Tema 4: Técnicas avanzadas de segmentación

- Introducción
- Planificación estática
- Planificación dinámica
- Predicción de saltos
- Ejemplos de predictores

4.1 Introducción

- Estudio de técnicas avanzadas para mejorar el rendimiento
- Técnicas:
  - ✓ Planificación
  - ✓ Predicción
- Planificación para reducir nº de detenciones debido a depen. entre instr.
  - ✓ Planificación *software* estática: desenrollamiento de bucles
  - ✓ Planificación *hardware* dinámica: algoritmo de Tomasulo
- Predicción hardware dinámica para reducir detenciones por saltos
  - ✓ Buffer de predicción de salto
  - ✓ Buffer de destino de salto

4.1 Introducción

- Segmentación:
  - Mejora el rendimiento del procesador solapando la ejecución de varias instrucciones
  - CPI = CPI_{ideal} + det. estructurales + det RAW + det WAR + det. WAW + det. control
- CPI_{ideal} mide el máximo rendimiento alcanzable por la implementación (=1)
- Hasta ahora: las técnicas básicas para reducir riesgos son:
  - Estructurales: replicación o segmentación de las unidades funcionales de ejecución
  - RAW: adelantamientos y planificación SW
  - WAR: no se dan por ahora
  - WAW: son poco frecuentes y de resolución sencilla
  - Control: salto retardado o predicción estática de salto
- Desde ahora: ILP: Instruction Level Parallelism
  - Paralelismo potencial entre las instrucciones de un programa
  - La implementación debe explotar al máximo el ILP
- Objetivo del tema:
  - Técnicas avanzadas (estáticas y dinámicas) para reducir el CPI de un procesador segmentado explotando el ILP

4.1 Introducción
Comparación planificación SW - HW

- Planificación SW (estática)
  - El compilador planifica: salto retardado, desenrollamiento, etc
    - Se rellenan los huecos de latencia en tiempo de compilación
  - Ventajas:
    - El HW es simple y no hace falta control de detenciones.
    - El compilador tiene una visión global del código y reordena sólo en tiempo de compilación
  - Inconvenientes:
    - Problemas con las dependencias no conocidas en tiempo de compilación (datos en memoria)
    - Necesidad de un buen compilador. Necesidad de recompilar si cambia la arquitectura.
- Planificación HW (dinámica)
  - El HW planifica (reordena) las instrucciones en tiempo de compilación.
  - Ventajas
    - Compilador simple
    - Mejora el rendimiento en ejecución de códigos compilados para otras arquitecturas compatibles
  - Inconvenientes
    - HW de control complejo y caro.

4.1 Introducción
Comparación planificación SW - HW

- Ejemplo
  - ✓ i486: procesador segmentado básico
  - ✓ Pentium: procesador superescalar (factor 2) con planificación estática
    - Para emitir dos inst./ck una debe ir al cauce U (int. compleja) y otra al V (int. simple)
  - ✓ Pentium Pro: procesador superescalar (factor 3) con planificación dinámica

- Supongamos un ejecutable binario compilado para el 486
  - ✓ Si se ejecuta sin recompilar en el Pentium
    - Las instrucciones no se reodenaron pensando en poner una instr. U seguida de otra V
    - No se gana rendimiento a no ser que recompilemos el código
  - ✓ Si se ejecuta sin recompilar en el Pentium Pro
    - La unidad de control reordena las instrucciones en tiempo de compilación
    - Mejora el tiempo de ejecución aunque no recompilemos el código

4.2 Planificación estática:
Desenrollamiento de lazos

- Técnica de planificación software realizada por el compilador
- Objetivo: reducir el número de detenciones por riesgos de control
- Ejemplo: lazo con iteraciones independientes
  - ✓ No existen dependencias verdaderas entre iteraciones
  - ✓ Las iteraciones se pueden ejecutar en cualquier orden (lazo paralelo)

```
double x[1000]          inicio: ld     f0, 0(r1)  ;f0=elemento vector
                                addd   f4, f0, f2 ;f2 contiene s
for (i=1;i<=1000;i++)           sd     0(r1), f4  ;almacena resultado
    x[i] = x[i] + s;            subi   r1, r1, #8 ; decrementa el
                                                   ; puntero (8bytes)
                                bnez   r1, inicio ; salta si r1≠0
```

El vector x se almacena en memoria en las posiciones 0-7999

r1 apunta inicialmente a la posición 7992 (última componente x[1000])

4.2 Desenrollamiento de lazos
Procesador segmentado usado en ejemplos

- Procesador carga/almacenamiento segmentado
- Saltos tienen latencia de un ciclo
  - ✓ Resuelto mediante salto retardado con un hueco de relleno
- Unidades funcionales PF segmentadas
  - ✓ Adelantamientos entre UFs PF

Ins produce resultado	Ins utiliza resultado	Latencia (ciclos)
Aritmética PF	Aritmética PF	3
Aritmética PF	Almacenamiento PF	2
Carga PF	Aritmética PF	1
Carga PF	Almacenamiento PF	0
Aritmética entera	Salto condicional	1

4.2.1 Desenrollamiento de lazos
Ejemplo

a) Ejecución sin planificación

```
inicio:  ld      f0, 0(r1)
         detención
         addd    f4, f0, f2
         detención
         detención
         sd      0(r1), f4
         subi    r1, r1, #8
         detención
         bnez    r1, inicio
         detención (hueco)
```

b) Con planificación y salto retardado

```
inicio:  ld      f0, 0(r1)
         subi    r1, r1, #8
         addd    f4, f0, f2
         detención ; 2ck de latencia al sd
         bnez    r1, inicio; la instrucción sd
         sd      8(r1), f4; ocupa el hueco de
                            retardo
```

sd y subi dependen por r1 pero el comp. resuelve el desorden sumando 8 en sd

- Sin planificación se necesitan 10 ciclos por iteración
- Con salto retardado, se necesitan 6 ciclos
- La gestión del lazo requiere 3 (2) ciclos por iteración

4.2.1 Desenrollamiento de lazos Ejemplo

- Desenrollamiento de 2 iteraciones
  - ✓ Repetir 2 veces el cuerpo del bucle
  - ✓ Modificar primer salto

```
inicio:  ld    f0, 0(r1)
         addd  f4, f0, f2
         sd    0(r1), f4
         subi  r1, r1, #8
         beqz  r1, final
         nop
         ld    f0, 0(r1)
         addd  f4, f0, f2
         sd    0(r1), f4
         subi  r1, r1, #8
         bnez  r1, inicio
final:
```

- Por ahora no hemos ganado nada
- Pero da pie a posibles mejoras:
  - ✓El primer salto no es necesario si el nº de iteraciones es par
  - ✓Al quitar el salto eliminas riesgos de control y aumenta el tamaño del bloque básico
  - ✓Eliminar el primer subi
    - ✓ cambiando el desplazamiento del ld y sd siguientes (-8 en vez de 0)
    - ✓ restando 16 en vez de 8 en el segundo subi
  - ✓Rellenar hueco de retardo

4.2.1 Desenrollamiento de lazos Ejemplo

- Desenrollamiento 2 iteraciones
  - ✓ Suponiendo un número de iteraciones par
  - ✓ Eliminando primer subi y ocupando hueco retardo salto

```
inicio:  ld    f0, 0(r1)
         addd  f4, f0, f2
         sd    0(r1), f4
         ld    f6, -8(r1)
         addd  f8, f6, f2
         subi  r1, r1, #16
         bnez  r1, inicio
         sd    8(r1), f8
final:
```

- Hemos ganado algo pero
  - ✓Tenemos RAW entre ld-addd, addd-sd y subi-bnez
  - ✓Antidependencias y dependencias de salida entre instrucciones que provienen de las dos iteraciones
  - ✓Total:8ck+5stall=13ck → 6.5ck por elemento de x[]
- Soluciones:
  - ✓Planificar para evitar los RAW
  - ✓Eliminar las dependencias falsas (antidependencias y dependencias de salida)
    - ✓ renombre de registros
    - ✓ utilizar distintos registros para ld y addd en cada iteración desenrollada
  - ✓OJO: sd y ld se pueden cambiar de orden sólo si no acceden a la misma posición de memoria

4.2.1 Desenrollamiento de lazos Ejemplo

- Desenrollamiento 2 iteraciones
  - ✓ Suponiendo un número de iteraciones par
  - ✓ Planificación y renombre

```
inicio:  ld     f0, 0(r1)
         ld     f6, -8(r1)
         addd   f4, f0, f2
         addd   f8, f6, f2
         subi   r1, r1, #16
         sd     16(r1), f4
         bnez   r1, inicio
         sd     8(r1), f8
final:
```

☑ Ventajas:
- Eliminadas todas las detenciones ld-addd
- Total: 8ck para dos elementos
- Es decir: 4ck por elemento

