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

ugr | Universidad de Granada

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

ATC
Departamento de Arquitectura y Tecnología de Computadores
UNIVERSIDAD DE GRANADA

Bibliografía

- Fundamental
  - Capítulo 4. Sección 2. M. Anguita, J. Ortega. *Fundamentos y problemas de Arquitectura de Computadores,* Editorial Técnica Avicam. ESIIT/C.1 ANG fun
  - Capítulos 3 y 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

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

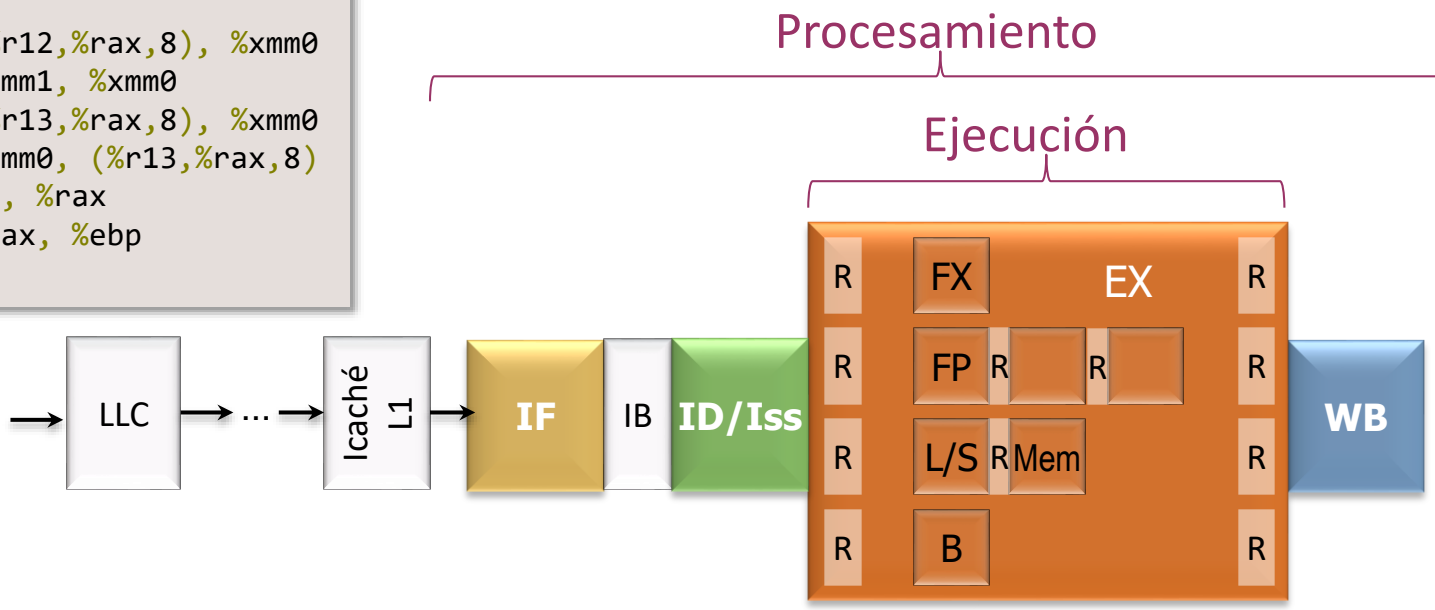
Arq. con **DLP** (*Data Level Parallelism*)	Arq. con **ILP** (*Instruction Level Parallelism*) Tema 4	Arq. con **TLP** (*Thread Level Parallelism*) explícito y **una** instancia de SO		Arq. con **TLP** explícito y **múltiples** instancias SO
Ejecutan las operaciones de una instrucción **concurr.** o en **paralelo**	Ejecutan múltiples instrucciones **concurr.** o en **paralelo**	Ejecutan múltiples flujos de control **concurr.** o en **paralelo**		Ejec. múltiples flujos de control en **paralelo**
Unidades funcionales vectoriales o SIMD	**Cores** escalares segmentados, superescalares o VLIW/EPIC	**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 **multicores**)	**Multi-computadores**: ejecutan threads en paralelo en un sistema con múltiples computadores

Apartados

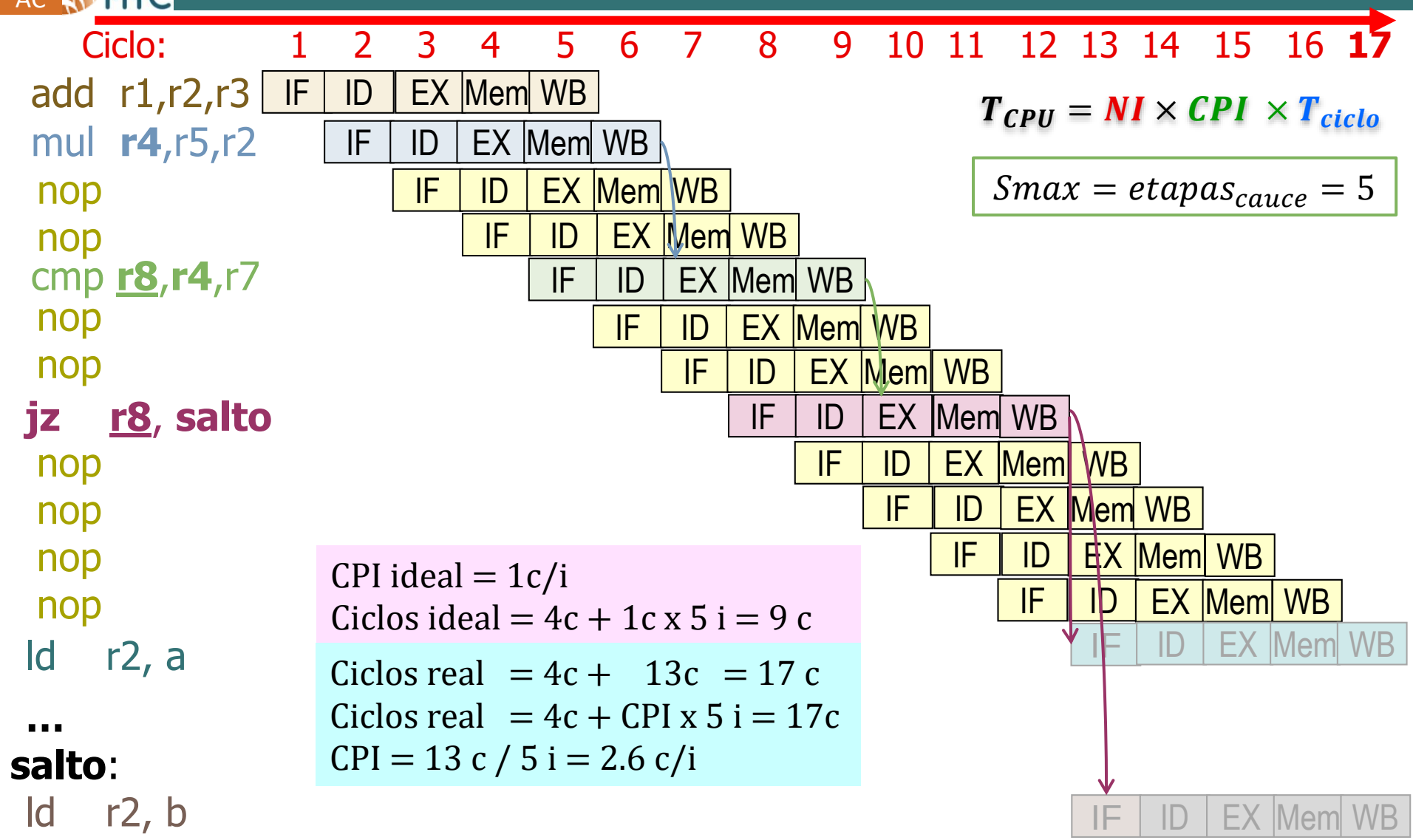
- Lección 11. Microarquitecturas ILP. Cauces Superescalares
- Lección 12. Consistencia del procesador y Procesamiento de Saltos
- Lección 13. Procesamiento VLIW

- Microarquitectura de núcleos ILP superescalar
- Microarquitectura de núcleos ILP VLIW

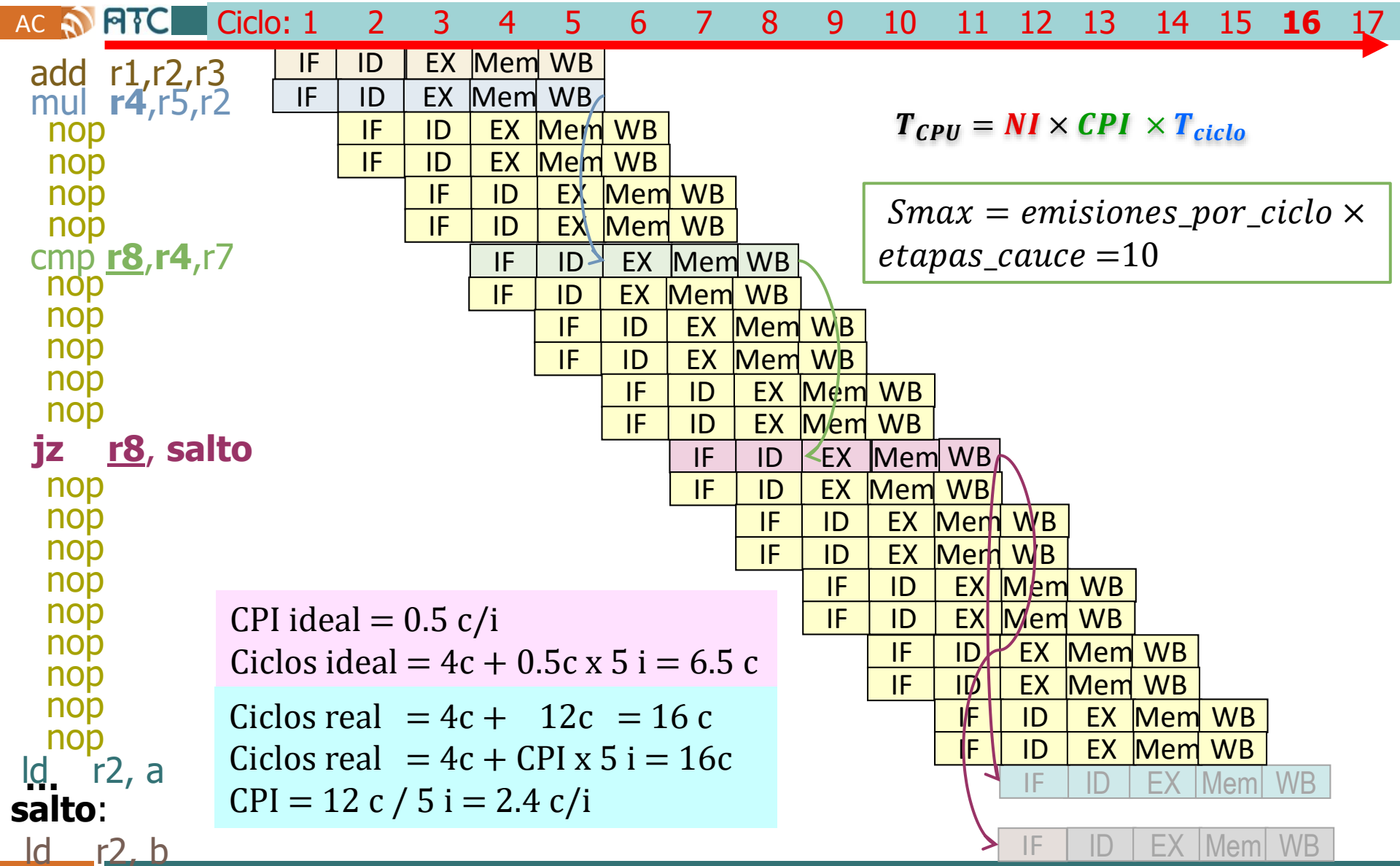
```
.L6:
  movsd (%r12,%rax,8), %xmm0
  mulsd %xmm1, %xmm0
  addsd (%r13,%rax,8), %xmm0
  movsd %xmm0, (%r13,%rax,8)
  addq  $1, %rax
  cmpl  %eax, %ebp
  jg .L6
```



