- La **consistencia se mantiene porque sólo las instrucciones que se retiran del ROB se completan** (escriben en los registros de la arquitectura) y se retiran en el orden estricto de programa.

# Buffer de Reordenamiento (ROB) II

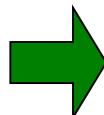
## Ejemplo de uso del ROB

I1: mult r1, r2, r3

I2: st r1, 0x1ca

I3: add r1, r4, r3

I4: xor r1, r1, r3



## Dependencias:

**RAW:** (I1,I2), (I3,I4)

**WAR:** (I2,I3), (I2,I4)

**WAW:** (I1,I3), (I1,I4), (I3,I4)

- I1: Se puede empezar a ejecutar inmediatamente (se suponen disponibles **r2** y **r3**)
- I2: Se envía a la unidad de almacenamiento hasta que esté disponible **r1**
- I3: Se puede empezar a ejecutar inmediatamente (se suponen disponibles **r4** y **r3**)
- I4: Se envía a la estación de reserva de la ALU para esperar a **r1**

Estación de Reserva (Unidad de Almacenamiento)

codop	dirección	op1	ok1
st	0x1ca	3	0

Estación de Reserva (ALU)

codop	dest	op1	ok1	op2	ok2
xor	6	5	0	[r3]	1

Líneas del ROB

# Buffer de Reordenamiento (ROB) III

## Ciclo 7

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	-	0	x
6	xor	10	r1	int_alu	-	0	i

**Ciclo 9** No se puede retirar **add** aunque haya finalizado su ejecución

#	codop	Nº Inst.	Reg.Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	-	0	x

**Ciclo 10** Termina **xor**, pero todavía no se puede retirar

#	codop	Nº Inst.	Reg.Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

# Buffer de Reordenamiento (ROB) IV

## Ciclo 12

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	33	1	f
4	st	8	-	store	-	1	f
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

## Ciclo 13

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

- Se ha supuesto que se pueden retirar dos instrucciones por ciclo.
- Tras finalizar las instrucciones **mult** y **st** en el ciclo 12, se retirarán en el ciclo 13.
- Después, en el ciclo 14 se retirarán las instrucciones **add** y **xor**.

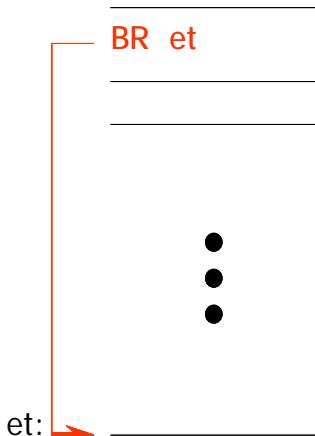
# Contenido de la Lección 12

- Consistencia. Reordenamiento
- Procesamiento especulativo de saltos

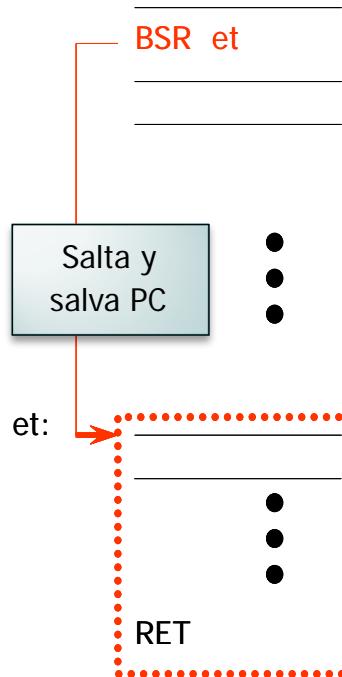
# Clasificación de los Saltos

## Saltos Incondicionales

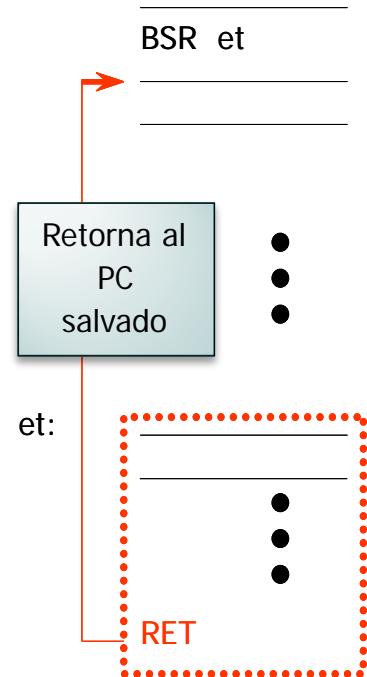
### Salto Incondicional



### Llamada a Subrutina

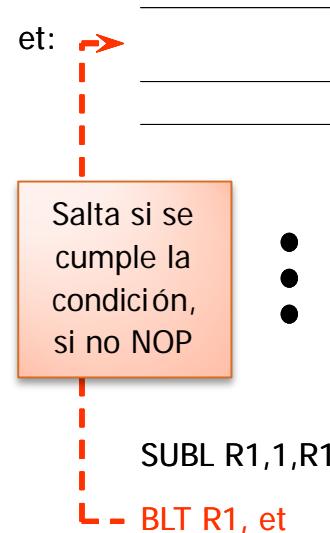


### Retorno de Subrutina



## Saltos Condicionales

### Condición de Bucle



### Otro Salto Condicional



Salto hacia atrás o hacia adelante

# Alternativas para la condición de salto

## Estado del Resultado

Existen bits de Estado que se modifican al realizar operaciones o mediante operaciones que comprueban específicamente el valor de los registros

```
add r1,r2,r3  
beq cero      ↗  
div r5,r4,r1  
.....
```

Dependencia  
que limita las  
Prestaciones  
en VLIW y  
superescalares

cero:

### Ejemplos:

IBM/360, PDP-11, VAX, X-86,  
Pentium, PowerPC, Sparc

## Comprobación Directa

Los resultados de las operaciones se comprueban directamente respecto a las condiciones específicas mediante instrucciones específicas

### Dos Instrucciones

```
add r1,r2,r3  
cmpeq r7, r1, 0  
bt r7, cero  
div r5,r4,r1  
.....
```

cero:

### Ejemplos:

Am 29000

### Una Instrucción

```
add r1,r2,r3  
bz r1, cero  
div r5,r4,r1  
.....
```

cero:

