Arquitectura de Computadores

TEMA 2

ILP, Predicción de saltos, Planificación dinámica y Especulación

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

Curso 2024-2025

Contenidos

- Introducción: ILP
- Técnicas SW: Compilador
- Tratamiento de dependencias de control: Predicción de saltos
  - Tratamiento de saltos
  - Técnicas de predicción: Estáticas y dinámicas
- Planificación dinámica y Especulación: Algoritmo de Tomasulo.
- Bibliografía
  - Básica: Capítulo 3 de Hennessy & Patterson, 6th ed., 2019
  - Complementaria: Capítulo 3 de Dubois, Annavaram & Stenström, 2012

Introducción

- OBJETIVO: Ejecutar el mayor número de instrucciones por ciclo
  - **Obtener el máximo número de instrucciones independientes**

CPI = CPI ideal + Penaliz. Media por Instr. (paradas "pipe")

¿Qué técnicas conocemos?

- Conflictos de recursos - Riesgos estructurales → Replicación/segmentación
- Dependencias de datos → Cortocircuitos
- Dependencias de control (Un salto cada 4-7 instrucciones) → Saltos retardados

Mecanismos para explotar ILP

- Basados en HW en tiempo de ejecución (dinámicos). Ej Intel Core, AMD, IBM, ARM…
  Toda la información disponible en ejecución
  Código independiente de la implementación

- Basados en SW en tiempo de compilación (estáticos). Ej Itanium
  Dependencias de memoria muy difíciles de determinar

Introducción

Paralelismo a nivel de instrucción ILP

- Es la técnica consistente en explotar paralelismo entre instrucciones próximas en la secuencia
- El bloque básico:
  - Un bloque básico (BB) es una secuencia de código sin saltos. Un solo punto de entrada y salida.
  - Típicamente muy pequeño: solo de 4 a 7 instrucciones
  - … y con fuertes dependencias entre ellas
- El camino es explotar ILP entre varios BB
- El caso más simple: paralelismo a nivel de bucle

```
for ( i =1000; i>0; i--)
        a (i) = a (i) + s ;
```

  - Todas las iteraciones son independientes (saltos)

- Técnicas para explotar ILP

	Técnica	Reduce
Dinámicas	Planificación Dinámica	Paradas por riesgos de datos
	Predicción dinámica de saltos	Paradas por riesgos de control
	Lanzamiento múltiple Varias instrucciones por ciclo	CPI Ideal
	Especulación	Riesgos de datos y control
	Dynamic memory disambiguation	Paradas por riesgos de datos en memoria
Estáticas	Desenrollado de bucles	Paradas por riesgos de control
	Planificación por el compilador	Paradas por riesgos de datos
	Software pipelining	CPI Ideal y Paradas por riesgos de datos
	Predicción estática y Especulación por el Compilador	CPI Ideal, paradas por riesgos de datos y control

Dependencias

- Determinar las dependencias es crítico para obtener el máximo paralelismo

  ¿Cuáles hay? ¿A qué recursos afectan?

Las dependencias son propias de los programas

- La presencia de una dependencia indica la posibilidad de aparición de un riesgo, pero la aparición de éste y la posible parada depende de las características del "pipe"
- Las dependencias
  - Indican la posibilidad de un riesgo
  - Determinan el orden de cálculo de los resultados
  - Imponen un límite al paralelismo que es posible obtener

Dependencias

Tipos de Dependencias

- Dependencias de datos
  - Dependencia verdadera (LDE)
  - Dependencias de nombre
    - Antidependencia (EDL)
    - Dependencia de salida (EDE)
- Dependencias de control

Dependencia verdadera (LDE)

- La instrucción j depende de i
  - i produce un resultado que usa j

```
i:    FLD    F0,0(X1)
j:    FADD.D F4,F0,F2
```

Dependencias

❑ Dependencias de nombre (Reutilización de los registros)

- Dos instrucciones i y j donde i precede a j presentan dependencias de nombre en las siguientes situaciones:

- Antidependencia WAR (EDL)
  - La instrucción j escribe (Reg o memoria) antes de que i lea.

```
FADD.D  F4,F0,F2
FLD     F0,-8(X1)
```

- Dependencia de salida WAW (EDE)
  - Las instrucciones i y j escriben el mismo reg. o memoria

```
FADD.D  F4,F0,F2
FSUB.D  F4,F3,F2
```

❑ ILP y Dependencias de datos

- Los mecanismos de ejecución deben preservar el comportamiento del programa. Mismo resultado que en ejecución secuencial
- Explotar todo el paralelismo posible sin afectar al resultado de la ejecución
- Para las dependencias de nombre eliminar la dependencia usando otros "nombres"

Dependencias

❑ Dependencias de control

o Cada instrucción depende de un conjunto de saltos y en general esta dependencia debe preservarse para preservar el comportamiento del programa

```
If  P1 (
            S1;
);
if  P2 (
            S2;
)
```

S1 depende de P1 ; S2 depende de P2

Pero… las dependencias de control pueden violarse, es decir, se pueden ejecutar instrucciones no debidas **si esto no afecta** al resultado correcto del programa (Ej. "delay slot" en los saltos retardados)

LO IMPORTANTE: el comportamiento de las excepciones y el flujo de datos deben preservase

Dependencias

❑ Dependencias de control y Excepciones

- Comportamiento de excepciones se debe preservar. Cualquier cambio en el orden de ejecución no debe cambiar cómo las excepciones son atendidas en la ejecución.

```
        ADD     X2,X3,X4
        BEQ     X2,X0,L1
        LW      X1,0(X2)
L1:     ---     ---
```

- LW no se puede mover antes de BEQ (posible fallo de página)

❑ Dependencias de control y flujo de datos

- Se debe mantener el flujo de datos entre instrucciones productoras y consumidoras de datos.

```
        ADD     X1,X2,X3
        BEQ     X4,X0,L1
        SUB     X1,X5,X6
L1:     ---     ---
        OR      X7,X1,X8
```

- OR usa el resultado de ADD o SUB dependiendo del comportamiento del salto. El flujo de datos se debe preservar.

Dependencias

Dependencias de datos

Fáciles de determinar para registros

Difíciles para direcciones de memoria
¿Son el mismo dato 100(X4) y 20(X6)?
En dos iteraciones diferentes 20(X6) y 20(X6) ¿son el mismo dato?

Implica: Más registros para evitar dependencias de nombre
Identificar dependencias entre load y store para permitir su reordenación

Dependencias de control

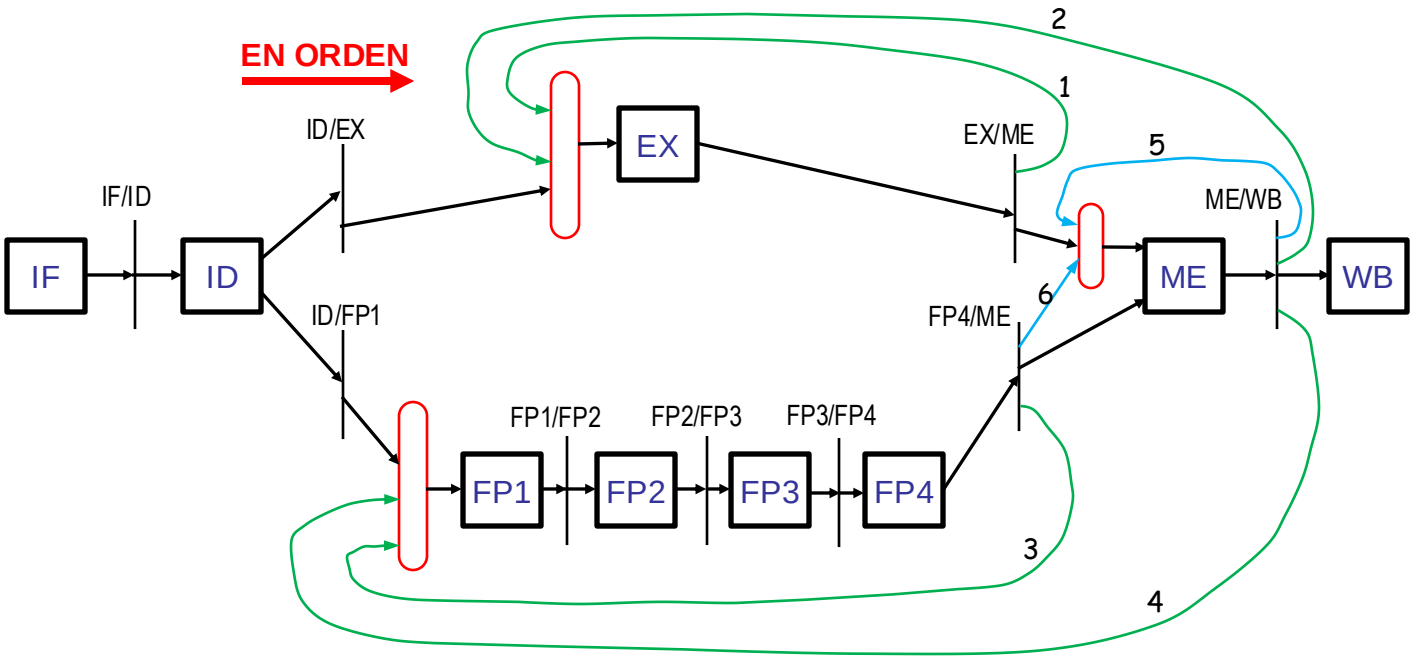
En general:
- Una instrucción dependiente de un salto no puede moverse antes del salto
- Una instrucción no dependiente de un salto no puede moverse después del salto

Efecto de las dependencias de control sobre el orden de las excepciones y el flujo de datos

SOLUCIÓN : HW + SW (PROCESADOR + COMPILADOR)

Técnicas SW para explotar ILP

- Modelo de computador para el estudio (muy similar al estudiado en FC2 y EC)
  - Lanzamiento a ejecución en orden. Detección de riesgos en ID. Si riesgo, congelar instrucción en ID.
  - Saltos resueltos en ID
  - Etapa de ejecución con varias UFs de diferente duración. Posibilidad de finalización en desorden.
    - EX: load, store, int
    - FP: cálculos en FP (segmentada en 4 etapas)
  - Bancos de registros separados para INT y FP



Red de anticipación de operandos (by-pass)

1: Resultado de op int

2: Res. de op int o load INT

3: Res. de op FP

4: Resultado de op FP o load FP

5: Res. de op int, load INT → store int
Res. de op FP, load FP → store FP

6: Res. de op FP → store FP

Técnicas SW para explotar ILP

- Latencia de uso: número mínimo de ciclos entre una instrucción que produce un resultado y otra que lo consume

Instrucción que produce resultado	Instrucción que usa el resultado	Latencia de uso
FP aritmética	FP aritmética	3
FP aritmética	STORE FP	2
LOAD FP	FP aritmética	1
LOAD FP	STORE FP	0
LOAD Int	Int aritmética	1
Int aritmética	Int aritmética	0

Técnicas SW para explotar ILP

- Un programa: Bucle simple

```
for ( i =1000; i > 0; i--)
    a (i) = a (i) + s ;
```

Código máquina RISC-V

```
Loop: FLD     F0,0(X1)    ; F0← elemento array
      FADD.D  F4,F0,F2    ; sumar escalar en F2
      FSD     F4,0(X1)    ; almacenar resultado
      ADDI    X1,X1,-8    ; decrementar puntero
      BNE     X1,X2,Loop  ; repetir si X1 ≠ X2
```

Ciclo lanzamiento			1	2	3	4	5	6	7	8	9	10
FLD	**F0,0(X1)**	IF	**ID**	EX(1)	ME	WB						
FADD.D	**F4,F0,F2**		IF	ID	**ID**	FP1	FP2	FP3	FP4	ME(2)	WB	
FSD	**F4,0(X1)**			IF	IF	ID	ID	**ID**(3)	EX(1)	ME		
ADDI	**X1,X1,-8**					IF	IF	IF	**ID**	EX	ME	WB
BNE	**X1,X2,Loop**								IF	ID	**ID**	DS(4)

(1) Cálculo de la dirección de memoria efectiva

(2) Pasa por la etapa ME, pero no hay un acceso real a memoria

(3) El FSD no avanza mientras su R. fuente coincida con alguno de los R. destino en FP1 o FP2. Si la coincidencia es con FP3, el FSD ya puede avanzar a EX, cuando llegue a ME recibirá el operando por forwarding desde FP4 (camino 6).

(4) Delay Slot

❑ Rendimiento en una iteración

			Ciclo de lanzamiento	
Loop:	FLD	F0,0(X1)	1	
	Espera		2	; FADD.D bloqueado en ID
	FADD.D	F4,F0,F2	3	
	Espera		4	; FSD bloqueado en ID
	Espera		5	; FSD bloqueado en ID
	FSD	F4,0(X1)	6	
	ADDI	X1,X1,-8	7	
	Espera		8	; BNE bloqueado en ID
	BNE	X1,X2,Loop	9	
	Espera		10	; Salto retardado

- El rendimiento se degrada por las dependencias, a pesar del by-pass de operandos
- El CPI ideal de 1 se duplica (10 ciclos para lanzar 5 instrucciones): CPI = 2

Técnicas SW para explotar ILP

- Planificación de instrucciones: Reordenar para ocultar latencias

```
Loop:   FLD     F0,0(X1)      Ciclo 1
        ADDI    X1,X1,-8          2
        FADD.D  F4,F0,F2          3
        Espera                    4
        BNE     X1,X2,Loop        5
        FSD     F4,8(X1)          6
```

Reordenamiento para ocultar latencias

6 ciclos por iteración. CPI ~ 1

2 ciclos de overhead por el salto

- Desenrollado 4 veces para más paralelismo (elimina saltos)

```
Loop:   FLD     F0,0(X1)
        FADD.D  F4,F0,F2
        FSD     F4,0(X1)
        FLD     F6,-8(X1)
        FADD.D  F8,F6,F2
        FSD     F8,-8(X1)
        FLD     F10,-16(X1)
        FADD.D  F12,F10,F2
        FSD     F12,-16(X1)
        FLD     F14,-24(X1)
        FADD.D  F16,F14,F2
        FSD     F16,-24(X1)
        ADDI    X1,X1,-32
        BNE     X1,X2,Loop
```

Expone más paralelismo y elimina saltos

Se elimina 3 saltos y 3 decrementos

Permanecen dependencias y paradas:
Hallar ciclos por iteración y CPI

MÁS REGITROS!! Renombrado por el Compilador
(Imprescindible ??)

❑ Desenrollado + Planificación

```
Loop:   FLD      F0,0(X1)
        FLD      F6,-8(X1)
        FLD      F10,-16(X1)
        FLD      F14,-24(X1)
        FADD.D   F4,F0,F2
        FADD.D   F8,F6,F2
        FADD.D   F12,F10,F2
        FADD.D   F16,F14,F2
        FSD      F4,0(X1)
        FSD      F8,-8(X1)
        ADDI     X1,X1,-32
        FSD      F12,16(X1)      ; 16-32= -16
        BNE      X1,X2,Loop
        FSD      F16,8(X1)       ;  8-32 = -24
```

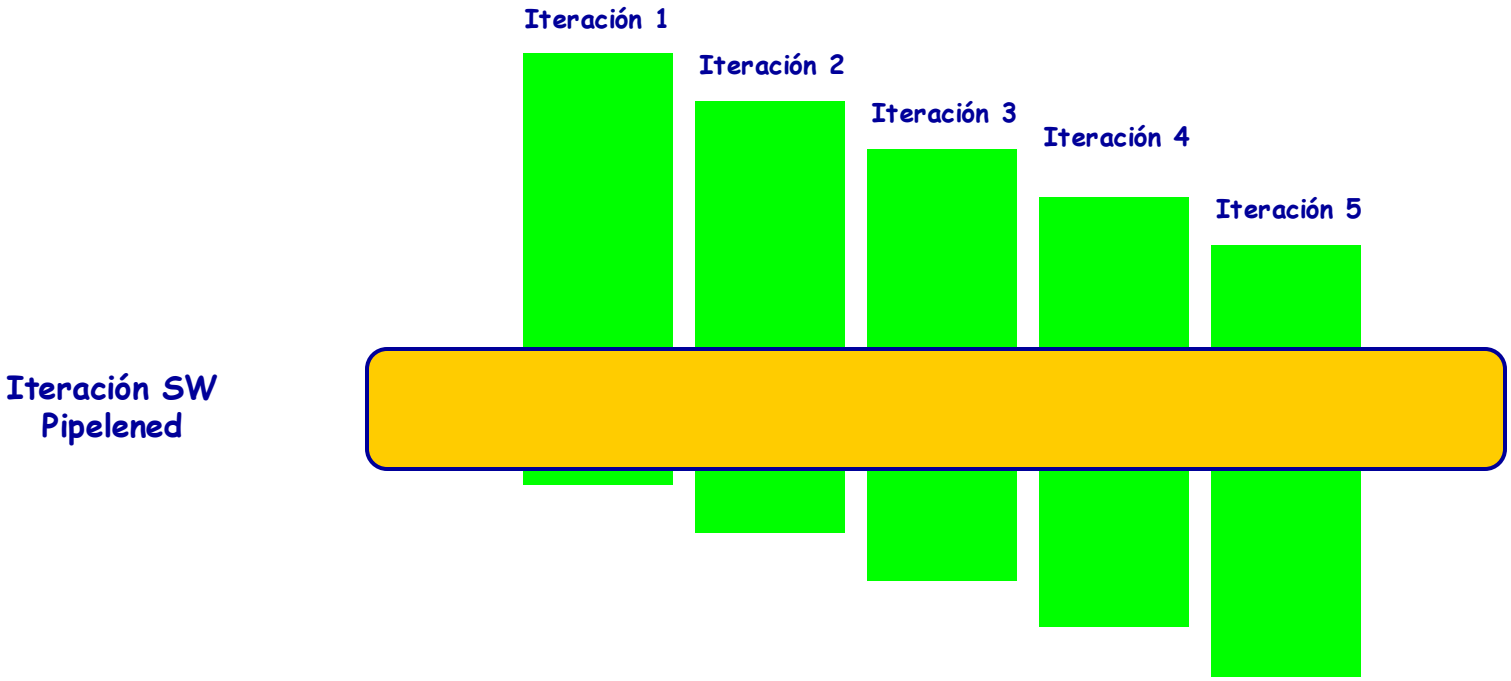
- ✓ Mover FSD después de ADDI: ojo al valor de X1
- ✓ 14 instrucciones lazadas en 14 ciclos: 3.5 ciclos por iteración
- ✓ CPI = 1
- ✓ Más registros (Imprescindible !!)

El compilador planifica para minimizar los riesgos y eliminar las paradas del "pipe"

❑ Software "pipelining"

- **Idea:**

Si las diferentes iteraciones de un bucle son independientes, tomemos instrucciones de diferentes iteraciones para aumentar el ILP
Reorganiza los bucles de manera que cada instrucción pertenece a una iteración diferente



Técnicas SW para explotar ILP

- **Software "pipelining": Ejemplo en pseudo-código de alto nivel.**
  - **Procesar un vector de 126 componentes**

```
// Bucle original
for (i=125; i>=0; i--) {
  f0=a(i);
  f4=f0+f2;
  a(i) = f4
}
```

Aumenta distancia entre instrucciones dependientes

```
// Versión SW Pipeline
f0=a(125);                  (cabecera)
f4=f0+f2;
f0=a(124);
for (i=125; i>1; i--) {     //bucle sw pl
  a(i) = f4;                //Store a(i)
  f4=f0+f2;                 // Add a(i-1)
  f0=a(i-2) }               //Load a(i-2)
a(1)=f4;                    (cola)
f4=f0+f2;
a(0)=f4;
```

❑Software "pipelining"

Antes: Unrolled 3 veces

```
1    FLD      F0,0(X1)
2    FADD.D   F4,F0,F2
3    FSD      F4,0(X1)
4    FLD      F6,-8(X1)
5    FADD.D   F8,F6,F2
6    FSD      F8,-8(X1)
7    FLD      F10,-16(X1)
8    FADD.D   F12,F10,F2
9    FSD      F12,-16(X1)
10   ADDI     X1,X1,-24
11   BNE      X1,X2,LOOP
```

Después: Software Pipelined

```
1    FSD      F4,0(X1) ; Stores a[i]
2    FADD.D   F4,F0,F2 ; Adds to a[i-1]
3    FLD      F0,-16(X1); Loads a[i-2]
4    ADDI     X1,X1,-8
5    BNE      X1,X2,LOOP;  X2 = cte. 8
```

Es necesario código previo al bucle (cabecera) y posterior al bucle (cola)

- Loop unrolling simbólico
  - Maximiza la distancia resultado-uso
  - Menor tamaño del código

Ejecución SW pipelined (suposición X1=1000)

Dir Mem	Dato
1000	a(125)
992	a(124)
984	a(123)
...	...
0	a(0)

```
F0 ← a(125)
F4 ← F0 + F2   ; a(125)+F2
F0 ← a(124)                    } Cabecera
                                    FLD     F0,0(X1)
                                    FADD.D  F4,F0,F2
                                    FLD     F0,-8(X1)
-------------------------------------------------------------
a(125) ← F4    ; a(125)+F2
F4 ← F0 + F2   ; a(124)+F2
F0 ← a(123)                    } Iter. i=125
X1 ← 992                            FSD     F4,0(X1) ; Stores a[i]
                                    FADD.D  F4,F0,F2 ; Adds to a[i-1]
                                    FLD     F0,-16(X1); Loads a[i-2]
-------------------------------------
a(124) ← F4    ; a(124)+F2
F4 ← F0 + F2   ; a(123)+F2
F0 ← a(122)                    } Iter. i=124
X1 ← 984                            FSD     F4,0(X1)
                                    FADD.D  F4,F0,F2
                                    FLD     F0,-16(X1)
-----------------------------------
        ...
        ...
F0 ← a(0)      ; Ahora X1=16   } Iter. i=2
X1 ← 8                              •
                                    •
                                    •
-------------------------------------------------------------
a(1) ← F4      ; a(1)+F2
F4 ← F0 + F2   ; a(0)+F2
a(0) ← F4      ; a(0)+F2       } Cola
                                    FSD     F4,0(X1)
                                    FADD.D  F4,F0,F2
                                    FSD     F4,-8(X1)
```