☒ Inconvenientes
- Se necesitan más registros

4.2.1 Desenrollamiento de lazos Ejemplo

- Desenrollamiento de 4 iteraciones

```
inicio:  ld     f0, 0(r1)
         ld     f6, -8(r1)
         ld     f10, -16(r1)
         ld     f14, -24(r1)
         addd   f4, f0, f2
         addd   f8, f6, f2
         addd   f12, f10, f2
         addd   f16, f14, f2
         sd     0(r1), f4
         sd     -8(r1), f8
         subi   r1, r1, # 32
         sd     16(r1), f12
         bnez   r1, inicio
         sd     8(r1), f16
```

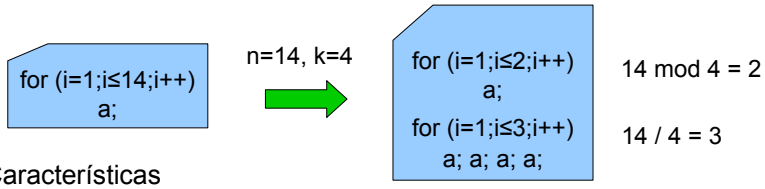
Número de iteraciones múltiplo de 4

- Se han eliminados 3 saltos y 3 subi
- Eliminadas todas las detenciones
- Total: 14 ciclos para 4 elementos
- Es decir: 3.5 ciclos por elemento

✓ Se necesita mayor número de registros

4.2.2 Desenrollamiento de lazos Conclusiones

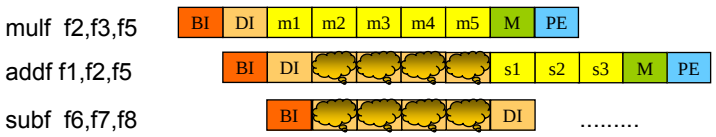
- En programas reales
  - ✓ Para un lazo genérico de *n* iteraciones y desenrollamiento k veces → implementar dos lazos consecutivos
    1. El lazo original con las primeras *n mod k* iteraciones
    2. El lazo desenrollado con índice contador *n/k*



- Características
  - ✓ Lo planifica el compilador en las primeras etapas del proceso de compilac.
  - ✓ Crea un bloque básico de mayor tamaño que el cuerpo del lazo original, lo que permite obtener mayor beneficio de la planificación.
  - ✓ Mejora el rendimiento pero incrementa el tamaño del código
  - ✓ Técnica aplicable a procesadores con emisión múltiple.

4.3 Planificación dinámica de instrucciones Concepto

- En DLX, si una instrucción se detiene por un riesgo RAW, las siguientes instrucciones se detienen también.



- Mejora: permitir que subf entre en el cauce y comience su fase de ejecución, ya que sus operandos están disponibles (no depende)

- Si una instrucción no puede avanzar se detiene, pero deja fluir las siguientes. La inst. subf comienza su etapa EJ antes que addf

4.3 Planificación dinámica de instrucciones

- Las instrucciones se buscan y decodifican en orden, pero pueden comenzar su fase de ejecución fuera de orden.
- Las instrucciones se ejecutan cuando tienen sus operandos disponibles (aunque no respeten el orden del programa)
  - ✓ Sus operandos están disponibles porque la instrucción no depende de las que aun están ejecutándose
- Necesario desacoplar la decodificación y la lectura de operandos.
- La etapa de decodificación (DI) se divide en dos:
  1. Emisión (*issue*): decodificar, comprobar riesgos estructurales.
  2. Lectura de operandos: lee cada operando cuando esté disponible. Espera en esta etapa mientras exista riesgo RAW.

4.3 Planificación dinámica de instrucciones

- Problemas:
  - ✓ Aparición de nuevos riesgos:
    - Antes: ejecución en orden: sólo riesgos RAW y también WAW con instr. multiciclo
    - Ahora: ejecución fuera de orden: más riesgos WAW y WAR

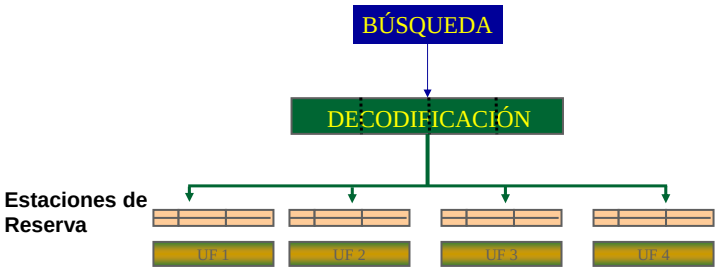
```
divd f0, f2, f4
addd f10, f0, f8
subd f8, f8, f14
```

- Si `subd` se ejecuta y escribe en f8 antes de que `addd` lea el operando f8, se produce un error → hay que detener la escritura en registro f8 (etapa PE) hasta que instrucciones previas lo hayan leído.
- Si fuese `subd f0, f8, f14` → WAW

  - ✓ Interrupciones imprecisas: dificultad para mantener las interrupciones precisas en una ejecución fuera de orden.
- Ejemplos de planificación dinámica:
  - ✓ Macador (*Scoreboard*). Anticuado. No lo vamos a estudiar.
  - ✓ Algoritmo de Tomasulo

4.3.1 Algoritmo de Tomasulo

- Origen: en el *IBM 360/91* (1967) con el objetivo de aumentar el rendimiento de las operaciones punto flotante.
- Cada instrucción se decodifica y se aloja en una estación de reserva, ER, donde permanece hasta que
  - ✓ disponga de sus operandos y
  - ✓ su UF correspondiente esté libre.



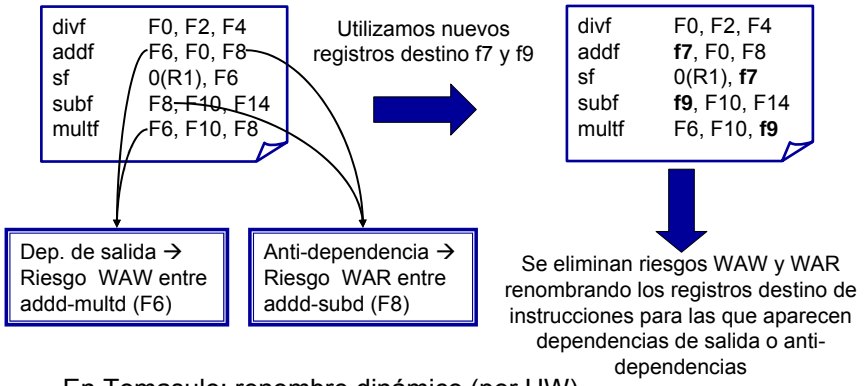
4.3.1 Algoritmo de Tomasulo

- Los riesgos RAW se evitan ejecutando una instrucción sólo cuando sus operandos están disponibles.
- Para evitar riesgos WAR y WAW, se hace un renombre dinámico de registros relacionando el registro destino de cada instrucción con la estación de reserva que aloja esa instrucción.
- Una estación de reserva almacena los operandos que estén disponibles (se copian desde el banco de registros) y el código de operación.
- Si algún operando no está disponible, la estación de reserva almacena el nombre de la estación de reserva que aloja la instrucción que produce ese operando.

Renombre de registros ➔ elimina riesgos WAR y WAW.

4.3.1 Algoritmo de Tomasulo

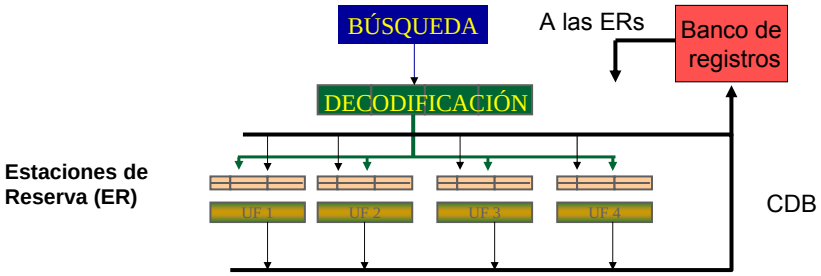
- Ejemplo de renombre de registros estático (por SW) :
  - ✓ La instrucción addd está parada esperando a que termine la división



- En Tomasulo: renombre dinámico (por HW)

4.3.1 Algoritmo de Tomasulo

- Características del Algoritmo de Tomasulo:
  - ✓ Bus de datos común (CDB): permite a las estaciones de reserva que esperan un operando la carga simultánea del mismo.