Riesgos (*hazards*): de **datos**, de **control**

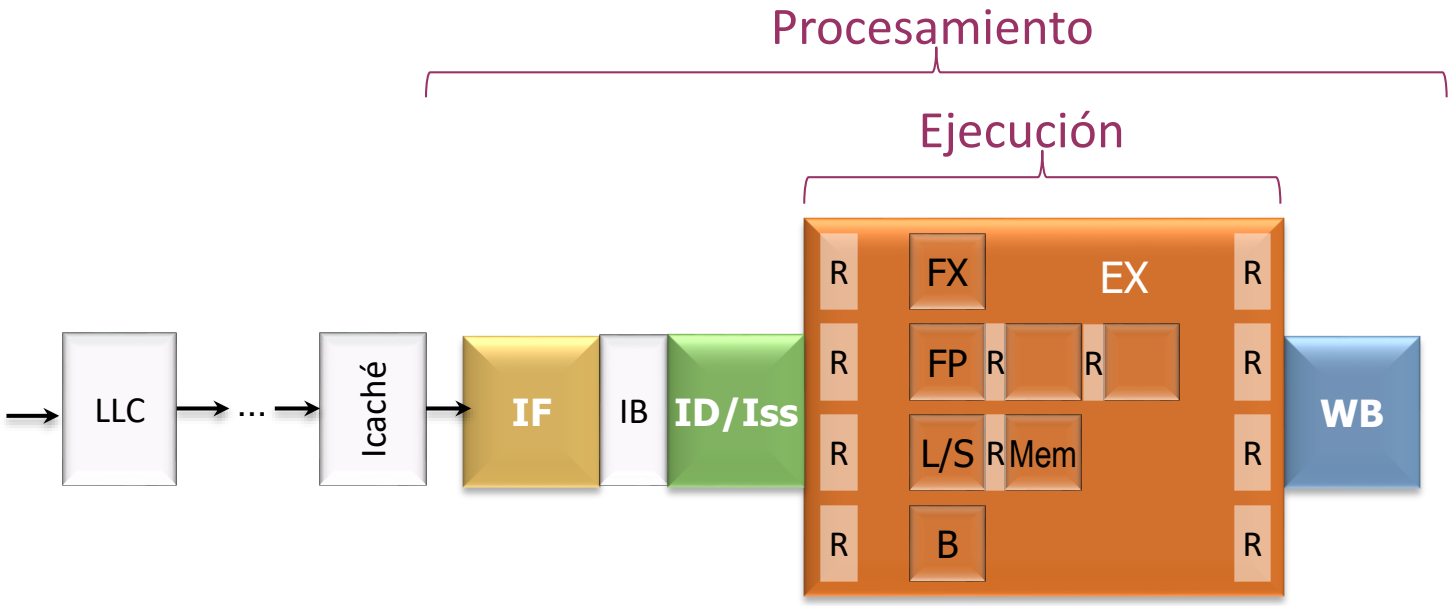


Riesgos (*hazards*): de **datos**, de **control**

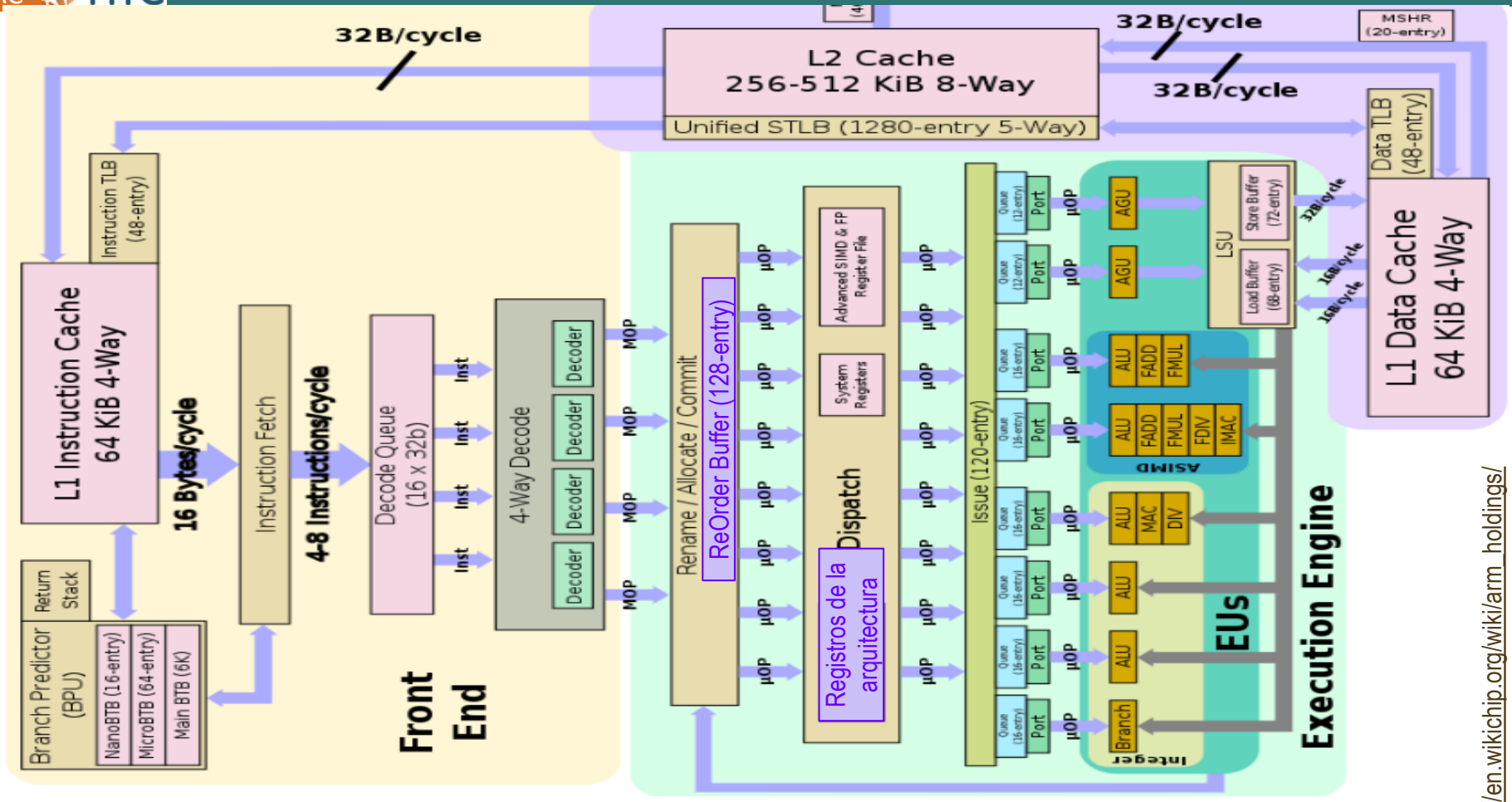


Apartados

- Microarquitectura de núcleos ILP superescalar
  - **Cauce superescalar**
  - Emisión
  - Consistencia del procesador y buffer de reorden (ROB)
  - Procesamiento de saltos
- Microarquitectura de núcleos ILP VLIW



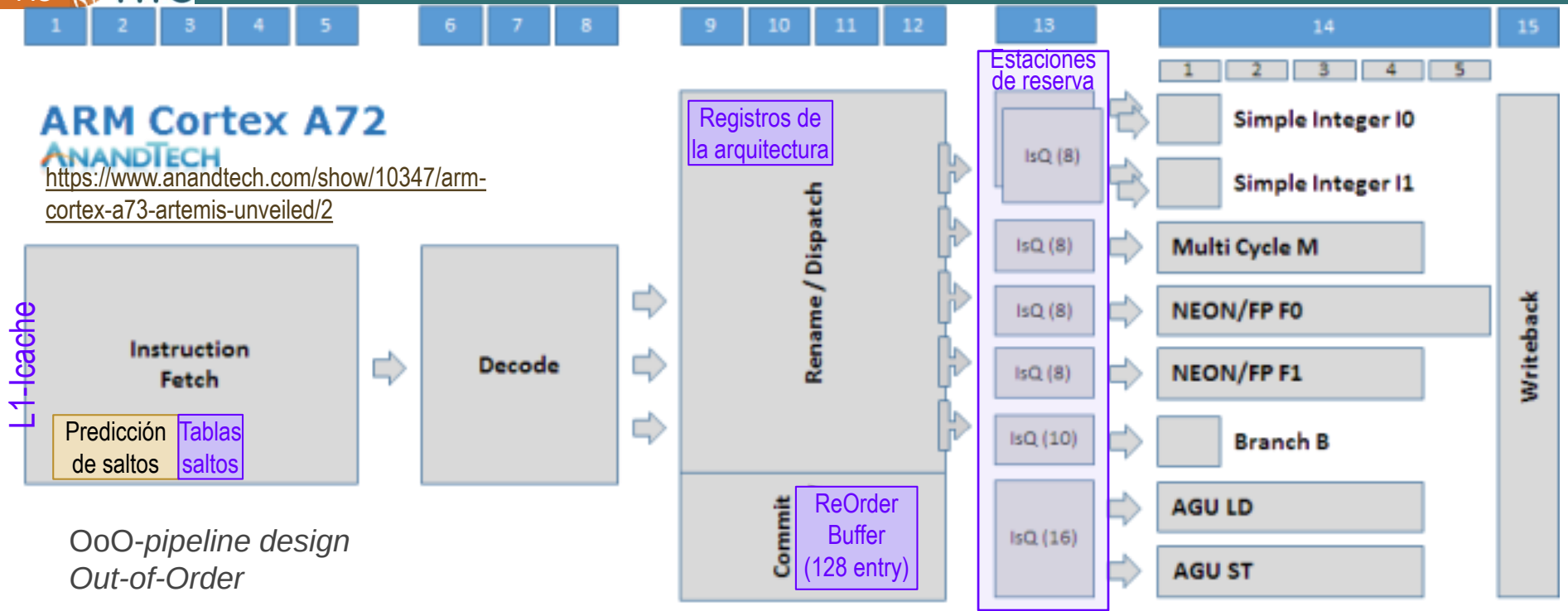
Ejemplo de Cauce (ARM cortex A76)