❑ Comparación

- Loop Unrolling
  - Bloque grande para planificar
  - Reduce el número de saltos
  - Incrementa el tamaño del código
  - Tiene que incluir iteraciones extra
  - Presión sobre el uso de registros

- Software Pipelining
  - No hay dependencias en el cuerpo del bucle
  - No reduce el número de saltos
  - Necesita inicio y finalización especial

Reduciendo la penalización de los saltos

❑ Tipos de saltos: Estadísticas

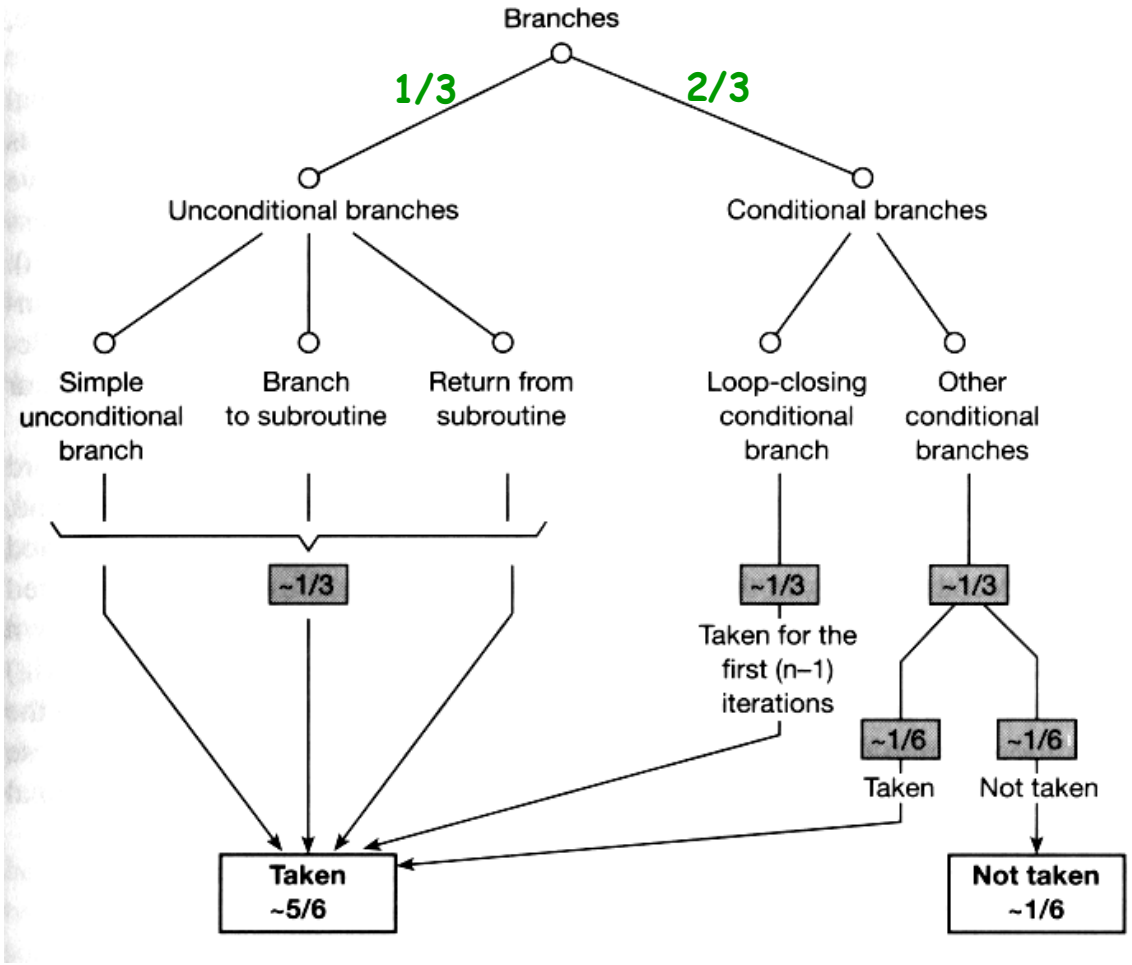
[Grohoski,G. (1990) IBM J. Res. Develop.; otros estudios dan resultados similares]

En promedio

- Instrucciones de salto
  1 de cada 5 instrucc.
- Saltos condicionales
  2 de cada 3 saltos
- Saltos incondicionales
  1 de cada 3 saltos
- Saltos tomados
  5 de cada 6 saltos
- Saltos condicionales tomados
  3 de cada 4 saltos condic.
- Saltos incondicionales tomados
  Todos

Conclusión (en promedio)

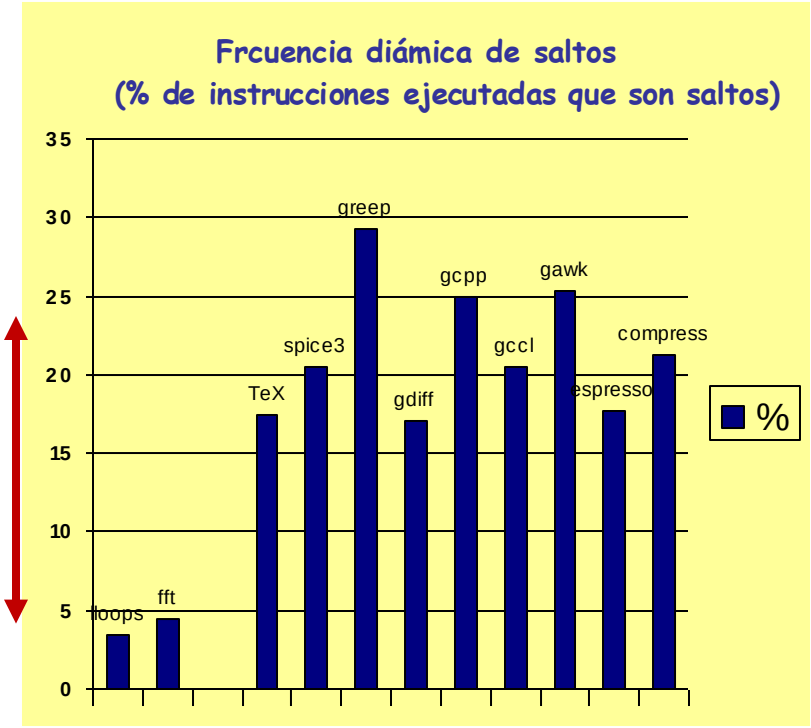
- 1 de cada 6 instrucciones es un salto tomado
- 1 de cada 8 instrucciones es un salto condicional
- 1 de cada 10 instrucciones es un salto condicional y tomado

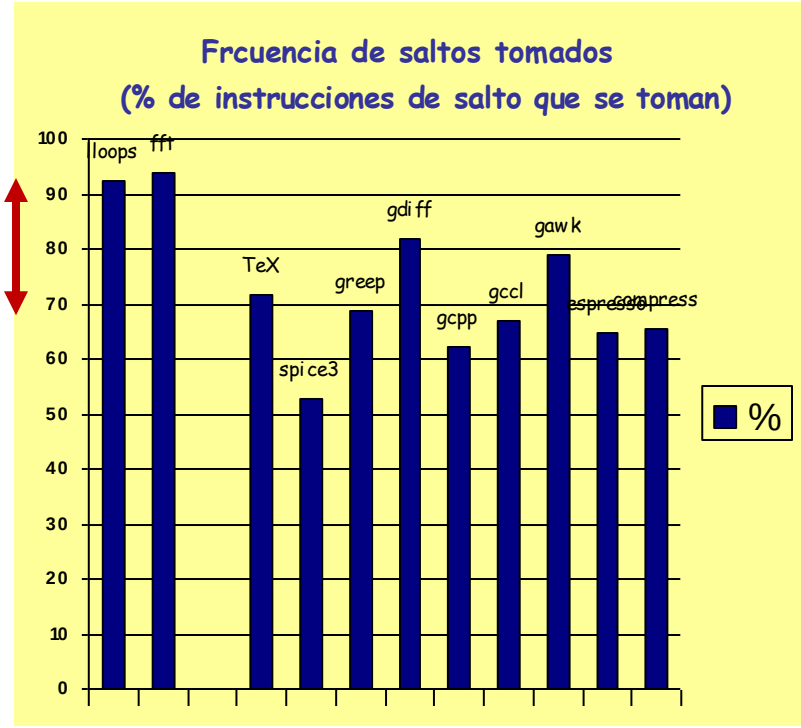


Programas enteros una de cada 4-5 instrucciones. Flotantes 1 de cada 10-20 instrucciones

Reduciendo la penalización de los saltos

❑ Tipos de saltos: Estadísticas





Conclusión

- Frecuencia de los saltos depende del tipo de programa
- El comportamiento depende del tipo de programa

❑ Predicción

Idea Básica

Cuando se detecta una instrucción de salto condicional sin resolver

- Se supone o predice el camino del salto: ***tomado o no tomado (Taken - Untaken)***
- Si el salto se predice como tomado se predice la dirección destino del salto
- La ejecución continúa de forma ***especulativa*** a lo largo del camino supuesto

Cuando se resuelve la condición

- Si la predicción fue correcta
  - ⇒ La ejecución se confirma y continúa normalmente
- Si la predicción fue incorrecta (fallo de predicción o *"misprediction"*)
  - ⇒ Se descartan todas las instrucciones ejecutadas especulativamente
  - ⇒ Se reanuda la ejecución a lo largo del camino correcto

Problemas a resolver en instrucciones de salto

1) Predecir la dirección de la instrucción destino del salto con un retardo mínimo (para saltos tomados)
2) Predecir el camino que tomará el salto
   - TAKEN (Tomado)
   - UNTAKEN (No Tomado)

❑ Acceso rápido a la dirección destino del salto I

Branch Target Address Cache (BTAC)

- Cache que almacena la dirección destino de los últimos saltos tomados
- Cuando se accede a una instrucción
  - Se accede simultáneamente a la BTAC utilizando la dirección de la instrucción
  - Si la dirección de la instrucción está en la BTAC, entonces es un salto. Si el salto se predice como tomado la dirección destino del salto se lee de la BTAC

Actualización de la BTAC

La BTAC se actualiza cuando se ejecuta la instrucción de salto y se conoce:
- *Si el salto fue tomado o no*
- *La dirección destino del salto*

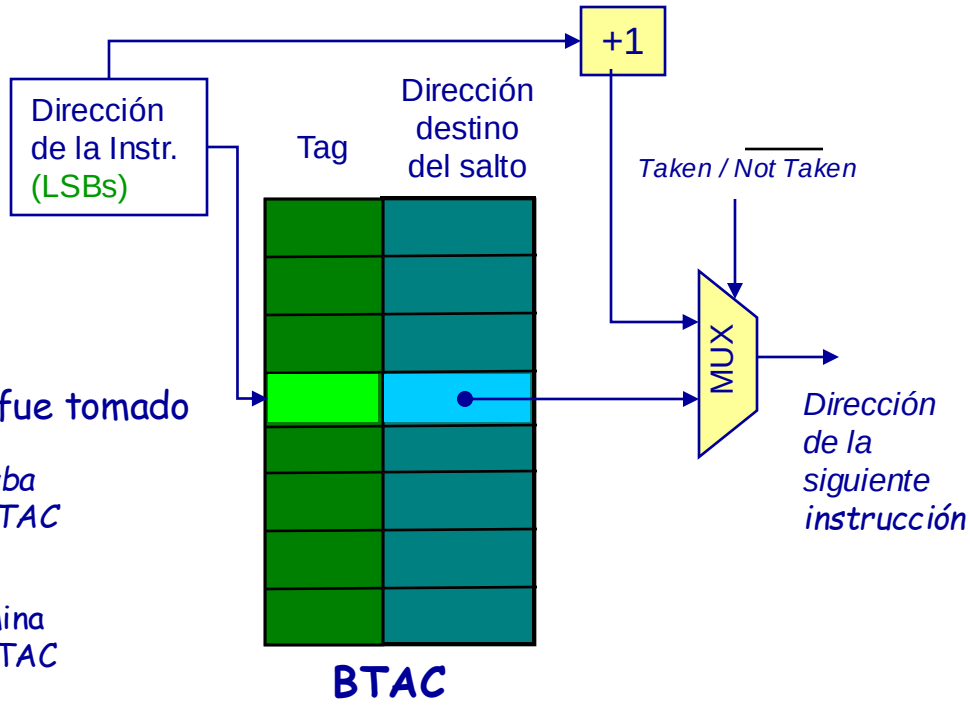
Si el salto fue tomado

Si no estaba en la BTAC → Se introduce en la BTAC

Si ya estaba en la BTAC → Se actualiza su dirección destino en la BTAC

Si el salto no fue tomado

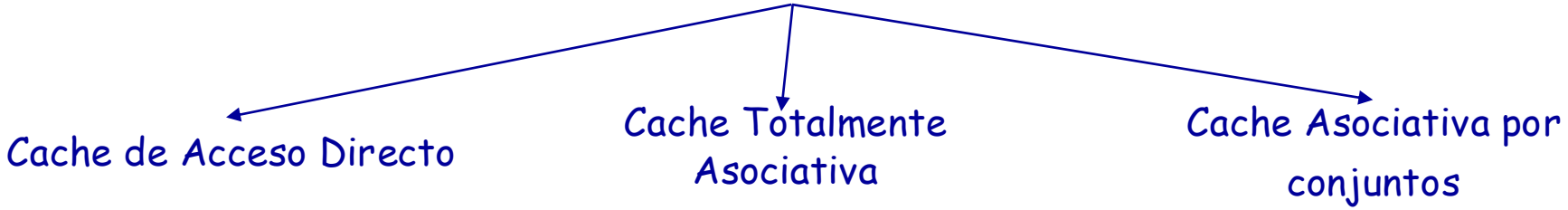
Si estaba en la BTAC → Se elimina de la BTAC



Tratamiento de Saltos: Predicción

❑ Acceso a la dirección destino del salto II

Alternativas de diseño de la BTAC



Ventaja: Menor coste
Desventaja: "*Aliasing*" (destrucción de información si dos saltos compiten por la misma entrada)

Ventaja: menos *Aliasing*
Desventaja: Mayor coste HW

Solución intermedia

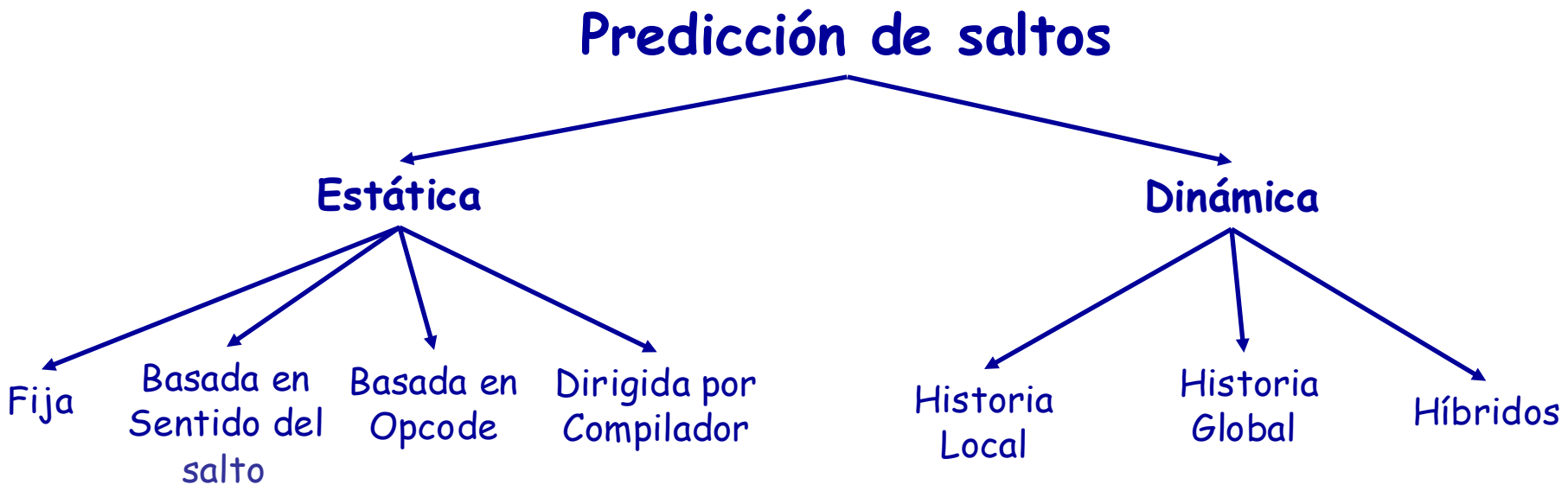
EJEMPLOS: Pentium (256) , Pentium II (512), Pentium 4 (4K) , AMD 64 (2K)

Variación Branch Target Instruction Cache

- Almacenar la instrucción: "más ventajas" si el tiempo de acceso a las instrucciones es alto
- Ejemplos: AMD K6,K7, NexGen Nx586

Tratamiento de Saltos: Predicción

- Clasificación de técnicas de predicción de saltos



Tratamiento de Saltos: Predicción

❑ Predicción estática

Predicción Fija

ALWAYS TAKEN

- Predecir todos los saltos como ***tomados***
- Mayor número de aciertos de predicción (3 de cada 4 saltos cond. son tomados)
- Mayor coste hardware (necesita almacenar la dirección destino del salto)

ALWAYS NOT TAKEN

- Predecir todos los saltos como ***no tomados***
- Menor número de aciertos de predicción (sólo 1 de cada 4 saltos cond. es no tomado)
- Menor coste hardware

Predicción basada en el SENTIDO del salto

Saltos hacia atrás : TOMADOS

La mayoría de saltos hacia atrás corresponden a bucles

Saltos hacia delante: NO TOMADOS

La mayoría de saltos hacia delante corresponden a IF-THEN-ELSE

Mal comportamiento en programas con pocos bucles y muchos IF-THEN-ELSE

Tratamiento de Saltos: Predicción

❑ Predicción estática

Predicción basada en el OPCODE de la instrucción de salto

Fundamento: La probabilidad de que un salto sea tomado depende del tipo de salto

El salto es tomado para ciertos códigos de operación y no tomado para otros

Predicción dirigida por el COMPILADOR

Basada en el tipo de CONSTRUCCIÓN

El compilador predice si el salto será tomado o no dependiendo del tipo de construcción de control

Basada en PROFILING

El compilador predice en función del comportamiento de esa instrucción en ejecuciones previas del programa

Especificado por el PROGRAMADOR

El programador indica al compilador si el salto debe ser tomado o no (mediante directivas específicas)

- Se añade un ***Bit de Predicción*** al opcode de la instrucción
- El compilador activa o desactiva este bit para indicar su predicción

❑ Predictores Dinámicos

Idea básica

La predicción se realiza observando el comportamiento de las instrucciones de salto en las últimas ejecuciones (Historia)

Necesario almacenar la historia de las últimas ejecuciones del salto

Predictores de 1 bit de historia	*Predictores de 2 bits de historia (bimodal)*	*Predictores de 3 bits de historia*
EJEMPLOS • Gmicro 100 (1991) • Alpha 21064 (1992) • R8000 (1994)	EJEMPLOS • MC68060 (1993) • Pentium (1994) • Alpha 21064A (1994) • Alpha 21164 (1995) • PA 8500 (1999) • UltraSparc (1995) • PowerPC 604 (1995) • PowerPC 620 (1996) • R10000 (1996)	EJEMPLOS • PA 8000 (1996)

Evolución

Predictores correlacionados
Predictores híbridos

Tratamiento de Saltos: Predicción

❑ Predictores Dinámicos

Predictor de un BIT

- Utilizan un bit de predicción por cada instrucción de salto
- El bit de predicción refleja el comportamiento de la última ejecución de la instrucción de salto

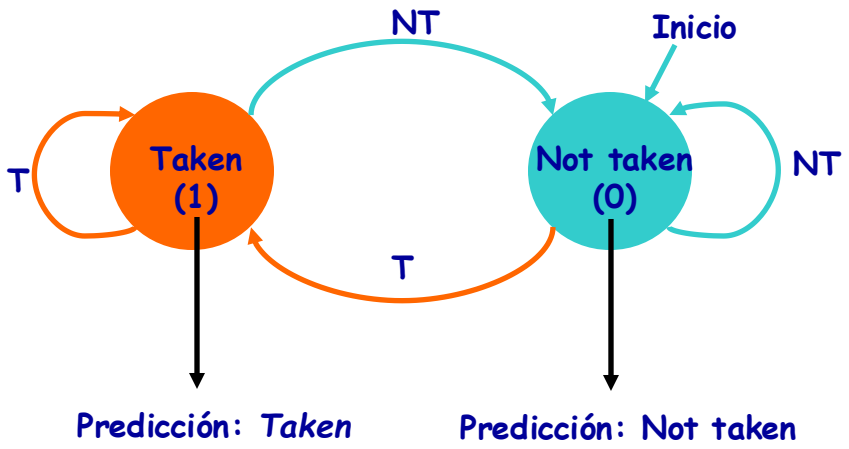
  ⇒*Indica si en la anterior ejecución el salto fue tomado o no*

Predicción

- El salto se predice como *Taken* si en la última ejecución fue tomado
- El salto se predice como *Not Taken* si en la última ejecución no fue tomado

FUNCIONAMIENTO

- *Máquina de dos estados:*
  - Not taken (0)
  - Taken (1)
- *Registro de historia*
  - Contador saturado de 1 bit
- *Predicción*
  - Valor del registro de historia



LIMITACIÓN

- Sólo se registra el comportamiento de la última ejecución del salto
- Dos malas predicciones en los cambios

Más Bits

Cambios de estado:
T: el salto ha sido tomado
NT: el salto no ha sido tomado

Tratamiento de Saltos: Predicción

❑ Predictores Dinámicos

Predictor de dos bits (BIMODAL)

- *Utilizan dos bits de predicción por cada instrucción de salto*
- *Estos bits reflejan el comportamiento de las últimas ejecuciones de ese salto*