### Ejemplos:

Cray, MIPS, MC881X0,  
HP PA, DEC Alpha

# Aspectos del Procesamiento de Saltos en un procesador Superescalar

AC ATC

- Detección de la Instrucción de Salto
  - Cuanto antes se detecte que una instrucción es de salto menor será la posible penalización. Los saltos se detectan usualmente en la fase de decodificación e incluso en la captación (si hay predecodificación) .
- Gestión de los Saltos Condicionales no Resueltos
  - Si en el momento en que la instrucción de salto evalúa la condición de salto ésta no se haya disponible se dice que el salto o la condición no se ha resuelto. Para resolver este problema se suele utilizar el procesamiento especulativo del salto.
- Acceso a las Instrucciones destino del Salto
  - Hay que determinar la forma de acceder a la secuencia a la que se produce el salto

**El efecto de los saltos en los procesadores superescalares es más pernicioso** ya que, al emitirse varias instrucciones por ciclo, *prácticamente en cada ciclo* puede haber una instrucción de salto.

# Gestión de Saltos Condicionales no resueltos

## Gestión de Saltos Condicionales no Resueltos

(Una condición de salto no se puede comprobar si no se ha terminado de evaluar)

<b>Gestión de Saltos Condicionales no Resueltos</b>  (Una condición de salto no se puede comprobar si no se ha terminado de evaluar)	<b>Bloqueo del Procesamiento del Salto</b>	Se bloquea la instrucción de salto hasta que la condición esté disponible  (68020, 68030, 80386)
	<b>Procesamiento Especulativo de los Saltos</b>	La ejecución prosigue por el camino más probable (se especula sobre las instrucciones que se ejecutarán). Si se ha errado en la predicción hay que recuperar el camino correcto.  (Típica en los procesadores superescalares actuales)
	<b>Múltiples Caminos</b>	Se ejecutan los dos caminos posibles después de un salto hasta que la condición de salto se evalúa. En ese momento se cancela el camino incorrecto.  (Máquinas VLIW experimentales: Trace/500 , URPR2)
<b>Evitar saltos condicionales</b>	<b>Ejecución Vigilada (<i>Guarded Exec.</i>)</b>	Se evitan los saltos condicionales incluyendo en la arquitectura instrucciones con operaciones condicionales  (IBM VLIW, Cydra-5, Pentium, HP PA, Dec Alpha)

# Esquemas de Predicción de Salto

## Predicción Fija

Se toma siempre la misma decisión: el salto siempre se realiza, '*taken*', o no, '*not taken*'

## Predicción Verdadera

La decisión de si se realiza o no se realiza el salto se toma mediante:

- **Predicción Estática:**

Según los atributos de la instrucción de salto (el código de operación, el desplazamiento, la decisión del compilador)

- **Predicción Dinámica:**

Según el resultado de ejecuciones pasadas de la instrucción (historia de la instrucción de salto)

↓  
**Prestaciones**

# Predicción Estática

## Predicción basada en el Código de Operación

Para ciertos códigos de operación (ciertos saltos condicionales específicos) se predice que el salto se toma, y para otros que el salto no se toma

MC88110 (93)  
PowerPC 603(93)

## Predicción basada en el Desplazamiento del Salto

Si el desplazamiento es positivo (salto hacia delante) se predice que no se toma el salto y si el desplazamiento es negativo (salto hacia atrás) se predice que se toma.

Alpha 21064 (92)  
PowerPC 603 (93)

## Predicción dirigida por el Compilador

El compilador es el que establece la predicción fijando, para cada instrucción el valor de un bit específico que existe en la instrucción de salto (bit de predicción)

AT&T 9210 (93)  
PowerPC 603 (93)  
MC88110 (93)

## Ejemplo: Predicción Estática en el MC88110

Formato	Instrucción		Predicción
	Condición Especificada	Bit 21 de la Instr.	
bcnd <i>(Branch Conditional)</i>	$\neq 0$	1	Tomado
	$= 0$	0	No Tomado
	$> 0$	1	Tomado
	$< 0$	0	No Tomado
	$\geq 0$	1	Tomado
	$\leq 0$	0	No Tomado
	bb1 <i>(Branch on Bit Set)</i>		Tomado
	bb0 <i>(Branch on Bit Clear)</i>		No Tomado

# Predictión Dinámica

- La predicción para cada instrucción de salto puede cambiar cada vez que se va a ejecutar ésta según la historia previa de saltos tomados/no-tomados para dicha instrucción.
- El presupuesto básico de la predicción dinámica es que es más probable que el resultado de una instrucción de salto sea similar al que se tuvo en la última (o en las  $n$  últimas ejecuciones)
- Presenta mejores prestaciones de predicción, aunque su implementación es más costosa

## • Predicción Dinámica Implícita

No hay bits de historia propiamente dichos sino que se almacena la dirección de la instrucción que se ejecutó después de la instrucción de salto en cuestión

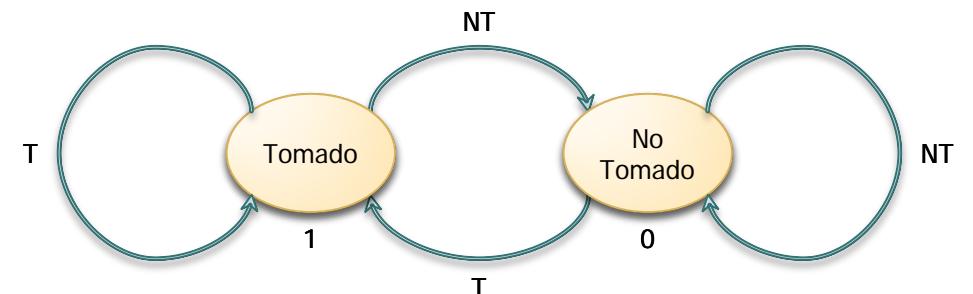
## • Predicción Dinámica Explícita

Para cada instrucción de salto existen unos bits específicos que codifican la información de historia de dicha instrucción de salto

# Ejemplos de Procedimientos Explícitos de Predicción Dinámica de Saltos

## Predicción con 1 bit de historia

La designación del estado, Tomado (1) o No Tomado (0), indica lo que se predice, y las flechas indican las transiciones de estado según lo que se produce al ejecutarse la instrucción (T o NT)