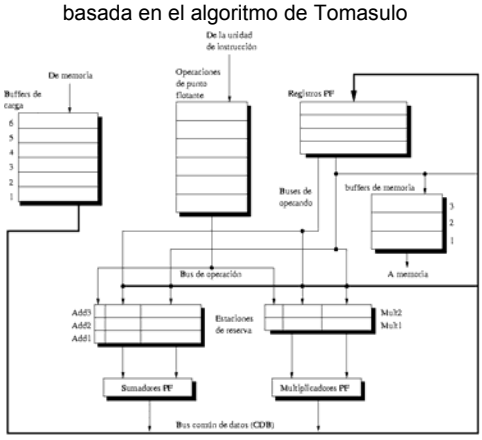
4.3.1 Algoritmo de Tomasulo

- Características del Algoritmo de Tomasulo:
  - ✓ Renombre dinámico de registros a través de las estaciones de reserva
  - ✓ Bus de datos común (CDB): permite a las estaciones de reserva que esperan un operando la carga simultánea del mismo.
    - Las instrucciones pendientes de un operando identifican la estación de reserva que contiene la instrucción que va a producir el operando
    - Los registros pendientes de recibir un operando también identifican la estación de reserva que contiene la instrucción que va a producir el operando
    - Cuando la instrucción se completa y se produce el operando esperado, éste se radia a todas las ER que lo necesitan y al registro destino.
    - Cuando hay sucesivas instrucciones que tienen idéntico registro destino, éste sólo identifica a la estación de reserva que contiene la última de estas instrucciones: Sólo la última de estas instrucciones es la que actualiza el registro.
  - ✓ Las ER obtienen cada operando cuando está disponible: desde el banco de registros o desde el CDB
  - ✓ Detección de riesgos y control de la ejecución distribuido (desde las estaciones de reserva).

4.3.1 Algoritmo de Tomasulo

- Contenido de las ER:
  - ✓ Instrucciones emitidas que esperan ejecución.
  - ✓ Operandos para esas instrucciones (si ya han sido calculados) o el origen de los operando en otro caso.
    - Es decir, en caso de que el operando no esté aun disponible, se vigila a la ER que contiene la instr. que produce ese operando
  - ✓ Información de control de ejecución (estado de la instr.)
  - ✓ Etiquetas para el control de los riesgos (flags de dependencia)
- Los resultados de las UF y de memoria (loads) van al CDB.
- Sólo consideramos operaciones PF y ld/sd de operandos PF

Unidad de punto flotante del MIPS basada en el algoritmo de Tomasulo



4.3.1 Algoritmo de Tomasulo

- Pasos para ejecutar una instrucción:
  1. ***Emisión***: lee una instrucción de la cola.
     - Si es operación aritmética en punto flotante
       - Si hay estaciones de reserva libres y operandos disponibles → se emite la instrucción a la estación de reserva
       - Si no hay estaciones de reserva libres → la instr. se detiene (así como todas las que siguen)
     - Si es carga/almacenamiento en punto flotante
       - Si hay buffers de carga o almacenamiento → se emite la instrucción al buffer.
  2. ***Ejecución***:
     - Si los operandos no están disponibles, espera a que aparezcan en el CDB → se comprueban riesgos RAW.
     - Cuando todos los operandos están disponibles y la UF está libre, se ocupa la UF y se ejecuta la instr.
  3. ***Escritura***:
     - Resultado disponible → al CDB → a registros y/o a cualquier ER que lo necesite
     - La estación de reserva y la UF se liberan cuando se ha completado la escritura

4.3.1 Algoritmo de Tomasulo

- Estación de reserva:
  - Ocupado
  - Op: Operación a realizar
  - V_j, V_k: valor de los operandos fuente
  - Q_j, Q_k: estaciones de reserva que producirán V_j, V_k
- Buffer de carga:
  - Ocupado
  - Dir: Dirección de memoria
- Buffer de almacenamiento:
  - Ocupado
  - Dir: Dirección de memoria
  - V: valor del operando fuente
  - Q_i: estación de reserva que producirá el resultado
- Banco de registros:
  - Q_i: estación de reserva que producirá el resultado

4.3.2 Algoritmo de Tomasulo: Ejemplo

- Ejemplo: ejecución del siguiente código:

```
ld    f6, 34(r2)
ld    f2, 45(r3)
multd f0, f2, f4
subd  f8, f6, f2
divd  f10, f0, f6
addd  f6, f8, f2
```

Unidades funcionales:

- Un sumador. Consume 2ck
- Un multiplicador. Consume 10ck
- Un divisor: Consume 40ck

- ✓ Anticipación en el banco de registros (las escrituras se realizan en la 1ª mitad del ciclo, mientras que las lecturas se efectúan en la 2ª mitad)
- ✓ 3 estaciones de reserva asociadas al sumador PF (add1, add2, y add3)
- ✓ 2 estaciones de reserva asociadas al multiplicador/divisor PF (mult1 y mult2)
- ✓ 6 estaciones/buffers para gestionar las cargas (ld) (load1 a load6)
- ✓ 3 estaciones/buffers para gestionar los almacenamientos (sd) (store1 a store3)

4.3.2 Algoritmo de Tomasulo: Ejemplo (2) Estado en el ciclo 3

- Estado de las estaciones de reserva y banco de registros cuando el primer ld se ha completado

Instrucción	Estado de las Instrucciones		
	Emisión	Ejecución	Escritura
ld f6, 34(r2)	1	2	3
ld f2, 45(r3)	2	3	
multd f0, f2, f4	3		
subd f8, f6, f2			
divd f10, f0, f6			
addd f6, f8, f2			

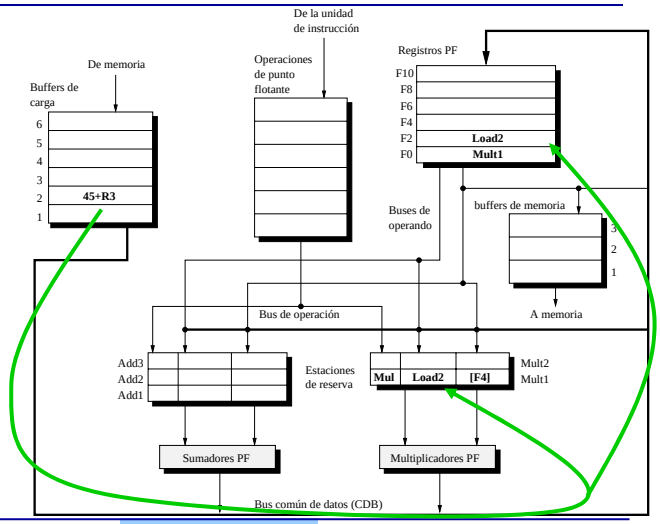
- ✓ El multiplicador y el reg. F2 están esperando la salida del Load2
- ✓ F0 espera la salida del mult1
- ✓ Load1 ya se ha liberado al escribir en F6

Estado de los registros									
Campo	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1	Load2							

Estaciones de reserva							
Nombre	Ocupado	Op.	V_j	V_k	Q_j	Q_k	Dir.
Load1	no						
Load2	si	LOAD	[R3]	45			45+R3
Add1	no						
Add2	no						
Add3	no						
Mult1	si	MUL		[F4]	Load2		
Mult2	no						

4.3.2 Algoritmo de Tomasulo: Ejemplo (3) Estado en el ciclo 3

Instrucción		Estado Em	Ej	Es
ld	f6, 34(r2)	1	2	3
ld	f2, 45(r3)	2	3	
multd	f0, f2, f4	3		
subd	f8, f6, f2			
divd	f10, f0, f6			
addd	f6, f8, f2			



4.3.2 Algoritmo de Tomasulo: Ejemplo (4) Estado en el ciclo 6

- Estado de las estaciones de reserva y banco de registros cuando subd va a escribir su resultado
  - ✓ En ck4 load2 escribe en F2 y en ER Mult1
  - ✓ En ck5 empieza la ejecución de multd
  - ✓ El divd esta esperando a Mult1
  - ✓ El addd en ck7 → riesgo estructural (1 sumador no segmentado) y RAW

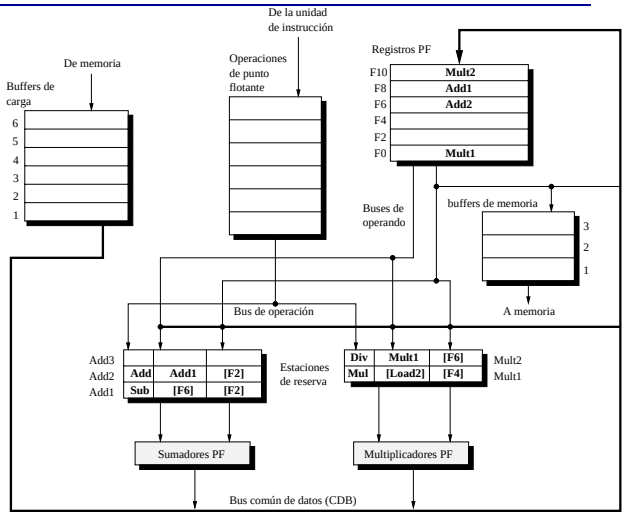
Instrucción		Estado de las Instrucciones Emisión	Ejecución	Escritura
ld	f6, 34(r2)	1	2	3
ld	f2, 45(r3)	2	3	4
multd	f0, f2, f4	3	5-	
subd	f8, f6, f2	4	5-6	
divd	f10, f0, f6	5		
addd	f6, f8, f2	6		

Estado de los registros									
Campo	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1			Add2	Add1	Mult2			

Estaciones de reserva							
Nombre	Ocupado	Op.	V_j	V_k	Q_j	Q_k	Dir.
Load1	no						
Load2	no						
Add1	si	SUB	[F6]	[Load2]			
Add2	si	ADD		[F2]	Add1		
Add3	no						
Mult1	si	MUL	[Load2]	[F4]			
Mult2	si	DIV		[F6]	Mult1		