Predicción

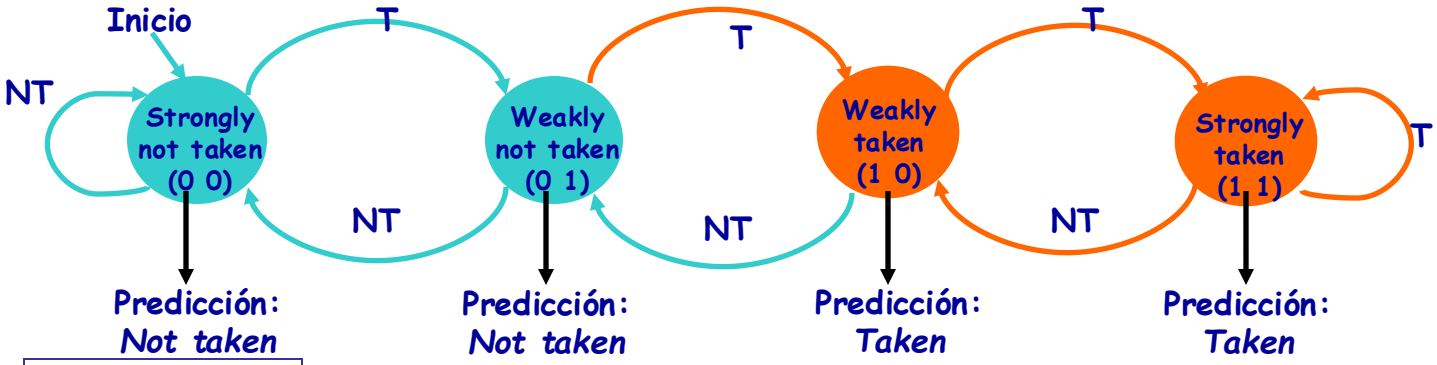
- Un salto que se toma repetidamente se predice como *Taken*
- Un salto que no se toma repetidamente se predice como *Not taken*
- Si un salto toma una dirección inusual una sola vez, el predictor mantiene la predicción usual

Funcionamiento

Máquina de cuatro estados:

- Strongly not taken (00)
- Weakly not taken (01)
- Weakly taken (10)
- Strongly taken (11)

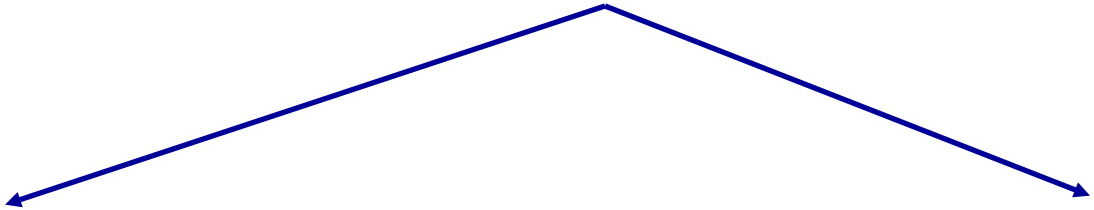
- *Registro de historia*
  - Contador saturado de 2 bits
- *Predicción*
  - bit más significativo del registro de historia



Cambios de estado:
T: el salto ha sido tomado
NT: el salto no ha sido tomado

❑ Predictores Dinámicos

Implementación de los bits de predicción



1) Branch Target Buffer (BTB)

Añade los bits de predicción a las entradas de la BTAC. La BTAC con bits de predicción se denomina BTB

EJEMPLOS	
• MC 68060	256 x 2 bit
• Pentium	256 x 2 bit
• R8000	1K x 1 bit
• PM1	1K x 2 bit
• Pentium II	512x2 bit
• Pentium 4	4kx2bits

2) Tabla de historia de saltos (BHT)

Utiliza una tabla especial, distinta de la BTAC, para almacenar los bits de predicción

EJEMPLOS	
• Gmicro 100	256 x 1 bit
• PowerPC 604	512 x 2 bit
• R10000	512 x 2 bit
• PowerPC 620	2K x 2 bit
• PA 8000	256 x 3 bit
• Alpha 21164A	2K x 2 bit
• AMD64	16Kx2bits

Actual: Samsung M3 (3 niveles; 128, 4k, 16K)

❑ Predictores Dinámicos: Implementación

1) Branch Target Buffer (BTB): bits acoplados

La BTB almacena

- La dirección destino de los últimos saltos tomados
- Los bits de predicción de ese salto

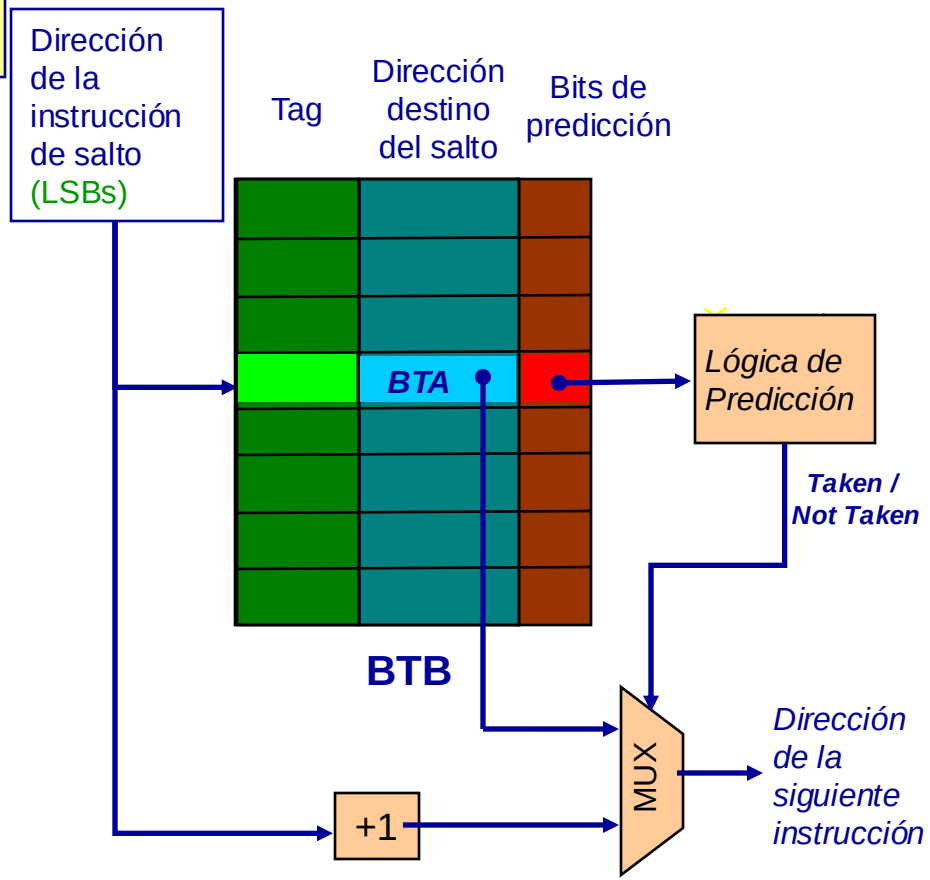
Actualización de la BTB

Los campos de la BTB se actualizan después de ejecutar el salto y se conoce:

- Si el salto fue tomado o no
  - ⇒ Actualizar bits de predicción
- La dirección destino del salto
  - ⇒ Actualizar BTA

***Predicción Implícita** (sin bits de predicción)*

Imita un sólo bit de predicción
- Si la instrucción de salto está en la BTB
  - ⇒ El salto se predice como tomado
- Si la instrucción de salto no está en la BTB
  - ⇒ El salto se predice como no tomado



DESVENTAJA: Sólo se pueden predecir aquellas instrucciones de salto que están en la BTB

❑ Predictores Dinámicos: Implementación

2) Tabla de historia de saltos (BHT): bits desacoplados

Existen dos tablas distintas:

- La BTAC, que almacena la dirección destino de los últimos saltos tomados
- La BHT, que almacena los bits de predicción de todas las instrucciones de salto condicional

Ventaja

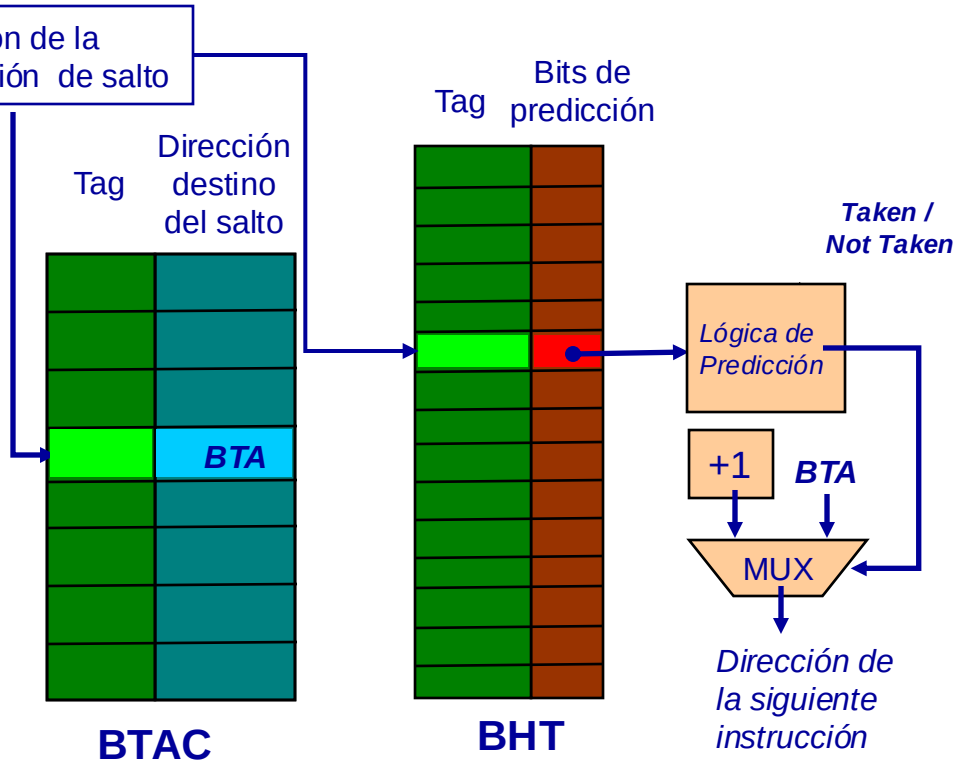
Puede predecir instruc.que no están en la BTAC (más entradas en BHT que en BTAC)

Desventaja

Aumenta el hardware necesario
⇒ 2 tablas (más bit de marca)

Acceso a la BHT

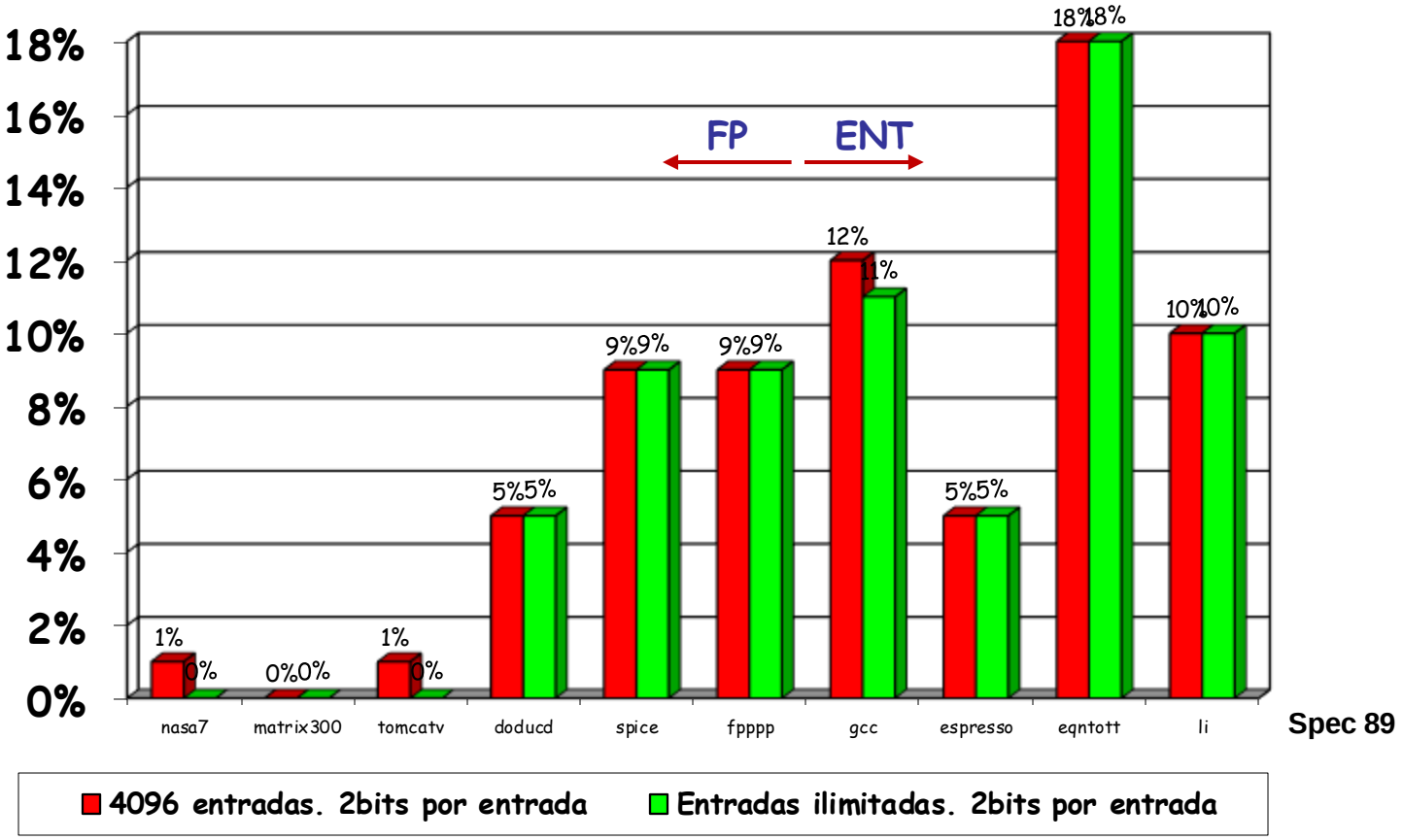
- Usando los bits menos significativos de la dirección
  - Sin TAGs ⇒ Menor coste (opción + habitual)
  - Compartición de entradas
    ⇒ Se degrada el rendimiento
- Asociativa por conjuntos
  - Mayor coste ⇒ Tablas pequeñas
  - Para un mismo coste hardware
    ⇒ Peor comportamiento



Tratamiento de Saltos: Predicción

❑ Predictores Dinámicos

Comportamiento: % de saltos mal predichos



- A partir de cierto valor, aumentar el tamaño del predictor no mejora el éxito de las predicciones
- Muchos fallos en algunos programas (enteros) ¿Por qué?

❑ Predictores Dinámicos

Otras formas de gestionar la historia

1. Muchas instrucciones de salto ejecutan **patrones repetitivos**

For (i=1; i<=4; i++) { } → Patrón: {1110}

- 1 = Taken
- 0 = Not Taken

Si conocemos el comportamiento del salto en las 3 últimas ejecuciones podemos predecir como se comportará en la siguiente ejecución

⇒ Predicción basada en **historia LOCAL**

Historia	Predicción
111	0 (NT)
011	1 (T)
101	1 (T)
110	1 (T)

Material de apoyo: H&P 6th ed. Sección 3.3

Tratamiento de Saltos: Predicción

❑ Predictores Dinámicos

2. Muchas instrucciones de salto dependen del comportamiento de otros **saltos recientes (historia global)**

Observación: Los saltos están relacionados; el comportamiento de los últimos saltos afecta a la predicción actual. **Idea:** Almacenar el comportamiento de los últimos n saltos y usarlo en la selección de la predicción.

Ejemplo:

(Suposición: d está almacenado en X1)

```
b1: if (d = 0) then          L1:   BNE   X1, X0, L1   ; salto b1 (Salto si d ≠ 0)
        d = 1                      ADDI  X1, X0, 1    ; Como d=0, hacer d=1
b2: if (d =1) then                 SUBI  X3, X1, 1    ; X3=d(X1)-1
                                   BNE   X3, X0, L2   ; salto b2  (Salto si d≠ 1)
                             L2:   .....
```

X3=0 ⇒ d=1
X3≠0 ⇒ d≠1

*Si conocemos el comportamiento de la última ejecución de b1 podemos predecir el comportamiento de b2 en la siguiente ejecución (**"si b1 no se toma, entonces b2 tampoco"**)*

↓

Predicción basada en historia ***GLOBAL***

❑ Predictores Dinámicos

Ejemplo (continuación)

Relación entre los dos saltos

```
        BNE   X1, X0, L1  ; salto b1 (Salto si d ≠ 0)
        ADDI  X1, X0, 1   ; Como d=0, hacer d=1
L1:     SUBI  X3, X1, 1   ; X3=d(X1)-1
        BNE   X3, X0, L2  ; salto b2  (Salto si d≠ 1)
        ..........
L2:
```

Caso 1: d=0,1,2,…

Valor de d	d ≠ 0?	salto b1	d antes de b2	d≠1?	salto b2
0	no	NT	1	no	NT
1	si	T	1	no	NT
2	si	T	2	si	T

***Si b1 no se toma, entonces b2 tampoco**: correlación entre saltos*

Predictores Dinámicos

Ejemplo (continuación)

Comportamiento del predictor de un bit (estado inicial "not taken" NT)

```
        BNE   X1, X0, L1  ; salto b1 (Salto si d ≠ 0)
        ADDI  X1, X0, 1   ; Como d=0, hacer d=1
L1:     SUBI  X3, X1, 1   ; X3=d(X1)-1
        BNE   X3, X0, L2  ; salto b2  (Salto si d≠ 1)
        ..........
L2:
```

Caso 2: d=2,0,2,0,…

Valor de d	Predicción de b1	b1	Nueva predicción de b1	Predicción de b2	b2	Nueva predicción de b2
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

Muchos fallos de predicción

Predictores Dinámicos

Solución

- Predictor de dos niveles (**1**,**1**):

  Para cada salto existen 2^1 predictores de **1** bit. El comportamiento del último salto (**1**) determina el predictor que se usa.

- Predictor de dos niveles (**m**,**n**)

  Para cada salto existen 2^m predictores de **n** bits. El comportamiento de los últimos **m** saltos determinan el predictor que se usa

Significado de los bit de predicción en un predictor (1,1)

Dos predictores de un bit (P0/P1)

Casos posibles

Bits de predicción: P0 / P1	Predicción si el último salto no tomado: usar P0	Predicción si el último salto tomado: usar P1
NT / NT	NT	NT
NT / T	NT	T
T / NT	T	NT
T / T	T	T

Ejemplo: Tamaño de un predictor (2,2) de 4k entradas: $2^2 \times 2 \times 4K = 32Kb$

❑ Predictores Dinámicos

Ejemplo (continuación)

Comportamiento del predictor de dos niveles (1,1) (estado inicial "not taken" NT)

```
        BNE   X1, X0, L1     ; salto b1 (Salto si d ≠ 0)
        ADDI  X1, X0, 1      ; Como d=0, hacer d=1
L1:     SUBI  X3, X1, 1      ; X3=d(X1)-1
        BNE   X3, X0, L2     ; salto b2  (Salto si d≠ 1)
        ..........
L2:
```

Caso 2: d=2,0,2,0,…

Sólo se predice mal la 1ª iteración (d=2)

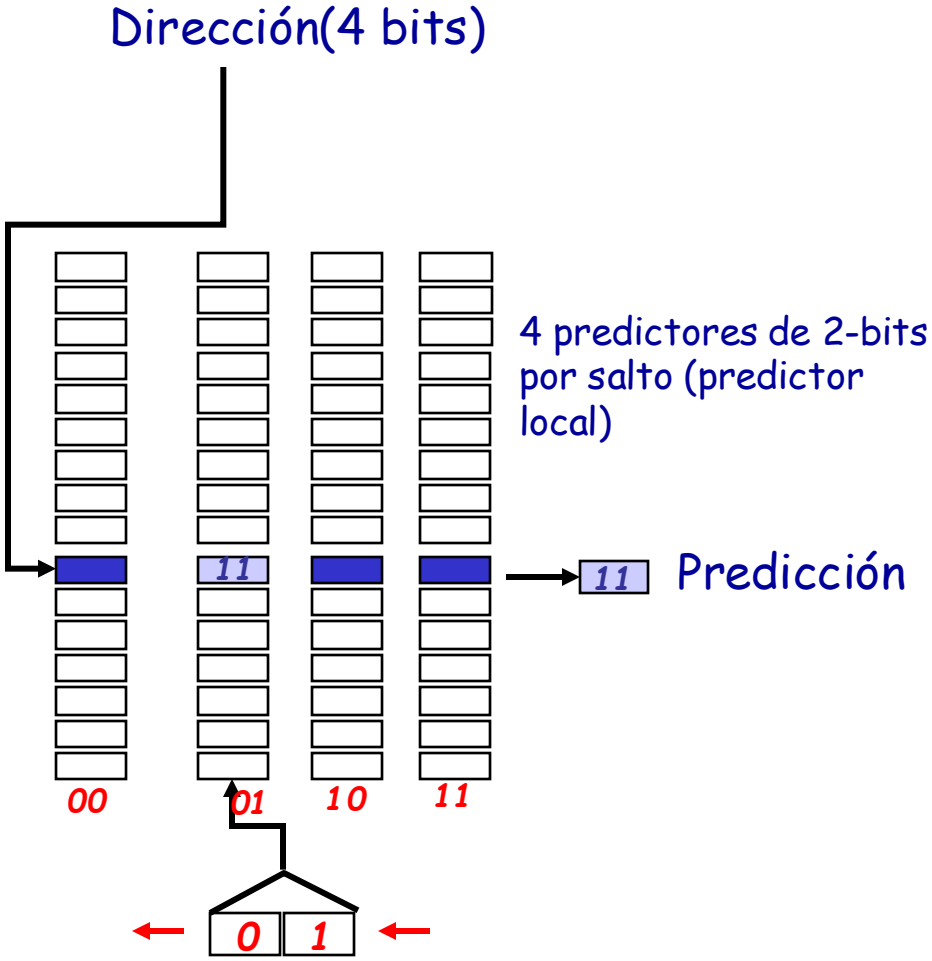
d = ?	Predicción de b1	b1	Nueva predicción de b1	Predicción de b2	b2	Nueva predicción de b2
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T /NT	NT	T/NT	NT/T	NT	NT/T