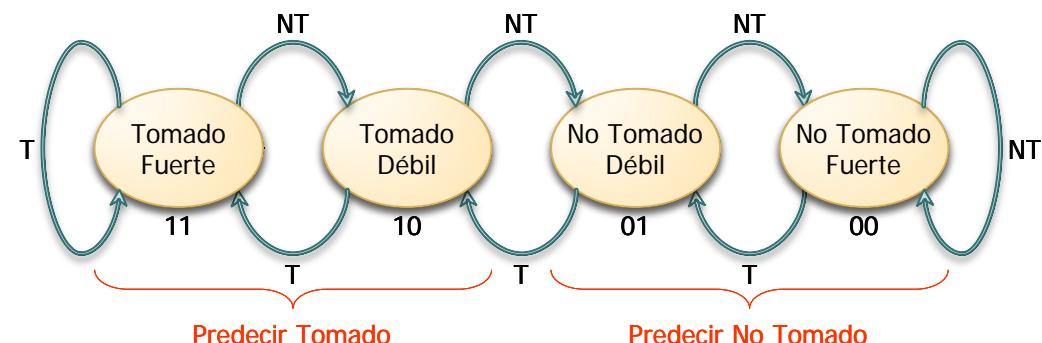


## Predicción con 2 bits de historia

Existen cuatro posibles estados. Dos para predecir Tomado y otros dos para No Tomado

La primera vez que se ejecuta un salto se inicializa el estado con predicción estática, por ejemplo 11

Las flechas indican las transiciones de estado según lo que se produce al ejecutarse la instrucción (T o NT)

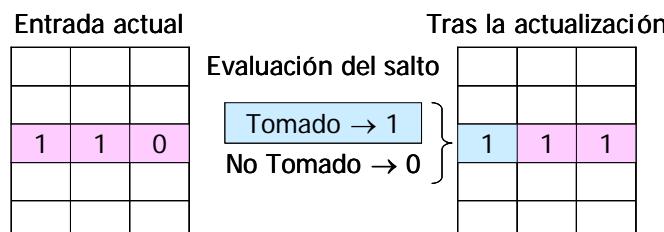


## Predicción con 3 bits de historia

Cada entrada guarda las últimas ejecuciones del salto

Se predice según el bit mayoritario (por ejemplo, si hay mayoría de unos en una entrada se predice salto)

La actualización se realiza en modo FIFO, los bits se desplazan, introduciéndose un 0 o un 1 según el resultado final de la instrucción de salto



# Extensión del Procesamiento Especulativo

- Tras la predicción, **el procesador continúa ejecutando instrucciones especulativamente hasta que se resuelve la condición.**
- El intervalo de **tiempo entre el comienzo de la ejecución especulativa y la resolución** de la condición **puede variar considerablemente y ser bastante largo.**
- En los procesadores superescalares, que pueden emitir varias instrucciones por ciclo, **pueden aparecer más instrucciones de salto condicional no resueltas durante la ejecución especulativa.**
- Si el número de instrucciones que se ejecutan especulativamente es muy elevado y la predicción es incorrecta, la penalización es mayor.

Así, **cuanto mejor es el esquema de predicción mayor puede ser el número de instrucciones ejecutadas especulativamente.**



- **Nivel de Especulación:** Número de Instrucciones de Salto Condicional sucesivas que pueden ejecutarse especulativamente (si se permiten varias, hay que guardar varios estados de ejecución). Ejemplos: Alpha21064, PowerPC 603 (1); Power 2 (2); PowerPC 620 (4); Alpha 21164 (6)
- **Grado de Especulación:** Hasta qué etapa se ejecutan las instrucciones que siguen en un camino especulativo después de un salto. Ejemplos: Power 1 (Captación); PowerPC 601 (Captación, Decodificación, Envío); PowerPC 603 (Todas menos la finalización)

# Instrucciones de Ejecución Condicional (Guarded Execution)

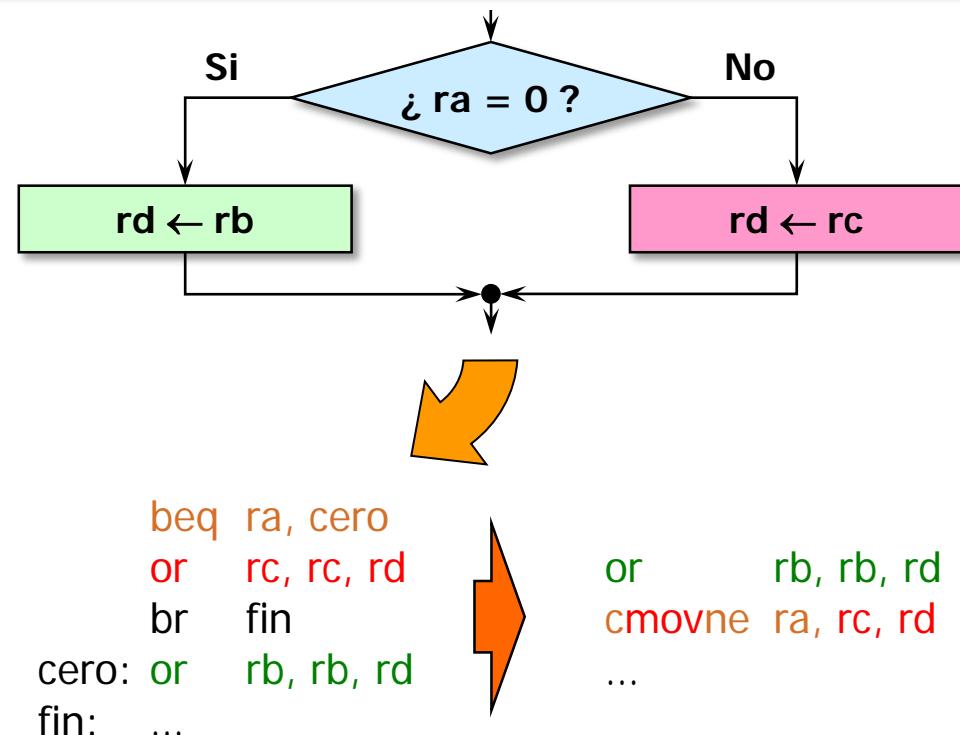
- Se pretende **reducir el número de instrucciones de salto** incluyendo en el **repertorio máquina instrucciones con operaciones condicionales** ('conditional operate instructions' o 'guarded instructions')
- Estas instrucciones tienen dos partes: la **condición** (denominada *guardia*) y la parte de **operación**

## Ejemplo: cmovexx de Alpha

cmove<sub>xx</sub> ra.rq, rb.rq, rc.wq

- xx** es una condición
- ra.rq, rb.rq** enteros de 64 bits en registros ra y rb
- rc.wq** entero de 64 bits en rc para escritura
- El registro **ra** se comprueba en relación a la condición **xx** y si se verifica la condición **rb** se copia en **rc**.

