

T4-AC.pdf



Phantone



Arquitectura de Computadores



2º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



Descarga la APP de Wuolah.
Ya disponible para el móvil y la tablet.



Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.

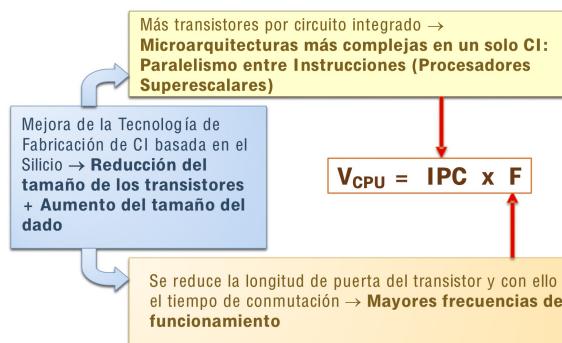


Tema 4: Arquitecturas con Paralelismo a nivel de Instrucción

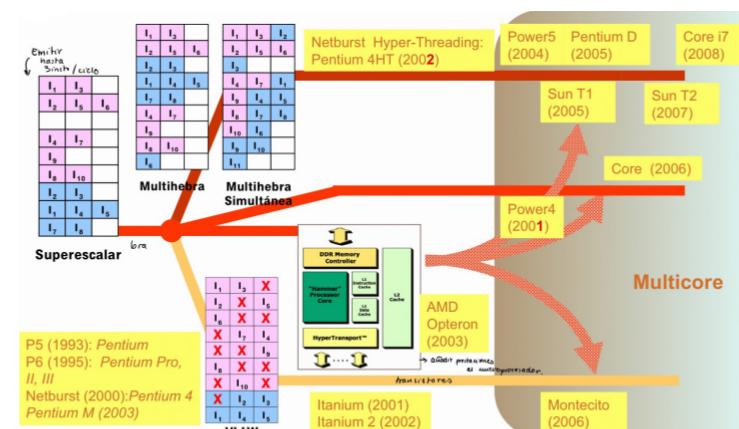
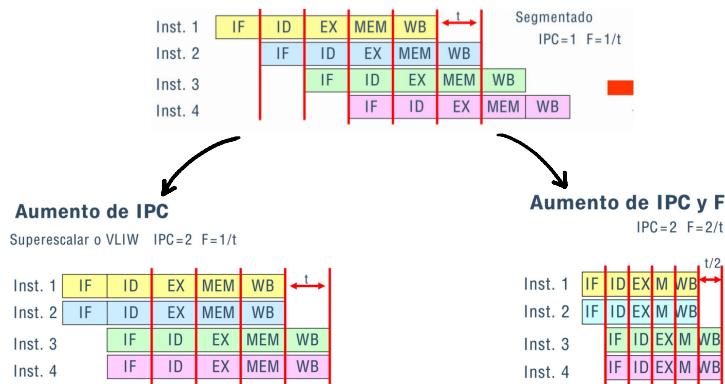
1. Microarquitecturas ILP. Cauces Superescalares

1.1 Introducción: motivación y nota histórica.

- Mejora de las prestaciones de los procesadores



- Evolución de Microarquitecturas



- Clasificación de cores multithread

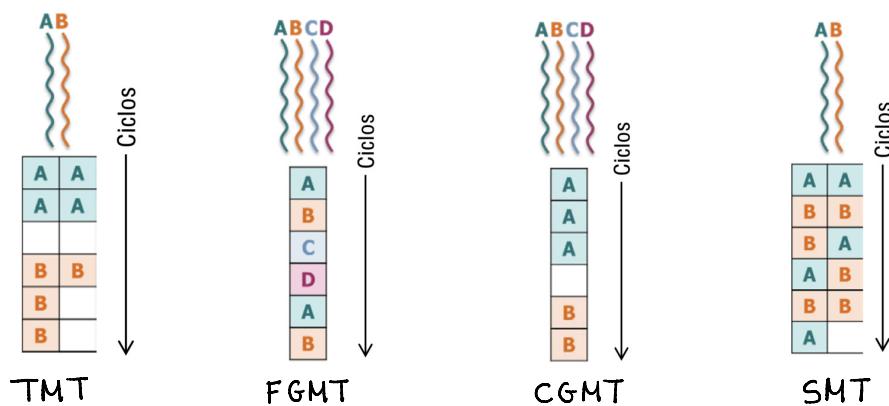
• **Temporal Multithreading (TMT):** estos tipos de core ejecutan varios threads **concurrentemente** en un mismo core. La conmutación entre threads la decide y controla el hardware. Emite instrucciones de **un único thread** en un ciclo.

- **Fine-grain multithreading (FGMT):** la conmutación entre threads la decide el hardware **cada ciclo** (coste 0). Esto puede ser por turno rotatorio (Round Robin) o por eventos* de cierta latencia combinado con alguna técnica de planificación (por ejemplo, el thread menos utilizado).

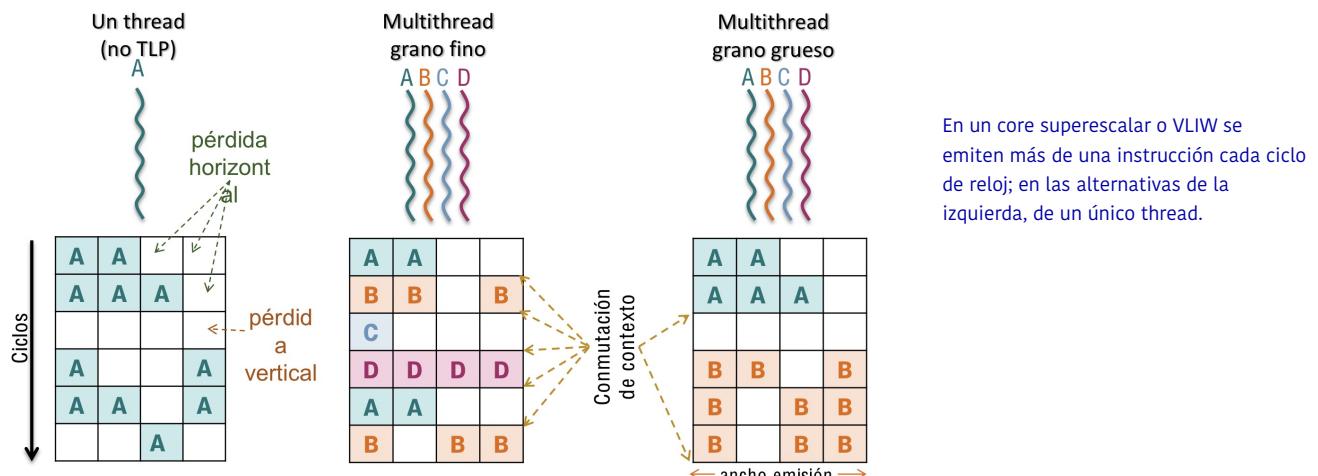
* Evento: dependencia funcional, acceso a los datos cache L1, salto no predecible, una operación de cierta latencia, etc.