https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76

Captación (IF)	Decodificación (ID)	Emisión (Iss)	Envío a UF	Ejecución (Ex)	WB

Ejemplo de Cauce (ARM cortex A72)



OoO-pipeline design
Out-of-Order

- Evitar problemas por dependencias RAW y estructurales.
- Renombrado de registros (elimina WAW y WAR)
- Ejecución especulativa de instrucciones
- Captura de operandos desde registros de la arquitectura o desde registros de renombrado.

- Detección de saltos y predicción de saltos incondicionales

- Implementar consistencia secuencial del procesador

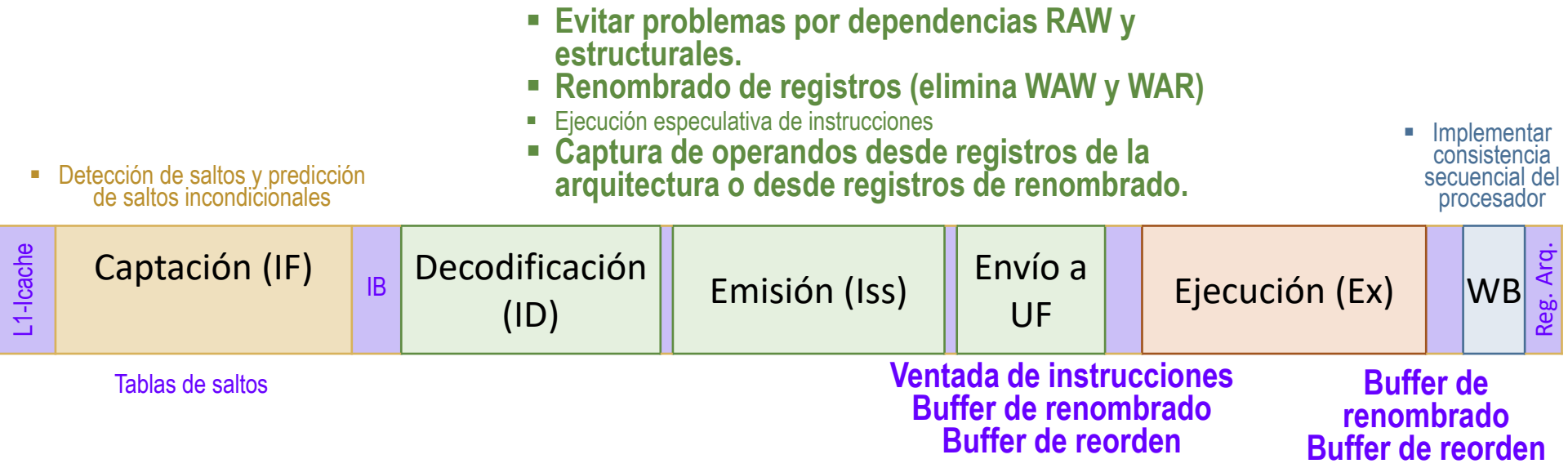
Tablas de saltos

Ventada de instrucciones
Buffer de renombrado
Buffer de reorden

Buffer de renombrado
Buffer de reorden

Apartados

- Microarquitectura ILP superescalar
  - Cauce superescalar
  - **Emisión (Emisión +Envío a UF) (Issue+Dispatch) (Algoritmo de Tomasulo)**
  - Consistencia del procesador y buffer de reorden
  - Procesamiento de saltos



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)**.**

Implementación Estática: Durante la Compilación

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

Características de los Buffers de Renombrado

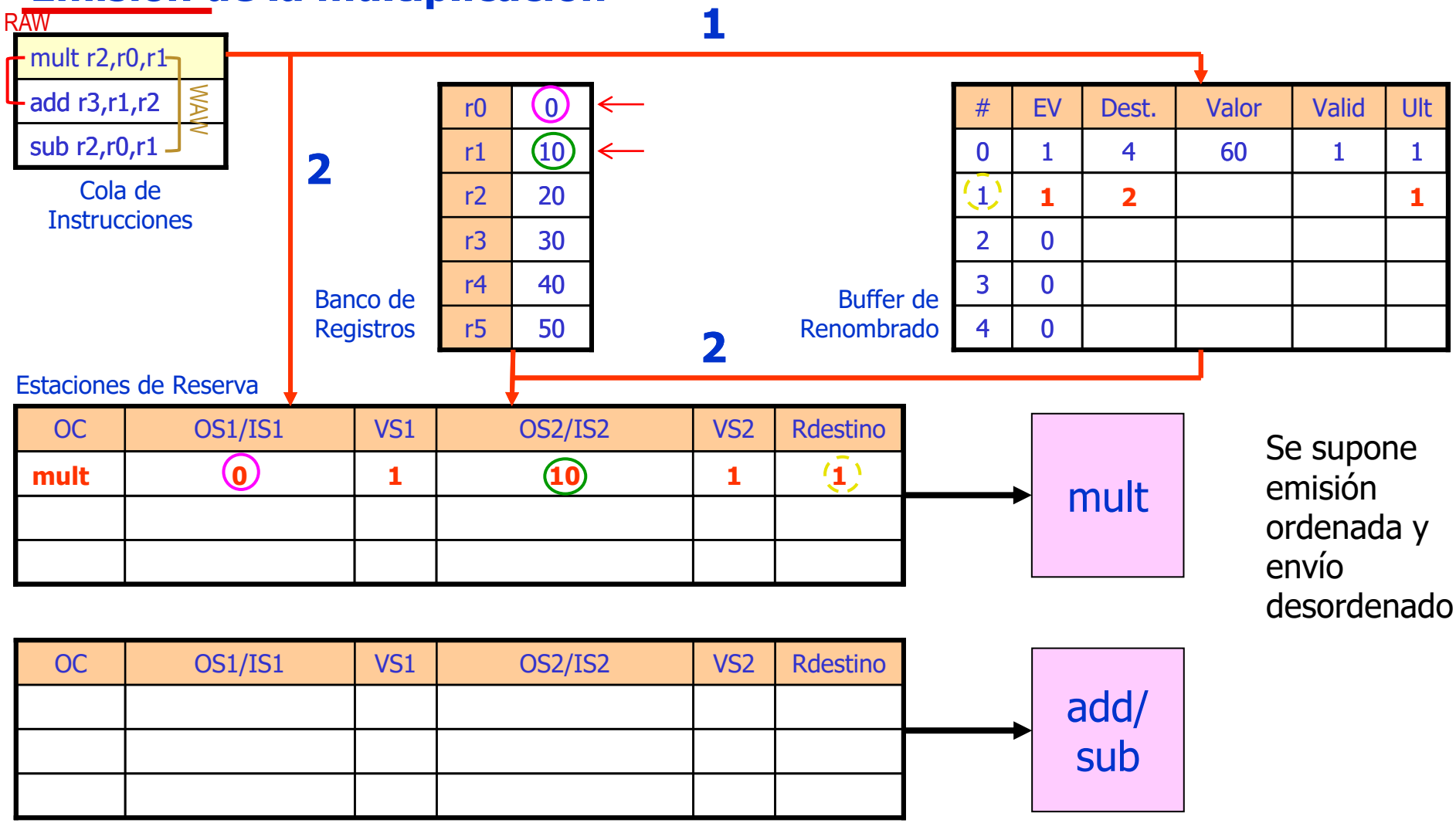
- **Tipos de Buffers** (separados o mezclados con los registros de la arquitectura)
- **Número de Buffers de Renombrado**
- **Mecanismos para acceder a los Buffers** (asociativos o indexados)