(Subrayado en rojo: Bit de predicción seleccionado en cada caso, en función del comportamiento del salto anterior)

❑ Predictores Dinámicos

Implementación para
Predictor de dos niveles (2,2)
con 16 entradas

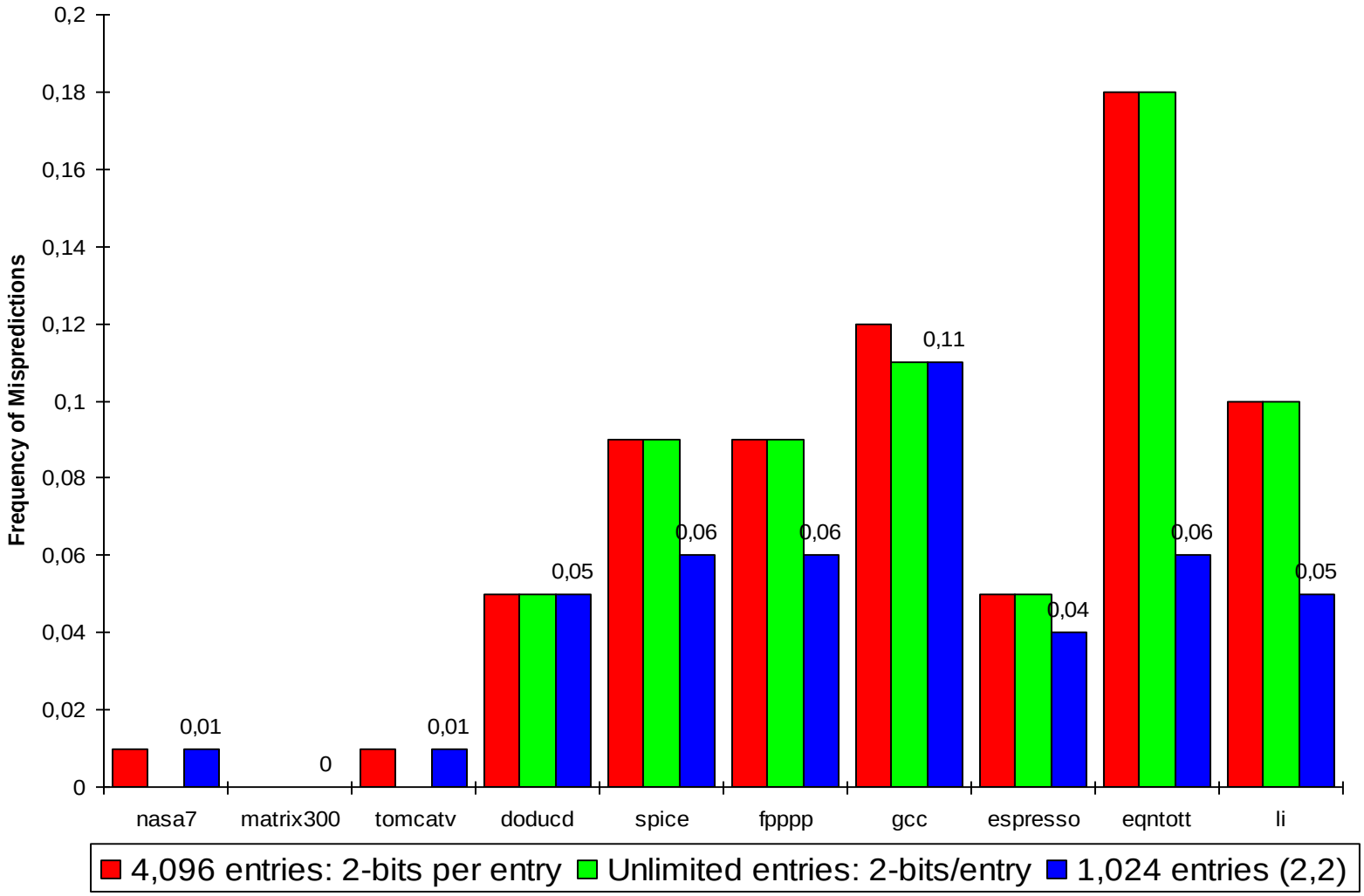
Tamaño = $2^2 \times 2 \times 16 = 128$ bits



Tratamiento de Saltos: Predicción

❑ Predictores Dinámicos

Comportamiento: Frecuencia de saltos ejecutados que se predicen mal



❑ Predictores híbridos

Idea básica

- Cada uno de los predictores estudiados tiene sus ventajas y sus inconvenientes
- Combinando el uso de distintos predictores y aplicando uno o otro según convenga, se pueden obtener predicciones mucho más correctas

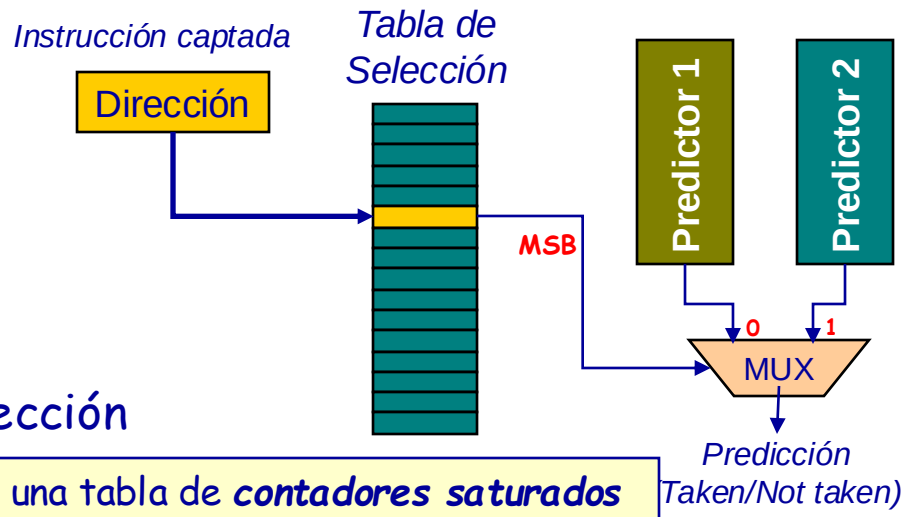
Predictor híbrido

Mezcla varios predictores y añade un mecanismo de selección del predictor

Mecanismo de selección

Elige, en cada caso, el predictor que haya dado mejores resultados hasta el momento

Implementación del mecanismo de selección



Para combinar dos predictores, P1 y P2, se utiliza una tabla de ***contadores saturados de dos bits*** indexada por la dirección de la instrucción de salto

P1	P2	Actualiz. del contador
Fallo	Fallo	Cont no varía
Fallo	Acierto	Cont = Cont +1
Acierto	Fallo	Cont = Cont -1
Acierto	Acierto	Cont no varía

- Si P2 acierta más que P1
  ⇒ *Cont* aumenta
- Si P1 acierta más que P2
  ⇒ *Cont* disminuye

Bit más signif. del contador	Predictor seleccionado
0	P1
1	P2

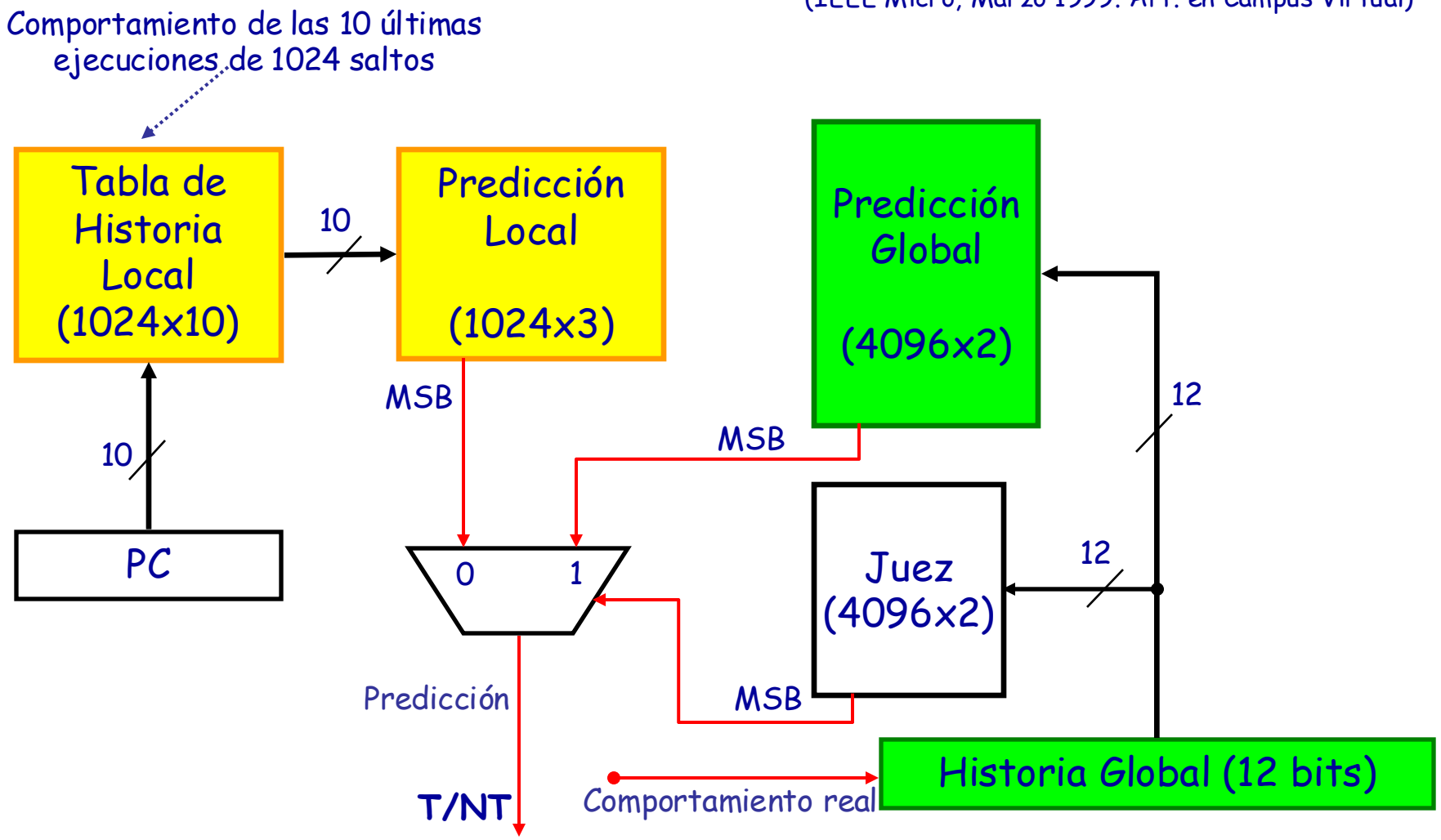
Ejemplo: Alpha 21264

- Predictor competitivo (Tournament Predictor)

- Predictor Local: Predicción de un salto en función del comportamiento previo de ese mismo salto
  - Considera las 10 últimas ejecuciones del salto
- Predictor global: Predicción de un salto en función del comportamiento de los últimos 12 saltos ejecutados
- Juez: Decide cuál de las dos predicciones se aplica
  - Selecciona el predictor que esté manifestando el mejor comportamiento
- Actualización: al resolver cada salto
  - Se actualizan los predictores en función de su acierto o fallo
  - Si los dos predictores hicieron una predicción distinta, se actualiza el juez para que favorezca al que acertó
- Gran importancia para la ejecución especulativa en 21264 (hasta 80 instrucciones en la ventana)
- Tasas de predicción correcta (benchmarks): 90-100%

Tournament predictor del Alpha 21264

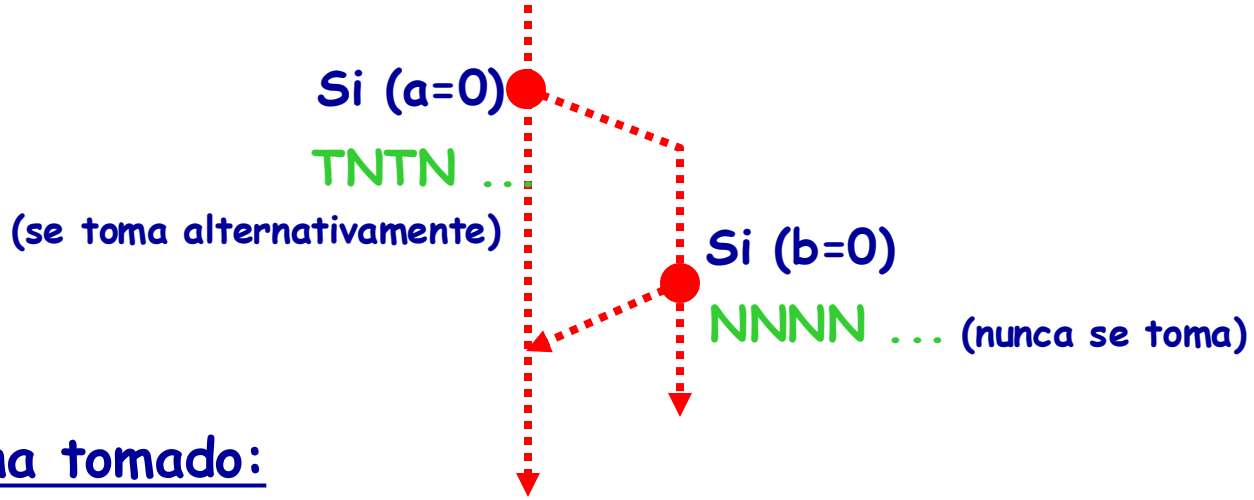
(IEEE Micro, Marzo 1999. Art. en Campus Virtual)



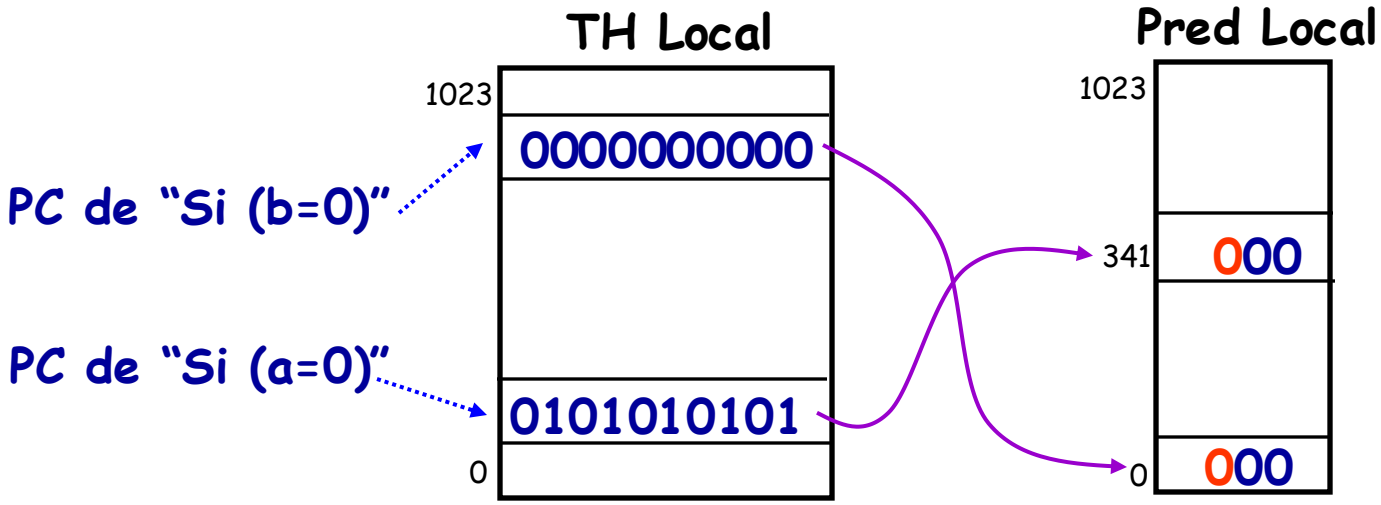
Juez: Acierto global y fallo local => incrementa
Fallo global y acierto local => decrementa

Ejemplos de funcionamiento (1)

Programa con dos saltos que tiene el comportamiento descrito

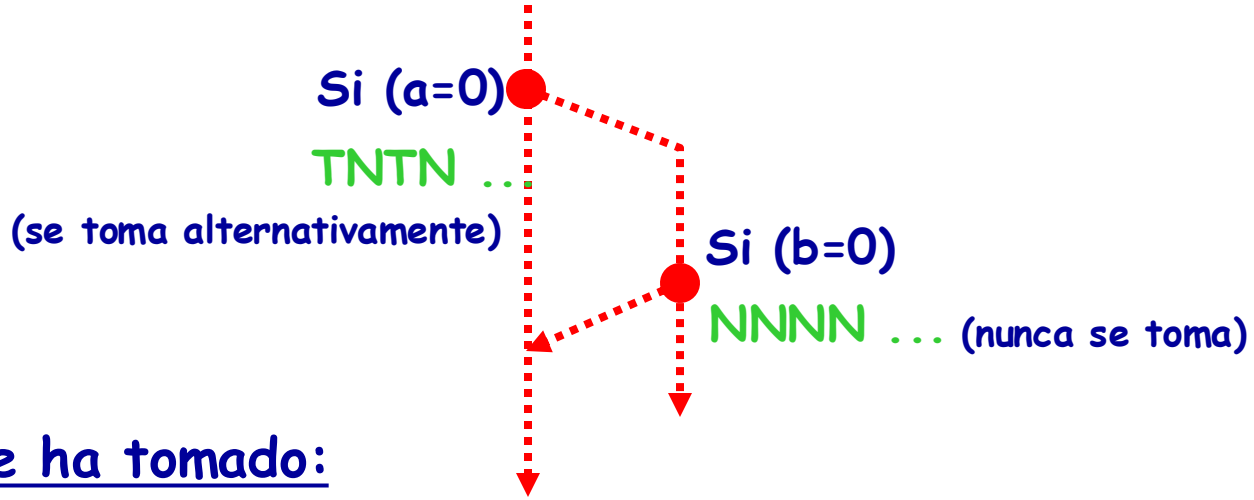


"Si(a=0)" se ha tomado:

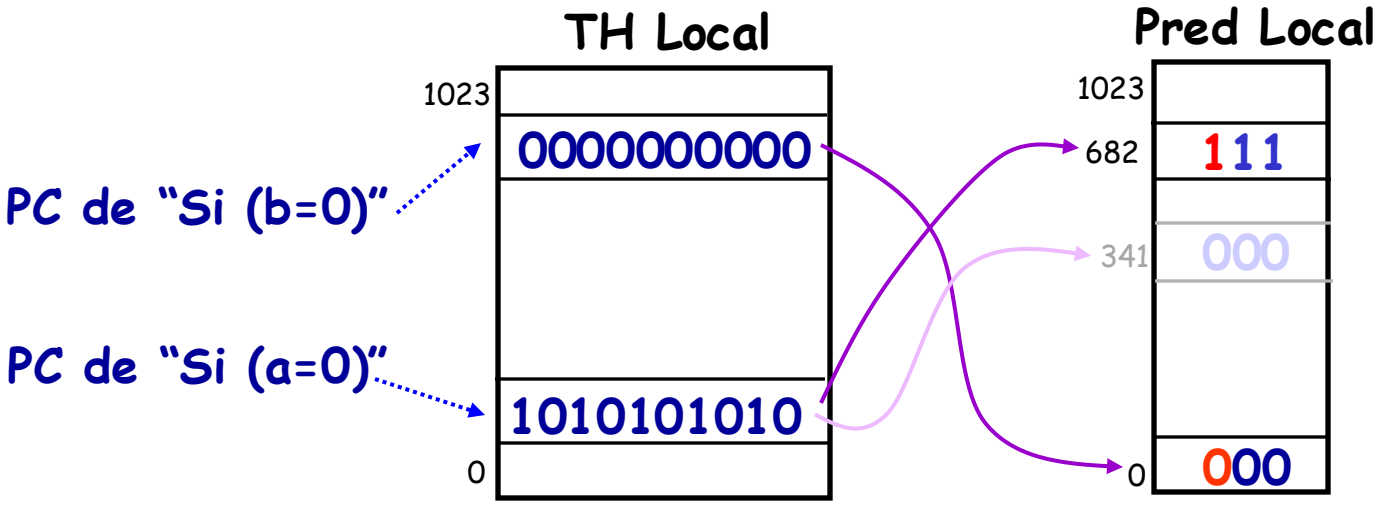


Ejemplos de funcionamiento (2)