- **Coarse-grain multithreading (CGMT):** la conmutación entre threads la decide el hardware tras intervalos de tiempo fijados (timeslice multithreading) o por eventos de cierta latencia (switch-on-event multithreading). Coste de 0 a varios ciclos.

• **Simultaneous Multithreading (SMT):** aquí ejecutan, en un core superescalar, varios threads en **paralelo**. Pueden emitir (para su ejecución) instrucciones de **varios threads** en un ciclo.

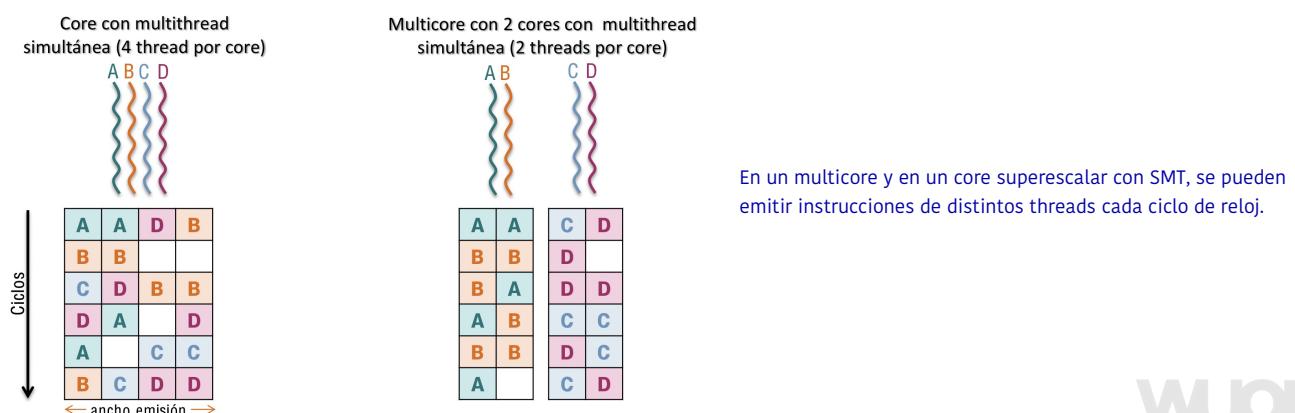


- Alternativas en un core con emisión múltiple de instrucciones de un thread



En un core superescalar o VLIW se emiten más de una instrucción cada ciclo de reloj; en las alternativas de la izquierda, de un único thread.

- Core multithread simultánea y multicores



En un multicore y en un core superescalar con SMT, se pueden emitir instrucciones de distintos threads cada ciclo de reloj.

1.2 Paralelismo entre instrucciones(ILP). Orden en Emisión y Finalización

En una secuencia de instrucciones se pueden distinguir tres tipos de ordenaciones:

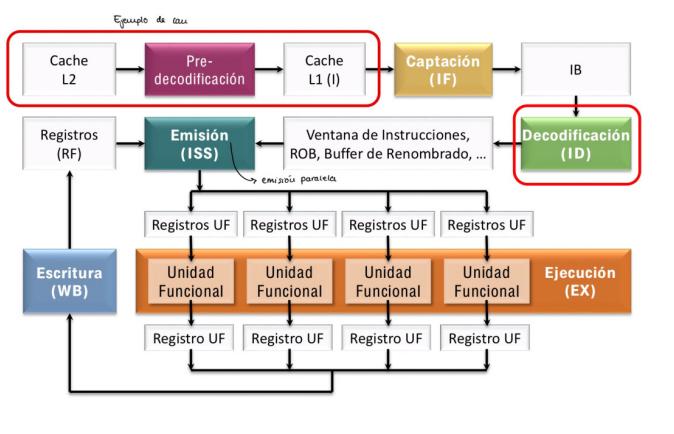
- El orden en el que se captan las instrucciones (el orden de las instrucciones en el código).
- El orden en el que las instrucciones se ejecutan.
- El orden en el que las instrucciones cambian los registros y la memoria.

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).

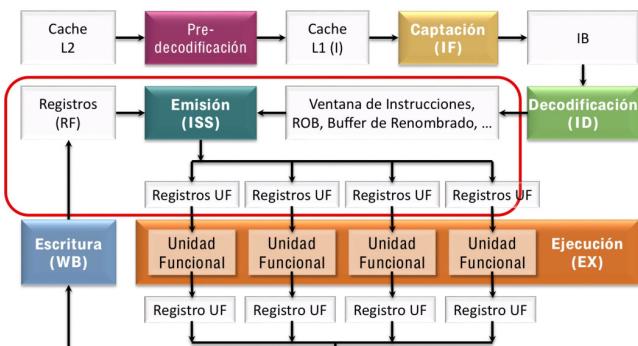
Cuanto más sofisticado sea un procesador superescalar, menos tiene que ajustarse a la ordenación de las instrucciones según se captan, para la ejecución y la modificación de los registros, de cara a mejorar los tiempos de ejecución. La única restricción es que el resultado del problema sea correcto.

1.3 Cauces superescalares

1.3.1 Decodificación paralela y predecodificación



1.3.2 Emisión paralela de instrucciones. Estaciones de reserva



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

→ Cola de registros.

Se irá rellenando conforme se van captando las instrucciones.

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

→ No tienen su operando	sub r5 r4 0 add [r3] 1 -	sub r6 r5 0 mult [r2] 1 -
→ Si tienen su operando	mult r5 [r1] 1 - [r5] 1 -	add r4 [r1] 1 - [r2] 1 -

Se han emitido [1] y [2]

Ha terminado [1] Ha terminado [2]: pueden emitirse [3] y [4]

→ Unidades Funcionales

add/sub: 2
mult: 1

[1] add r4,r1,r2 (2) → Ciclos que necesitan para ejecutarse por completo
[2] mult r5,r1,r5 (5)
[3] sub r6,r5,r2 (2)
[4] sub r5,r4,r3 (2)