Velocidad del Renombrado

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

Algoritmo de Tomasulo I

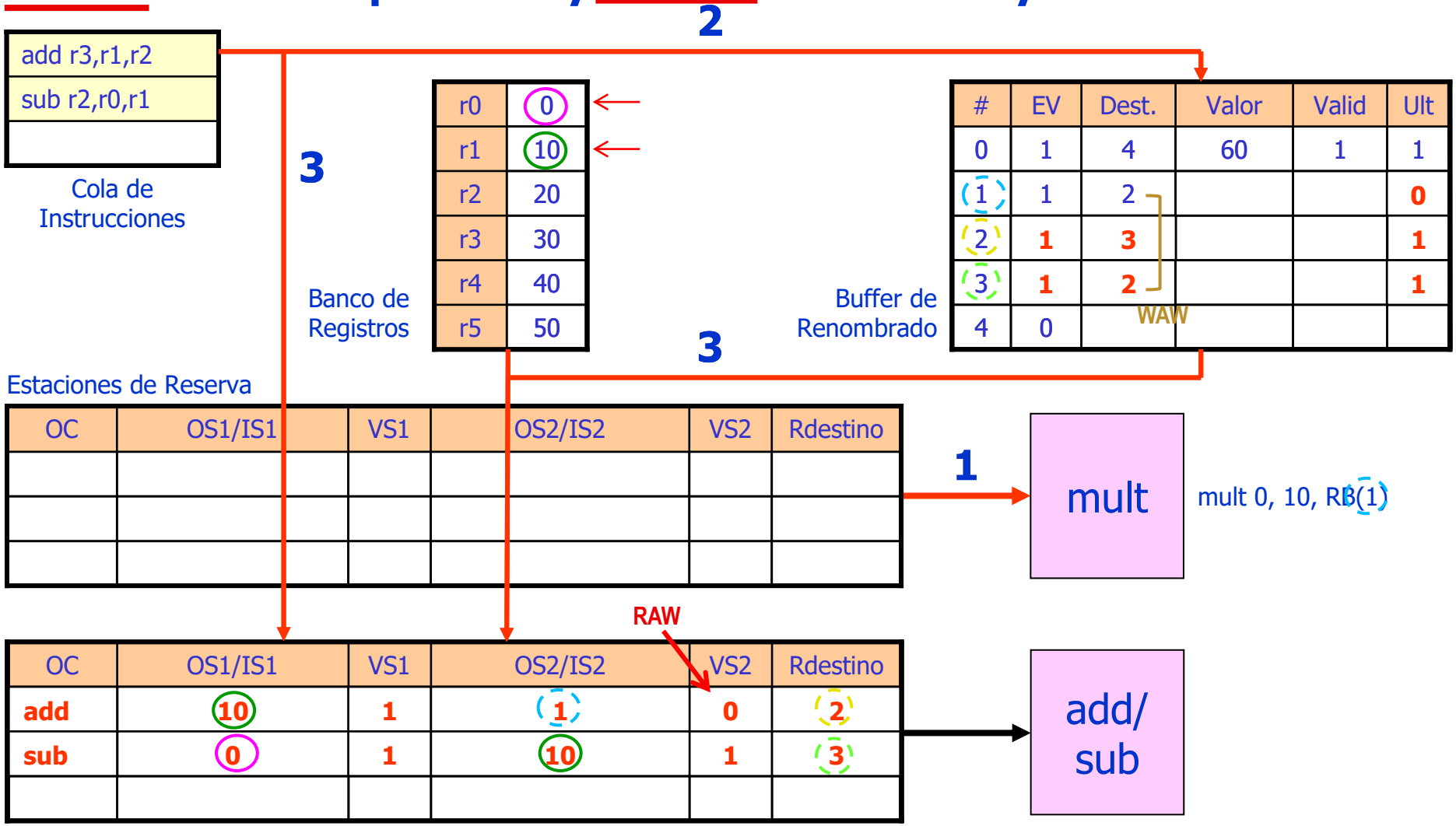
Emisión de la multiplicación



Se supone emisión ordenada y envío desordenado

Algoritmo de Tomasulo II

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



Algoritmo de Tomasulo III

Envío de la resta

Cola de Instrucciones

Banco de Registros

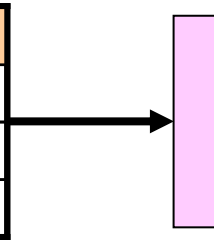
r0	0
r1	10
r2	20
r3	30
r4	40
r5	50

Buffer de Renombrado

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

Estaciones de Reserva

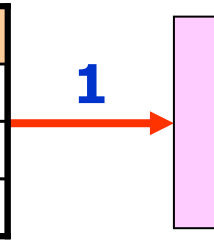
OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino



mult 0, 10, RB(1)

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

1



sub 0, 10, RB(3)

Algoritmo de Tomasulo IV

Termina la resta

Cola de Instrucciones

Banco de Registros

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

Buffer de Renombrado

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

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

add/ sub

sub 0, 10, RB(3)

1

Algoritmo de Tomasulo V

Termina la multiplicación

Cola de Instrucciones

Banco de Registros

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

Buffer de Renombrado

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

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

Banco de Registros

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

Buffer de Renombrado

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

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

Banco de Registros

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

Buffer de Renombrado

#	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				

Estaciones de Reserva

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

mult

OC	OS1/IS1	VS1	OS2/IS2	VS2	Rdestino

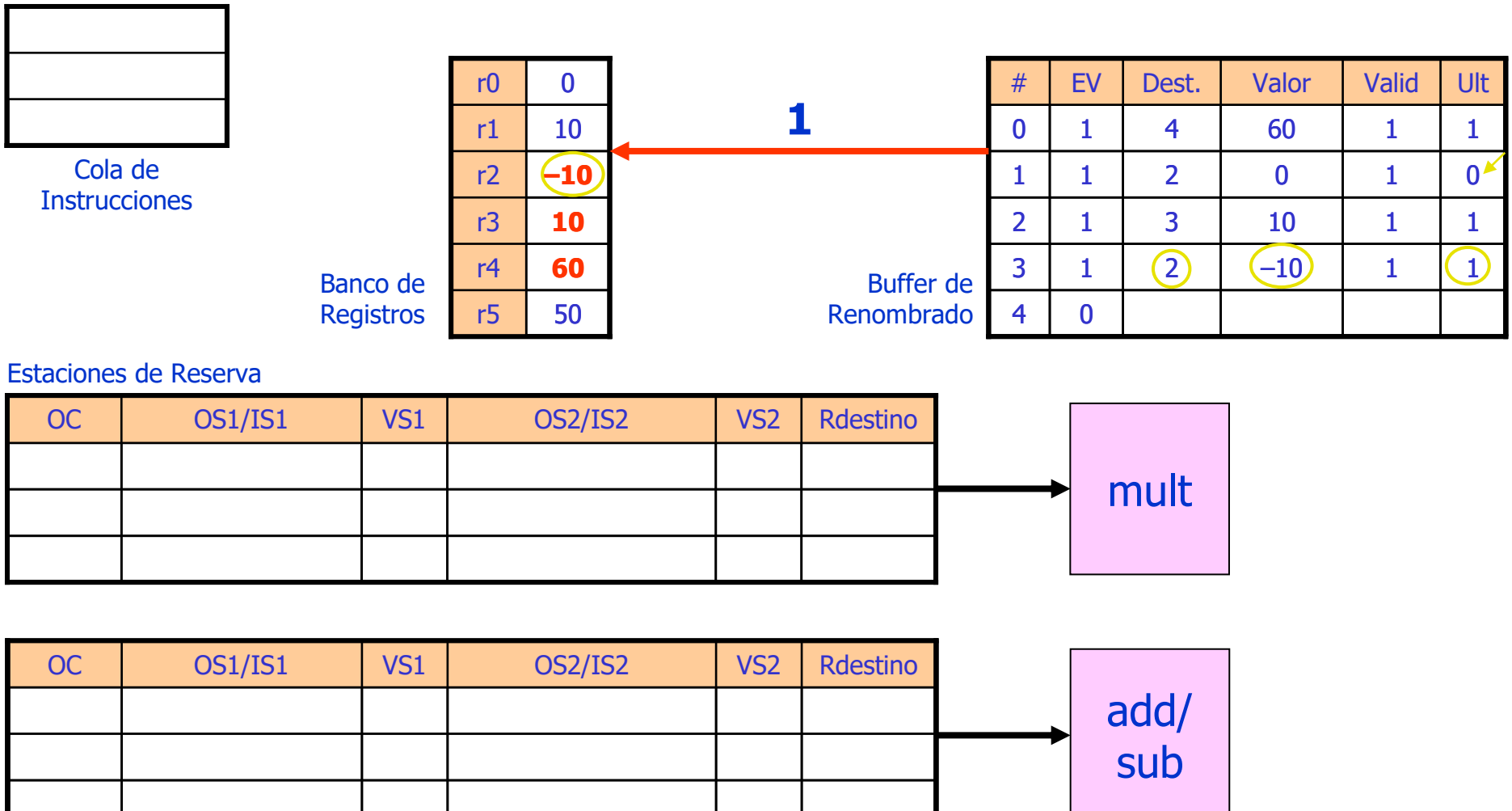
add/ sub

add 10, 0, RB(2)

1

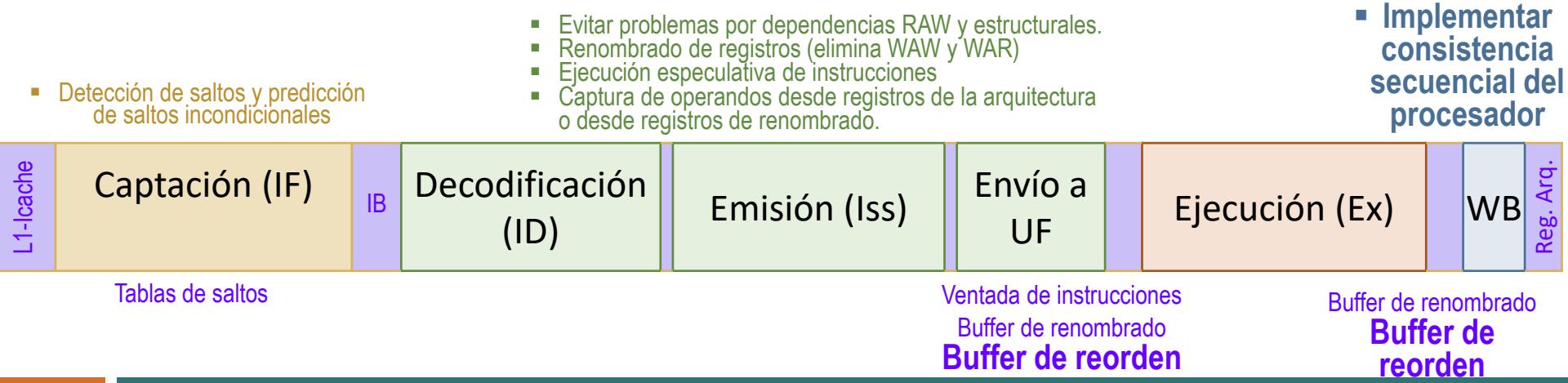
Algoritmo de Tomasulo VIII

Se actualizan los registros (etapa WB - *commit*)