Programa con dos saltos que tiene el comportamiento descrito

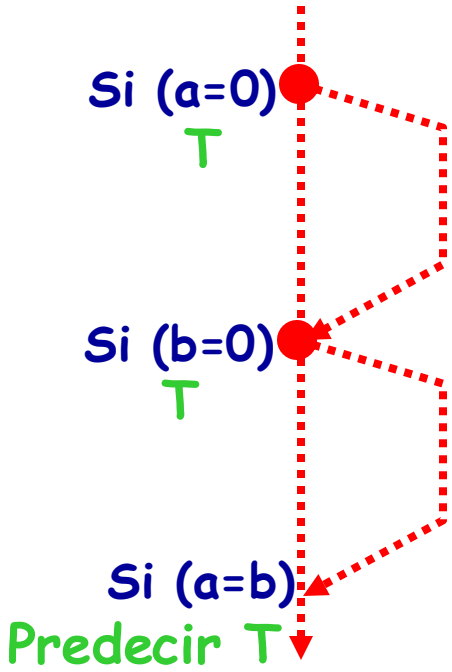


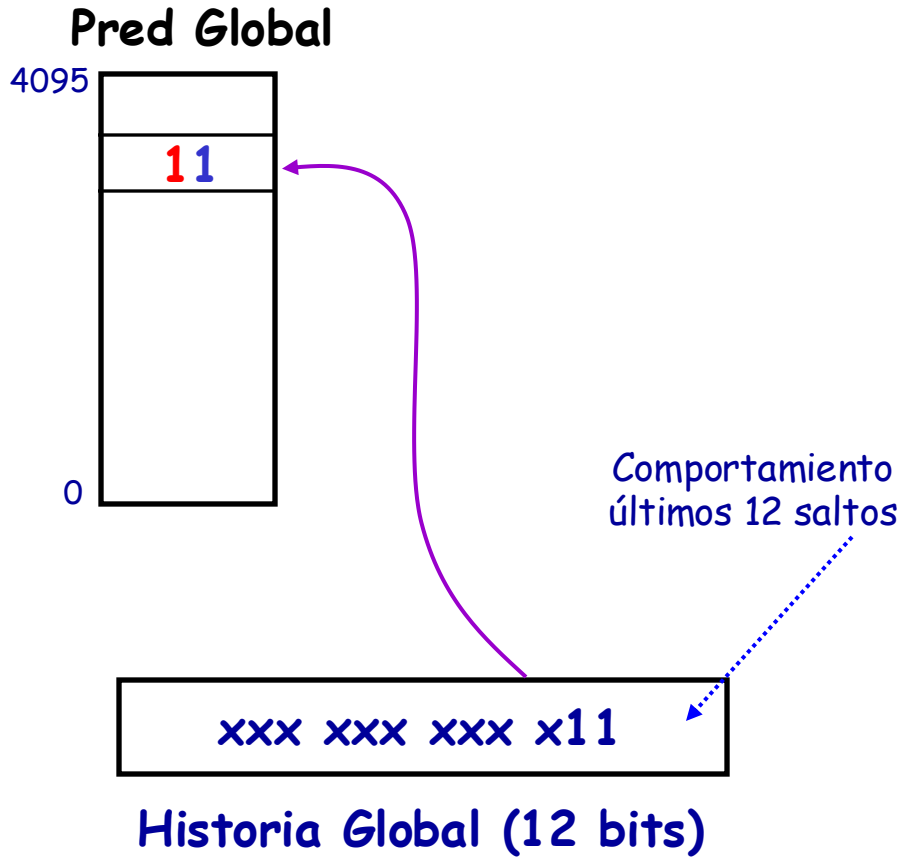
"Si(a=0)" no se ha tomado:



Ejemplos de funcionamiento (3)

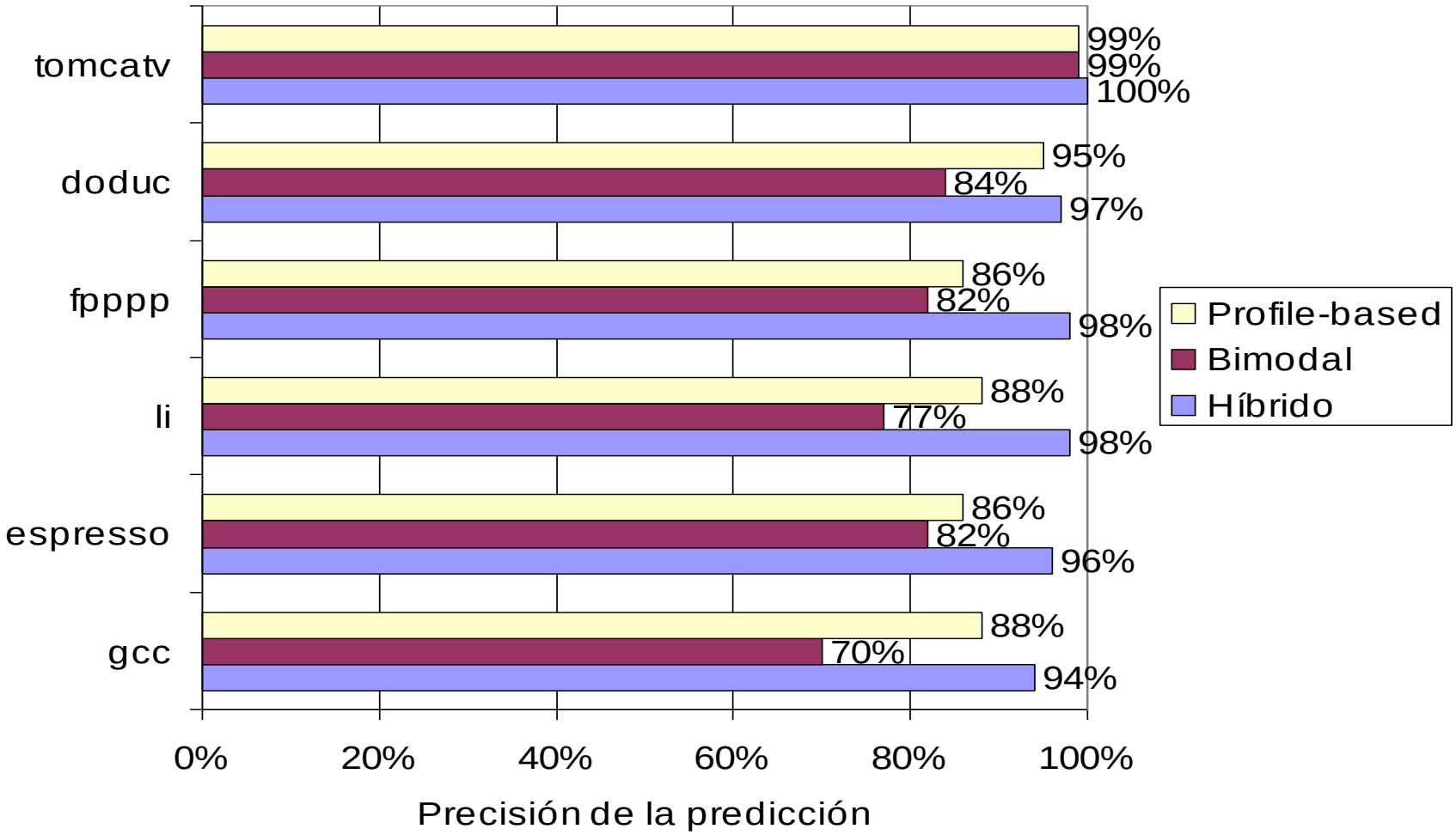
Programa con tres saltos que tiene el comportamiento descrito





Tratamiento de Saltos: Predicción

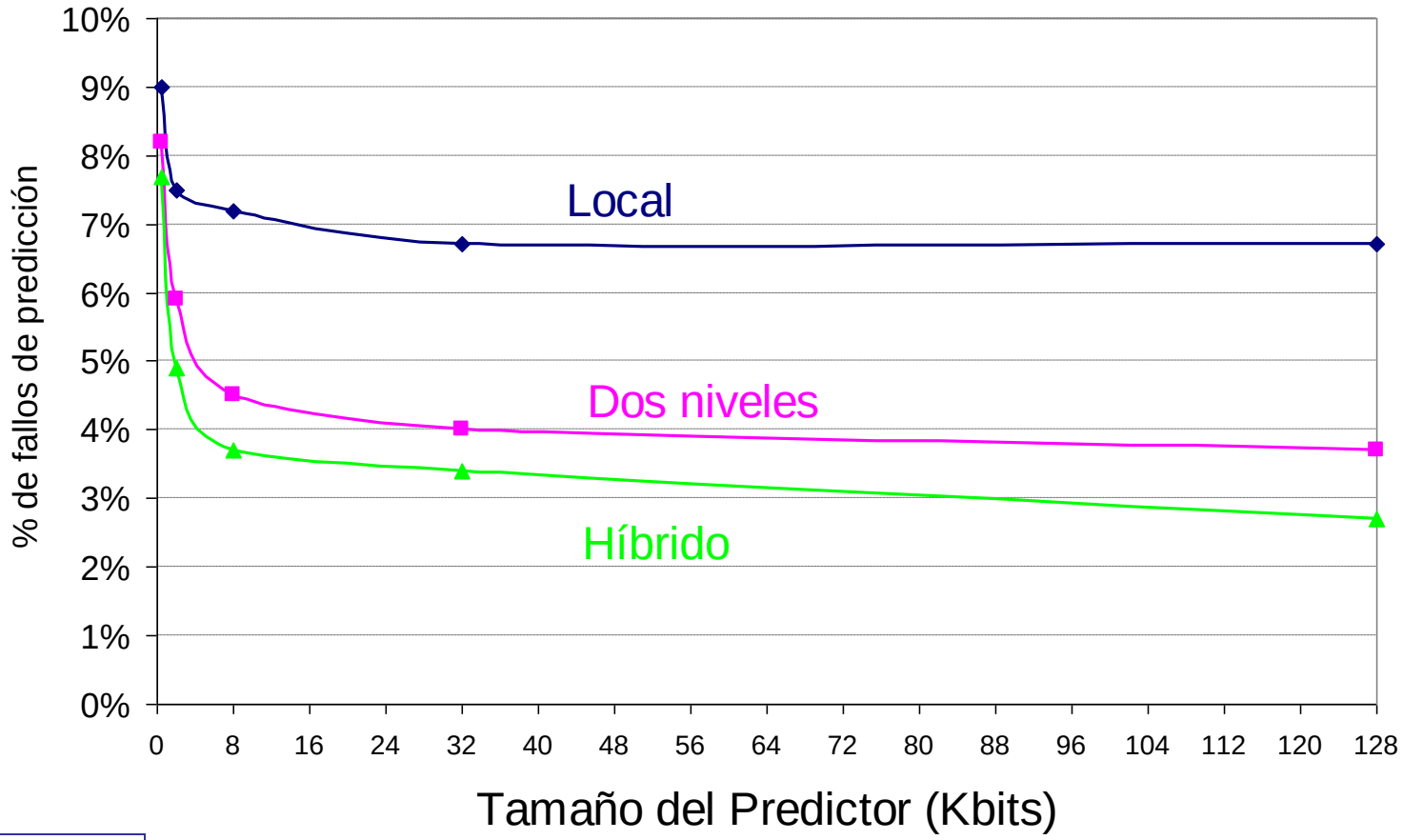
❑ Predictores: Comportamiento



Profile_based- Predictor estático

Predictores: Comportamiento

- La ventaja del predictor híbrido es su capacidad de seleccionar el predictor correcto para un determinado salto
- Muy importante para programas enteros
  - Un predictor híbrido selecciona el global casi 40% de las veces para SPEC integer y menos del 15% de las veces para SPEC FP

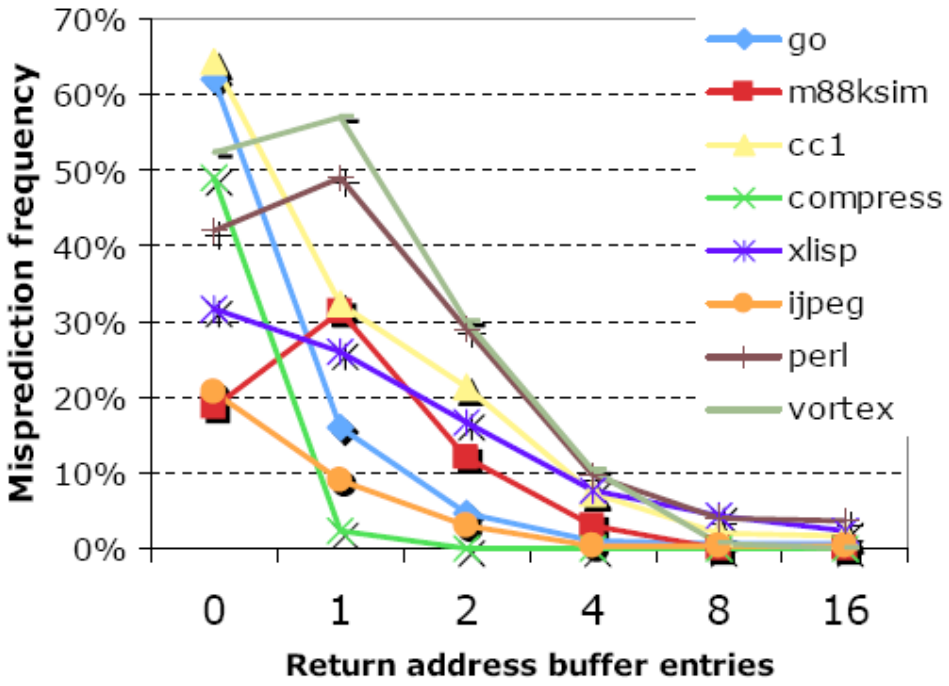


❑ Predicción de los retornos

- La precisión de los predictores con los retornos es muy baja: La dirección de retorno es diferente en función de la llamada
- Solución : Pila de direcciones de retorno (8 a 16 entradas)

EJEMPLOS

• UltraSparc I, II	4 entradas
• Pemtium Pro	16 entradas
• R10000	1 entrada



❑ Recuperación de fallos de predicción (misprediction)

Tareas básicas

1) Descartar los resultados de las instrucciones ejecutadas especulativamente

2) Reanudar la ejecución por el camino correcto con un retardo mínimo

1) Descarte de los resultados

- Los resultados de estas instrucciones especulativas se almacenan en registros temporales (registros de renombramiento o *Buffer* de reordenamiento)
- Estas instrucciones no modifican los contenidos de los registros de la arquitectura ni de la memoria

Si la ejecución fue correcta

Se actualizan los registros de la arquitectura y/o la memoria

Si la ejecución fue incorrecta

Se descartan los resultados de los registros temporales

Tratamiento de Saltos: Predicción

❑ Recuperación de fallos de predicción (misprediction)

2) Reanudación de la ejecución por el camino correcto

El procesador debe guardar, al menos, la dirección de comienzo del camino alternativo

Si la predicción fue "*Not taken*"

El procesador debe calcular y almacenar la dirección destino del salto

Si la predicción fue "*Taken*"

El procesador debe almacenar la dirección de la instrucción siguiente al salto

Ejemplos: PowerPC 601 – 603 - 605

Reducción de los retardos en la recuperación de fallos

El procesador puede guardar, no solo la dirección del camino alternativo, sino prebuscar y almacenar algunas instrucciones de este camino

Si la predicción fue "*Taken*"

- El procesador almacena la dirección del camino secuencial
- El procesador prebusca y almacena las primeras instrucciones secuenciales

Si la predicción fue "*Not taken*"

- El procesador calcula y almacena la dirección destino del salto
- El procesador prebusca y almacena las primeras instrucciones del destino del salto

Ejemplos: 2 buffer Power1, Power2, Pentium, UltraSparc (16), R10000 (256 bits)
3 buffer Nx586 (2 pendientes)

Tratamiento de Saltos: Otras alternativas

❑ Ejecución condicional de instrucciones

Idea básica

- Eliminar, parcialmente, los saltos condicionales mediante ***instrucciones de ejecución condicional***
- Una instrucción de ejecución condicional está formada por:
  - Una condición
  - Una operación
- Ejecución condicional
  - Si la condición es ***cierta*** ⇒ La instrucción se ejecuta
  - Si la condición es ***falsa*** ⇒ La instrucción se comporta como NOP

Ejemplo

```
        BNEZ    R1, L
        MOV     R2, R3
L
```

⟶

```
CMOVZ   R2, R3, R1
```

Ventaja: Buena solución para implementar alternativas simples de control
Desventaja: Consumen tiempo en todos los casos. Más lentas que las incondicionales
Ejemplos: ARM, Alpha, Hp-Pa, MIPS, Sparc

Resumen

- ✓ Predictor bimodal bueno para Loop (programas FP)
- ✓ Predictores de dos niveles buenos para IF then else
- ✓ Predicción de la dirección destino importante
- ✓ Ejecución condicional reduce el número de saltos

Tratamiento de dependencias de datos en ejecución

❑ Planificación dinámica: Procesador

Modifica la secuencia de instrucciones resolviendo las dependencias en tiempo de ejecución. Disponibilidad de más unidades funcionales. Código válido para diferentes implementaciones

- **Problema** : Lanzamiento de instrucciones en orden.

```
FDIV.D  F0,F2,F4      S1
FADD.D  F10,F0,F8     S2   S2 depende de S1
FSUB.D  F12,F8,F14    S3   S3 es independiente de las demás
```

La etapa ID bloquea la ejecución en S2 hasta que se resuelve la dependencia (F0 disponible) y FSUB.D no puede ejecutarse.

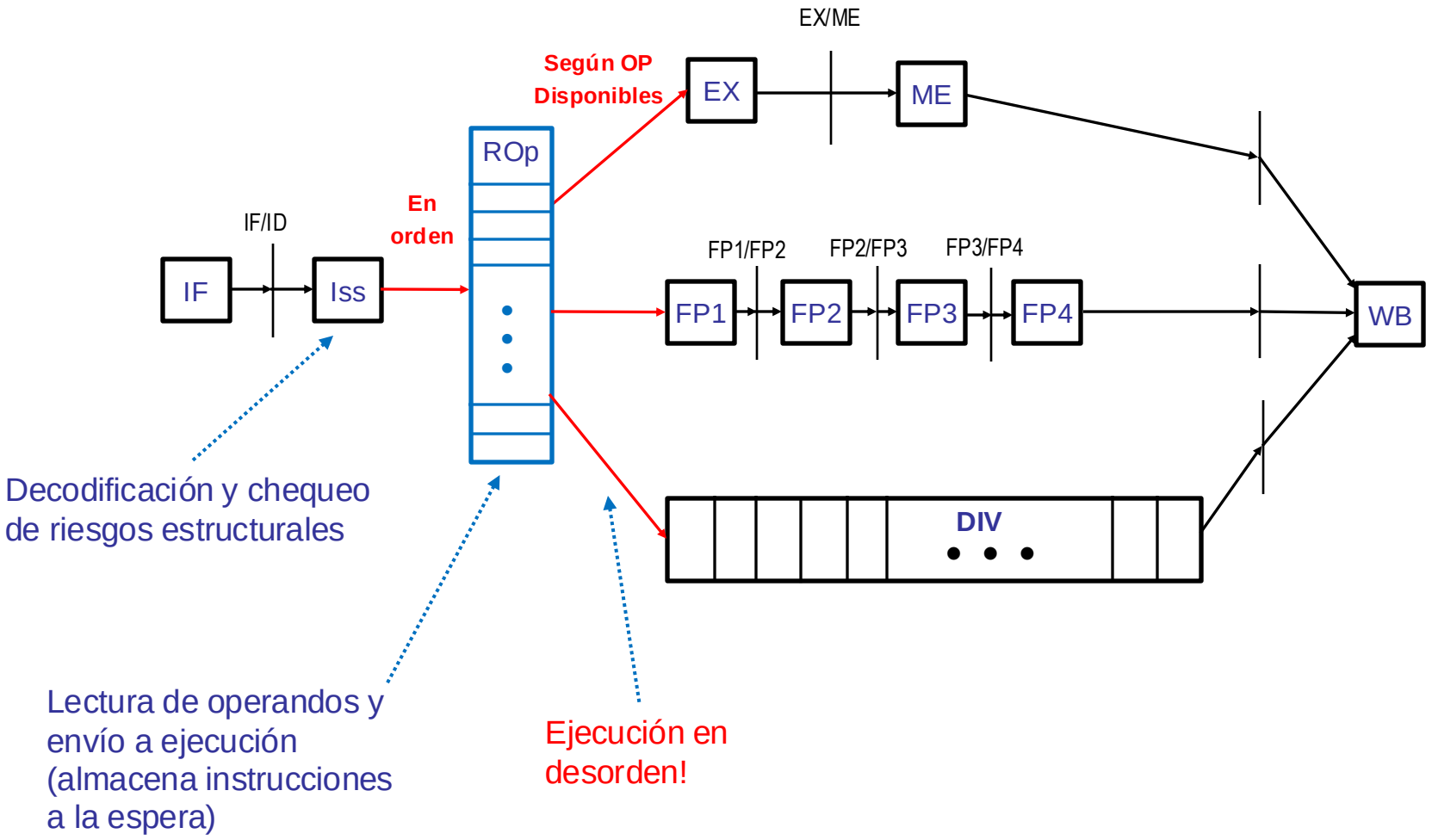
- **Solución** : Dividir la etapa ID en dos etapas diferenciadas.

Issue: Decodifica y chequea riesgos estructurales.
Lectura de operandos: Chequea disponibilidad de operandos. Debe implementarse para permitir el flujo de instrucciones.

Ejecución fuera de orden ➔ ¿Finalización fuera de orden?