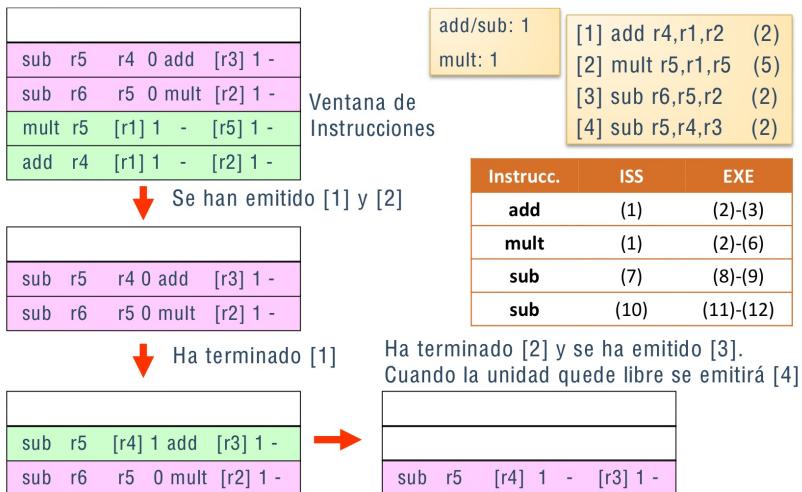
Instrucc. ISS EXE

add	(1)	(2)-(3)
mult	(1)	(2)-(6)
sub	(7)	(8)-(9)
sub	(7)	(8)-(9)

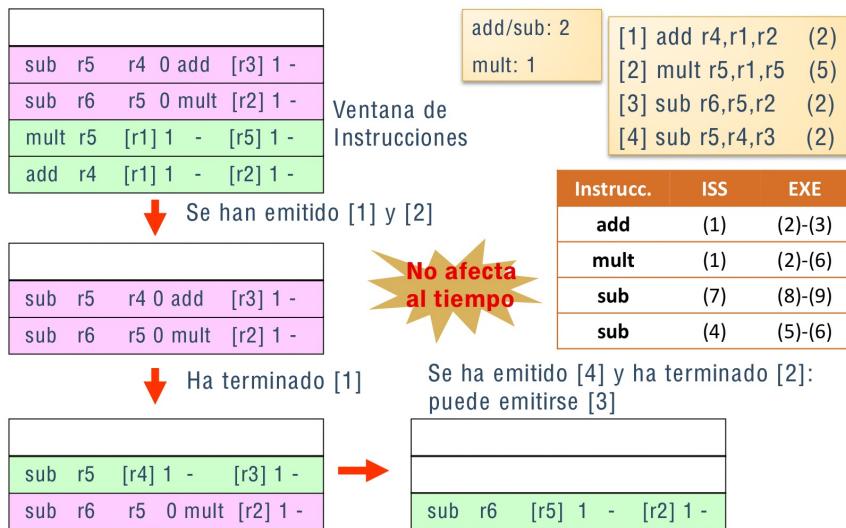
→ Ciclos en los que se ejecuta

→ Ciclo en el que se captó

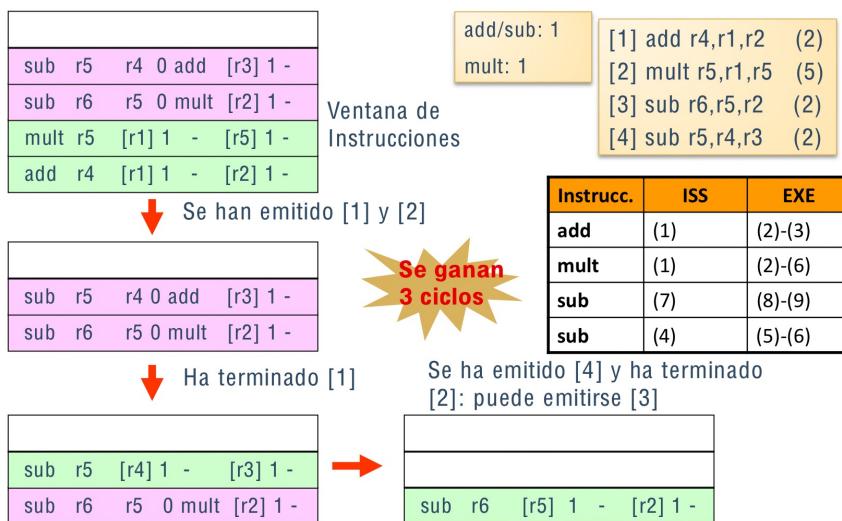
EJEMPLO



• Emisión paralela de instrucciones Desordenada



EJEMPLO

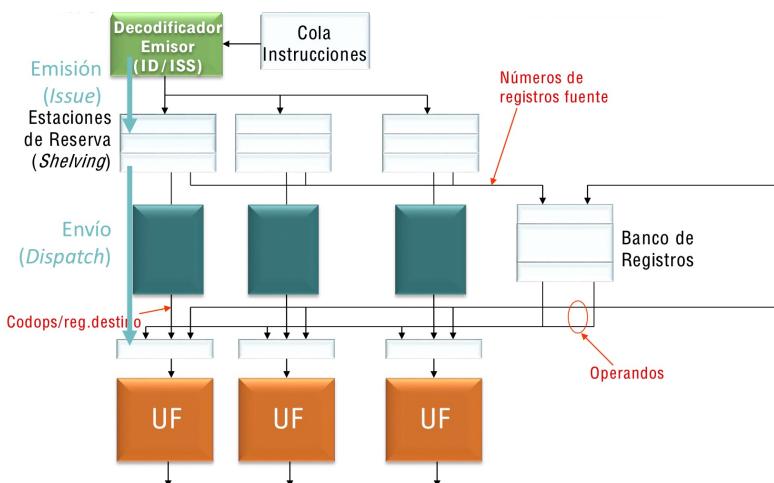


Estudiar sin publi es posible.

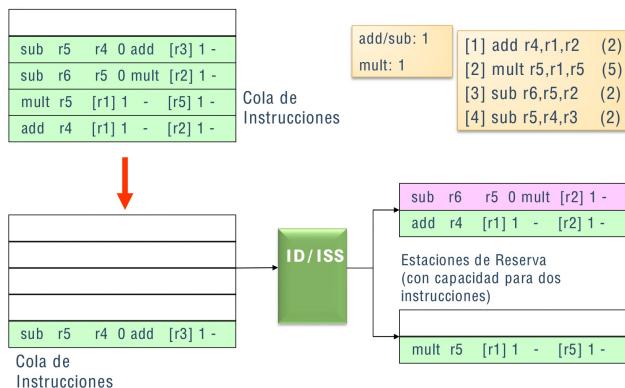
Compra Wuolah Coins y que nada te distraiga durante el estudio.