4.3.2 Algoritmo de Tomasulo: Ejemplo (5) Estado en el ciclo 6

Instrucción		Estado Em	Ej	Es
ld	f6, 34(r2)	1	2	3
ld	f2, 45(r3)	2	3	4
multd	f0, f2, f4	3	5-	
subd	f8, f6, f2	4	5-6	
divd	f10, f0, f6	5		
addd	f6, f8, f2	6		



4.3.2 Algoritmo de Tomasulo: Ejemplo (6) Estado en el ciclo 14

- Estado de las estaciones de reserva y banco de registros cuando multd va a escribir su resultado

Instrucción		Estado de las Instrucciones Emisión	Ejecución	Escritura
ld	f6, 34(r2)	1	2	3
ld	f2, 45(r3)	2	3	4
multd	f0, f2, f4	3	5-14	
subd	f8, f6, f2	4	5-6	7
divd	f10, f0, f6	5		
addd	f6, f8, f2	6	8-9	10

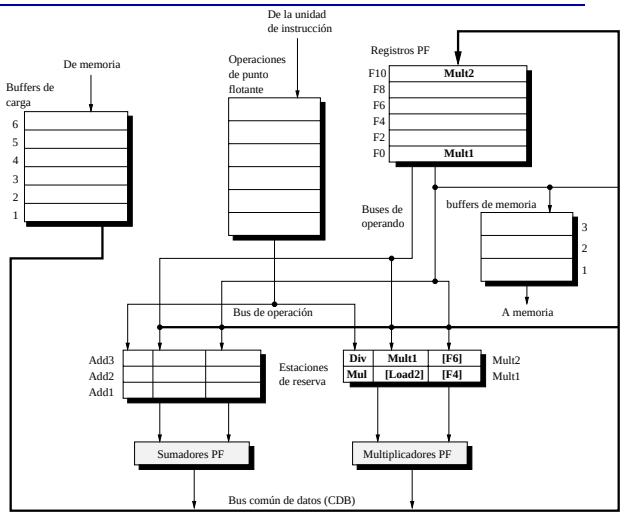
- ✓ subd y addd han terminado
- ✓ En ck15 Mult1 va a escribir en F0 y en Mult2
- ✓ El divd esta esperando a mult1, pero podrá empezar en ck16.

Estado de los registros									
Campo	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1					Mult2			

Estaciones de reserva							
Nombre	Ocupado	Op.	V_j	V_k	Q_j	Q_k	Dir.
Load1	no						
Load2	no						
Suma1	no						
Suma2	no						
Suma3	no						
Mult1	si	MUL	[Load2]	[F4]			
Mult2	si	DIV		[F6]	Mult1		

4.3.2 Algoritmo de Tomasulo: Ejemplo (7) Estado en el ciclo 14

Instrucción		Estado Em	Ej	Es
ld	f6, 34(r2)	1	2	3
ld	f2, 45(r3)	2	3	4
multd	f0, f2, f4	3	5-14	
subd	f8, f6, f2	4	5-6	7
divd	f10, f0, f6	5		
addd	f6, f8, f2	6	8-9	10



4.3.2 Algoritmo de Tomasulo: Ejemplo (8)

- Tabla completa del estado de las instrucciones y ciclo en que se completa cada etapa

Instrucción		Estado de las Instrucciones Emisión	Ejecución	Escritura
ld	f6, 34(r2)	1	2	3
ld	f2, 45(r3)	2	3	4
multd	f0, f2, f4	3	5-14	15
subd	f8, f6, f2	4	5-6	7
divd	f10, f0, f6	5	16-55	56
addd	f6, f8, f2	6	8-9	10

- WAR evitado:
  - ✓ addd ha escrito en f6 antes de que divd se ejecute pero divd ya tenía una copia del f6 “bueno” en la estación de reserva (renombre de registros)

4.3.2 Algoritmo de Tomasulo: Ejemplo (9)

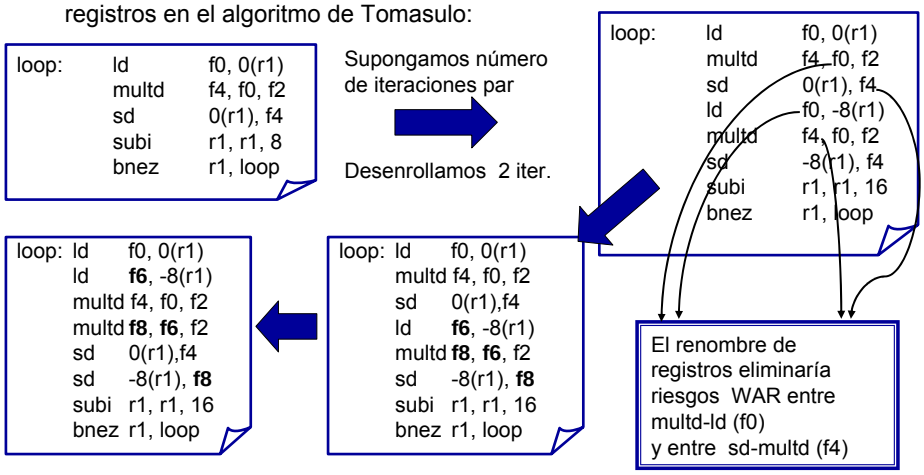
- Suponiendo una sola ER de suma (Add1) y addd f0,f8,f2

Instrucción		Estado de las Instrucciones		
		Emisión	Ejecución	Escritura
ld	f6, 34(r2)	1	2	3
ld	f2, 45(r3)	2	3	4
multd	f0, f2, f4	3	5-14	15
subd	f8, f6, f2	4	5-6	7
divd	f10, f0, f6	5	16-55	56
addd	f0, f8, f2	8	9-10	11

- Habría riesgo estructural en la ER al intentar emitir addd
- WAW evitado:
  - ✓ Aunque parece que muld escribe en f0 en ck15, no es así!!!
  - ✓ Cuando se emite muld, f0 "vigila" a Mult1, pero cuando luego se emite addd, f0 pasa a "vigilar" a Add1 en lugar de a Mult1 → mult no escribe en f0

4.3.3 Algoritmo de Tomasulo Ejemplo con lazo

- Veamos con el ejemplo de un lazo la verdadera potencia del renombre de registros en el algoritmo de Tomasulo:



4.3.3 Algoritmo de Tomasulo
Ejemplo con lazo (2): Ciclo 5

- Estado de las ER y banco de registros cuando el 2º ld va a escribir el resultado (ignoramos inst. subi y suponemos pred. *salto tomado*). Ciclo 5

		Estado de las Instrucciones		
Instrucción		**Emisión**	**Ejecución**	**Escritura**
ld	f0, 0(r1)	1	2	3
multd	f4, f0, f2	2	4-	
sd	0(r1), f4	3		
ld	f0, 0(r1)	4	5	
multd	f4, f0, f2	5		
sd	0(r1), f4			

- El primer ld ha acabado y liberado Load1
- El primer sd está esperando a Mult1
- F0 "vigila" al segundo ld y F4 a Mult2

Estado de los registros									
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**…**	**F30**
Qi	Load2		Mult2						

Estaciones de reserva							
Nombre	**Ocupado**	**Op.**	**V_j**	**V_k**	**Q_j**	**Q_k**	**Dir.**
Load1	no						
Load2	si	LOAD					-8(r1)
Store1	si	STORE			Mult1		0(r1)
Store2	no						
Suma1	no						
Mult1	si	MUL	[Load1]	[F2]			
Mult2	si	MUL		[F2]	Load2		

4.3.3 Algoritmo de Tomasulo
Ejemplo con lazo (3): Ciclo 13

- Estado de las ER y banco de registros cuando la 1ª multd va a escribir el resultado

		Estado de las Instrucciones		
Instrucción		**Emisión**	**Ejecución**	**Escritura**
ld	f0, 0(r1)	1	2	3
multd	f4, f0, f2	2	4-13	
sd	0(r1), f4	3		
ld	f0, 0(r1)	4	5	6
multd	f4, f0, f2	5	7-	
sd	0(r1), f4	6		

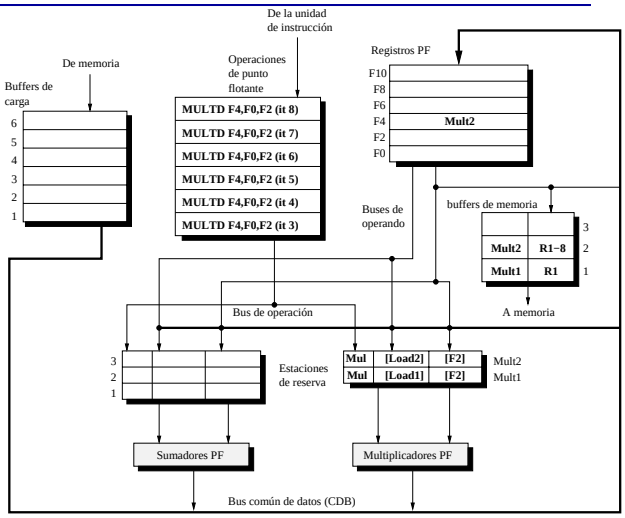
- Los ld's han terminado
- Los sd's están esperando a sus respectivas multiplicaciones
- En F4 sólo va a escribir la última multiplicación