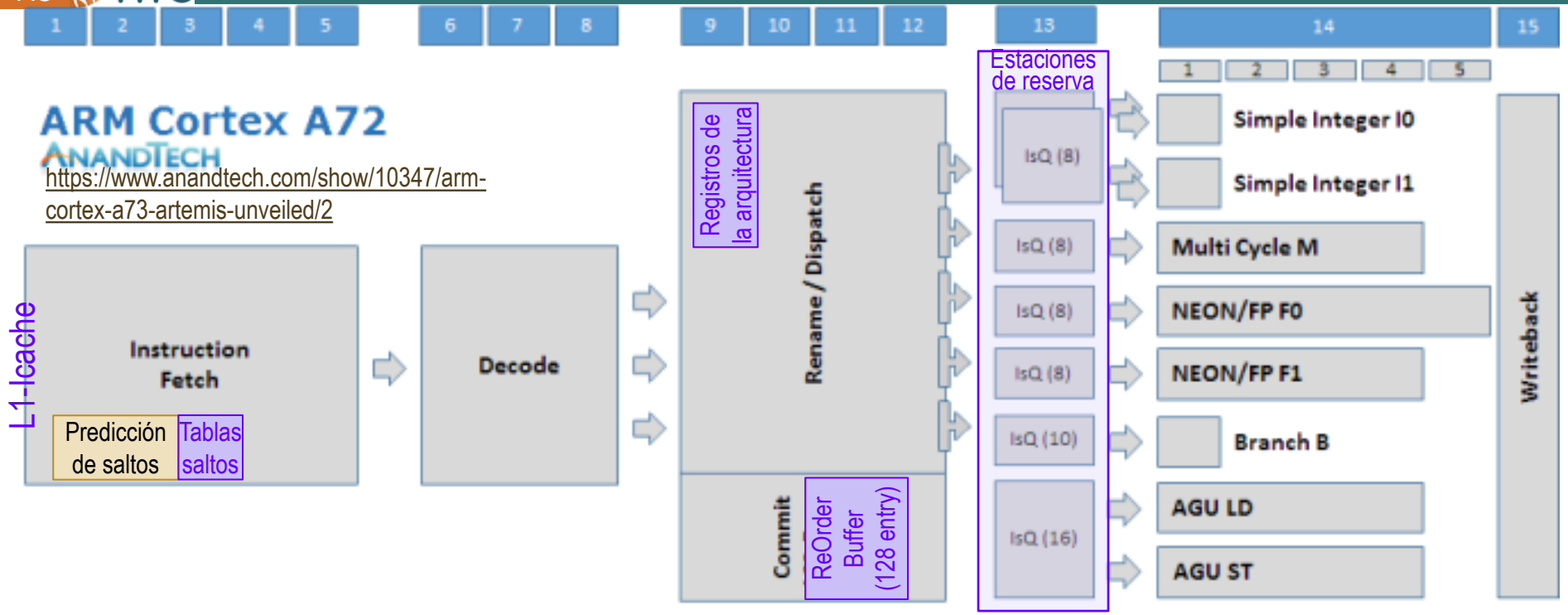


Apartados

- Microarquitectura ILP superescalar
  - Cauce superescalar
  - Emisión (Algoritmo de Tomasulo)
  - **Consistencia del procesador y buffer de reorden**
  - Procesamiento de saltos



Ejemplo de Cauce (ARM cortex A72)



- Evitar problemas por dependencias RAW y estructurales.
- Renombrado de registros (elimina WAW y WAR)
- Ejecución especulativa de instrucciones
- Captura de operandos desde registros de la arquitectura o desde registros de renombrado.

- Implementar consistencia secuencial del procesador

Detección de saltos y predicción de saltos incondicionales

L1-Icache	Captación (IF)	IB	Decodificación (ID)	Emisión (Iss)	Envío a UF	Ejecución (Ex)	WB	Reg. Arq.

Tablas de saltos

Ventada de instrucciones
Buffer de renombrado
Buffer de reorden

Buffer de renombrado
Buffer de reorden

Consistencia

		cauce	
Consistencia de Procesador Consistencia en el orden en que se completan las instrucciones	Débil/relajado: 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)
Consistencia de Memoria Consistencia del orden de los accesos a memoria	Débil/relajado: 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

Tendencia / Prestaciones

Reordenamiento Load/Store


```
loop: ld   r1, 0x1C(r2)
      mul  r1, r1, r6
W(1C+r2) st   r1, 0x1C(r2)
R(2D+r2) ld   r3, 0x2D(r2)
      mul  r3, r3, r6
WAR   st   r3, 0x2D(r2)
      addi r2, r2, #1
      subi r4, r4, #1
      bnz  r4, loop
```

```
→ ld   r1, 0x1C(r2)
  mul  r1, r1, r6
  st   r1, 0x1C(r2)
```

```
→ ld   r3, 0x2D(r2)
  mul  r3, r3, r6
  st   r3, 0x2D(r2)
```

```
→ addi r2, r2, #1
  subi r4, r4, #1
  bnz  r4, loop
```

0x1C(r2)

Se evita tener que esperar a las multiplicaciones y se pueden ir adelantando cálculos correspondientes al control del número de iteraciones (**suponiendo que hay renombrado en hardware**)

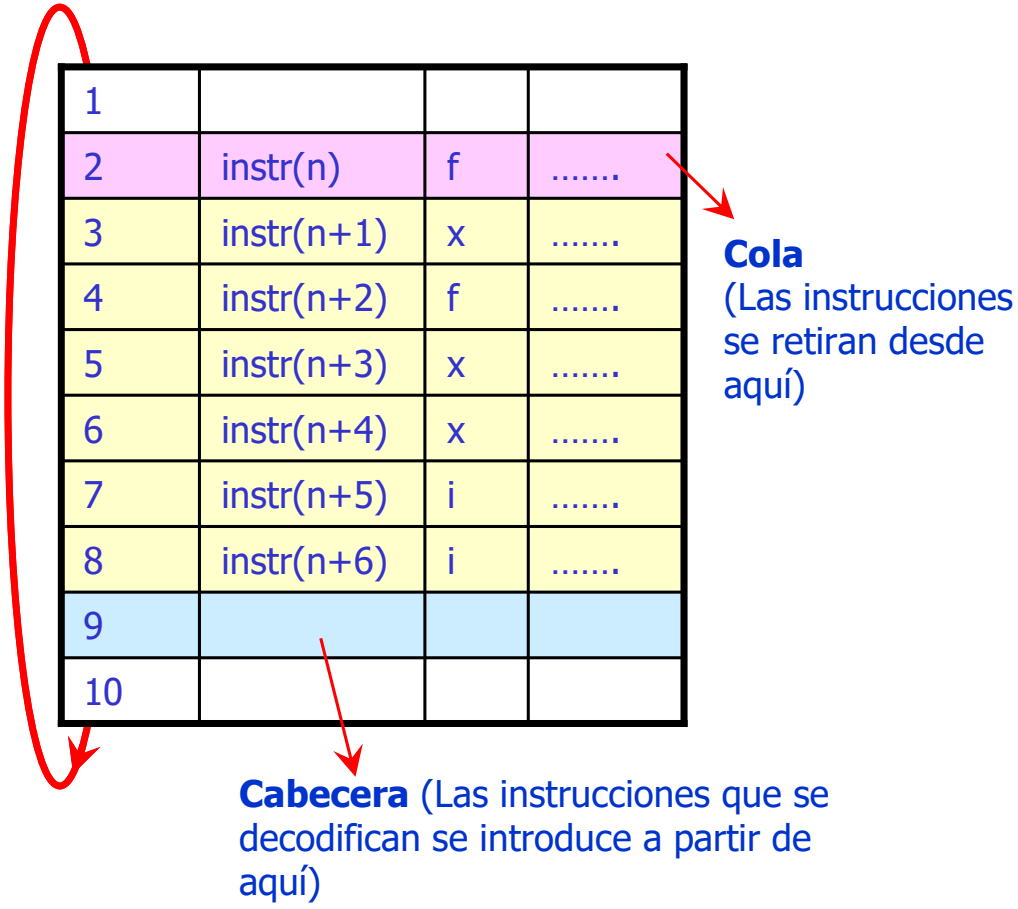
Si se tuviera: st r1,0x1C(r2) W(1C+r2)

ld r3,0x2D(**r7**) R(2D+**r7**)

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

Se tendría un **load especulativo**

Buffer de Reordenamiento (ROB) I



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)**
- Una **instrucciones sólo se puede retirar** (y modificar los registros de la arquitectura) **si ha terminado su ejecución**, 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 emite 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 emite 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

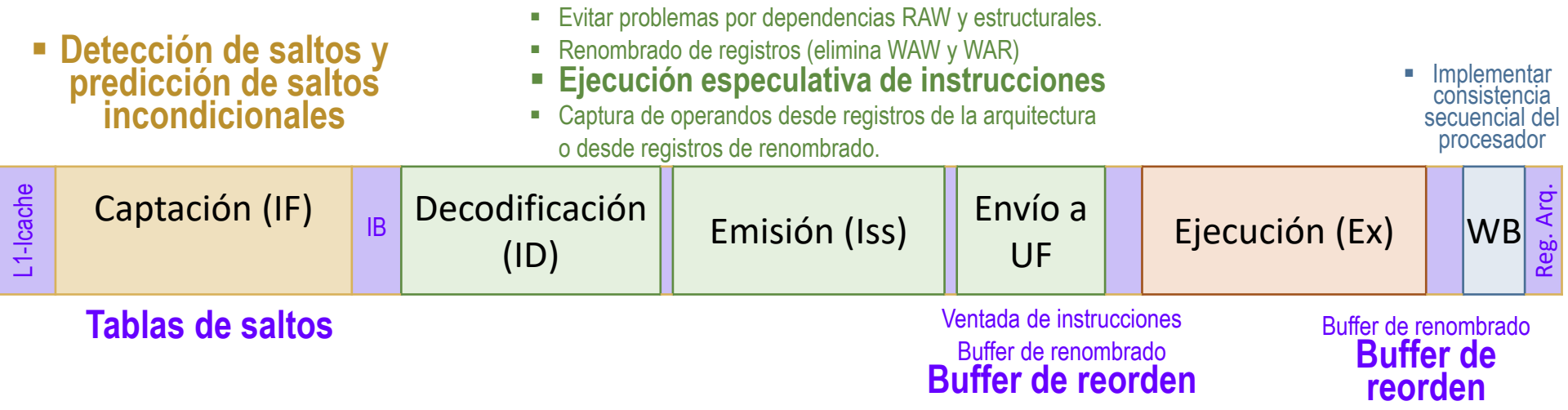
#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

- Se ha supuesto que se pueden retirar (completar) ***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**.