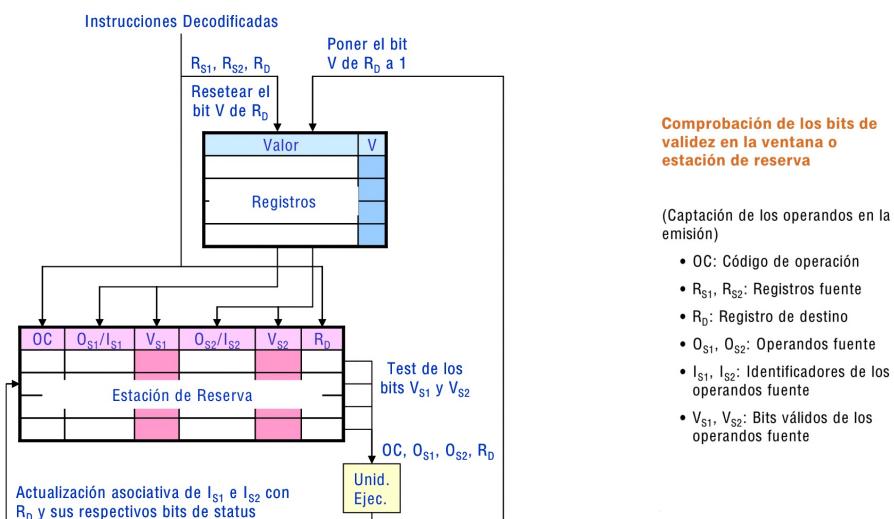
• Estaciones de reserva. Ventana de instrucciones distribuida



EJEMPLO



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



WUOLAH

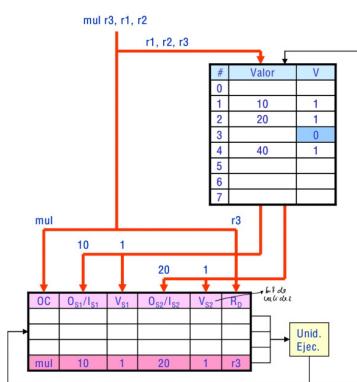
EJEMPLO

1

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

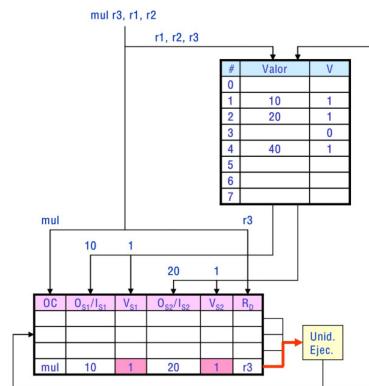


2

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

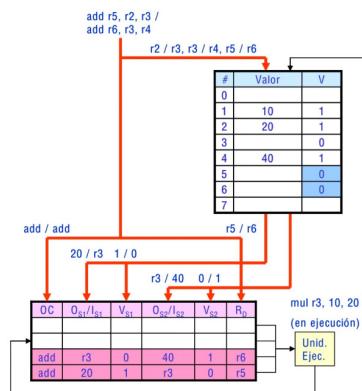


3

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

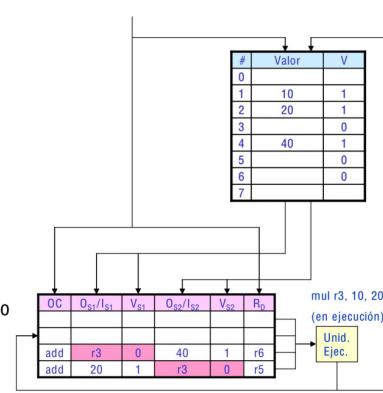


4

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)

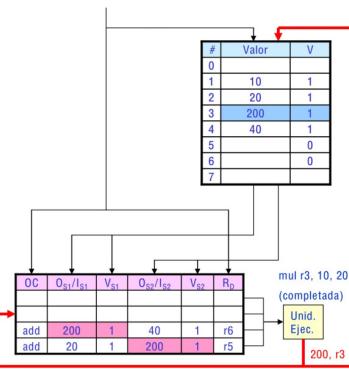


5

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

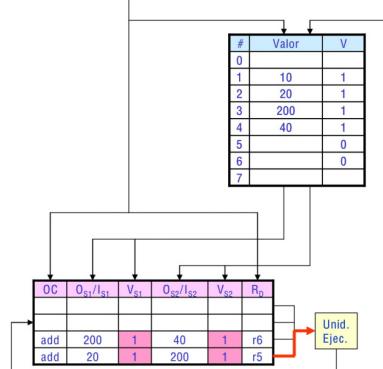


6

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 ($V_{S1} = 1$ y $V_{S2} = 1$)
- Así que pueden enviarse a la unidad de ejecución

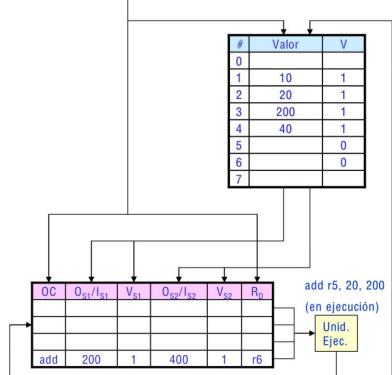


7

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

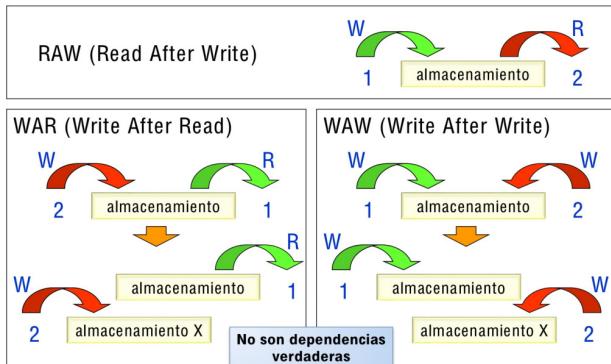
Ciclo i + 6 (cont.):

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



1.3.3 Renombramiento de registros

• Riesgos de datos



• Buffers de renombramiento

	Entrada Válida	Registro Destino	Valor	Valor Válido	Último
1	5	50	1	1	1
1	12	1200	1	1	1
Búsq. asoc. de r2	1	2	20	1	1
1	1	3	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮

Permite escrituras pendientes a un mismo registro. El bit de último de utilza para marcar cual ha sido el más reciente.