Planificación dinámica: Ventajas...Problemas...Soluciones

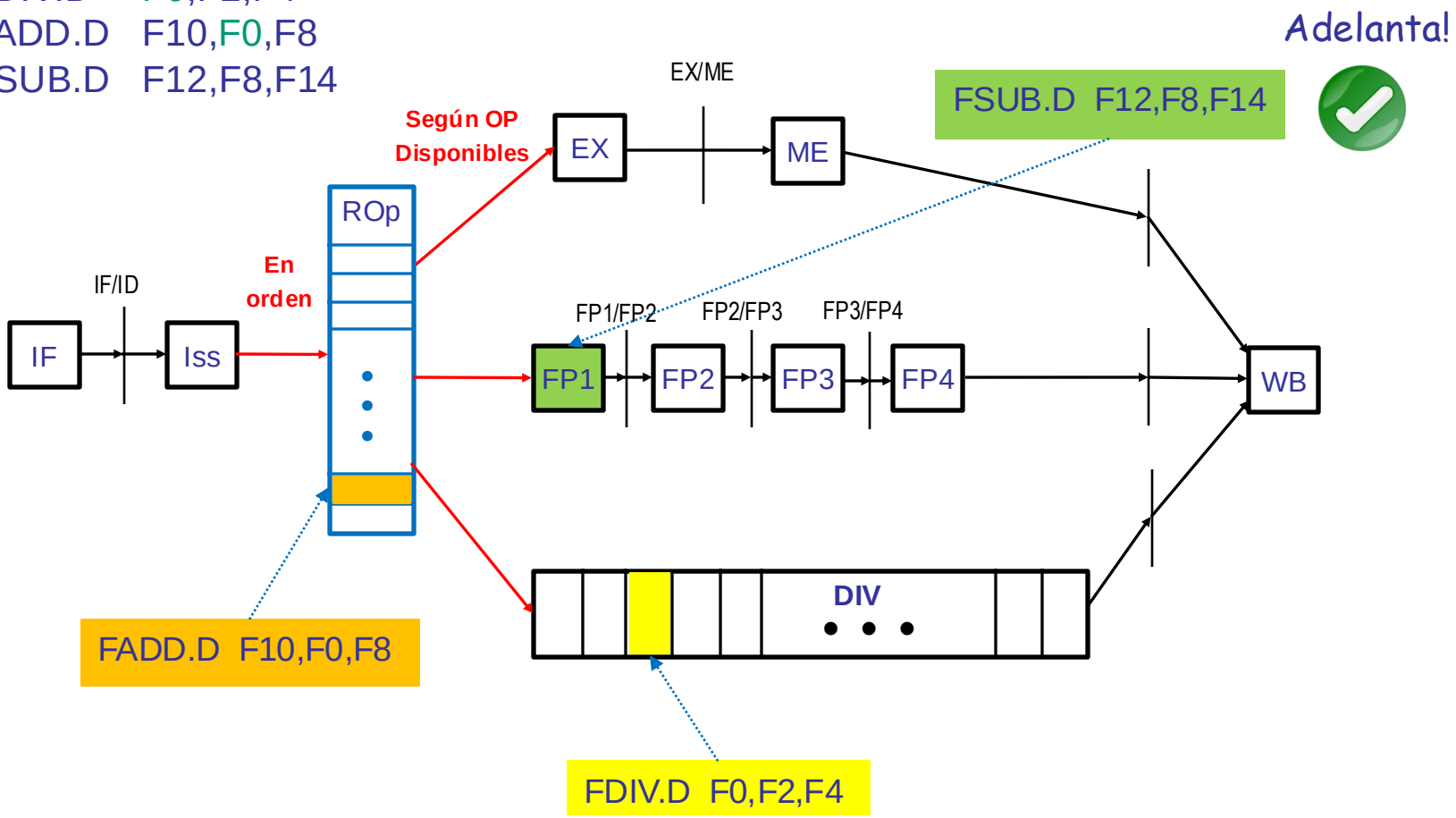
- Etapas Issue (Iss) y Lectura de Operandos (ROp) separadas



Planificación dinámica: Ventajas...Problemas...Soluciones

- Ejecución de la secuencia

FDIV.D F0,F2,F4
FADD.D F10,F0,F8
FSUB.D F12,F8,F14

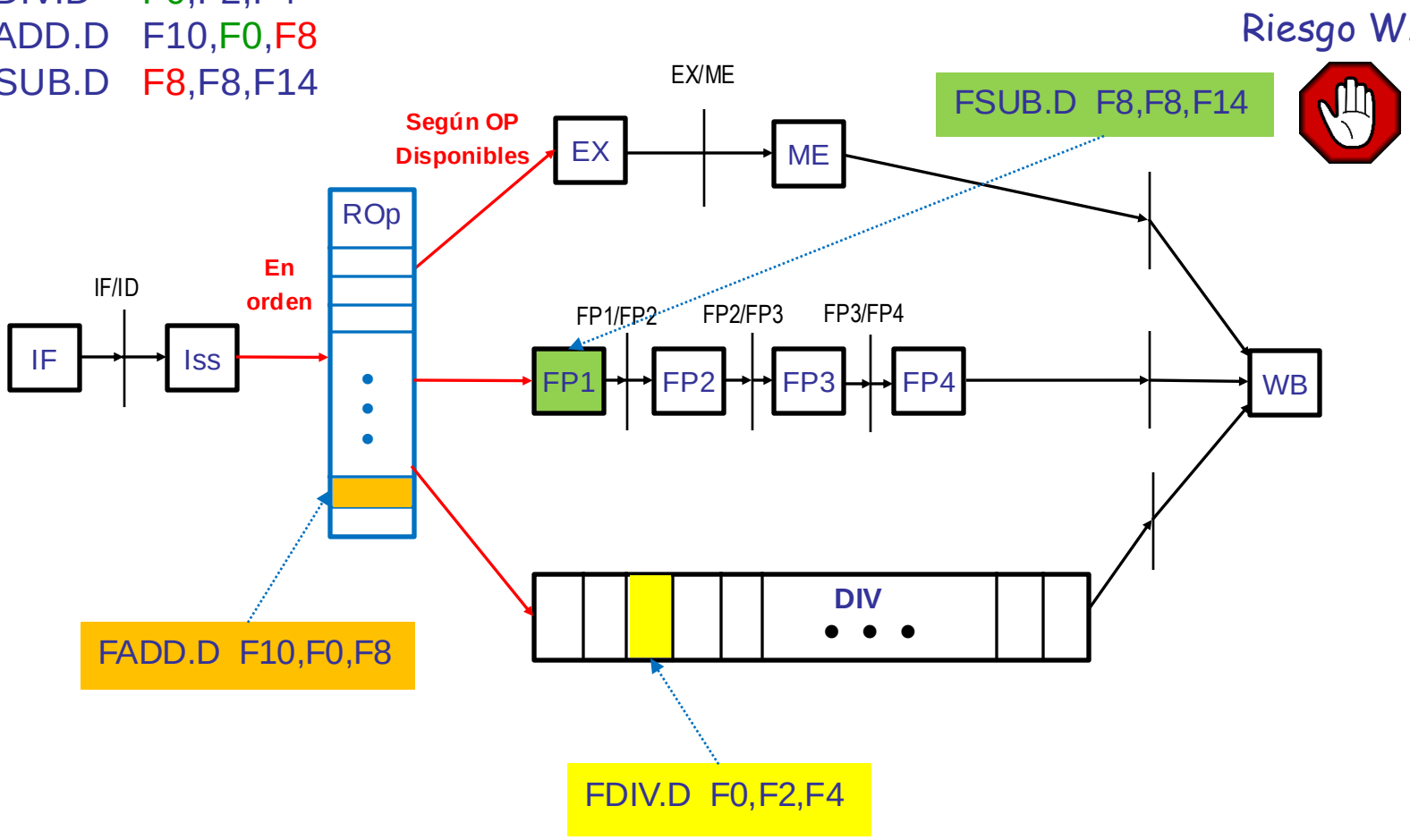


FSUB.D acabará antes que FDIV.D y que FADD.D: Se evita el bloqueo.

Planificación dinámica: Ventajas...Problemas...Soluciones

- Y si la secuencia es ...

FDIV.D F0,F2,F4
FADD.D F10,F0,F8
FSUB.D F8,F8,F14



Si FSUB.D acaba antes de que FADD.D lea sus operandos, destruirá el valor de F8!

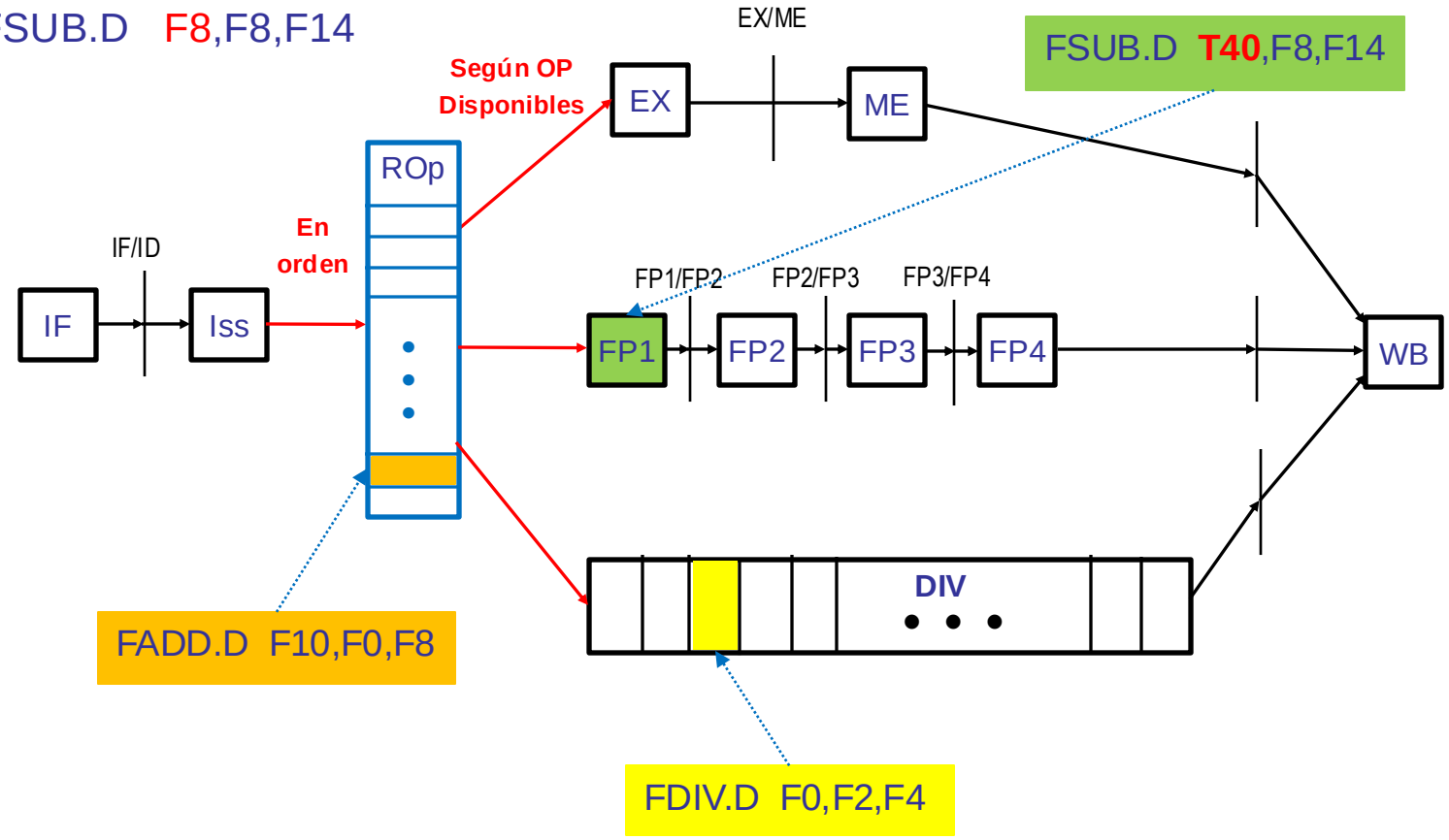
Planificación dinámica: Ventajas...Problemas...Soluciones

- Y si la secuencia es ...

```
FDIV.D   F0,F2,F4
FADD.D   F10,F0,F8
FSUB.D   F8,F8,F14
```

Podría el Hw suplantar el reg destino de SUB.D?

IDEA: Renombramiento Dinámico



Las instrucciones posteriores al FSUB.D que tengan F8 como operando, por supuesto deben usar el valor almacenado en T40! (¿Hasta cuándo?)

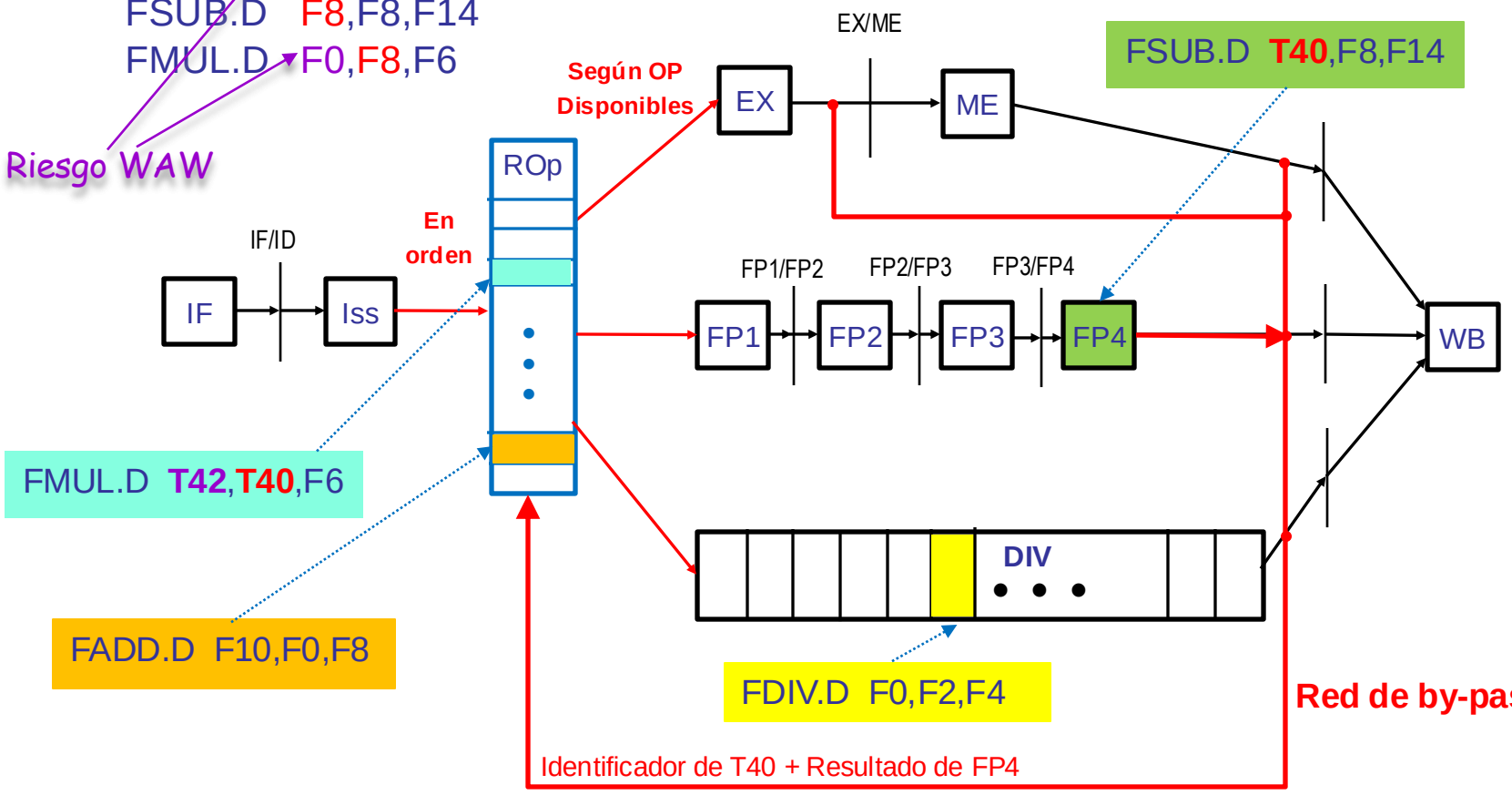
Planificación dinámica: Ventajas...Problemas...Soluciones

- Y si la secuencia es ...

FDIV.D F0,F2,F4
FADD.D F10,F0,F8
FSUB.D F8,F8,F14
FMUL.D F0,F8,F6

Riesgo WAW

Podrían las instrucciones en Etapa ROp leer los operandos en cuanto estén calculados?
IDEA: By-pass de operandos

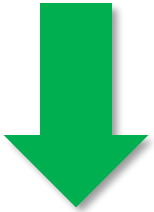


Las instrucciones en la etapa ROp pueden obtener sus operandos a través de la red de by-pass. En el ejemplo, FMUL.D está a la espera del valor de T40.

Una instrucción posterior a FMUL.D que referencie F0, leerá el resultado de FMUL.D, incluso aunque FDIV.D acabe más tarde.

Planificación dinámica: Ventajas...Problemas...Soluciones

- Múltiples alternativas para implementar las ideas esbozadas en las anteriores transparencias.
- Esfuerzos desde los años 60
  - Scoreboard (CDC 6600):
    - Limita el efecto de dependencias RAW (no bloqueo en lanzamiento)
  - Algoritmo de Tomasulo
    - También limita el efecto de dependencias RAW
    - Además, introduce el concepto de "renombramiento dinámico de registros" ➔ elimina el efecto de dependencias WAR y WAW
    - Complejidad hardware considerable para la época: éxito modesto



Evolución: Ley de Moore

    - Idea implementada (con diversos matices) en **casi todos** los μ-procesadores modernos

Especulación

- La predicción de saltos introduce ESPECULACIÓN
  - Dos tipos de instrucciones en el procesador
    - Las independientes
    - Las que dependen de una predicción de salto. Su finalización depende del acierto o fallo en la predicción.

¿Cómo podemos implementar esta distinción con un modelo de ejecución con finalización Fuera de orden?

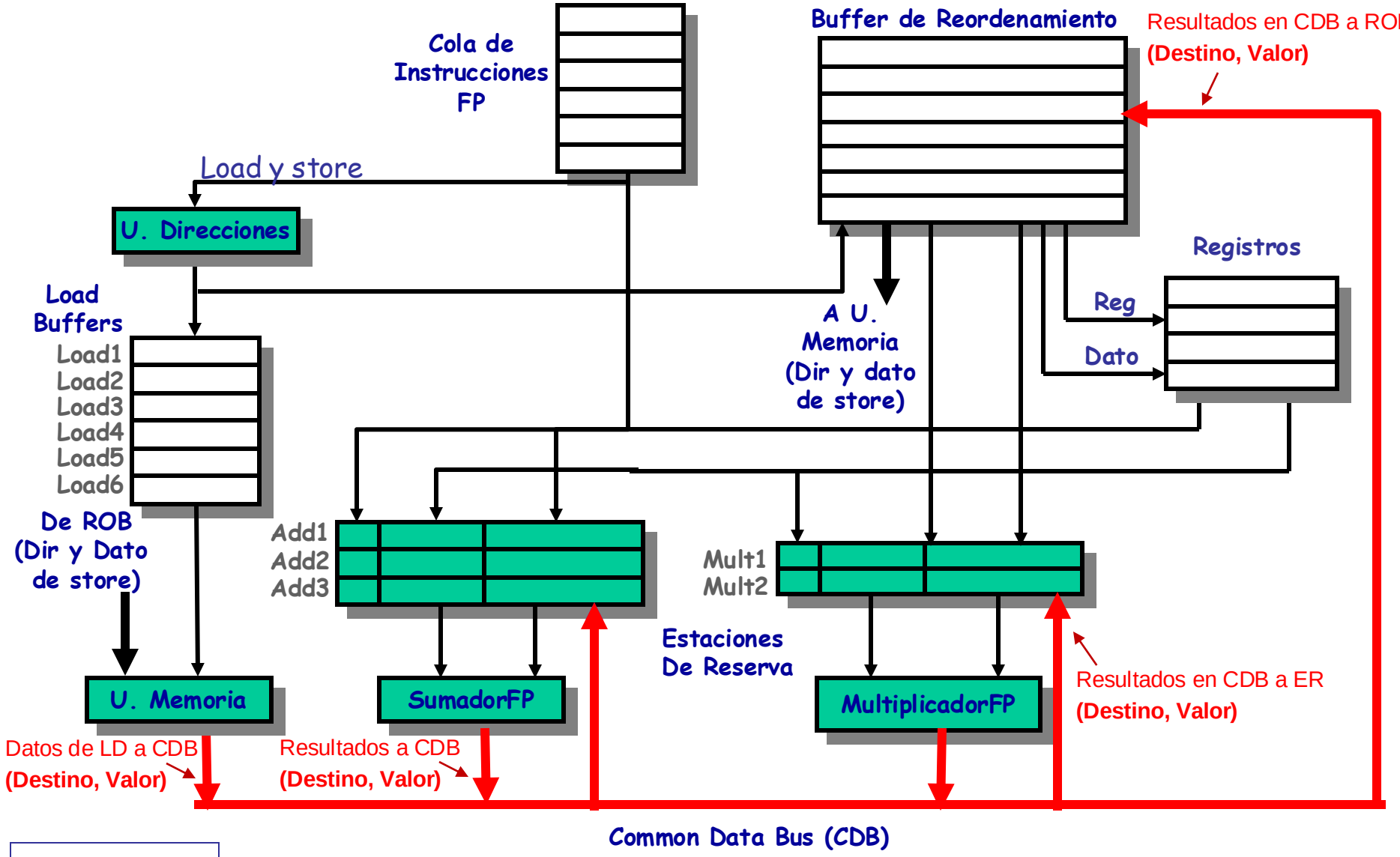
Forzando finalización en orden mediante el

ALGORITMO DE TOMASULO CON ESPECULACIÓN

- La etapa WB no almacena resultados en registros
  - Los resultados se guardan temporalmente en un buffer (ROB)
- Se añade una etapa de finalización (commit): se escriben resultados en reg. o memoria
- Las instrucciones hacen la etapa de finalización en orden
- Una instr. que depende de un salto sin resolver no puede hacer la etapa de finalización

Especulación

- Algoritmo de TOMASULO con especulación



Planificación Dinámica : Tomasulo

Componentes de las Estaciones de Reserva (ER)

- Op: Operación a realizar
- Valorj, Valork: Valores de los operandos fuente (si están calculados)
- Qj, Qk: Nº del ROB que está produciendo los operandos fuente. Notar: Qj,Qk=0 ⇒ operando calculado.
- Busy: Indica ER ocupada
- Destino: Número de entrada del ROB al que va el resultado

Registros FP

- Vi: Valor almacenado en el registro (puede estar obsoleto)
- Qi: Nº del ROB que está produciendo el valor a almacenar en el registro. Si Qi=0 => no hay ningún ROB que vaya a almacenar un nuevo valor en el reg (el valor del registro no está obsoleto)

Load buffers

- Valor: Valor leído en la memoria (si ya se ha completado la lectura).
- Dir: Dirección de lectura
- Destino: Número de entrada del ROB al que va el resultado

Especulación

- El Buffer de Reordenamiento (ROB)
  - Almacena resultados de instrucciones cuya ejecución ha finalizado, pero…
    - están a la espera de actualizar registros o memoria (finalización en orden)
    - son dependientes de un salto (ejecución especulativa)
  - Permite el paso de operandos entre instrucciones especuladas con dependencia LDE.
  - Puede implementarse como cola circular con dos punteros
    - 1: instrucción más antigua; 2: primer hueco libre

