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

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

Objetivos Lección 13

- Realizar una planificación estática de instrucciones para un procesador VLIW sencillo
- Usar instrucciones predicado para eliminar saltos

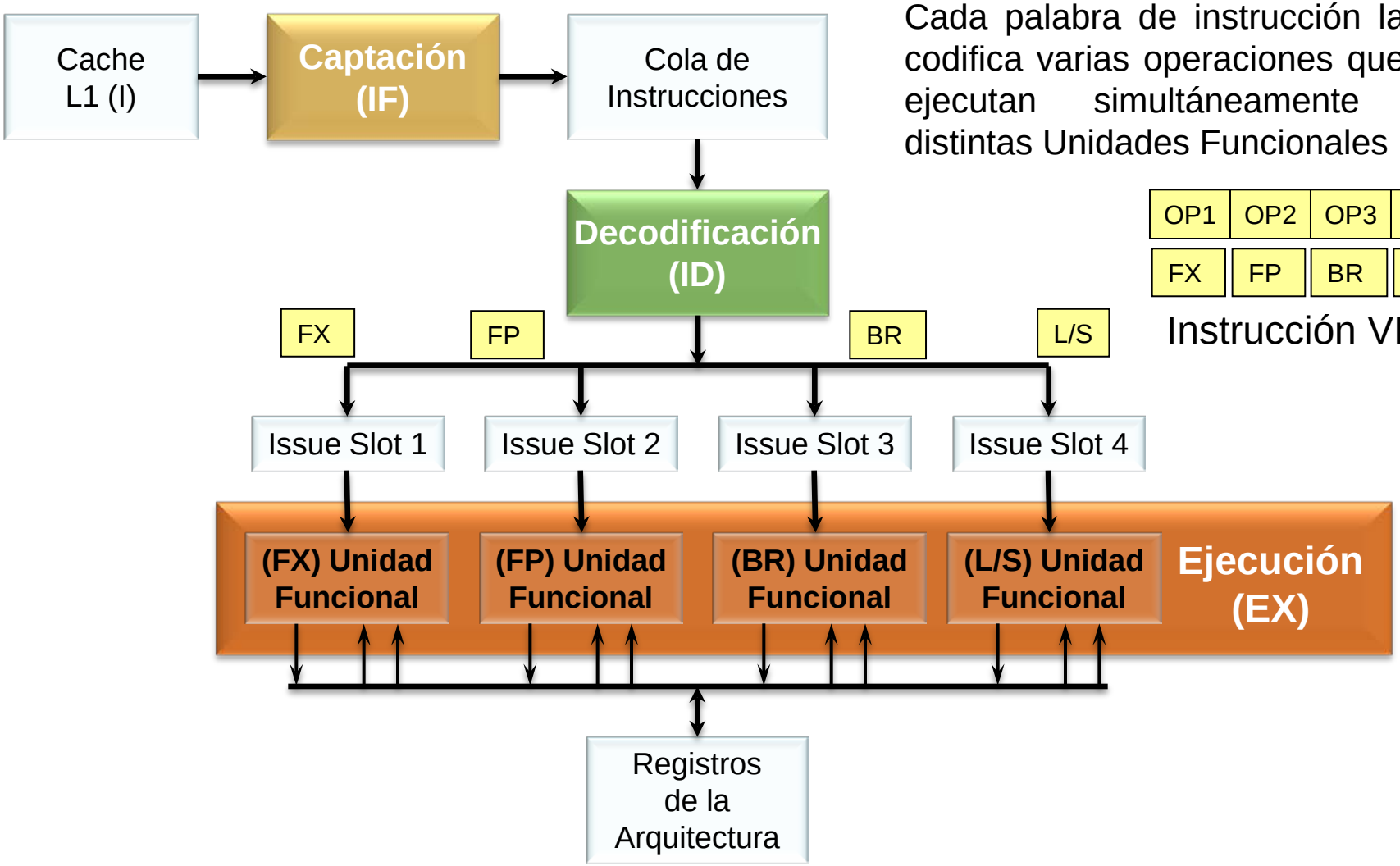
Bibliografía

- Fundamental
  - 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 (Very Large Inst. Word) II



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

OP1	OP2	OP3	OP4
FX	FP	BR	L/S

Instrucción VLIW

Contenido de la Lección 13

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

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 que aparecen en las tablas:

Instr. que produce el resultado	Instr. que usa el resultado	Espera
Operación FP	Operación FP	3
Operación ALU	Store	2
Load	Operación FP	1
Load	Store	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)	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) ; f4<-M[0+r1]	sd f4,**8**(r1) ; f4<-M[8+r1]	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**	