• 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

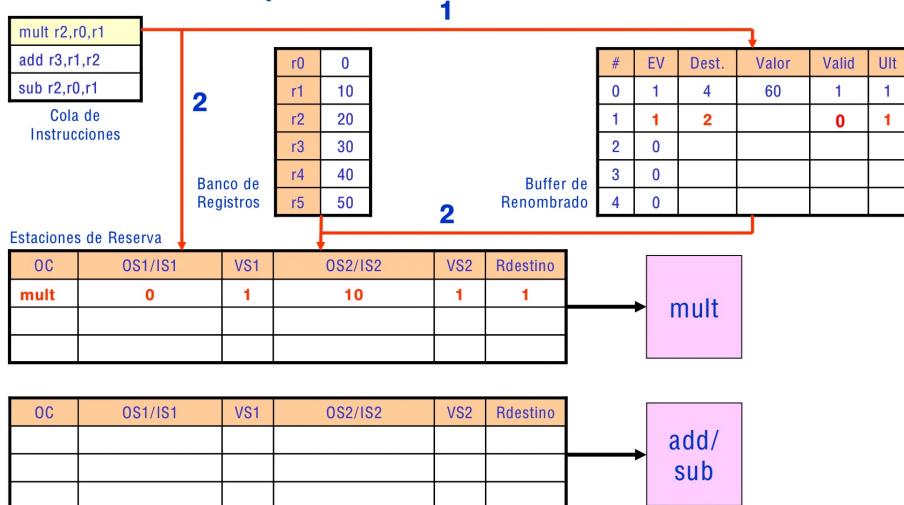
- Tipos de Buffers** (separados o mezclados con los registros de la arquitectura)
- Número de Buffers de Renombrado**
- Mecanismos para acceder a los Buffers** (asociativos o indexados)

Velocidad del Renombrado

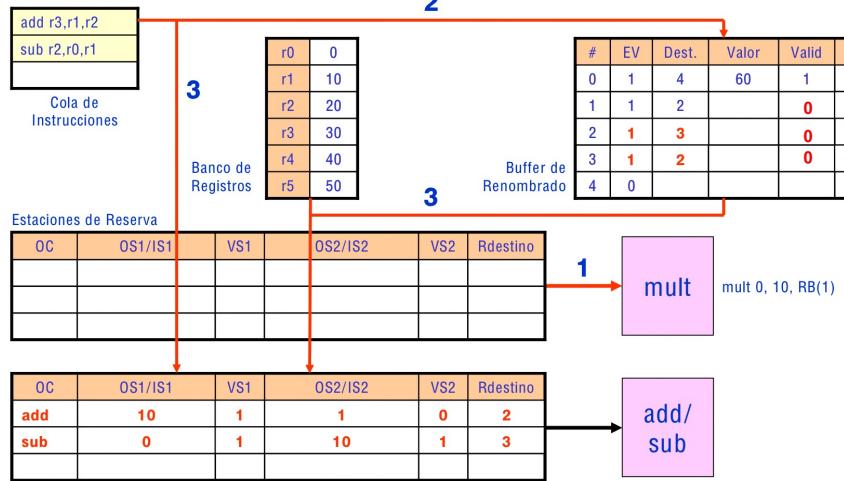
- Máximo número de nombres asignados por ciclo** que admite el procesador

• Algoritmo de Tomasulo

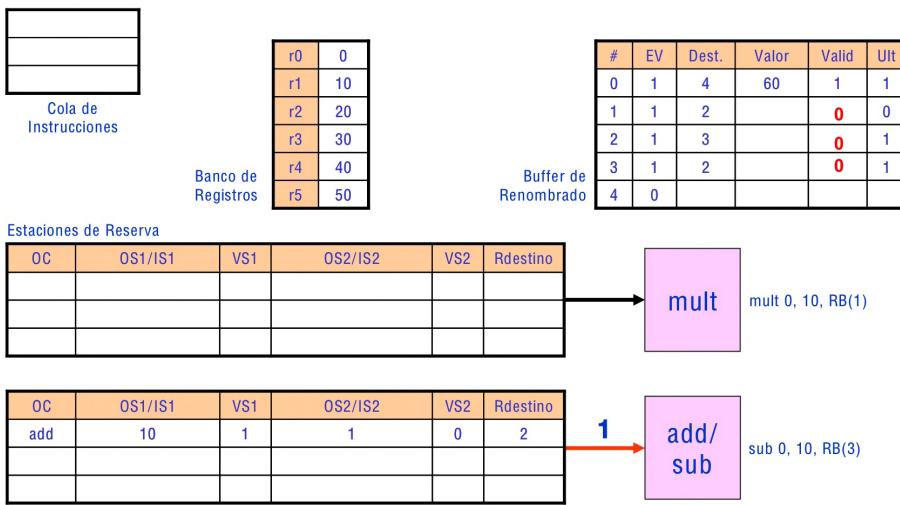
1. Emisión de la multiplicación



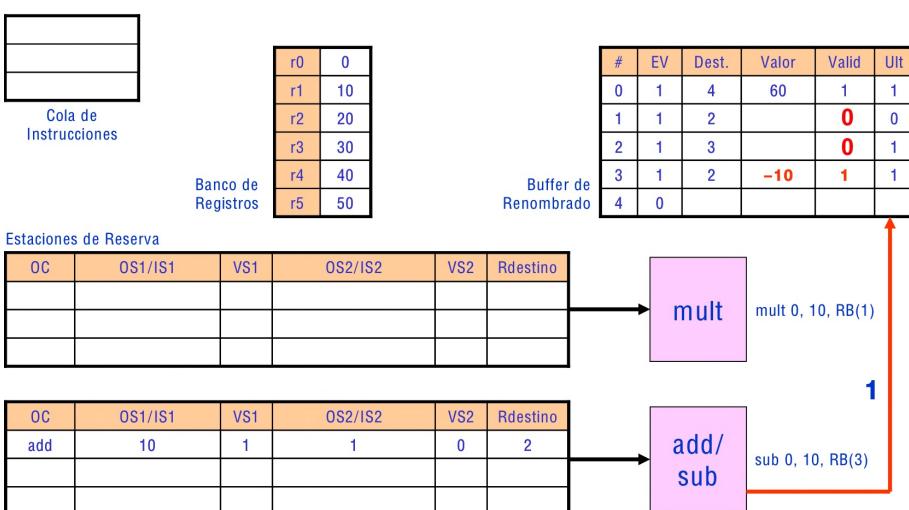
2. Envío de la multiplicación y emisión de la suma y la resta



3. Envío de la resta



4. Termina la resta

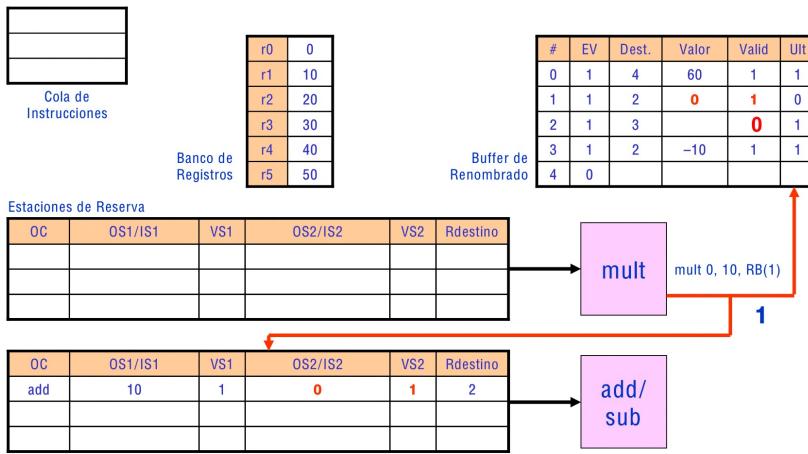


Estudiar sin publi es posible.