- Los operandos de una instrucción pueden llegar hasta la ER desde:
  - CDB (la instrucción que genera el operando todavía no ha realizado la fase de escritura)
  - ROB (la instrucción que genera el operando se ha ejecutado, pero no ha actualizado el banco de registros)
  - Registros (la instrucción que genera el operando ha finalizado completamente)

Especulación

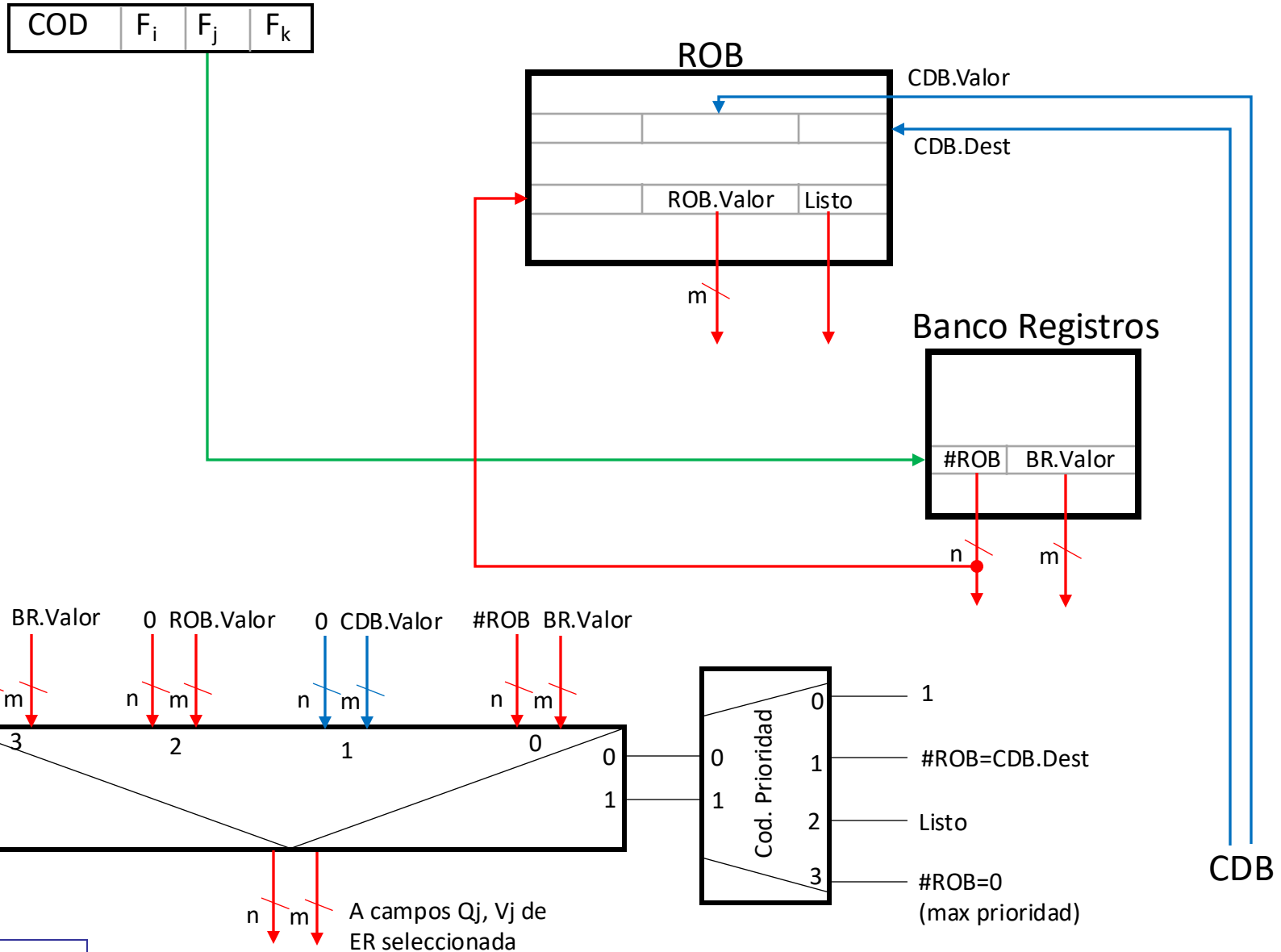
- Estructura del ROB: cada entrada contiene 4 campos
  - Tipo de instrucción
    - Salto (sin reg destino), Store (destino en memoria), Aritmética/Load (con destino en registro)
  - Destino
    - Número de registro (Aritmética/Load)
    - Dirección de memoria (Store)
  - Valor
    - Resultado de la ejecución de la instrucción. Guarda el valor hasta que se actualiza registro destino o memoria.
      - En Store: Mientras el valor no está disponible, guarda el nº de la entrada del ROB donde está la instrucción que lo producirá.
  - Listo
    - La instrucción ha completado la fase de ejecución y el resultado está disponible en el campo "Valor"

Especulación: fases

- Algoritmo de TOMASULO con especulación
  - Las 4 fases del algoritmo de Tomasulo especulativo:
    - **Issue**: Toma la instrucción de la cola
      - Es necesario ER con entrada libre y Buffer de Reordenamiento (ROB) con entrada libre
      - Toma operandos de registros o de resultados almacenados en ROB por instrucciones previas (ver transparencia siguiente)
      - Marca Reg. Destino con nº de entrada del ROB asignada
      - Marca ER asignada con nº de entrada del ROB asignada
    - **Ejecución**: Opera sobre los operandos
      - Espera hasta que los operandos estén disponibles. Chequea CDB.
    - **Escritura de resultados**: Finaliza ejecución
      - Escribe a través de CDB en todas las ER de Fus y entradas del ROB que estén a la espera del resultado
      - Libera ER
      - No escribe en registros ni en memoria
      - Envía por CDB resultado + nº de entrada del ROB a la que se dirige
    - **Commit**: Actualiza registros desde el ROB
      - Cuando la Instrucción está en la cabecera del ROB y resultado presente: Actualiza Registro (o escribe en memoria) y elimina la instrucción del ROB

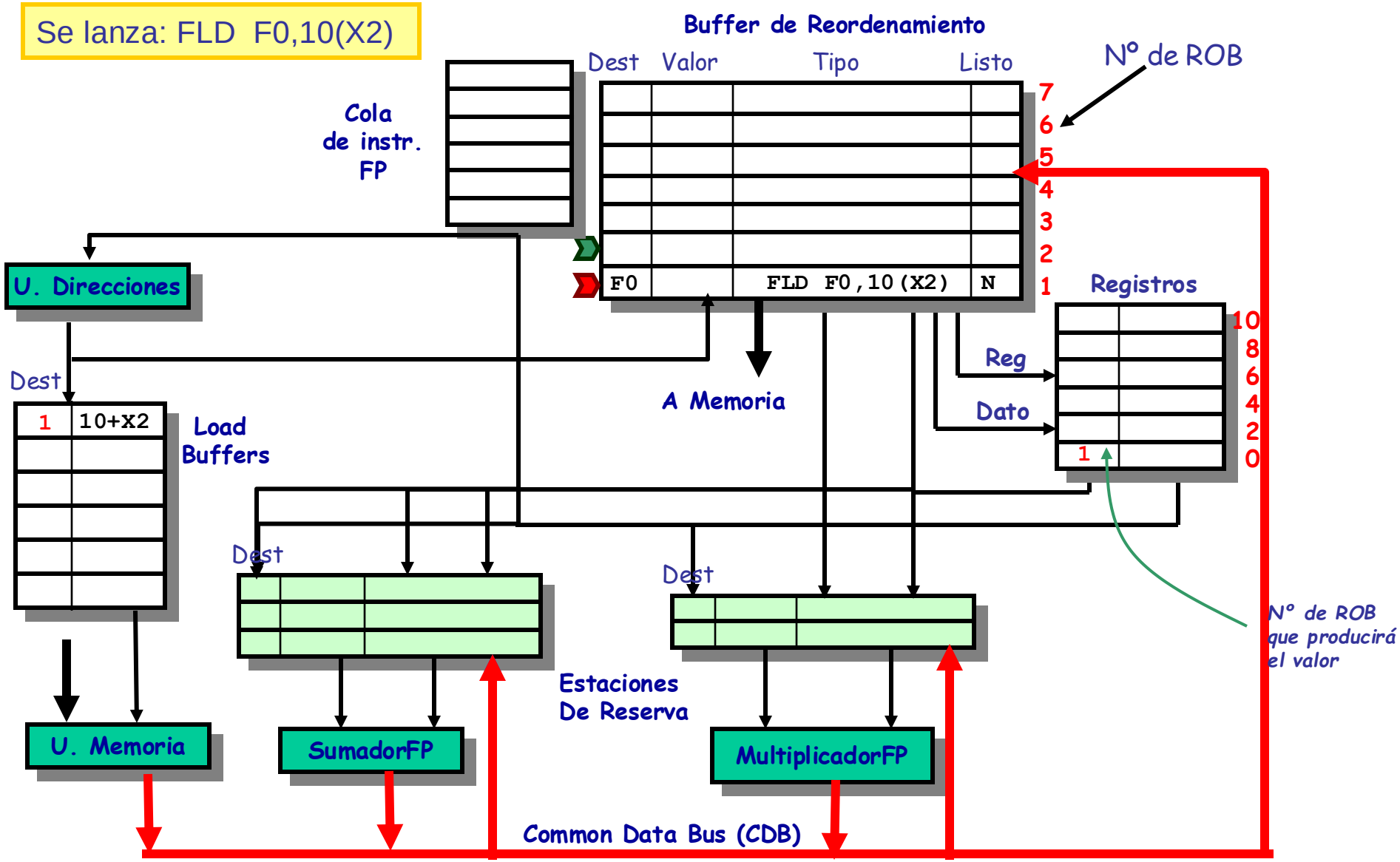
Especulación: fases

Gestión de operandos en la fase "issue" de una instrucción: operando F_j (similar para F_k)



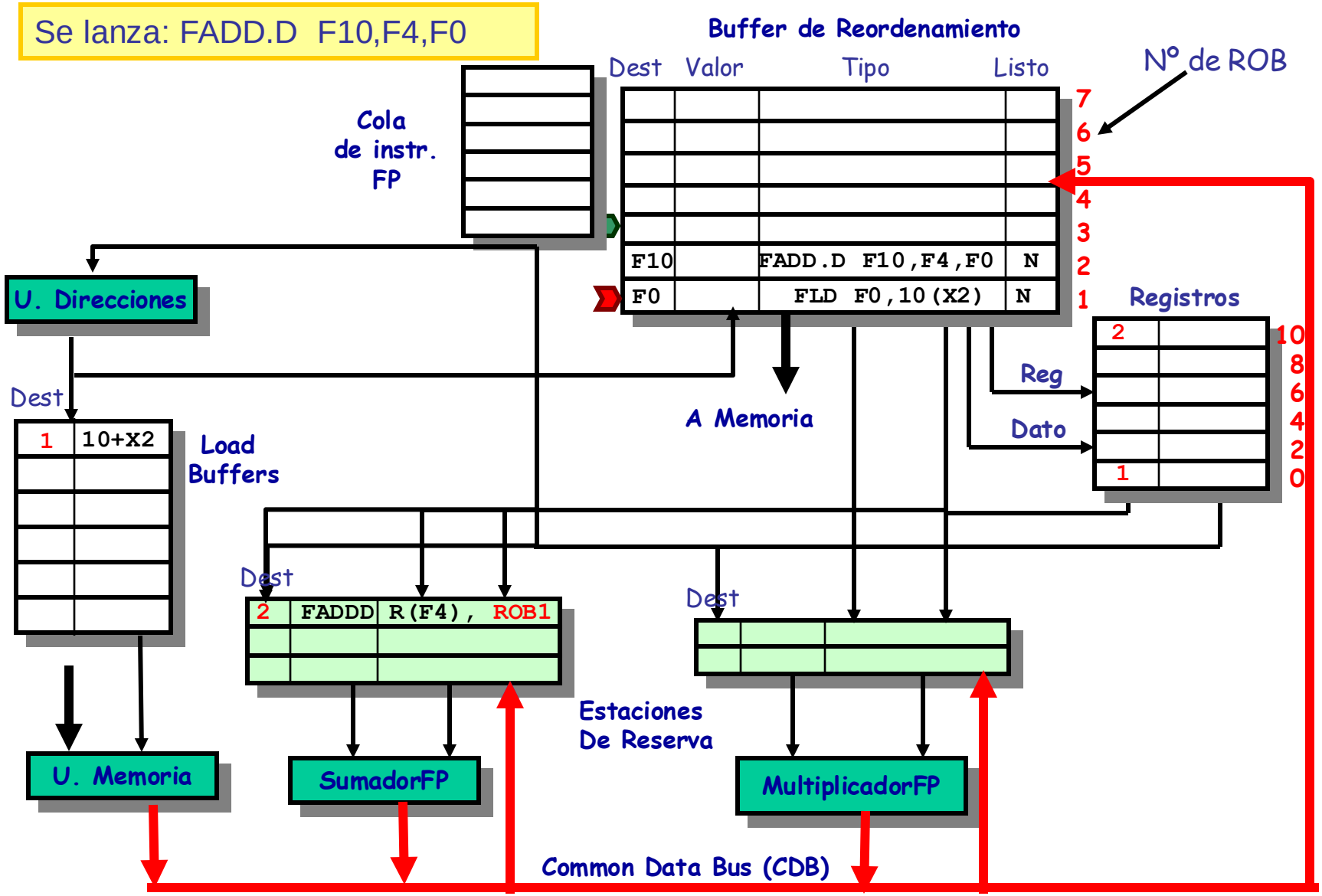
Especulación: Ejemplo

Se lanza: FLD F0,10(X2)



Especulación: Ejemplo

Se lanza: FADD.D F10,F4,F0

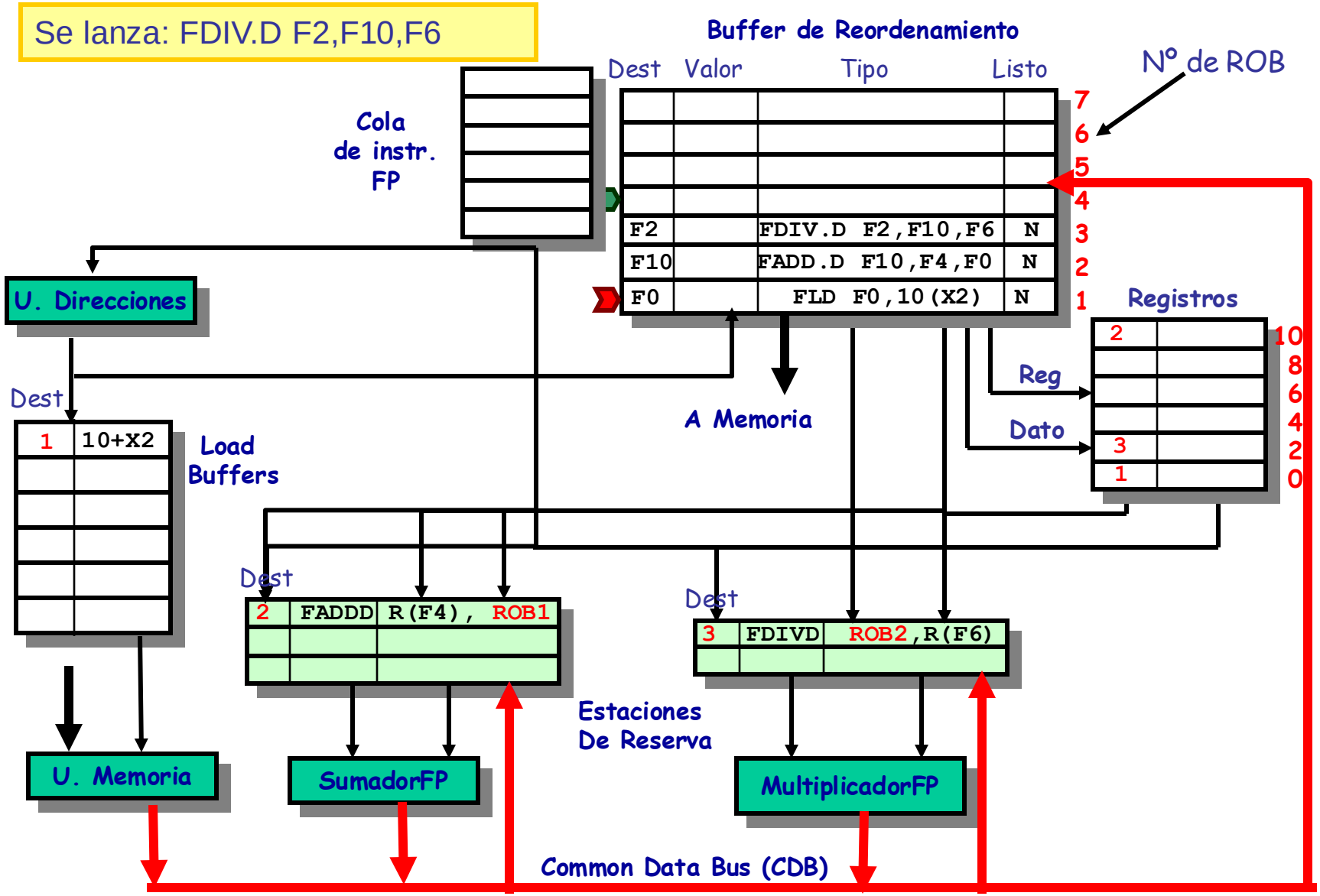


El reg F4 no tiene marca ➔ copiar valor en ER

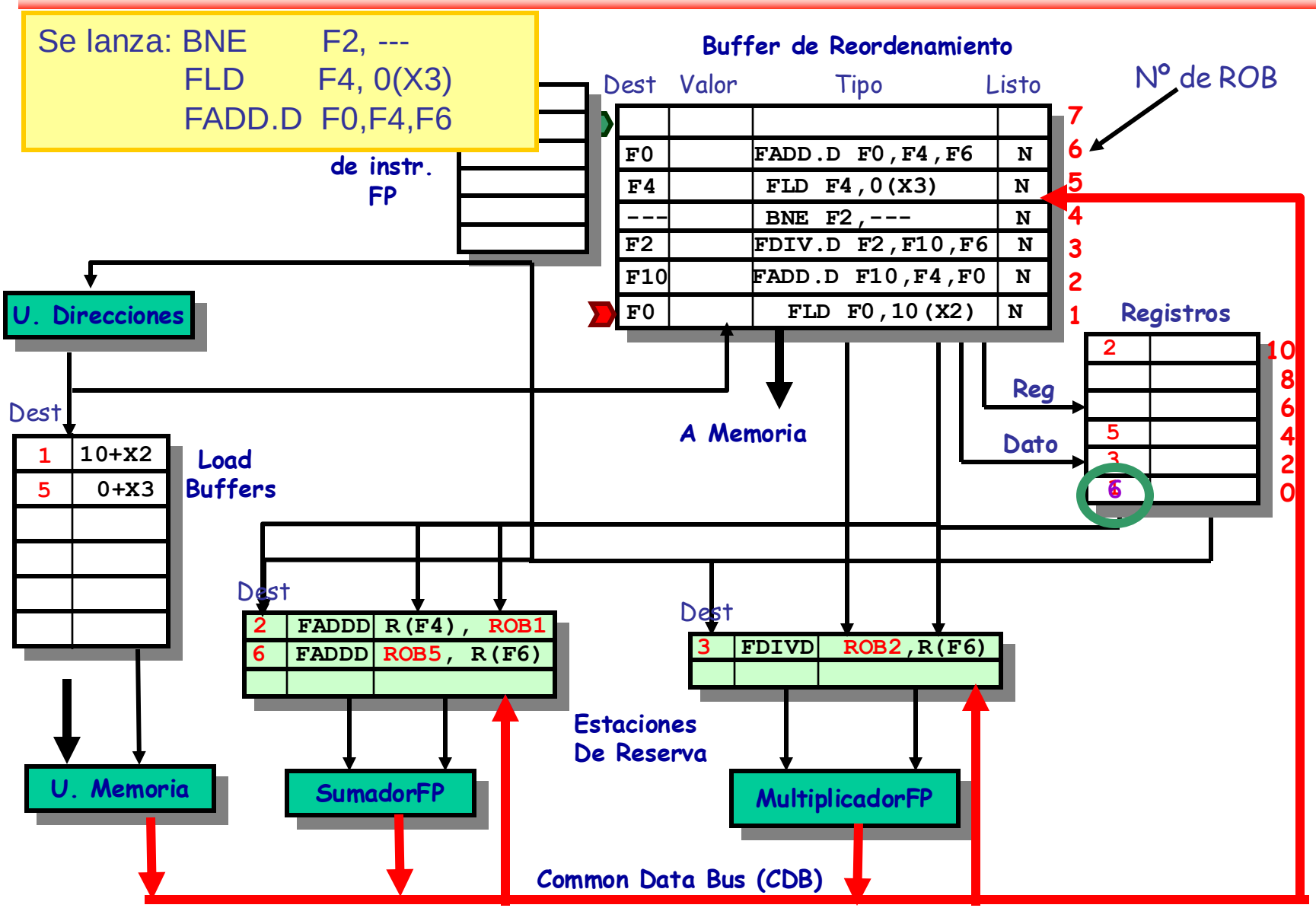
El reg F0 tiene marca (1) y campo "Listo" de ROB1=N ➔ copiar marca en ER