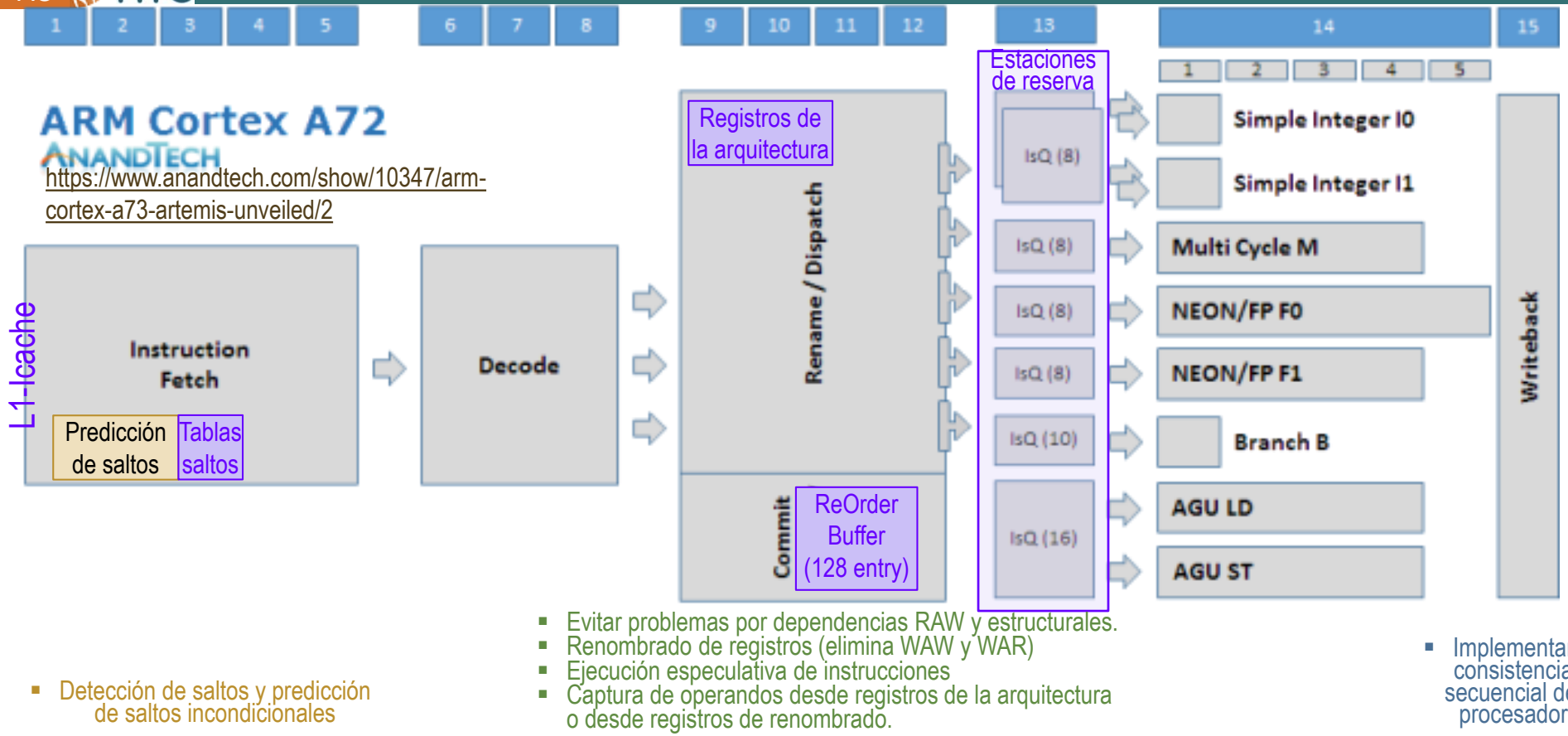
cauce

Apartados

- Microarquitectura ILP superescalar
  - Cauce superescalar
  - Emisión (Algoritmo de Tomasulo)
  - Consistencia del procesador y buffer de reorden
  - **Procesamiento de saltos**



Ejemplo de Cauce (ARM cortex A72)



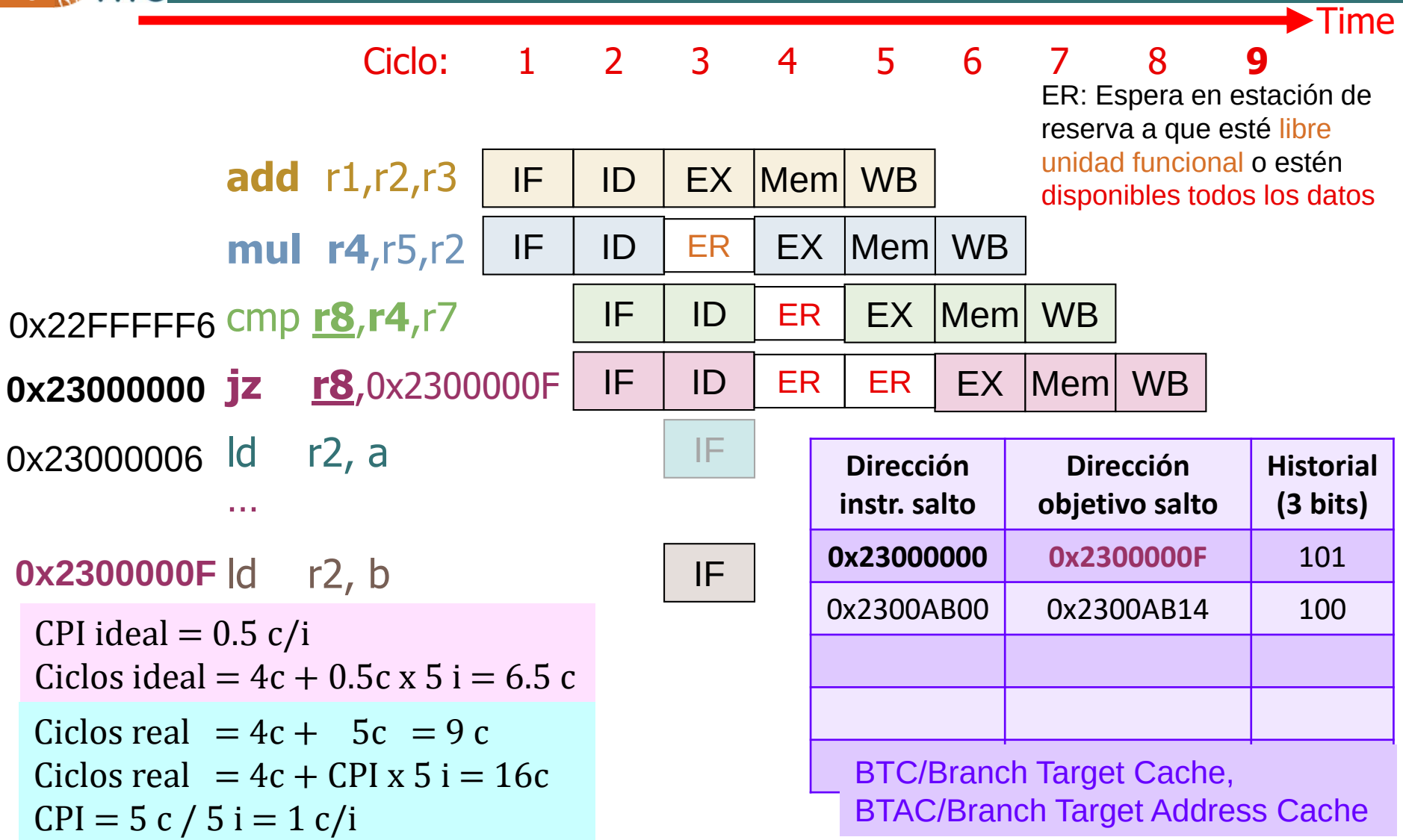
- Detección de saltos y predicción de saltos incondicionales
- Evitar problemas por dependencias RAW y estructurales.
- Renombrado de registros (elimina WAW y WAR)
- Ejecución especulativa de instrucciones
- Captura de operandos desde registros de la arquitectura o desde registros de renombrado.
- Implementar consistencia secuencial del procesador

Tablas de saltos

Ventada de instrucciones
Buffer de renombrado
Buffer de reorden

Buffer de renombrado
Buffer de reorden

Tabla de saltos (BTC: *Branch Target Cache*)



Dirección instr. salto	Dirección objetivo salto	Historial (3 bits)
0x23000000	**0x2300000F**	101
0x2300AB00	0x2300AB14	100

BTC/Branch Target Cache,
BTAC/Branch Target Address Cache

CPI ideal = 0.5 c/i
Ciclos ideal = 4c + 0.5c x 5 i = 6.5 c

Ciclos real = 4c + 5c = 9 c
Ciclos real = 4c + CPI x 5 i = 16c
CPI = 5 c / 5 i = 1 c/i

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

jg .L6 saltar

```
.L6:
  movsd (%r12,%rax,8), %xmm0
  mulsd %xmm1, %xmm0
  addsd (%r13,%rax,8), %xmm0
  movsd %xmm0, (%r13,%rax,8)
  addq  $1, %rax
  cmpl  %eax, %ebp
  jg .L6
```

jle .L7 no saltar

```
.L6:
  addq  $1, %rax
  cmpl  %eax, %ebp
  jle .L7
  movsd (%r12,%rax,8), %xmm0
  mulsd %xmm1, %xmm0
  addsd (%r13,%rax,8), %xmm0
  movsd %xmm0, (%r13,%rax,8)
  jmp .L6
.L7:
```

Predicción 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*

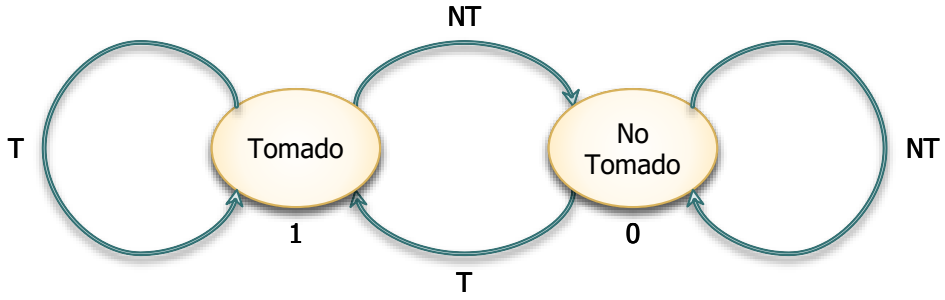
- **Predicción Dinámica Explícita**

  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)