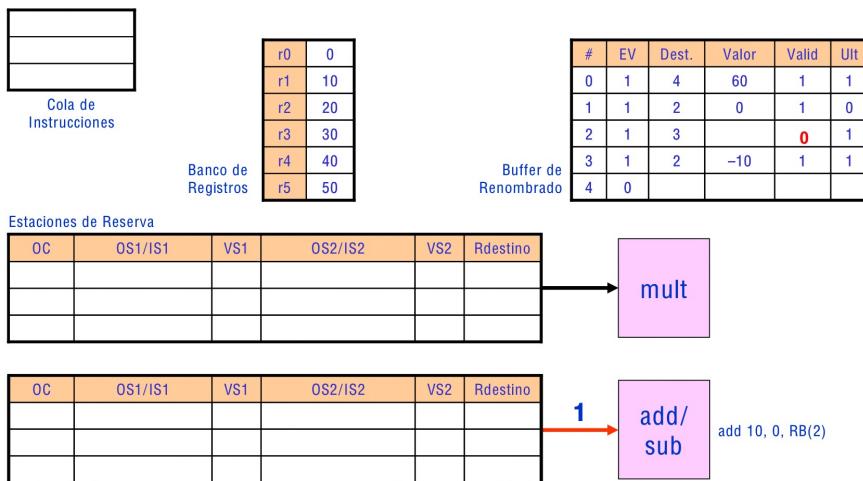
Compra Wuolah Coins y que nada te distraiga durante el estudio.



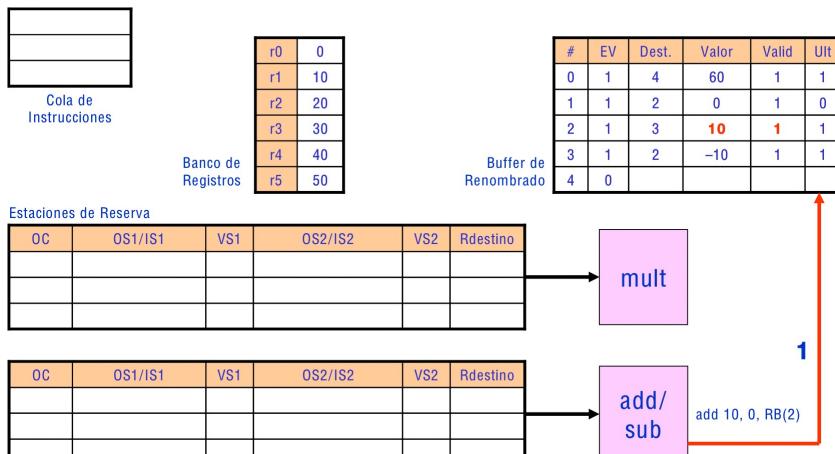
5. Termina la multiplicación



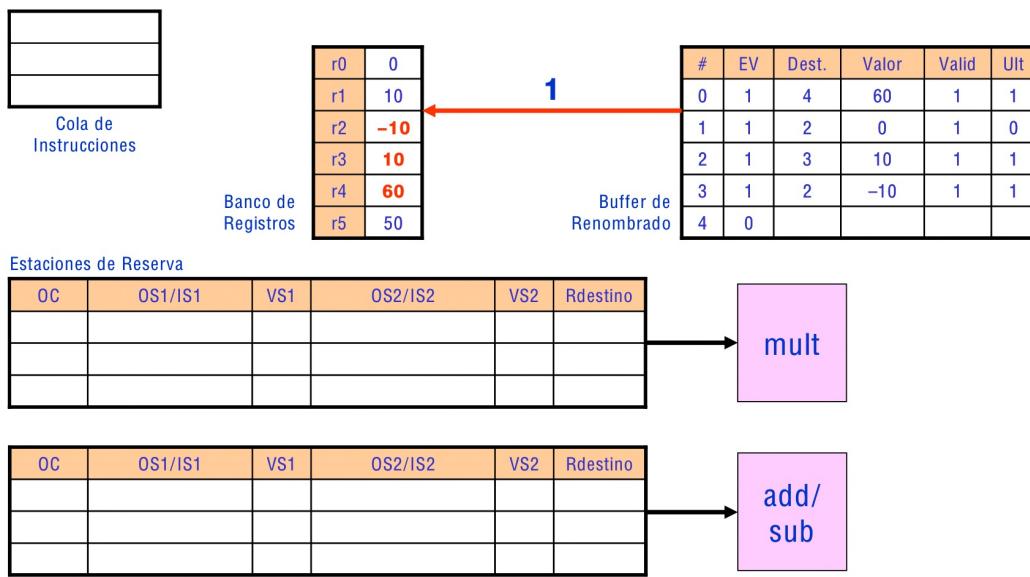
6. Envío de la suma



7. Termina la suma



8. Se actualizan los registros



2. Consistencia del procesador y procesamiento de datos

2.1 Consistencia. Reordenamiento

• Consistencia

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 (unidades funcionales) pero no se han modificado los registros de la arquitectura.
- El **final del procesamiento de la instrucción o momento en el 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 Complete).

La **consistencia** de un programa se refiere a:

- El **orden** en el que las instrucciones se **completan**.
- El **orden** en el 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 del programa**.

Consistencia de Procesador	Débil:	Deben detectarse y resolverse las dependencias	Power1 (90) PowerPC 601 (93) Alpha R8000 (94) MC88110 (93)	Tendencia
	Consistencia en el orden en que se completan las instrucciones			
Consistencia de Memoria	Fuerte:	Se consigue mediante el uso de ROB	PowerPC 620 PentiumPro (95) UltraSparc (95) K5 (95) R10000 (96)	↓ ↓
	Consistencia del orden de los accesos a memoria			

Tendencia / Prestaciones

WUOLAH

- Reordenamiento

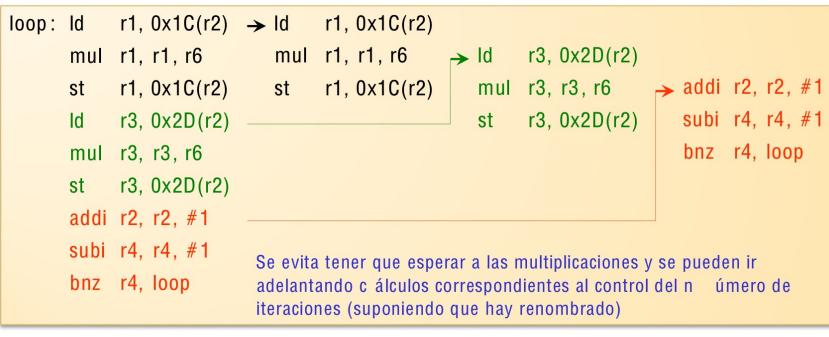
Las instrucciones de LOAD y STORE implican cambios en el procesador y en memoria.