Sparc V9, HP PA, y Pentium ofrecen también estas instrucciones.



# Para ampliar .....

## ➤ Artículos de Revistas:

- Smith, J.E.: "A study of branch prediction strategies". Proc. Eighth Symp. on Computer Architecture, pp.135-148, 1981.
- Smith, J.E.; Lee, J.: "Branch prediction strategies and branch target buffer design". IEEE Computer, pp.6-22. Enero, 1984.
- Yeh, T.; Patt, Y.N.: "A comparison of dynamic branch prediction that use two levels of branch history". Proc. 20th Symp. On Computer Architecture, pp.257-266, 1993.
- Mahlke, S.A.: "A comparison of full and partial predicated execution support for ILP processors". Proc. 22nd Symp. On Computer Architecture, pp.138-150. Junio, 1995.
- Smith, J.E.; Plezskun, A.R.: "Implementing Precise Interrupts in Pipelined Processors". IEEE Trans. On Computers, Vol.37, No.5, pp.562-573. Mayo, 1988.
- Torng, H.C.; Day, M.: "Interrupt Handling of Out-of-Order Execution Processors". IEEE Trans. On Computers, Vol.42, No.1, pp.122-127. Enero, 1993.
- Walker, W.; Cragon, H.G.: "Interrupt Processing in Concurrent Processors". IEEE Computer, pp.36-46. Junio, 1995.
- Moudgil, M.; Vassiliadis, S.: "Precise Interrupts". IEEE Micro, pp.58-67. Febrero, 1996.
- Samadzadeh, M.; Garalnabi, L.E.: "Hardware/Software Cost Analysis of Interrupt Processing Strategies". IEEE Micro, pp.69-76. Mayo-Junio, 2001.

# Para ampliar ...

- Páginas Web:
  - <http://www.delphion.com/research>  
(Se pueden consultar las patentes sobre elementos de los procesadores. En particular las relativas a ROBs y elementos para mantener la consistencia secuencial en el procesador)
  - [http://www.aceshardware.com/Spades/read.php?article\\_id=51](http://www.aceshardware.com/Spades/read.php?article_id=51)  
(Artículo: J. De Gelas: “The Secrets of High Performance CPUs, Part2”)
- Artículos de Revistas:
  - Smith, J.E.; Pleszkun, A.R.: “Implementing precise interrupts in pipelined processors”. IEEE Trans. Computers, Vol.C37, pp.562-573. Mayo, 1988.  
(Se propone por primera vez el uso del ROB)

2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores

## Tema 4

## Lección 13. Procesamiento VLIW

Material elaborado por los profesores responsables de la asignatura:  
Julio Ortega – Mancia Anguita

*Licencia Creative Commons*



*ugr*

Universidad  
de Granada



# Bibliografía

## ➤ Fundamental

- Capítulo 4. M. Anguita, J. Ortega. *Fundamentos y Problemas de Arquitectura de Computadores*. Ed. Avicam, 2016
- Capítulo 5. J. Ortega, M. Anguita, A. Prieto. *Arquitectura de Computadores*. Thomson, 2005. ESIIT/C.1 ORT arq

## ➤ Complementaria

- Sima and T. Fountain, and P. Kacsuk.  
*Advanced Computer Architectures: A Design Space Approach*. Addison Wesley, 1997. ESIIT/C.1 SIM adv