FX/BR	FP	L/S

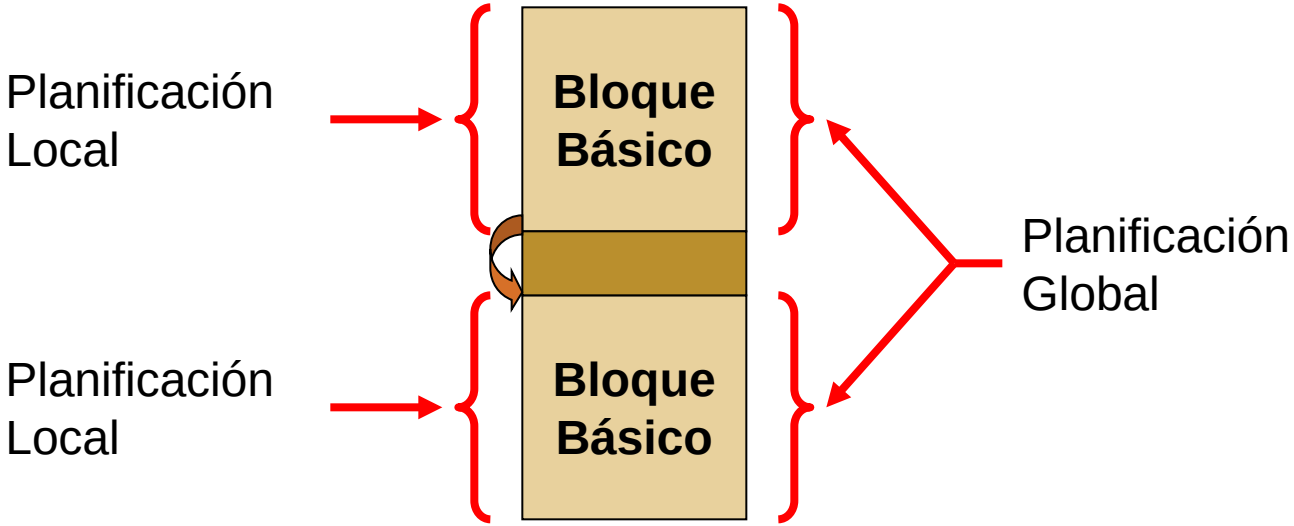
Instrucción VLIW

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**	

1 (ld f0 → ld f6)

2 (addd f4 → sd f4)

1 (subui → bne)

1 (bne → sd f20)

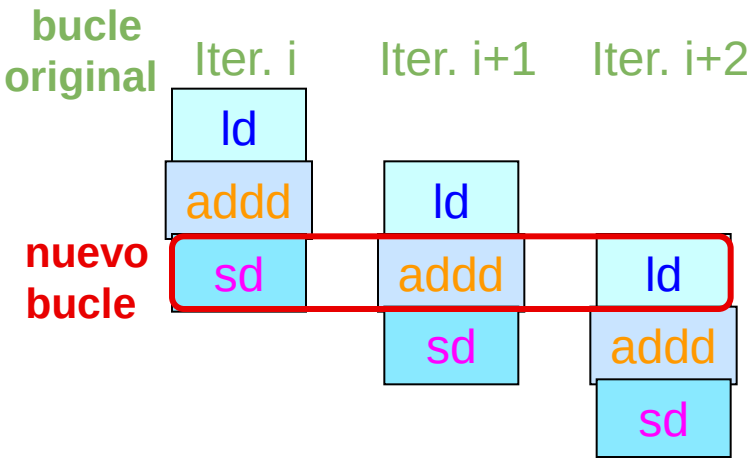
Se tardarían 10x(1000/5)=2000 ciclos, frente a los 10x1000=10000 ó 6x1000=6000 ciclos del bucle sin desenrollar.

escalar

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
```



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
		1		3
	bne r1,loop			4
		1		5
	Slot 1	**Slot 2**	**Slot 3**	

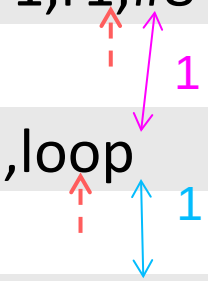
- Se tardarían 5x1000=5000 ciclos (algunos más si se consideran las instrucciones previas al inicio del bucle con segmentación software y las posteriores al final del bucle).
- Si se desenrolla este bucle ya segmentado, se pueden mejorar mucho más las prestaciones.

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

escalar

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,**8**(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

escalar

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    ; 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;
```

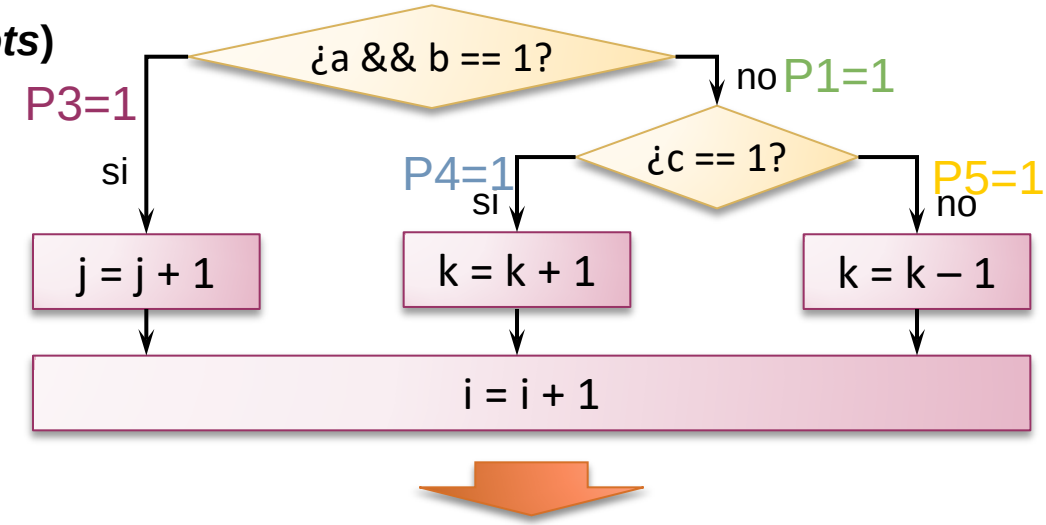
r2 ← r1
r2 ← – r1 (condicional a que r1 < 0)