LOAD: - Cálculo de dirección en ALU o unidad de direcciones.
- Acceso a cache.
- Escritura de dato en registro.

STORE: - Cálculo de dirección en ALU o unidad de direcciones.
- Esperar que esté disponible el dato a almacenar.

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 o otro que dio lugar a una falta de cache y accede a MP (memoria principal).



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)

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			

- 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 le preceden también.
- La consistencia se mantiene porque sólo las instrucciones que se retiran del ROB se completan (escribe en los registros de la arquitectura) y se retiran en el orden estricto de programa.

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

EJEMPLO:

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

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

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**.

2.2 Procesamiento especulativo de saltos

Los aspectos del procesamiento de saltos en un procesador de superescalares son:

- **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 esté disponible de 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 perjudicial ya que, al emitirse varias instrucciones por ciclo, prácticamente en cada ciclo puede haber una instrucción de salto.

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Gestión de Saltos Condicionales no Resueltos (Una condición de salto no se puede comprobar si no se ha terminado de evaluar)	Bloqueo del Procesamiento del Salto	Se bloquea la instrucción de salto hasta que la condición esté disponible (68020, 68030, 80386)
	Procesamiento Especulativo de los Saltos	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. (Tipica en los procesadores superescalares actuales)
	Múltiples Caminos	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)
Evitar saltos condicionales	Ejecución Vigilada (<i>Guarded Exec.</i>)	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 el salto de 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).

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.
- **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.
- **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)

EJEMPLO :

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

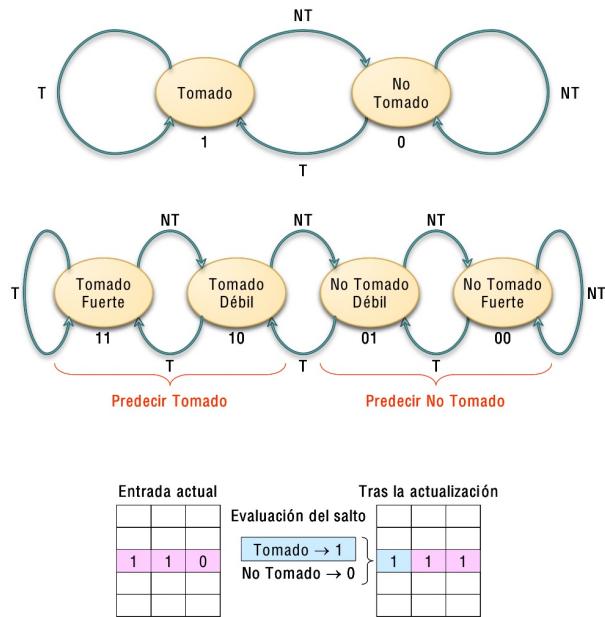
Predicción dinámica

- 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



WUOLAH

EJEMPLO



Instrucciones de ejecución condicional (Guarded Execution)

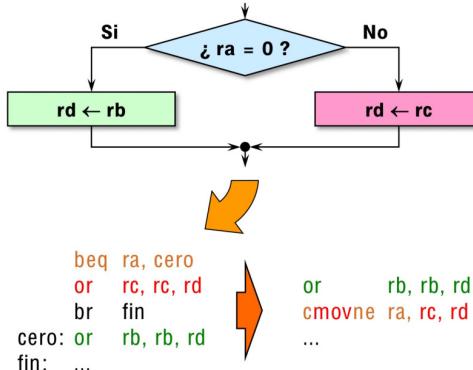
- Se pretende **reducir el número de instrucciones de salto** incluyendo en el **repertorio máquina instrucciones con operaciones condicionales** ('conditional operate instructions' o 'guarded instructions')
- Estas instrucciones tienen dos partes: la **condición** (denominada **guardia**) y la parte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

EJEMPLO

cmoveq 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**.

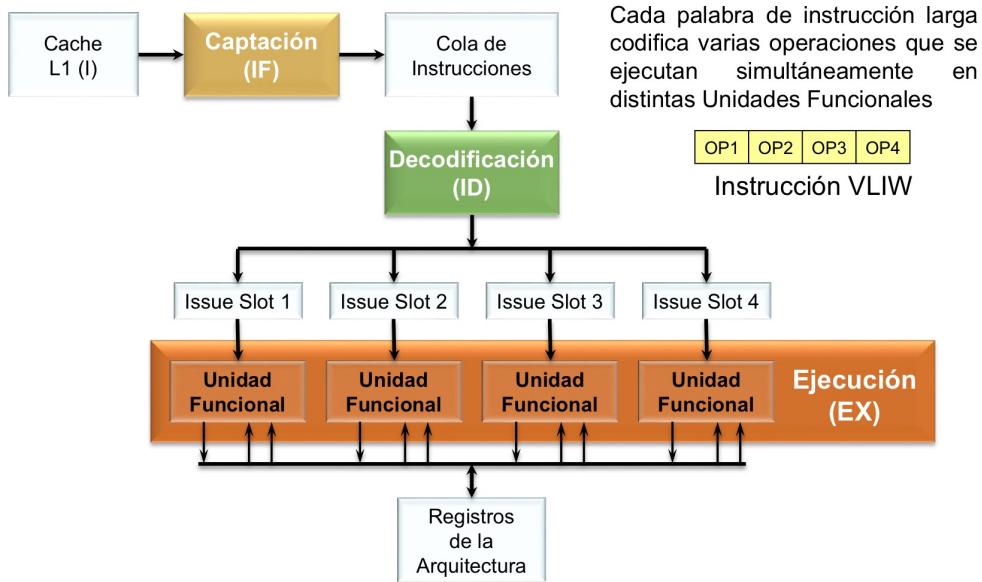
Sparc V9, HP PA, y Pentium ofrecen también estas instrucciones.



3. Procesamiento VLIW (Very Long Instruction Word)

3.1 Características generales y motivación (ILP hardware vs ILP software)

- 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) o 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 emite dos instrucciones por ciclo es relativamente pequeño, pero crece a medida que se pretenden emitir más instrucciones por ciclo).



3.2 Planificación estática

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,...