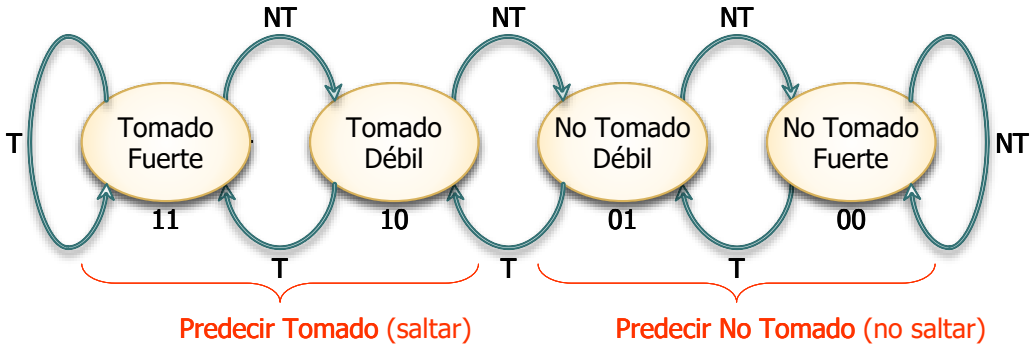


Predicción con 2 bits de historia

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 instrucci ón (T o NT)

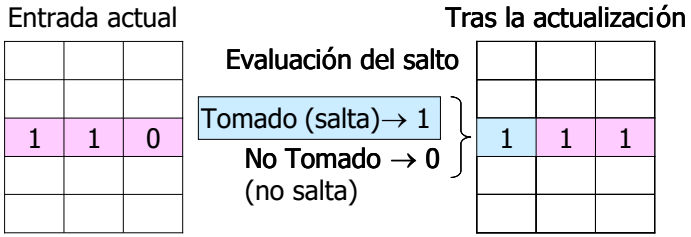


Predicción con 3 bits de historia

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



Instrucciones de Ejecución Condicional (*Guarded Execution*)

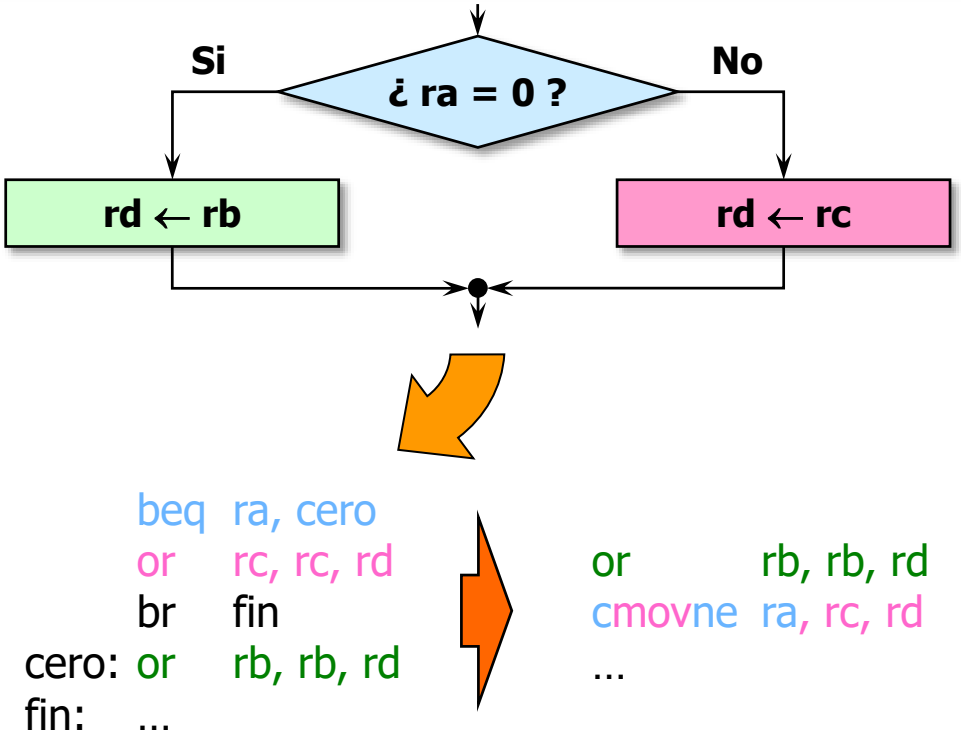
- Se pretende **reducir el número de instrucciones de salto** incluyendo en el **repertorio máquina instrucciones con operaciones condicionales** (*'conditional operate instructions'* o *'guarded instructions'*)
- Estas instrucciones tienen dos partes: la **condición** (denominada *guardia*) y la parte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

Ejemplo: `cmovxx` de Alpha

`cmovxx ra.rq, rb.rq, rc.wq`

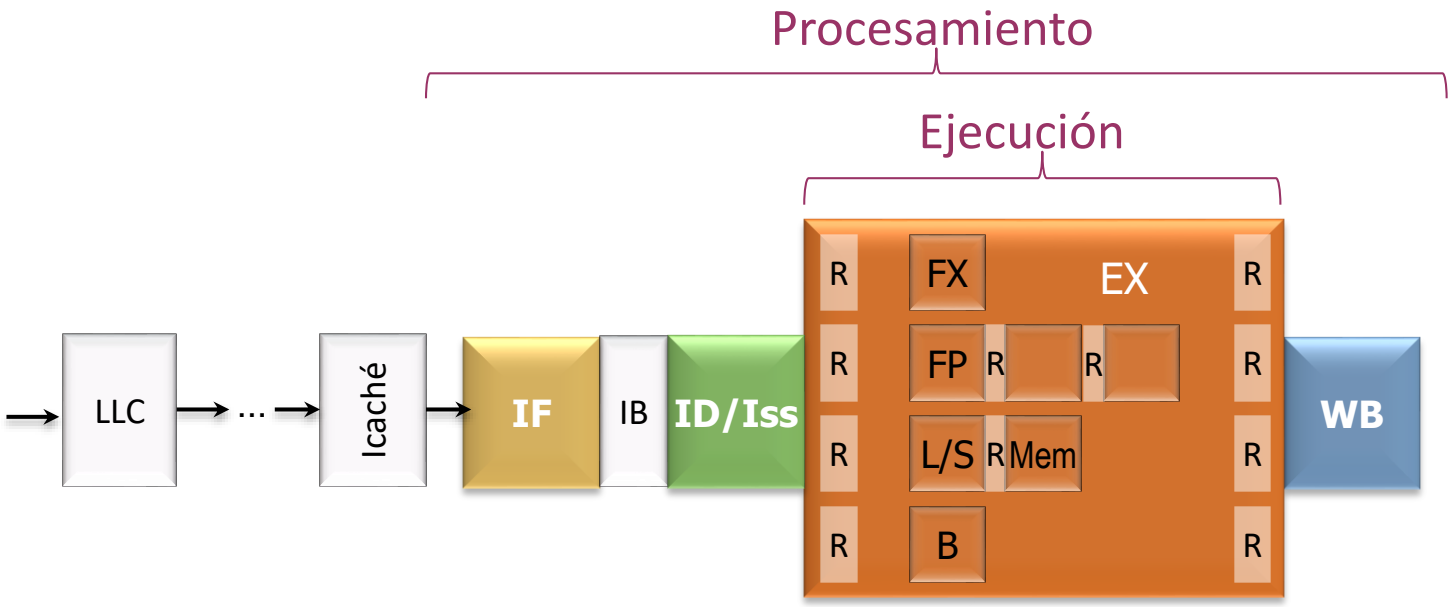
- **`xx`** es una condición
- **`ra.rq, rb.rq`** enteros de 64 bits en registros ra y rb
- **`rc.wq`** entero de 64 bits en rc para escritura
- El registro `ra` se comprueba en relación a la condición `xx` y si se verifica la condición `rb` se copia en `rc`.

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

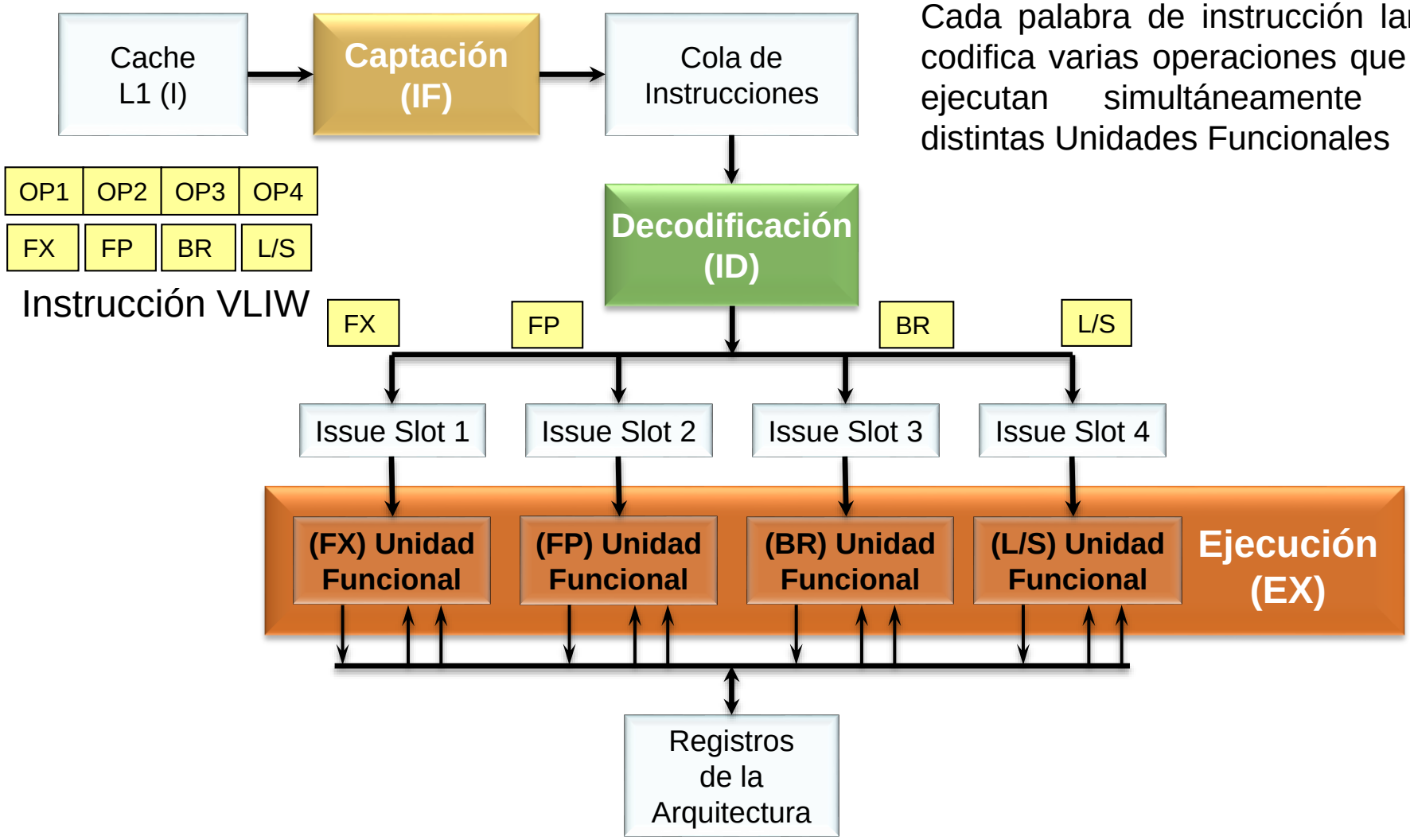


Apartados

- Microarquitectura ILP superescalar
- **Microarquitectura ILP VLIW**



Características generales de los procesadores VLIW (Very Large Inst. Word) II



Cada palabra de instrucción larga codifica varias operaciones que se ejecutan simultáneamente en distintas Unidades Funcionales