Estado de los registros									
Campo	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**…**	**F30**
Qi			Mult2						

Estaciones de reserva							
Nombre	**Ocupado**	**Op.**	**V_j**	**V_k**	**Q_j**	**Q_k**	**Dir.**
Load1	no						
Load2	no						
Store1	si	STORE			Mult1		0(r1)
Store2	si	STORE			Mult2		-8(r1)
Suma3	no						
Mult1	si	MUL	[Load1]	[F2]			
Mult2	si	MUL	[Load2]	[F2]			

4.3.3 Algoritmo de Tomasulo
Ejemplo con lazo (4): Ciclo 13

Instrucción		Estado It.	Em	Ej	Es
ld	f0, 0(r1)	1	1	2	3
multd	f4, f0, f2	1	2	4-13	
sd	f4, 0(r1)	1	3		
ld	f0, 0(r1)	2	4	5	6
multd	f4, f0, f2	2	5	7-	
sd	f4, 0(r1)	2	6		



4.3.3 Algoritmo de Tomasulo
Ejemplo con lazo (5)

- Tabla completa del estado de las instrucciones y ciclo en que se completa cada etapa

Instrucción		Estado de las Instrucciones It.	Emisión	Ejecución	Escritura
ld	f0, 0(r1)	1	1	2	3
multd	f4, f0, f2	1	2	4-13	14
sd	f4, 0(r1)	1	3	15	16
ld	f0, 0(r1)	2	4	5	6
multd	f4, f0, f2	2	5	7-16	17
sd	f4, 0(r1)	2	6	18	19

- Parecido al desenrollamiento:
  - ✓ Las dos instrucciones ld se ejecutan primero, seguidas de las instrucciones multd, y finalmente las instrucciones sd
  - ✓ Hemos supuesto: predicción de salto correcta y subi ejecutándose aparte (inst entera)

4.3.4 Algoritmo de Tomasulo Conclusiones

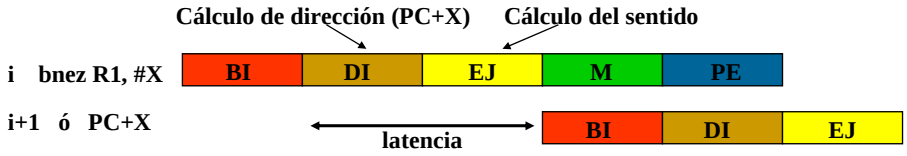
- El algoritmo de Tomasulo implementa a nivel HW dos técnicas de compilación:
  - ✓ Renombre de registros
  - ✓ Desenrollamiento dinámico de lazos
- Resumen de características:
  - ✓ Las cargas y almacenamientos se tratan como unidades funcionales
  - ✓ Adelantamiento de datos desde el CDB a las unidades funcionales
  - ✓ Los riesgos RAW se resuelven en la etapa de ejecución
  - ✓ No aparecen riesgos WAR ni WAW (se eliminan cuando se renombran los registros operando en las ER)
    - Las etiquetas para el control de los riesgos indican qué estación de reserva contiene la instrucción que produce el resultado que se necesita → *renombre de registros*

4.3.4 Algoritmo de Tomasulo Conclusiones

- Inconvenientes:
  - ✓ Alto coste hardware → no apropiado para procesador segmentado simple.
  - ✓ Rendimiento limitado por el uso del CDB → ensanchar el CDB para que no represente un cuello de botella → complicar mucho el hardware
- Apropiado para segmentar una arquitectura para la que es difícil planificar el código o que tiene pocos registros.

4.4 Predicción de salto
Recordatorio

- Hemos visto en el tema anterior las técnicas de:
  - ✓ Parar siempre. La peor solución desde el punto de vista del rendimiento
  - ✓ Salto retardado. Válido para uno o dos ck de latencia.
  - ✓ Predicción estática. El porcentaje de aciertos es poco alentador.
- Vamos a estudiar ostras alternativas:
  - ✓ Predicción dinámica de salto (buffer de predicción de salto)
  - ✓ Buffer de destino de salto (BTB)
  - ✓ Otras alternativas



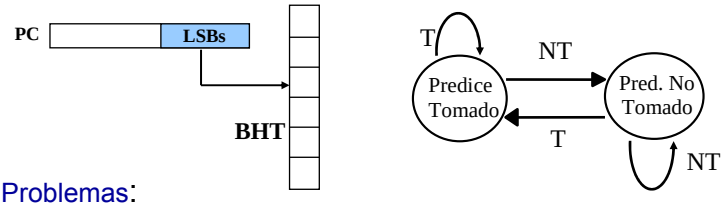
  - ✓ 20% de saltos condicionales. 70% de saltos efectivos. El resto: ideal.

4.4.1 Predicción dinámica

- El HW realiza una predicción basada en la historia computacional
- Las prestaciones de un predictor de salto dependen de
  - ✓ Precisión en la predicción: aumenta con mejores predictores
  - ✓ Coste del fallo de predicción
- El coste del fallo en predicción depende de muchos aspectos:
  - ✓ Longitud del cauce (mejor cuanto más corto)
  - ✓ Organización del pipeline (donde se resuelve el sentido y la dirección, ...)
- En general, recuperarse de un fallo en predicción es caro:
  - ✓ 4 a 9 ciclos en el Alpha 21264,
  - ✓ 11 o más ciclos en el Pentium III.
  - ✓ Hasta más de 31 ciclos en el Pentium 4
- Por otro lado, la predicción dinámica suele ser más efectiva que la estática, pero con un coste HW y complejidad también superior.

4.4.1 Predictor de un bit

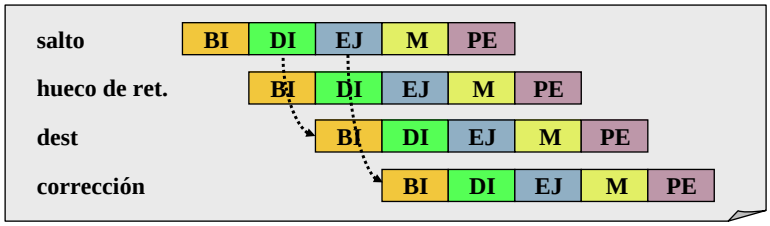
- Idea: si un salto fue efectivo es probable que vuelva a serlo
- Buffer de predicción de salto
  - ✓ *Branch predictor buffer* o *branch history table* BHT
  - ✓ Pequeña memoria direccionada por la dir. (o parte) de la instrucción de salto
  - ✓ La posición de memoria tiene un bit que indica si el salto fue o no efectivo
  - ✓ Si la predicción es errónea se invierte el bit



- Problemas:
  - Sólo útil si conocemos antes la dirección que el sentido
  - Colisiones en la BHT: dos instrucciones de salto en dir. con mismos LSBs

4.4.1 Predictor de un bit

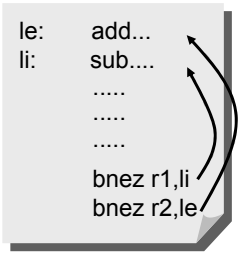
- Ejemplo: dirección conocida en DE, sentido en EJ. Salto retardado



- Pierde un ck si falla la predicción (otro ck si hueco no útil)
- Ejemplo: 85% de huecos útiles.
  - ✓ 10% de colisiones en BHT. Predicción correcta el 50%
  - ✓ 90% restante con predicción correcta el 75% de las veces
    - CPI_{salto}= 1 + 0.15x1 + (0.1x0.5 + 0.9x0.25)x1 = 1.425
    - CPI = 0.8x1 + 0.2x1.425 = 1.085

4.4.1 Predictor de un bit

- Problema de precisión: lazo anidado o en una rutina frecuente
- Ejemplo: dos lazos anidados.
  - ✓ El interno se predice 2 veces incorrectamente por cada iteración externa

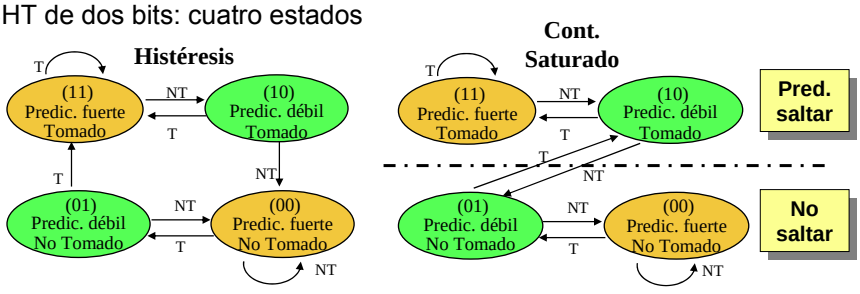


Externo	Interno	Predice	Real	Fallo	Nueva Pred.
1	1	NT	T	S	T
	2	T	T	N	T
	.....	T	T	N	T
	última	T	NT	S	NT
2	1	NT	T	S	T
	2	T	T	N	T
	.....	T	T	N	T
	última	T	NT	S	NT

Solución: Predictor de dos bits

4.4.2 Predictor de dos bits

- Idea: una segunda oportunidad antes de cambiar la predicción (inercia)
- BHT de dos bits: cuatro estados



- ❑ Caso general de n bits: contador saturado
  - ❍ Decrementa cuando no salta. Incrementa cuando salta
  - ❍ Predice NO saltar si cuenta < 2^{n-1}
  - ❍ En la práctica, n=2 es tan efectivo como n>2 o incluso mejor.