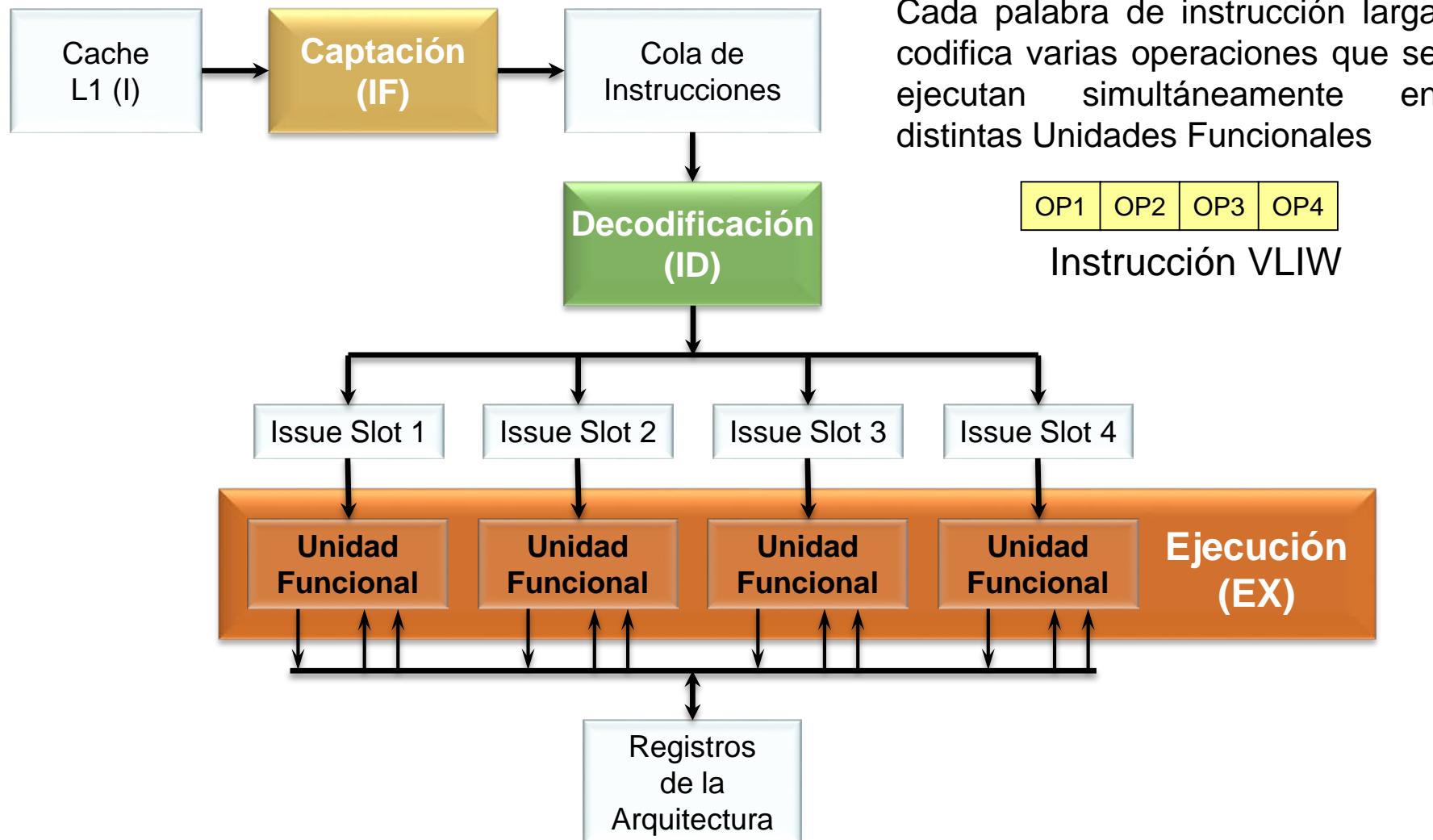
# Contenido de la Lección 13

- Características generales y motivación (ILP hardware vs. ILP software)
- Planificación estática
- Procesamiento especulativo

# Características generales de los procesadores VLIW I

- Las arquitecturas VLIW utilizan varias unidades funcionales independientes.
- En lugar de tratar de enviar varias instrucciones independientes a las unidades funcionales, una arquitectura VLIW empaqueta varias operaciones en una única instrucción (muy larga, Very Long, por ejemplo, entre 112 y 128 bits) u ordena las instrucciones en el paquete de emisión con las mismas restricciones de independencia.
- La decisión de qué instrucciones se deben emitir simultáneamente corresponde al compilador (hardware más sencillo que el de un superescalar).
- Las ventajas de la aproximación VLIW crecen a medida que se pretende emitir más instrucciones por ciclo (el hardware adicional para un superescalar que emita dos instrucciones por ciclo es relativamente pequeño, pero crece a medida que se pretenden emitir más instrucciones por ciclo)

# Características generales de los procesadores VLIW II



# Contenido de la Lección 13

- Características generales y motivación (ILP hardware vs. ILP software)
- Planificación estática
- Procesamiento especulativo

# El papel del Compilador

## Planificación estática

Necesita asistencia del compilador, que puede realizar renombrados, reorganizaciones de código, etc., para mejorar el uso de los recursos disponibles, el esquema de predicción de saltos,...



VLIW

## Planificación dinámica

Requiere menos asistencia del compilador pero más coste hardware. Facilita la portabilidad del código entre la misma familia de procesadores



Superescalar

El compilador construye paquetes de instrucciones (*ventanas de emisión*) sin dependencias, de forma que el procesador no necesita comprobarlas explícitamente.

# Planificación Estática I

```
for ( i = 1000 ; i > 0 ; i = i - 1 )  
    x[i] = x[i] + s;
```



```
loop:    ld      f0, 0(r1)  
          addd   f4, f0, f2  
          sd      f4, 0(r1)  
          subui r1, r1, #8  
          bne    r1, loop
```

Existen dependencias RAW entre cada dos instrucciones consecutivas, y además existe una instrucción de salto que controla el final del bucle:  
Parece que no se puede aprovechar mucho ILP

# Planificación Estática II

Suponiendo que hay suficientes unidades funcionales para la suma, y que cuando hay dependencias de tipo RAW (las de tipo WAW y WAR las puede evitar el compilador mediante renombrado) los retardos introducidos son:

Instrucción que produce el resultado	Instrucción que usa el resultado	Latencia
Operación FP	Operación FP	3
Operación ALU	Store	2
Load	Operación FP	1
Load	Store	0

	Opción 1 (Sin desenrollar)	Opción 2 (Sin desenrollar)	Ciclos
loop	ld f0,0(r1)	ld f0,0(r1)	1
	-----	subui r1,r1,#8	2
	addd f4,f0,f2	addd f4,f0,f2	3
	-----	-----	4
	-----	bne r1,loop	5
	sd f4,0(r1)	sd f4,8(r1)	6
	subui r1,r1,#8	-----	7
	-----	-----	8
	bne r1,loop	-----	9
	-----	-----	10

# Planificación Estática III

	Instrucción Entera	Instrucción FP	Load/Store	Ciclo
loop	subui r1, r1, #8		ld f0, 0(r1)	1
				2
		addd f4, f0, f2		3
				4
	bne r1, loop			5
			sd f4, 0(r1)	6
Slot 1		Slot 2	Slot 3	

La arquitectura VLIW no ganaría nada frente a la opción 2 del bucle sin desenrollar. Además, se necesitarían 12 palabras en el código VLIW frente a las 5 que se utilizaría en la codificación escalar.