Planificación Estática I


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

```
loop:   ld      f0, 0(r1)     ; r1 ⇔i, f0<-M[0+r1]
        addd    f4, f0, f2    ; f2=s, f4<-f0+f2
        sd      f4, 0(r1)     ; M[0+r1]<-f4
        subui   r1, r1, #8    ; r1<-r1-8
        bne     r1, loop      ; salto si no 0
```

1 (ld → addd)
2 (addd → sd)
1 (subui → bne)
1 (bne → ld)

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** los retardos introducidos son los de la tabla:

```
loop:  ld    f0, 0(r1)     ; r1 ⇔i, f0<-M[0+r1]
       addd  f4, f0, f2    ; f2=s, f4<-f0+f2
       sd    f4, 0(r1)     ; M[0+r1]<-f4
       subui r1, r1, #8    ; r1<-r1-8
       bne   r1, loop      ; salto si no 0
```

Pr.	Co.	Esp.
FP	FP	3
ALU	St	2
Ld	FP	1
Ld	St	0
Fx	Br	1
Br	-	1

	Opción 1 (Sin desenrollar)	Opción 2 (Sin desenrollar)	Ciclos
loop	ld f0,0(r1) ; f4<-M[0+r1]	ld f0,0(r1)	1
	nop	**subui r1,r1,#8** ; r1<-r1-8	2
	addd f4,f0,f2	addd f4,f0,f2	3
	nop	nop	4
	nop	bne r1,loop	5
	sd f4,0(r1) ; M[0+r1] <-f4	sd f4,**8**(r1) ; M[8+r1] <-f4	6
	subui r1,r1,#8		7
	nop		8
	bne r1,loop		9
	nop		10

Planificación Estática III

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

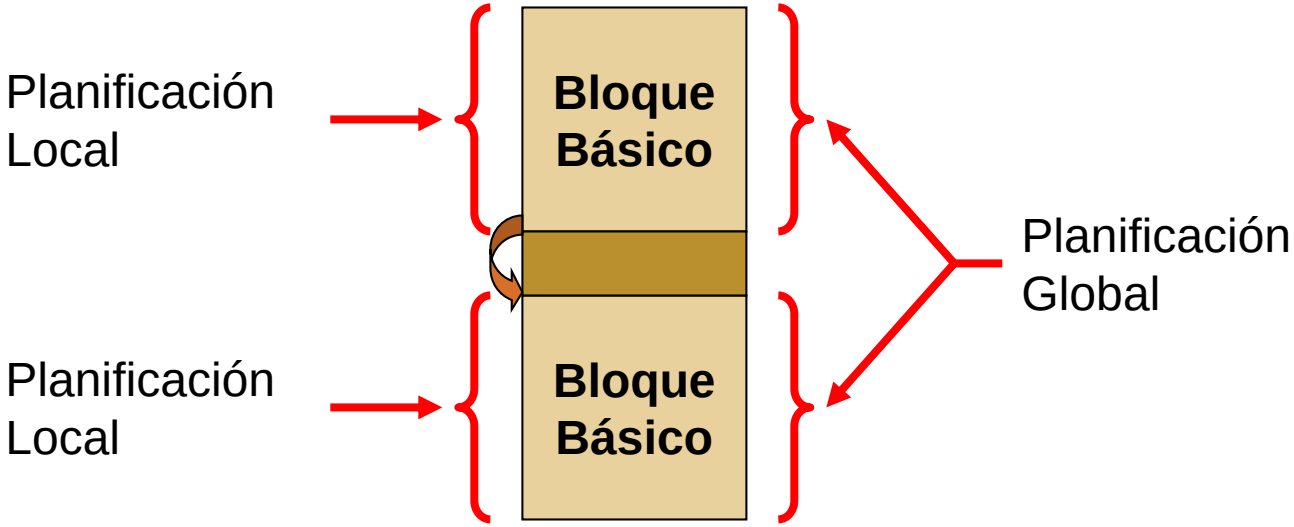
Instrucción VLIW

Pr.	Co.	Esp.
FP	FP	3
ALU	St	2
Ld	FP	1
Ld	St	0
Fx	Br	1
Br	-	1

La arquitectura VLIW no ganaría nada frente a la opción 2 del bucle sin desenrollar. Además, se necesitarían 18 palabras en el código VLIW frente a las 6 que se utilizaría en la codificación escalar (suponiendo que no pueden parar la entrada de nuevas instrucciones en el cauce)

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 básicos 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
```

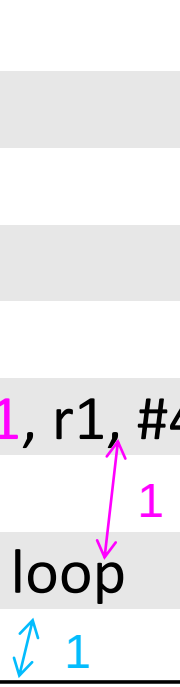
Existen dependencias RAW entre cada dos instrucciones consecutivas, y además existe una instrucción de salto que controla el final del bucle:
Parece que no se puede aprovechar mucho ILP

```
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
```

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 10x(1000/5)=2000 ciclos, frente a los 10x1000=10000 ó 6x1000=6000 ciclos del bucle sin desenrollar.

FX/BR	FP	L/S

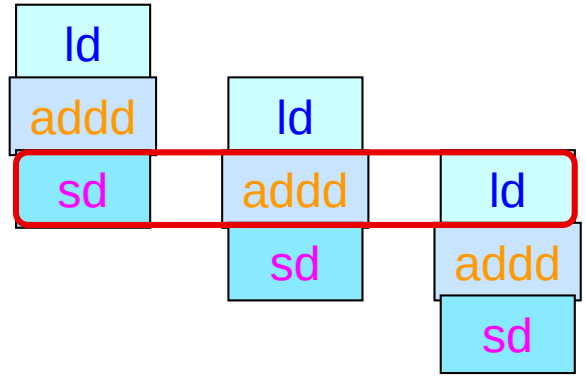
Instrucción VLIW

Planificación Estática con Segmentación Software 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
```

bucle original



nuevo bucle

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

bucle original

```
Iter. i:    ld      f0, 16(r1)
            addd    f4, f0, f2
            sd      f4, 16(r1)

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

Iter. i+2:  ld      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 5x1000=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.

Pr.	Co.	Lat.
FP	FP	3
ALU	St	2
Ld	FP	1
Ld	St	0
Fx	Br	1
Br	-	1

FX/BR	FP	L/S

Instrucción VLIW

Planificación Estática con Segmentación Software II

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

- Se tardarían 5x1000=**4**000 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.

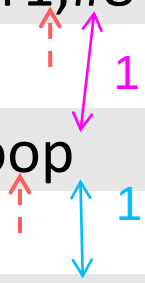
Pr.	Co.	Lat.
FP	FP	3
ALU	St	2
Ld	FP	1
Ld	St	0
Fx	Br	1
Br	-	1

FX/BR	FP	L/S

Instrucción VLIW

Planificación Estática con Segmentación Software II

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



- Se tardarían 5x1000=4000 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.

Pr.	Co.	Lat.
FP	FP	3
ALU	St	2
Ld	FP	1
Ld	St	0
Fx	Br	1
Br	-	1

FX/BR	FP	L/S

Instrucción VLIW

Instrucciones de Ejecución Condicional


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

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

```
cmovz r2, r3, r1
```

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

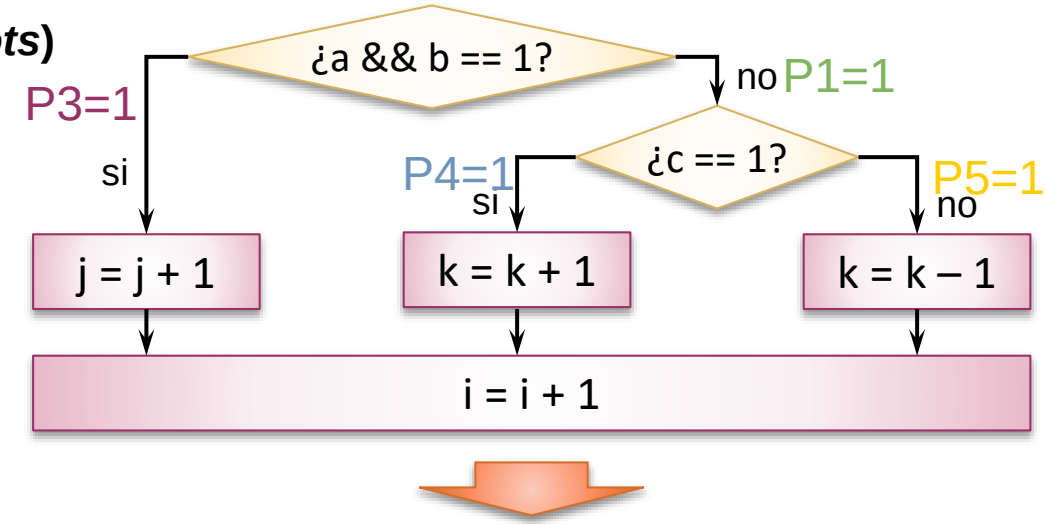
```
if ( B < 0 )
    A= – B;
else
    A = B;
```

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

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]

```
[1]          P1, P2 = cmp (a==0)
[2]          P3 = cmp (a!=a)      (pone P3 a 0)
[3]          P4 = cmp (a!=a)      (pone P4 a 0)
[4]          P5 = cmp (a!=a)      (pone P5 a 0)
[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
```

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