4.4.2 Predictor de dos bits

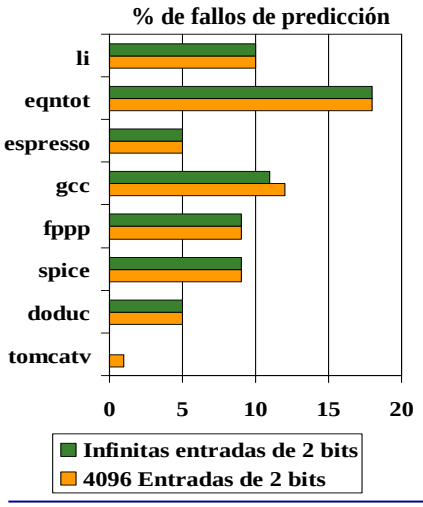
- Resuelve el problema de los dos lazos anidados
  - ✓ El interno se predice sólo 1 vez incorrectamente por cada iteración externa (excepto en la primera iteración si la predicción inicial es NT)

```
le:   add...
li:   sub....
      .....
      .....
      .....
      bnez r1, li
      bnez r2, le
```

Externo	Interno	Predice	Real	Fallo	Nueva Pred.
1	1	NT	T	S	DNT
	2	DNT	T	S	DT
	.....	DT	T	N	T
	última	T	NT	S	DT
2	1	DT	T	N	T
	2	T	T	N	T
	.....	T	T	N	T
	última	T	NT	S	DT

Estudiemos como se comporta el predictor con benchmarks

4.4.2 Predictor de dos bits



- Mejorar la predicción
  - ✓ ¿Más entradas?
    - A partir de 4096 no afecta
  - ✓ ¿Mejor algoritmo?
    - Pasar de 2 a n bits
      - Ya hemos visto que no!
    - Otra idea
      - Estudiar qué le pasa al peor código
- Código eqntot (18% de error)
  - ✓ Contiene saltos correlacionados

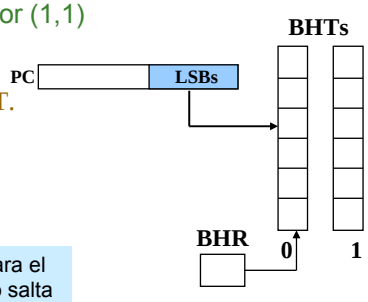
```
if (a==2)  a=0;
if (b==5)  b=0;
if (a!=b) {
..................... }
```

4.4.3 Predictor correlacionado

- Para predecir saltos relacionados con otros saltos (correlacionados)
- También llamados predictores en dos niveles
- Idea: Para predecir un salto "b"
  - ✓ No sólo miran lo que le ocurrió en el pasado al salto "b"
  - ✓ Sino que también miran lo que les ocurrió a algunos saltos previos a "b"
- Implementación más simple: Predictor (1,1)

❍ BHR: *Branch History Register* de 1 bit para seleccionar una de entre 2^1 tablas BHT. El BHR almacena el resultado del último salto ejecutado (0 = NT, 1 = Tomado)

❍ 2 BHTs: *Branch History Tables* de 1 bit



Se almacena el comportamiento del salto "b" para el caso en que "b-1" salta y para el caso en que no salta

4.4.3 Ejemplo (predictor correlacionado)

```
if (d==0)   /* salto b1 */
    d=1;
if (d==1)   /* salto b2 */
   ...
```

```
      bnez R1,L1      ; salto b1 (d≠0)
      addi R1, R0,#1  ; d==0, entonces d=1
L1:   subi R3, R1,#1
      bnez R3, L2     ; salto b2 (d≠1)
      ...
L2:   ...
```

- Consideremos una secuencia en que "d" alterna los valores 2 y 0

d inicial	¿*d*==0?	b1	*d* antes de b2	¿*d*==1?	b2
0	si	NT	1	si	NT
2	no	T	2	no	T

Predictor de un bit (100% de fallos)

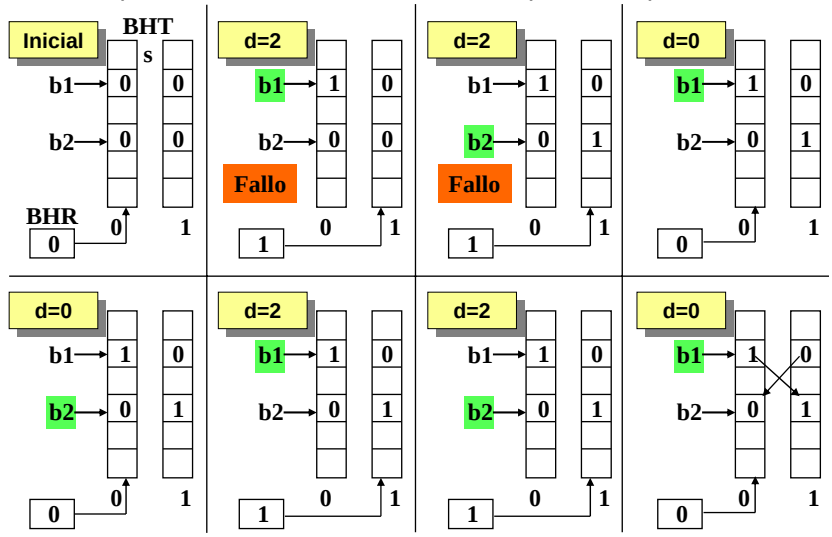
	Predic. inicial	d==2	d==0	d==2
b1:	NT	T	NT	T
b2:	NT	T	NT	T

Predictor de dos bits (50% de fallos)

	Predic. inicial	d==2	d==0	d==2
b1:	NT	DNT	NT	DNT
b2:	NT	DNT	NT	DNT

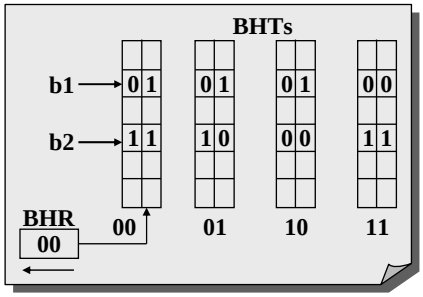
4.4.3 Ejemplo (predictor correlacionado)

- Con el predictor correlacionado sólo se equivoca la primera iter.

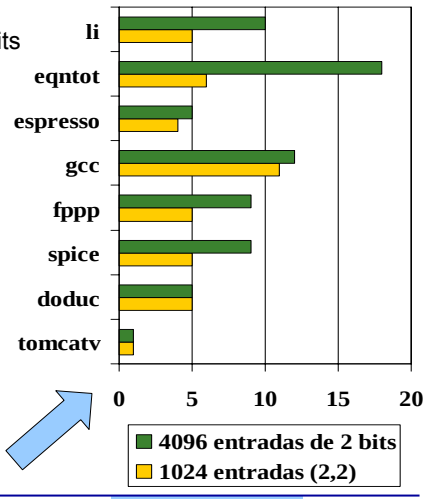


4.4.3 Predictor correlacionado general Generalización: predictor (m,n)

- Usa información de m saltos
  - ✓ BHR: reg. de desplazamiento de m bits
- Las tablas BHT son de n bits
- Ejemplo: predictor (2,2):



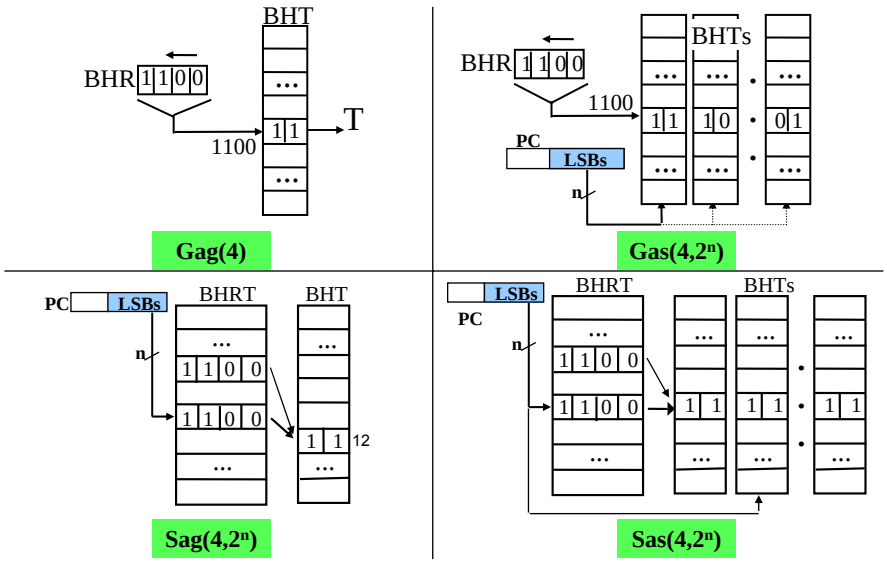
Comparación predictor de 2 bits con un (2,2) del mismo tamaño