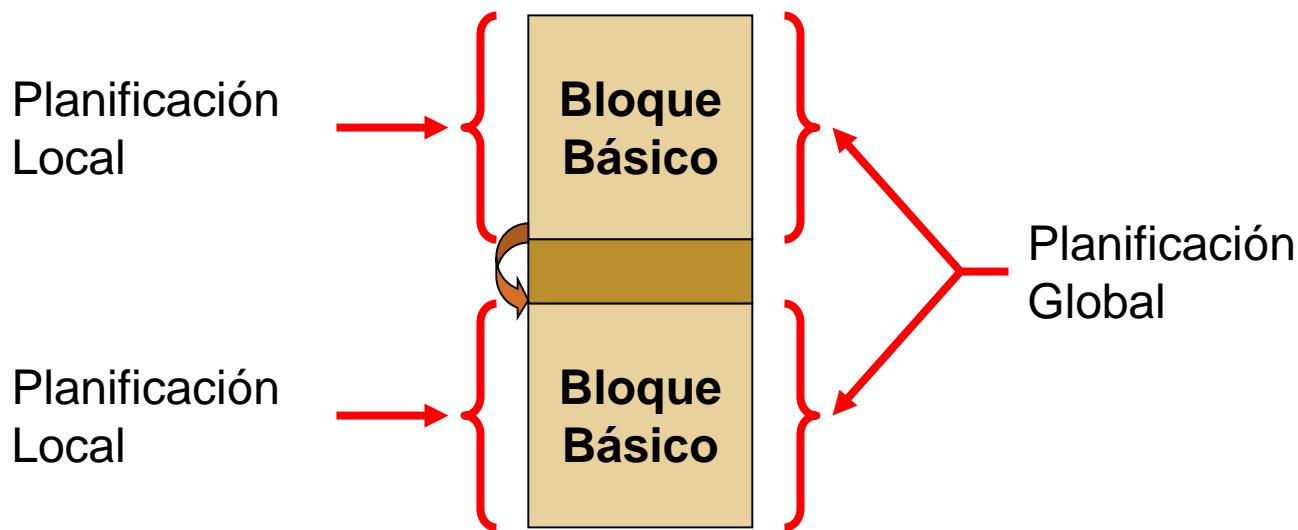
# Planificación estática local y global

## ➤ Planificación local:

- actúa sobre un bloque básico (mediante desenrollado de bucles y planificación de las instrucciones del cuerpo aumentado del bucle).

## ➤ Planificación global:

- actúa considerando bloques de código entre instrucciones de salto.



# Planificación Estática Local

- Desenrollado de bucles:
  - Al desenrollar un bucle se crean bloques más largos, lo que facilita la planificación local de sus sentencias
  - Además de disponer de más sentencias, éstas suelen ser independientes, ya que operan sobre diferentes datos
- Segmentación software (software pipelining):
  - Se reorganizan los bucles de forma que cada iteración del código transformado contiene instrucciones tomadas de distintas iteraciones del bloque original
  - De esta forma se separan las instrucciones dependientes en el bucle original entre diferentes iteraciones del bucle nuevo

# Planificación Estática con Desenrollado de Bucles I

```
for ( i = 1000 ; i > 0 ; i = i - 1 )  
    x[i] = x[i] + s;
```



```
loop:    ld      f0, 0(r1)  
         addd   f4, f0, f2  
         sd     f4, 0(r1)  
         subui r1, r1, #8  
         bne    r1, loop
```



```
loop:    ld      f0, 0(r1)  
         ld      f6, -8(r1)  
         ld      f10, -16(r1)  
         ld     f14, -24(r1)  
         ld     f18, -32(r1)  
         addd   f4, f0, f2  
         addd   f8, f6, f2  
         addd   f12, 10, f2  
         addd   f16, f14, f2  
         addd   f20, f18, f2  
         sd     f4, 0(r1)  
         sd     f8, -8(r1)  
         sd     f12, -16(r1)  
         sd     f16, -24(r1)  
         sd     f20, -32(r1)  
         subui r1, r1, #40  
         bne    r1, loop
```

Existen dependencias RAW entre cada dos instrucciones consecutivas, y además existe una instrucción de salto que controla el final del bucle:  
Parece que no se puede aprovechar mucho ILP

El desenrollado pone de manifiesto un mayor paralelismo ILP

# Planificación Estática con Desenrollado de Bucles II

	Instrucción Entera	Instrucción FP	Load/Store	Ciclo
loop			ld f0, 0(r1)	1
			ld f6, 0(r1)	2
		addd f4, f0, f2	ld f10, -16(r1)	3
		addd f8, f6, f2	ld f14, -24(r1)	4
		addd f12, f10, f2	ld f18, -32(r1)	5
		addd f16, f14, f2	sd f4, 0(r1)	6
	subui r1, r1, #40	addd f20, f18, f2	sd f8, -8(r1)	7
			sd f12, 24(r1)	8
	bne r1, loop		sd f16, 16(r1)	9
			sd f20, 8(r1)	10
Slot 1		Slot 2	Slot 3	

Se tardarían  $10 \times (1000/5) = 2000$  ciclos, frente a los  $10 \times 1000 = 10000$  ó  $6 \times 1000 = 6000$  ciclos del bucle sin desenrollar.

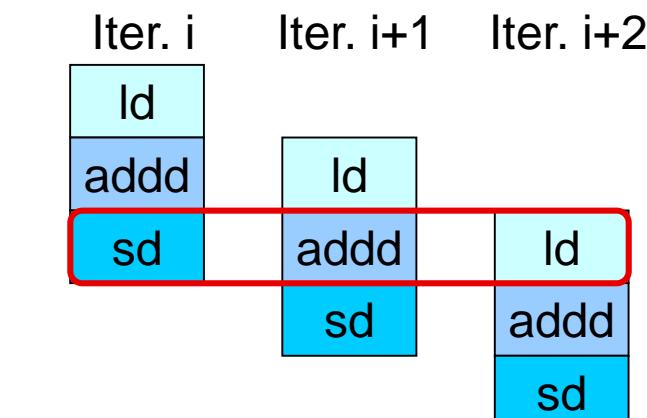
# Planificación Estática con Segmentación

## Software I

```
for ( i = 1000 ; i > 0 ; i = i - 1 )  
    x[i] = x[i] + s;
```

loop:    Id            f0, 0(r1)  
          addd        f4, f0, f2  
          sd           f4, 0(r1)  
          subui      r1, r1, #8  
          bne          r1, loop

loop:    sd           f4, 16(r1)  
          addd        f4, f0, f2  
          Id           f0, 0(r1)  
          subui      r1, r1, #8  
          bne          r1, loop



Iter. i:    Id            f0, 16(r1)  
          addd        f4, f0, f2  
          **sd**           f4, 16(r1)

Iter. i+1:    Id            f0, 8(r1)  
          **addd**        f4, f0, f2  
          sd            f4, 8(r1)

Iter. i+2:    **Id**            f0, 0(r1)  
          addd        f4, f0, f2  
          sd            f4, 0(r1)

# Planificación Estática con Segmentación Software II

	Instrucción Entera	Instrucción FP	Load/Store	Ciclo
loop		addd f4,f0,f2	sd f4,16(r1)	1
	subui r1,r1,#8		ld f0,0(r1)	2
	-----	-----	-----	3
	bne r1,loop			4
	-----	-----	-----	5
Slot 1		Slot 2	Slot 3	

- Se tardarían  $5 \times 1000 = 5000$  ciclos (algunos más si se consideran las instrucciones previas al inicio del bucle con segmentación software y las posteriores al final del bucle).
- Si se desenrolla este bucle ya segmentado, se pueden mejorar mucho más las prestaciones.