Ej cmov

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**

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.

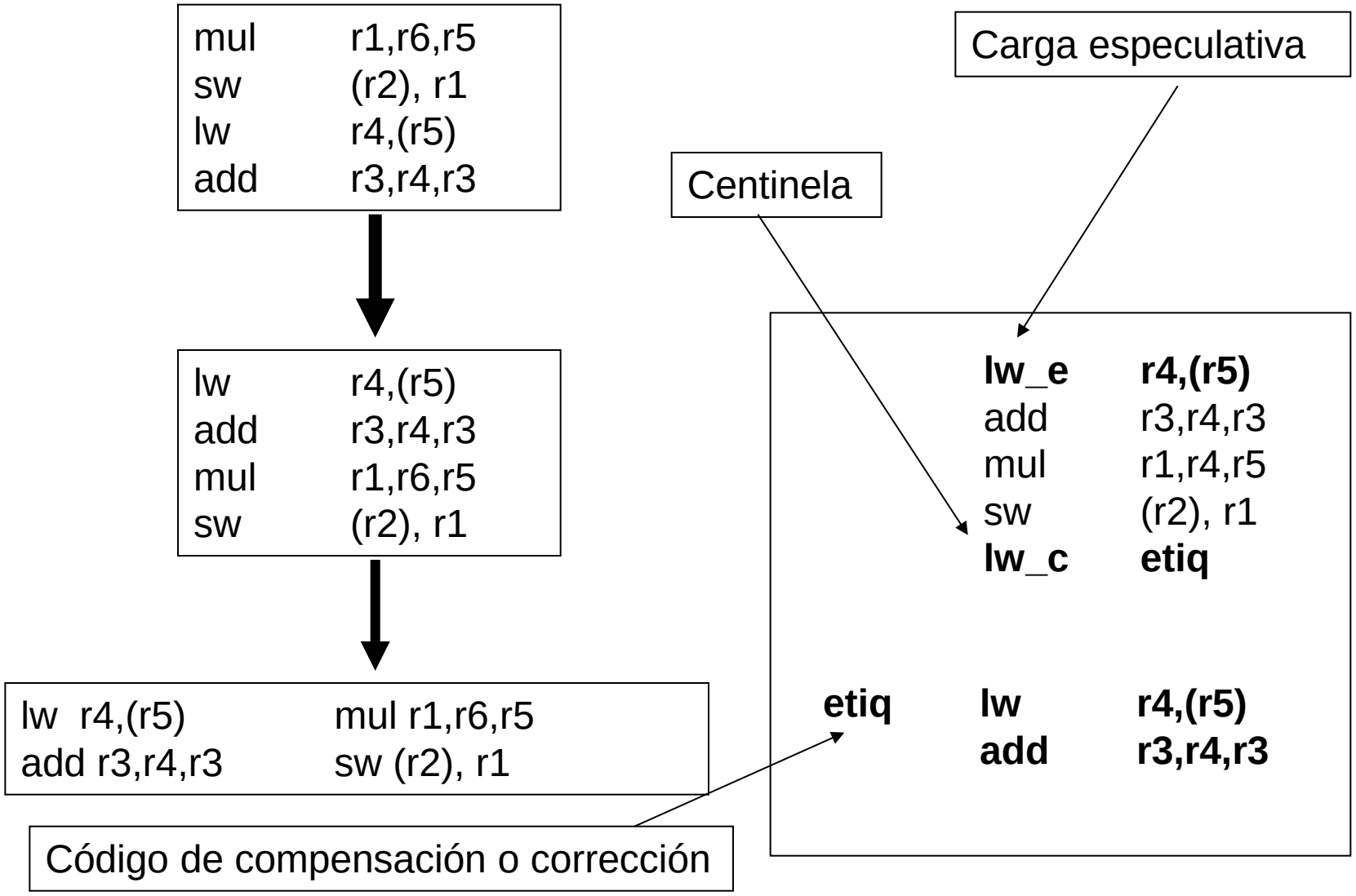
Uso de Centinelas para Permitir la Especulación de las Referencias a Memoria

- 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

Load especulativo

- 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



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 departamental 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.