Planificación dinámica

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

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



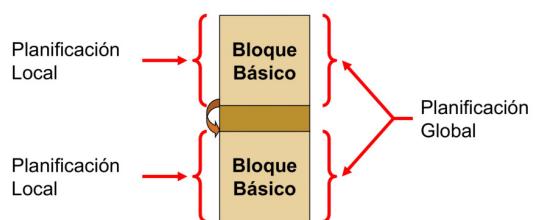
`loop: ld f0, 0(r1)
 addd f4, f0, f2
 sd f4, 0(r1)
 subui r1, r1, #8
 bne r1, loop`

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

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

Hay dos tipos de planificación estática:

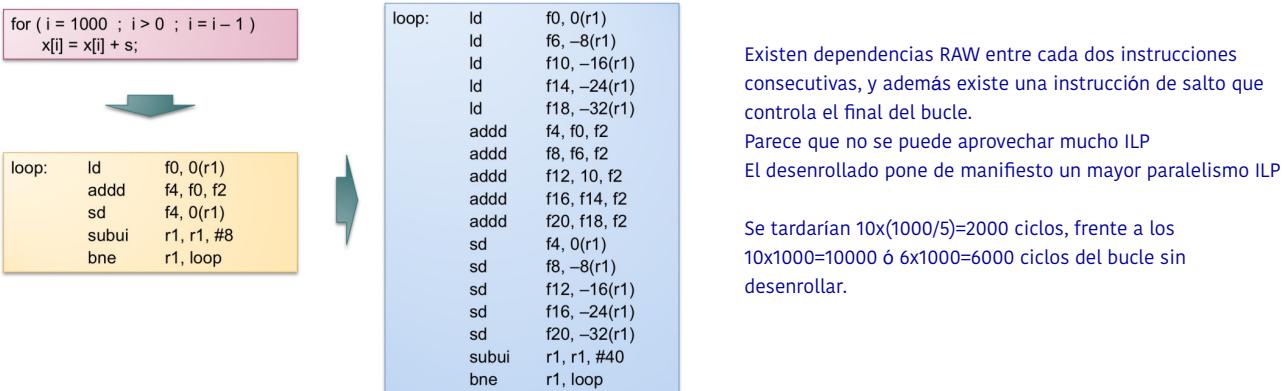
- **Planificación local:** actúa sobre un bloque básico (mediante desenrollado de bucles y planificación de las instrucciones del cuerpo aumentado del bucle).
- **Planificación global:** actúa considerando bloques de código entre instrucciones de salto.



3.2.1 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

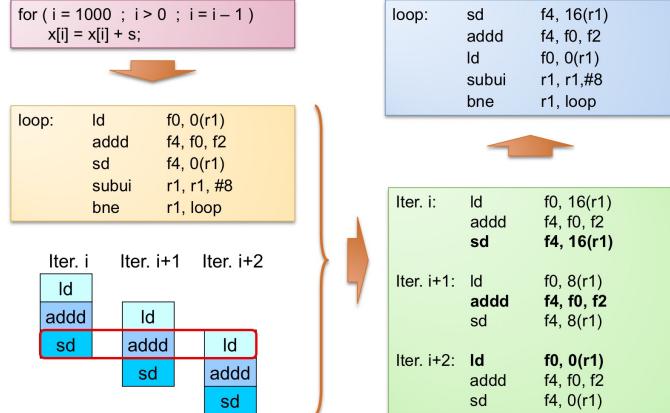


	Instrucción Entera	Instrucción FP	Load/Store	Ciclo
loop			Id f0, 0(r1)	1
			Id f6, 0(r1)	2
			addd f4, f0, f2	3
			addd f8, f6, f2	4
			addd f12, f10, f2	5
			addd f16, f14, f2	6
	subui r1, r1, #40		sd f8, -8(r1)	7
			sd f12, 24(r1)	8
			bne r1, loop	9
			sd f16, 16(r1)	10

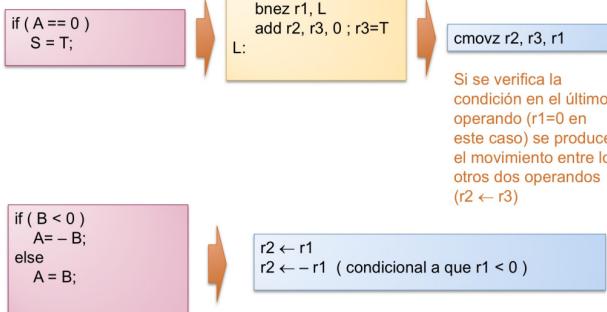
3.2.2 Planificación estática global

- 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
 - Especulación

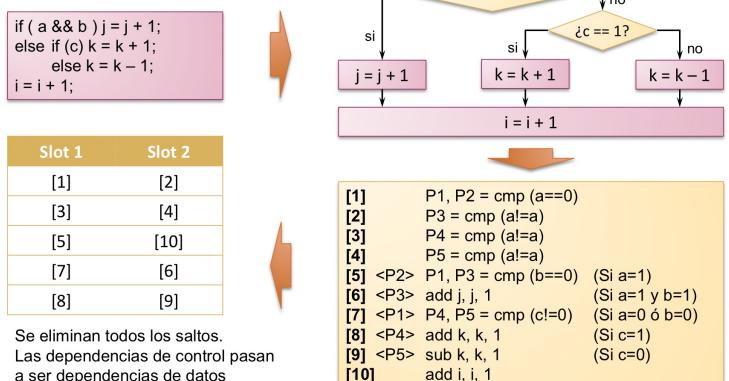
Planificación estática con segmentación software



Instrucciones de ejecución condicional



Ejemplo de Ejecución VLIW (con dos slots)



Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



3. 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.

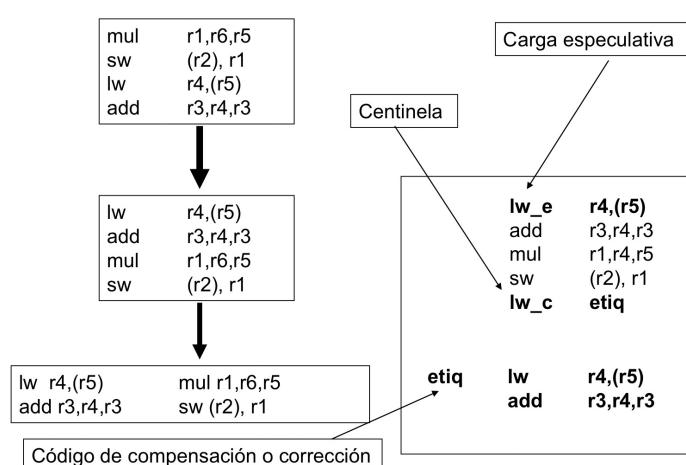


Se aprovecha el slot de acceso a memoria. Si beqz da lugar al salto se ha ejecutado una instrucción de forma innecesaria. Otro problema son las excepciones.

Uso de centinelas para permitir la especulación de las referencias a memoria

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)



WUOLAH