4.4.4 Predictores adaptativos en dos niveles

- Idea de Yeh y Patt simultánea (1992) al pred. correlacionado
- Notación: XAx(m,2^n)
  - X: "**G**" (un único BHR global) ó "**S**" (una tabla de BHRs -BHRT- indexada por los "n" bits menos significativos de la dir. de la instrucción de salto)
  - A: Adaptativo
  - x: "**g**" (una única BHT global) ó "**s**" (varias BHTs de las cuales se selecciona una con los "n" bits menos significativos de la dir. de la inst. de salto)
  - m: número de bits del registro de desplazamiento BHR o ancho de BHRT
- Ejemplo: Gas(4,2^2)
  - Un BHR de 4 bits contiene la historia de los 4 últimos saltos para indexar la BHT que seleccionen los 2 últimos bits de la dir. de la instrucción de salto.
- Todas las BHTs de Yeh y Patt son de dos bits de ancho
- Experimentos de Yeh y Patt
  - Usar G/g (globales) en códigos enteros y S/s (locales) en códigos flotantes.

4.4.4 Predictores adaptativos en dos niveles

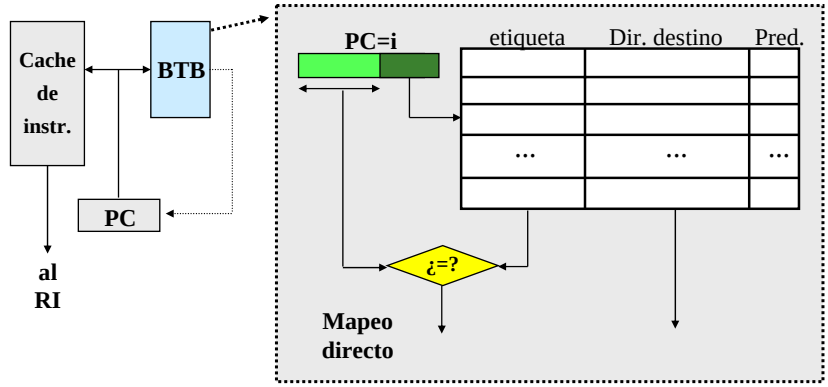


4.4.5 Predictores híbridos

- Usar G para códigos enteros y S para códigos flotantes
  - ✓ O combinar G y S para conseguir buenas predicciones en los dos casos
- Predictores gselect y gshare
  - ✓ gselect indexa la BHT concatenando algunos bits de PC con BHR
  - ✓ gshare indexa la BHT con el XOR de algunos bits del PC y BHR
  - ✓ Estudio de McFarling: gshare es algo mejor que gselect
- Predictores híbridos
  - ✓ Idea: un predictor se comporta bien en ciertas situaciones y en otras no
  - ✓ Poner dos predictores y luego seleccionar la predicción de uno de ellos
    - McFarling: uno de dos bits combinado con un gshare
    - Young y Smith: estático fijado por el compilador combinado con adaptativo
    - Alpha 21264: uno global combinado con uno local
    - Y muchos más!
  - ✓ En general dan mejores resultados los predictores híbridos

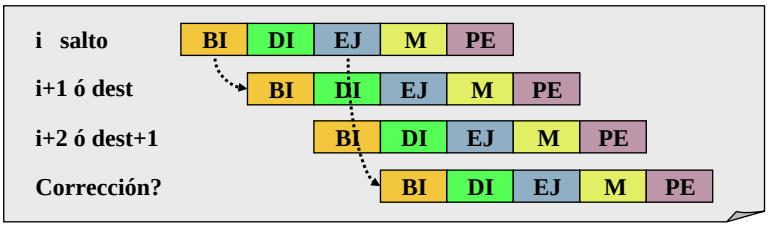
4.4.6 Buffer de destino de salto (BTB)

- ¿De qué sirve predecir el sentido si no tenemos la dirección?
- Solución: almacenar la dirección de salto junto con la predicción
- Acceso a la BTB en paralelo con la Búsqueda de Instrucción (BI)



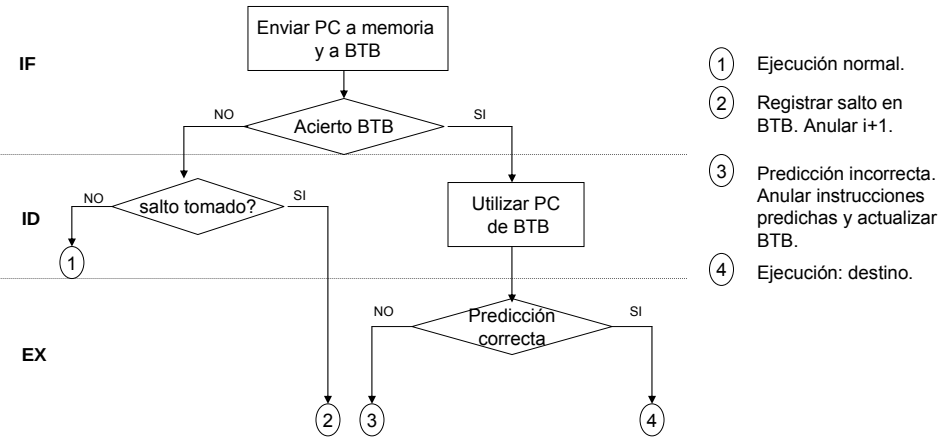
4.4.6 Buffer de destino de salto (BTB)

- Si la predicción es sólo de un bit (tomado / no tomado)
  - ✓ No hace falta el campo predicción
  - ✓ Sólo se almacenan en la BTB los saltos de los que se predice saltar
  - ✓ Acierto en BTB: Encuentras el salto en BTB y predices saltar
  - ✓ Acierto en predicción:
    - Si hay acierto en BTB (encuentras el salto) y efectivamente salta
    - Si no hay acierto en BTB (no encuentras el salto) y efectivamente no salta
  - ✓ Ejemplo: Dirección conocida en DI y sentido en EJ



4.4.6 Buffer de destino de salto (BTB) En el DLX: dirección y sentido en DI

Funcionamiento detallado de la BTB para el caso de predicción de un solo bit



4.4.6 Buffer de destino de salto (BTB) En el DLX: dirección y sentido en DI

Instrucción no de salto o salto no registrado que no salta → no se registra (penalización=0 ciclos).

①

	Ciclo de reloj										
Instrucción	BTB	2	3	4	5	6	7	8	9	10	11
i: BEQZ r1,#X	IF	ID	EX	MEM	WB						
i+1		IF	ID	EX	MEM	WB					
i+2			IF	ID	EX	MEM	WB				
i+3				IF	ID	EX	MEM	WB			

Salto no registrado que salta → se registra (penalización=2 ciclos).

②

	Ciclo de reloj										
Instrucción	BTB	2	3	4	5	6	7	8	9	10	11
i: BEQZ r1,#X	IF	ID	EX	MEM	WB						
i+1		IF	X								
k			aBTB	IF	ID	EX	MEM	WB			
k+1					IF	ID	EX	MEM	WB		

La actualización de la BTB consume 1ck. Mientras se actualiza la BTB no se puede realizar IF ya que la etapa IF implica una consulta de la BTB (y ésta está ocupada)

4.4.6 Buffer de destino de salto (BTB) En el DLX: dirección y sentido en DI

Salto registrado en BTB, predicción incorrecta → anula predicción y actualiza BTB (penalización=2 ciclos).

③

	Ciclo de reloj										
Instrucción	BTB	2	3	4	5	6	7	8	9	10	11
i: BEQZ r1,#X	IF	ID	EX	MEM	WB						
k		IF	X								
i+1			aBTB	IF	ID	EX	MEM	WB			
i+2					IF	ID	EX	MEM	WB		

Salto registrado en BTB, predicción correcta → penalización=0 ciclos.