# Planificación estática global

- La planificación global mueve código a través de los saltos condicionales (que no correspondan al control del bucle)
- Se parte de una estimación de las frecuencias de ejecución de las posibles alternativas tras una instrucción de salto condicional
- Apoyo para facilitar la planificación global:
  - Instrucciones con predicado y
  - Especulación

# Instrucciones de Ejecución Condicional

```
if ( A == 0 )  
    S = T;
```

```
bnez r1, L  
add r2, r3, 0 ; r3=T
```

L:

```
cmovez r2, r3, r1
```

```
if ( B < 0 )  
    A = -B;  
else  
    A = B;
```

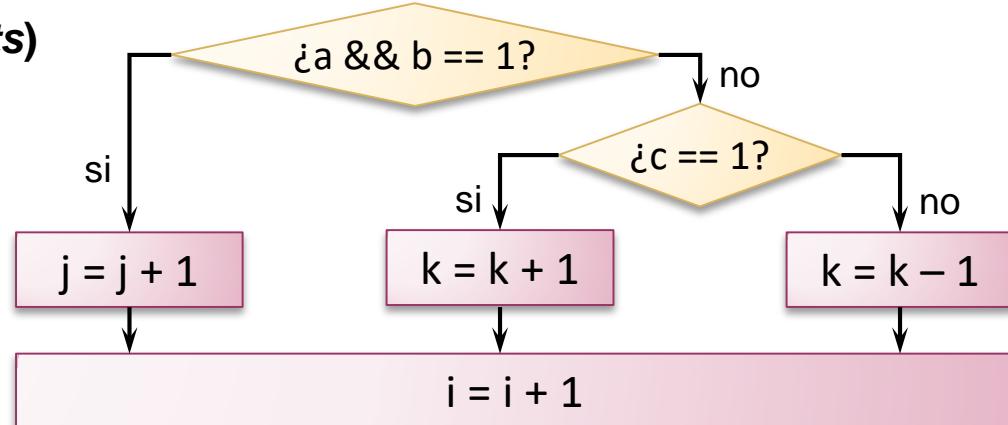
```
r2 ← r1  
r2 ← -r1 ( condicional a que r1 < 0 )
```

Si se verifica la condición en el último operando ( $r1=0$  en este caso) se produce el movimiento entre los otros dos operandos ( $r2 \leftarrow r3$ )

# Instrucciones con Predicado I

## Ejemplo de Ejecución VLIW (con dos slots)

```
if ( a && b ) j = j + 1;  
else if (c) k = k + 1;  
    else k = k - 1;  
i = i + 1;
```



Slot 1	Slot 2
[1]	[2]
[3]	[4]
[5]	[10]
[7]	[6]
[8]	[9]

Se eliminan todos los saltos.  
Las dependencias de control pasan  
a ser dependencias de datos

[1]	P1, P2 = cmp (a==0)	
[2]	P3 = cmp (a!=a)	
[3]	P4 = cmp (a!=a)	
[4]	P5 = cmp (a!=a)	
[5]	<P2> P1, P3 = cmp (b==0)	(Si a=1)
[6]	<P3> add j, j, 1	(Si a=1 y b=1)
[7]	<P1> P4, P5 = cmp (c!=0)	(Si a=0 ó b=0)
[8]	<P4> add k, k, 1	(Si c=1)
[9]	<P5> sub k, k, 1	(Si c=0)
[10]	add i, i, 1	

# Contenido de la Lección 13

- Características generales y motivación (ILP hardware vs. ILP software)
- Planificación estática
- Procesamiento especulativo

# Procesamiento Especulativo

El procesamiento especulativo se basa en la predicción de que determinada instrucción, condición, etc. será muy probable, para adelantar su procesamiento, mejorando las prestaciones del procesador.

El procesamiento especulativo tiene un coste si la predicción que se ha hecho no es correcta. Este coste va desde el correspondiente a haber ejecutado una instrucción que no tendría que haberse ejecutado, hasta la necesidad de incluir código que deshaga el efecto de la operación implementada, vigilar el comportamiento frente a las excepciones, etc.

Slot 1	Slot 2
lw r1,40(r2)	add r3,r4,r5
	add r6,r3,r7
beqz r10,loop	
lw r8,0(r10)	
lw r9,0(r8)	



Slot 1	Slot 2
lw r1,40(r2)	add r3,r4,r5
lwc r8,0(r10),r10	add r6,r3,r7
beqz r10,loop	
lw r9,0(r8)	

Se produce el lw r8,0(r10) si r10!=0

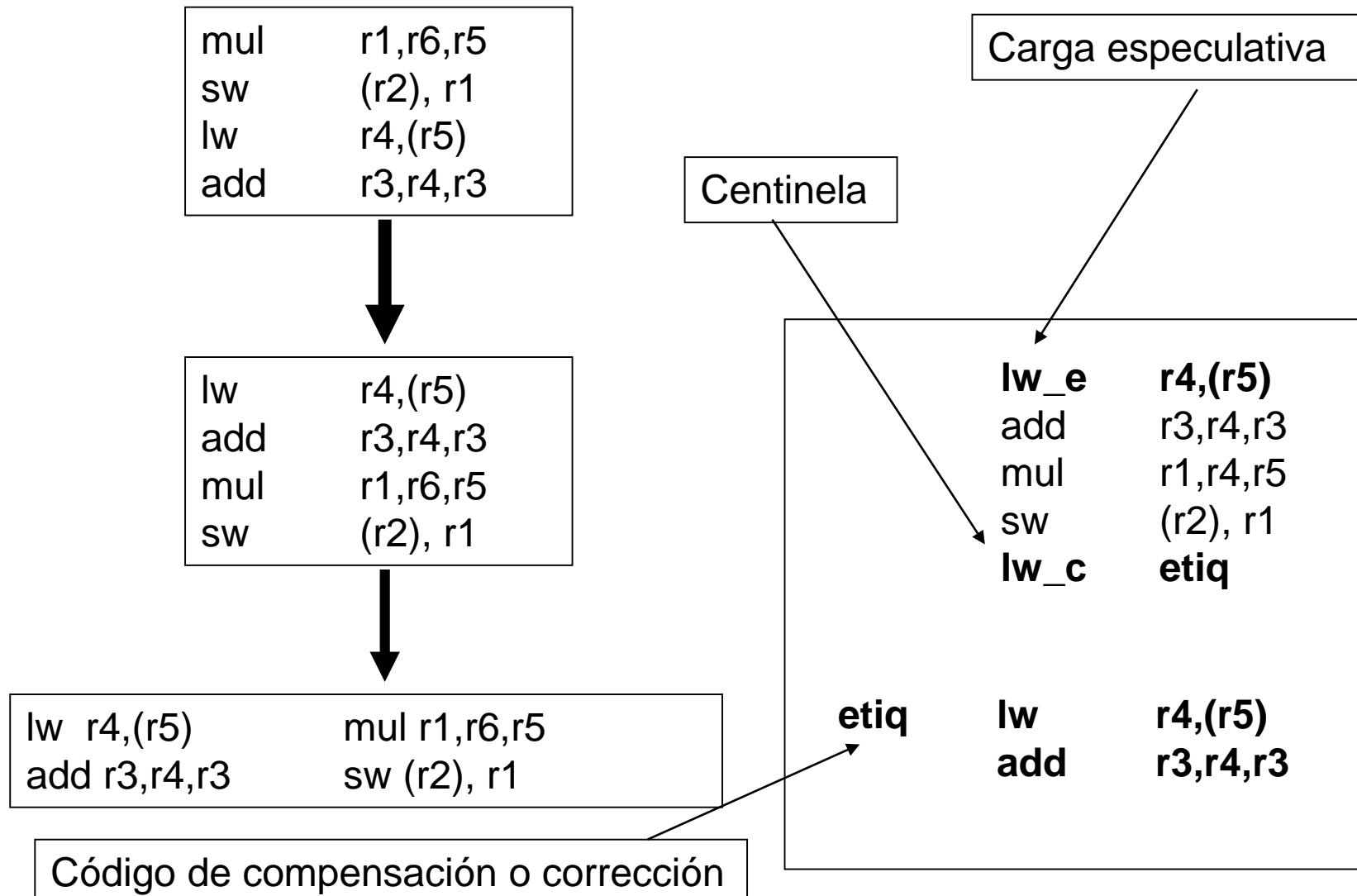
Se aprovecha el slot de acceso a memoria. Si **beqz** da lugar al salto se ha ejecutado una instrucción de forma innecesaria. **Otro problema son las excepciones**

# Uso de Centinelas para Permitir la Especulación de las Referencias a Memoria

AC  ATC

- Cuando no existe ninguna ambigüedad, el compilador adelanta los LOADs con respecto a los STOREs para reducir la longitud del camino crítico en el código
- Cuando existe ambigüedad:
  - Se incluye en la arquitectura una instrucción para comprobar los conflictos de direcciones.
  - La instrucción se sitúa en la posición original del LOAD (**centinela**)
  - Cuando se ejecuta el LOAD especulativo, el hardware guarda la dirección a la que se ha realizado el acceso.
  - Si los sucesivos STOREs no han accedido a esa dirección, la especulación es correcta. En caso contrario, la especulación ha fallado.
  - Si la especulación ha fallado:
    - Si la especulación afecta al LOAD solamente, se vuelve a ejecutar cuando se llega al centinela.
    - Si se han ejecutado instrucciones que dependen del LOAD habrá que repetir todas esas instrucciones (se necesita mantener información de todas ellas en un trozo de código cuya dirección se incluye en la instrucción centinela)

# Uso de Centinelas para Permitir la Especulación de las Referencias a Memoria



# Especulación Software vs. Hardware

- Al final, los procesadores que implementan ILP intensivo en hardware han acabado tomando ideas de la especulación software y viceversa:

## Técnicas Software en ILP intensivo en Hardware

- Soporte para instrucciones condicionales
- Precaptación de instrucciones y otros elementos para mejorar el acceso a caches
- Elementos para la predicción de saltos
- Soporte especial para los LOADs especulativos

## Técnicas Hardware en ILP intensivo en Software

- Planificación de instrucciones utilizando campos de marca
- Predicción dinámica de saltos
- Soporte para procesamiento de excepciones
- Hardware para verificar la corrección de LOADs especulativos

# Para ampliar ...

- Páginas Web:
  - <http://www.research.ibm.com/vliw> (Investigación sobre VLIW en IBM).
- Artículos de Revistas:
  - Fisher, J.A.: “Very Long Instruction Word Architectures and ELI-512”. Proc. 10th Symp. On Computer Architecture, pp.140-150, 1983. (Aparece el término VLIW).
  - Rau, B.R.; et al.: “The Cydra 5 departmental supercomputer: design philosophies, decisions, and trade-offs”. IEEE Computers, pp.22-34, 22:1, 1989.

# Para ampliar ...

- Páginas Web:
  - <http://www.cs.ucsd.edu/classes/sp00/cse231/mdes.pdf>  
**(artículo sobre compiladores para VLIW y EPIC)**
  - <http://www.compilerconnection.com/index.html>  
**(página bastante completa con información relativa a compiladores: compañías, investigación, grupos de noticias,...)**
- Artículos de Revistas:
  - Lam, M.: "Software Pipelining: An effective scheduling technique for VLIW processors". Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation, pp.318-328, 1988.
  - Wilson, R.P.; Lam, M.: "Efficient context-sensitive pointer analysis for C programs". Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation, pp.1-12, 1995.
  - Mahlke, S.A., et al.: "Sentinel Scheduling for VLIW and superscalar processors". Proc. 5th Conf. On Architectural Support for Prog. Languages and Operating Systems, pp.238-247, 1992.