④

	Ciclo de reloj										
Instrucción	BTB	2	3	4	5	6	7	8	9	10	11
i: BEQZ r1,#X	IF	ID	EX	MEM	WB						
k		IF	ID	EX	MEM	WB					
k + 1			IF	ID	EX	MEM	WB				
k + 2				IF	ID	EX	MEM	WB			

4.4.6 Buffer de destino de salto (BTB)

- Resumen de resultados

(Ciclos perdidos)	Salta	No salta
Predice saltar	**0**	**2**
Predice NO saltar	**2**	**0**

- Ejemplo:
  - ✓ 90% de aciertos en buffer con 95% de aciertos en predicción. En caso de fallo en BTB (predice NT) suponemos un 80% de acierto en predicción.
  - ✓ CPI_{salto}= 1 + 0.9x0.05x2 + 0.1x0.2x2= 1.13 CPI=0.8x1+0.2x1.13=1.026
- Variaciones
  - ✓ Cache de destinos de salto (BTC)
    - Almacena una o más instrucciones destino. Al terminar BI tienes que decodificar la instrucción de salto y la que acabas de leer de BTC (superescalar)
  - ✓ Pila de direcciones de retorno (RAS)
    - Almacena la dirección de retorno de procedimientos. Adicional a la BTB

4.4.7 Instrucciones predicadas

- Requerimientos:
  - ✓ instrucciones **predicadas** o **condicionales** y
  - ✓ registros **predicado**
- Cada registro predicado, P, de un bit puede ser escrito tras chequear una condición
- Las instrucciones predicadas usan un registro predicado como operando adicional
- Las instrucciones predicadas se buscan, decodifican y emiten normalmente
- Dependiendo de la arquitectura del procesador:
  - ✓ Una instrucción predicada se ejecuta sólo si el predicado es cierto
  - ✓ En la arquitectura ISA IA64, la instrucción se puede ejecutar, pero no escribe en el banco de registros si el predicado es falso.

4.4.7 Instrucciones predicadas

```
if (r1==r2) {
        r3=r4+r5
;
}
else {
        r6=r4-r5;
}
```

```
        sub r3, r1, r2
        bnez r3, no
si:     add r3, r4, r5
        jmp sigue
no:     sub r6, r4, r5
sigue:  .....
```

```
        cmpeq r1, r2, p4
        cmpne r1, r2, p5
<p4>    add r3, r4, r5
<p5>    sub r6, r4, r5
```

☺ **Ventajas**

- ☺ Capaz de eliminar saltos y su predicción asociada. Cambia dep. control por dep. datos.
- ☺ Bloques básicos más grandes ⇒ el compilador puede planificar mejor.

☹ **Inconvenientes**

- ☹ Afecta al conjunto de instrucciones y al diseño del procesador (cortocircuitos, control,...)
- ☹ Las instrucciones descartadas consumen recursos y ancho de banda.

- Predicar es más efectivo cuando elimina totalmente los riesgos de control
- Uso limitado si el control de flujo es más complejo que una simple alternativa.

4.4.8 Ejecución multicamino

(eager ó multipath)

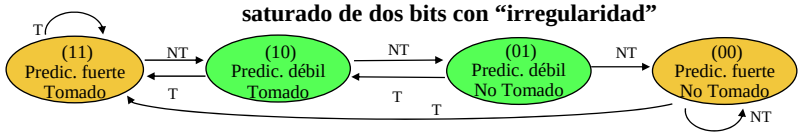
- Se ejecutan las dos ramas de una bifurcación sin hacer predicción.
- Cuando se resuelve el salto, todas las instr. del camino no tomado se cancelan
- Ejecución oracle: ejecución eager con recursos ilimitados
  - ✓ Proporciona las prestaciones máximas teóricas de la predicción de salto perfecta
- Con recursos limitados, la ejecución eager se debe aplicar con cuidado.
- Raramente implementado ⇒ IBM mainframes y proyectos de investigación

4.5 Ejemplos: Implementaciones comerciales

Técnica	Procesadores comerciales
Sin predicción	Intel 8086
Predicción estática:	
siempre no tomado	Intel i486
siempre tomado	SuperSPARC
atrás tomado, adelante no	HP PA-7x00
basada en perfiles	primeros PowerPCs
Predicción dinámica:	
1-bit	DEC Alpha 21064, AMD K5
2-bits (saturado)	PowerPC 604, MIPS R10000, 21164, UltraSparc Cyrix 6x86 and M2, NexGen 586
adaptativo dos niveles	Intel PentiumPro, Pentium III, AMD K6
Predicción híbrida	DEC Alpha 21264, Pentium 4 (no especificado)
Predicado	Itanium y la mayoría de los proc. de señal: de ARM, TI TMS320C6201 y muchos otros
Ejecución multicamino (limitada)	IBM mainframes: IBM 360/91, IBM 3090

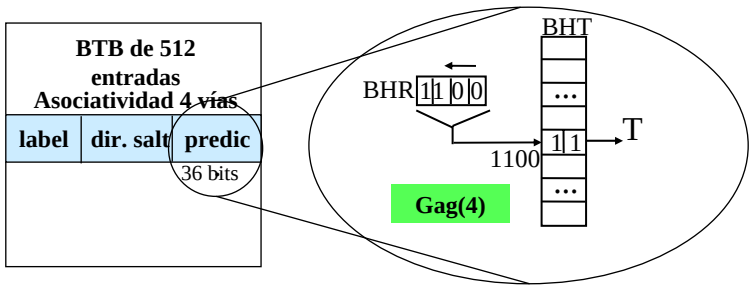
4.5 Predictor Pentium y Pentium III

Pentium: BTB de 256 entradas, cada una con un contador saturado de dos bits con "irregularidad"

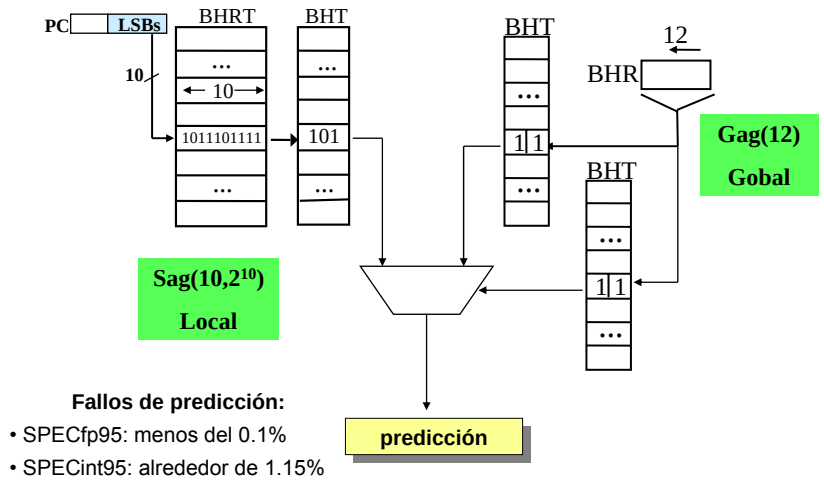


Un salto que rara vez salta falla 3 veces más con este esquema que con un contador saturado normal

Pentium Pro a Pentium III: BTB de 512 entradas, cada una con 36 bits para implementar un predictor adaptativo tipo Pap(4)



4.5 Predictor del Alpha 21264



Fallos de predicción:

- SPECfp95: menos del 0.1%
- SPECint95: alrededor de 1.15%

4.5 Predictor del HP 8700

- Predictor híbrido: combina
  - ✓ BHT de dos bits: contador saturado
  - ✓ Predicción estática dirigida por el compilador
- Pasos en la predicción
  - ✓ Inicialmente se sigue el consejo del compilador mirando el código de operación de la instrucción de salto (predicción estática)
  - ✓ Si falla la predicción del compilador se decrementa el contador saturado y si acierta se incrementa dicho contador para el salto en cuestión
  - ✓ Futuras predicciones se hacen atendiendo al contador:
    - Si el contador contiene un valor mayor o igual que 2 se usa la predicción del compilador almacenada en el código de operación
    - Si el contador contiene un valor menor que 2 se hace lo contrario de lo que el compilador almacenó en el código de operación

Conclusiones

- En los procesadores de hoy y del futuro son esenciales excelentes técnicas de gestión de saltos.
- Requerimientos para la gestión eficiente de saltos:
  - ✓ Resolución del salto temprana en el cauce
  - ✓ Almacenar el destino del salto en una BTB grande y rápida
  - ✓ Un buen algoritmo de predicción de salto y ejecución especulativa
  - ✓ A veces un salto se predice mientras el previo aun no se ha resuelto
    - Soportar dos o más niveles de especulación.
  - ✓ Mecanismo de recuperación eficiente
    - Minimizar la penalización por fallo en predicción
- Próximas clases:
  - ✓ Emisión múltiple. Superescalares y VLIW.
  - ✓ Especulación y planificación dinámica con especulación.

Bibliografía

- J.L HENNESSY, D. PATTERSON. Computer Architecture: a quantitative approach. 3 Ed. Morgan Kaufman, 2003.
  - ✓ **Desenrollamiento de bucles: Capitulo 4.1**
  - ✓ **Algoritmo de Tomasulo: Capítulo 3.2 y 3.3**
  - ✓ **Predicción salto: Capítulo 3.4 (predicción) y 3.5 (BTB)**
- J. SILC, B. ROBIC, T. UNGERER. Processor Architecture. Springer. 1999.
  - ✓ **Predicción de salto: Capítulo 4.3**
- ANDREW S. TANENBAUM. Structured Computer Organization. Cuarta Ed., Ed. Prentice Hall, 1999.
  - ✓ **Predicción: Capítulo 4.5.2**
  - ✓ **Instrucciones predicadas: Capítulo 5.8.3**