Especulación: Ejemplo

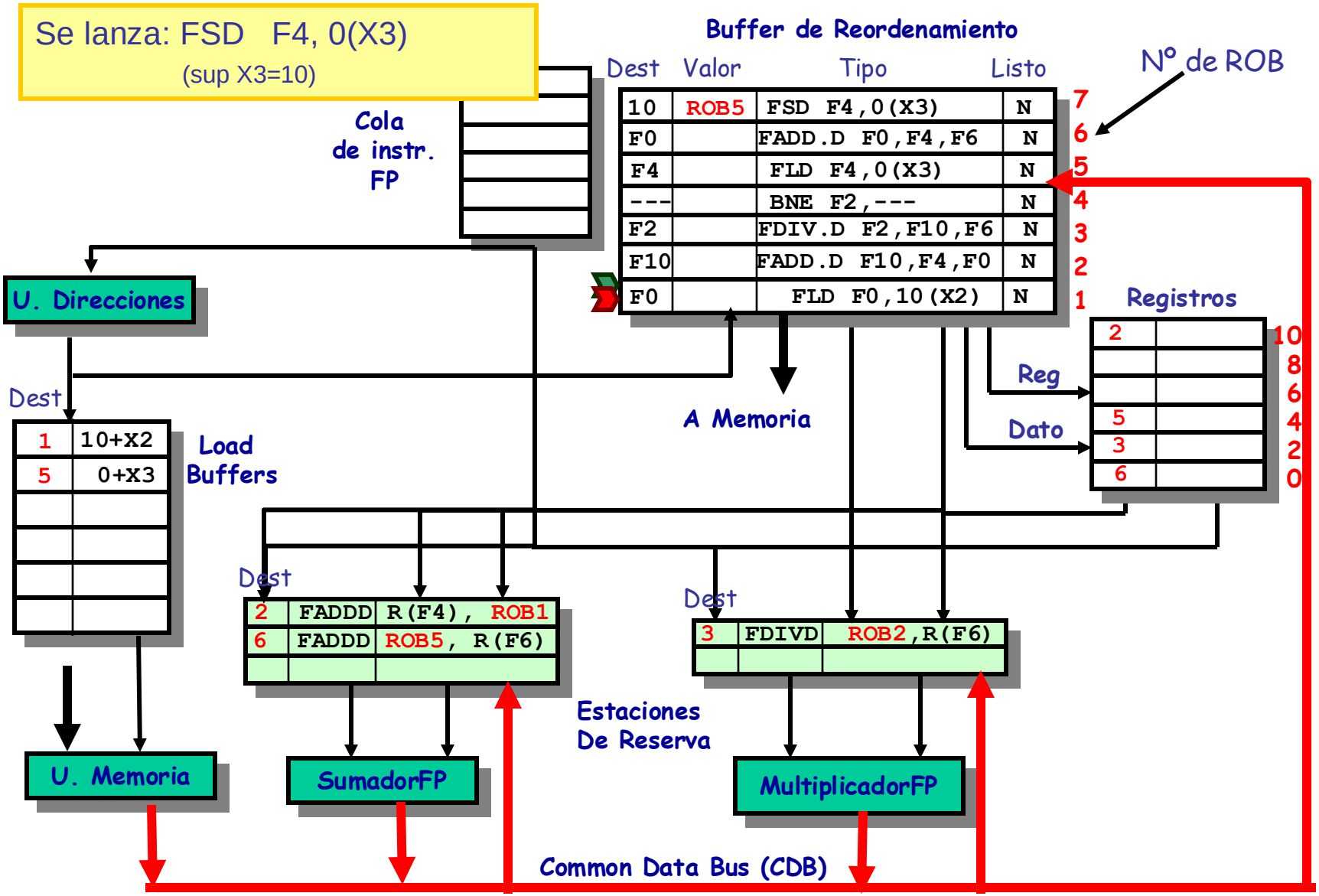
Se lanza: FDIV.D F2,F10,F6



Especulación: Ejemplo

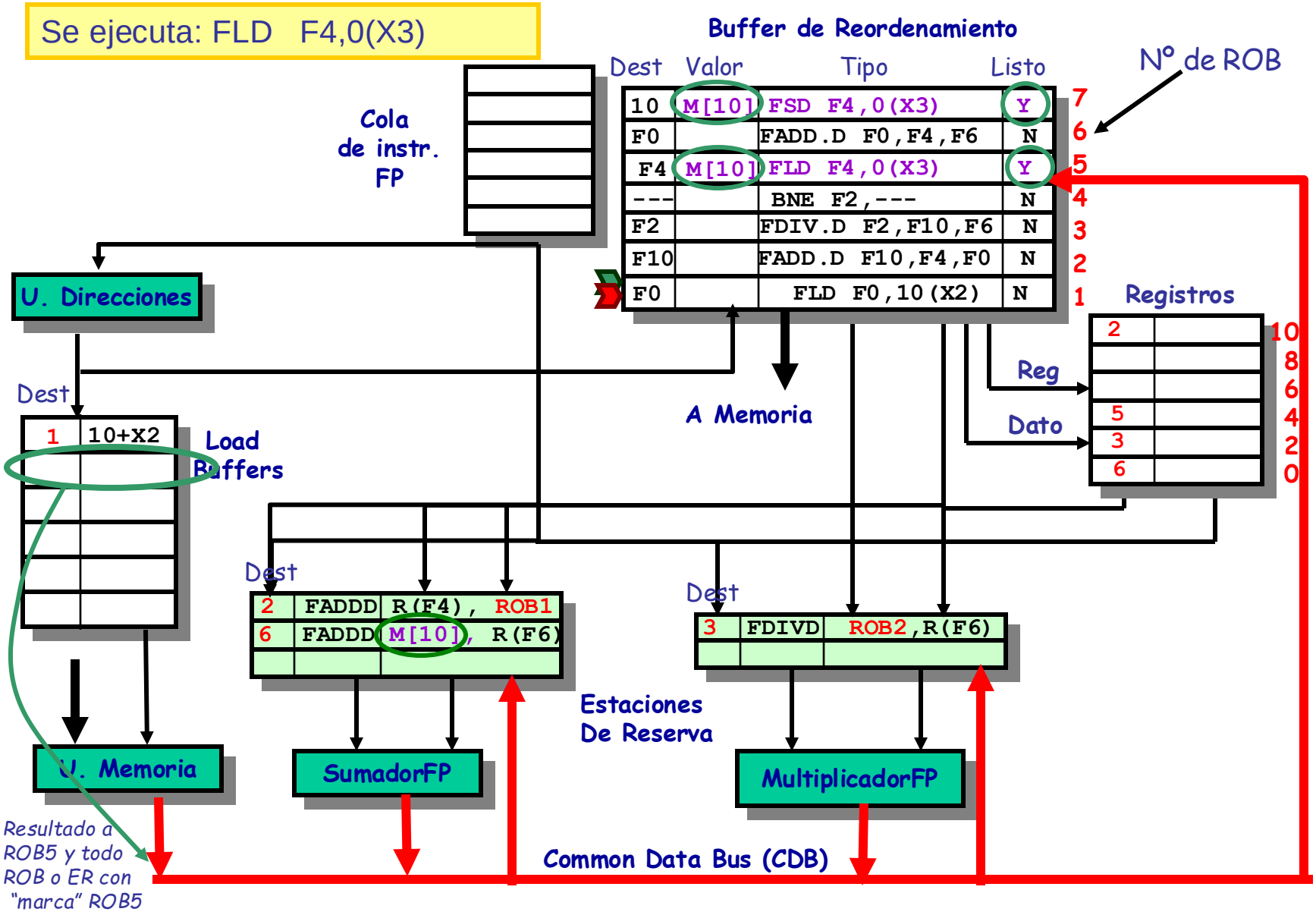


Especulación: Ejemplo

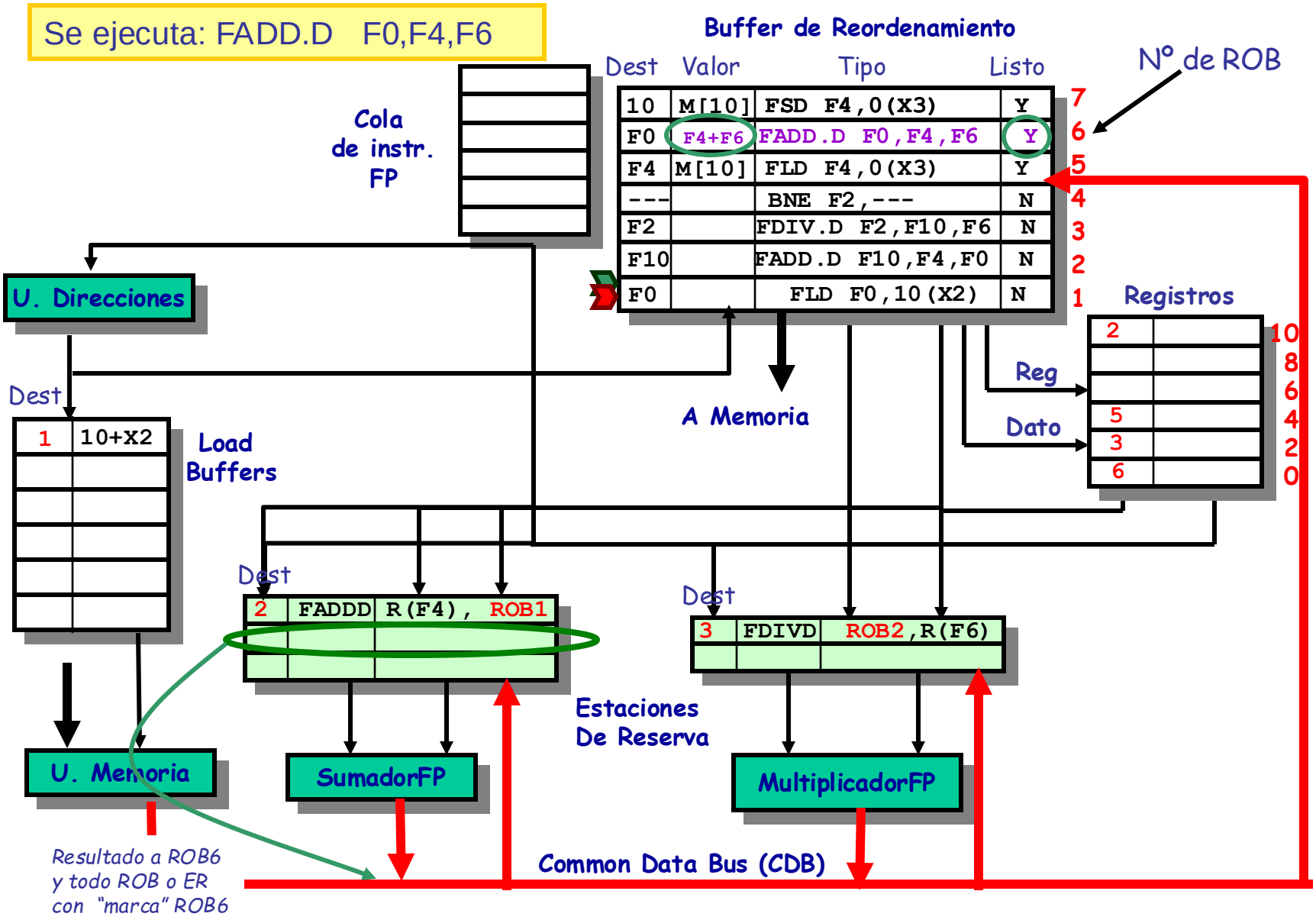


Especulación: Ejemplo

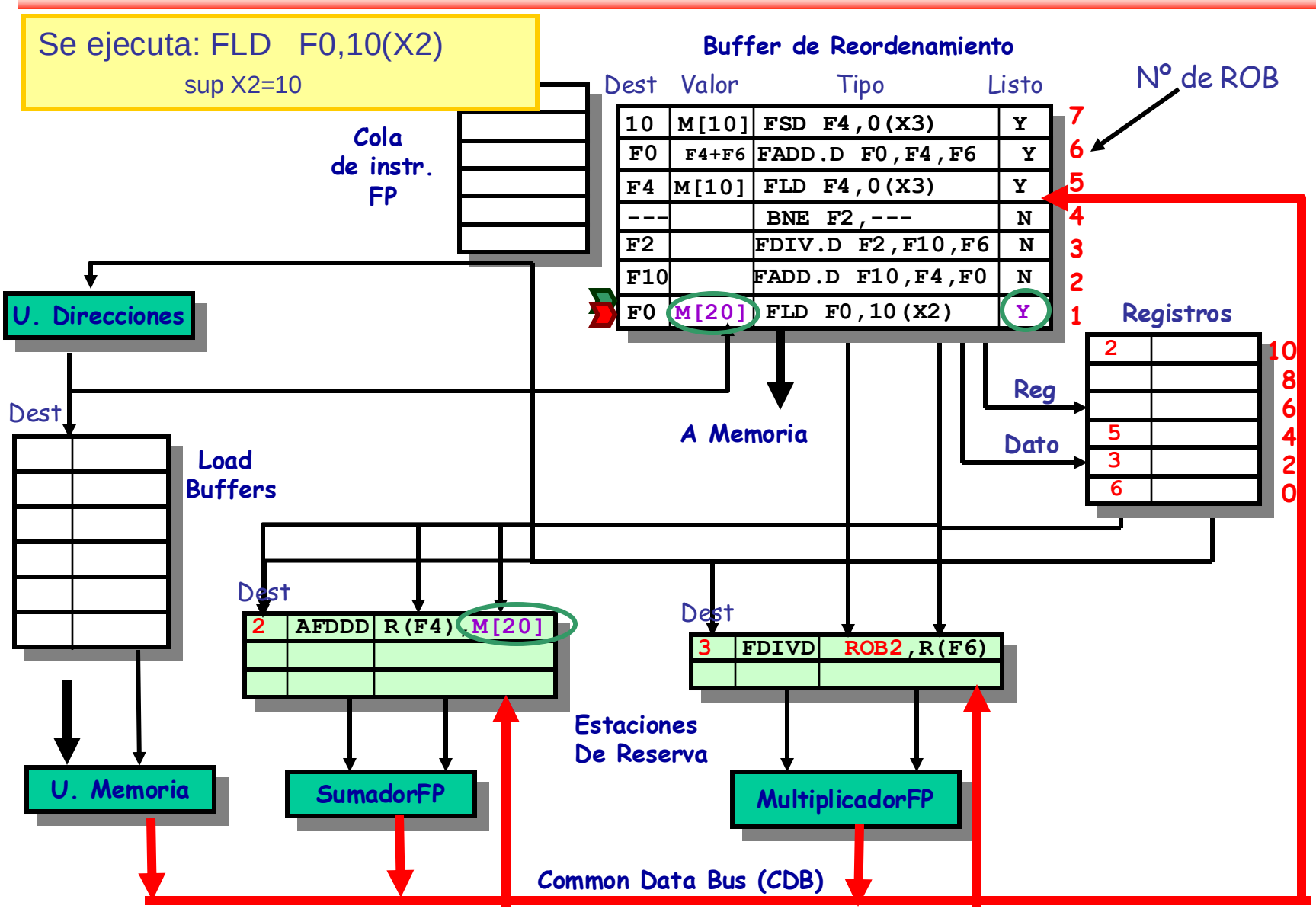
Se ejecuta: FLD F4,0(X3)



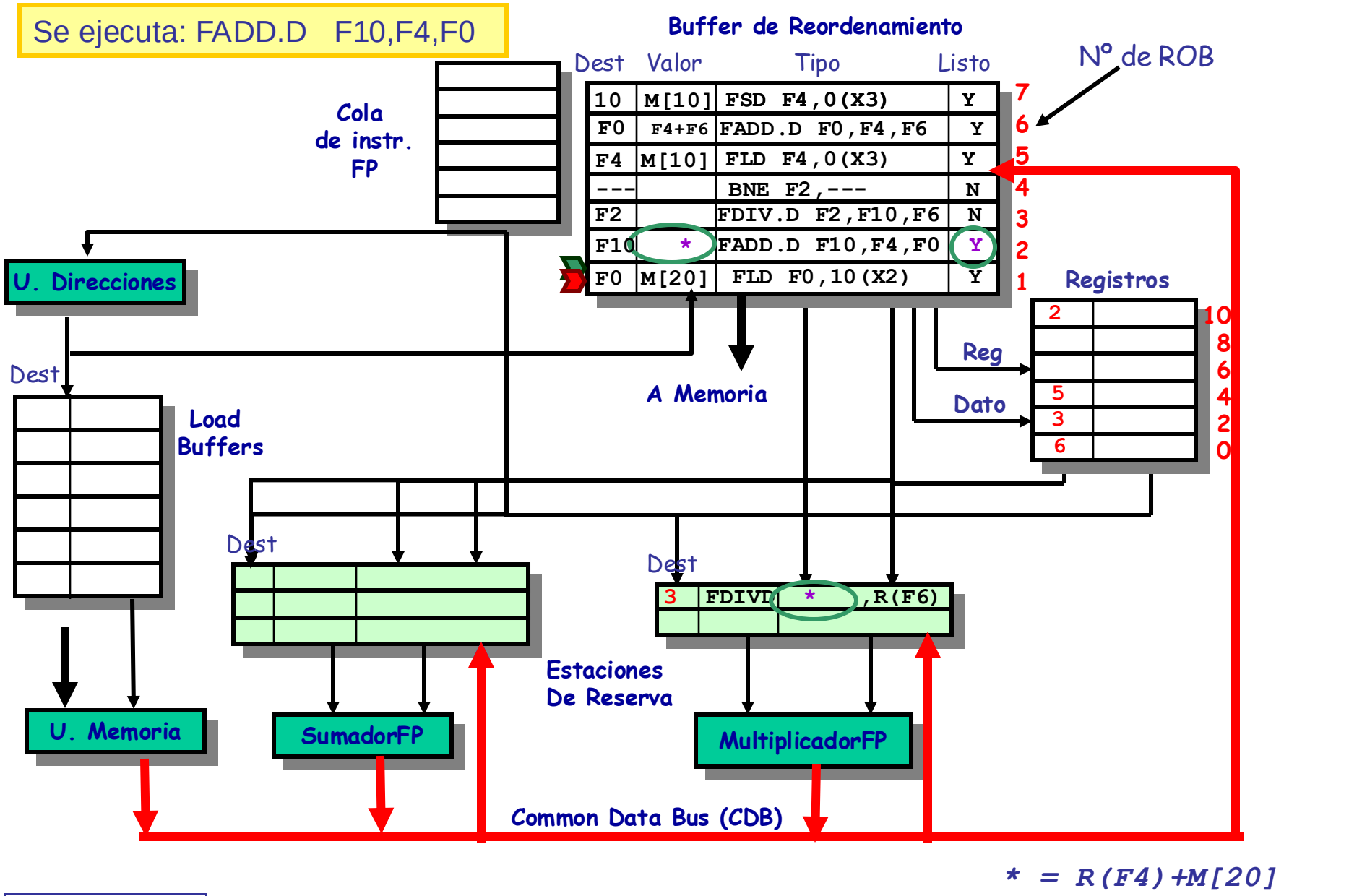
Especulación: Ejemplo



Especulación: Ejemplo

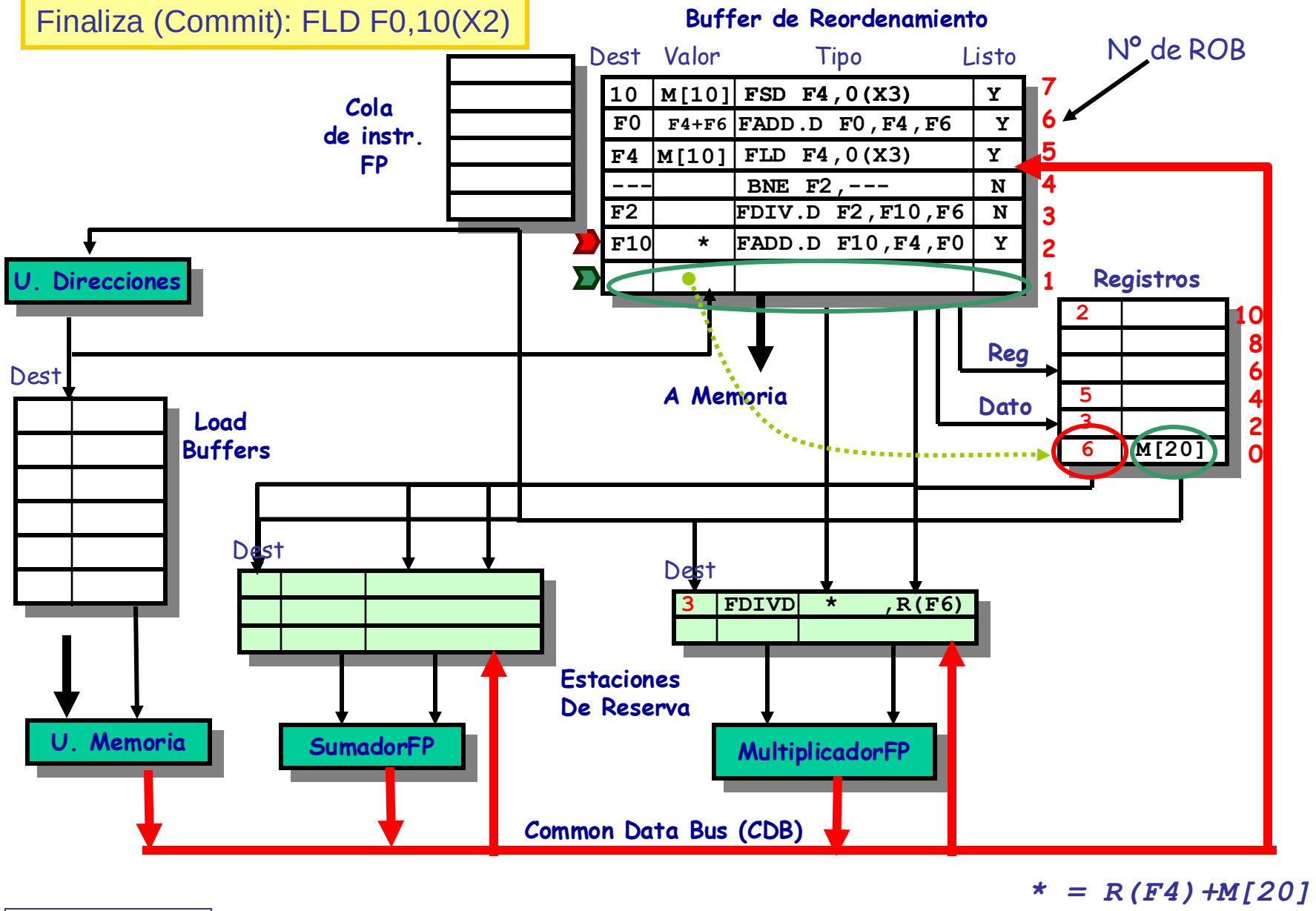


Especulación: Ejemplo



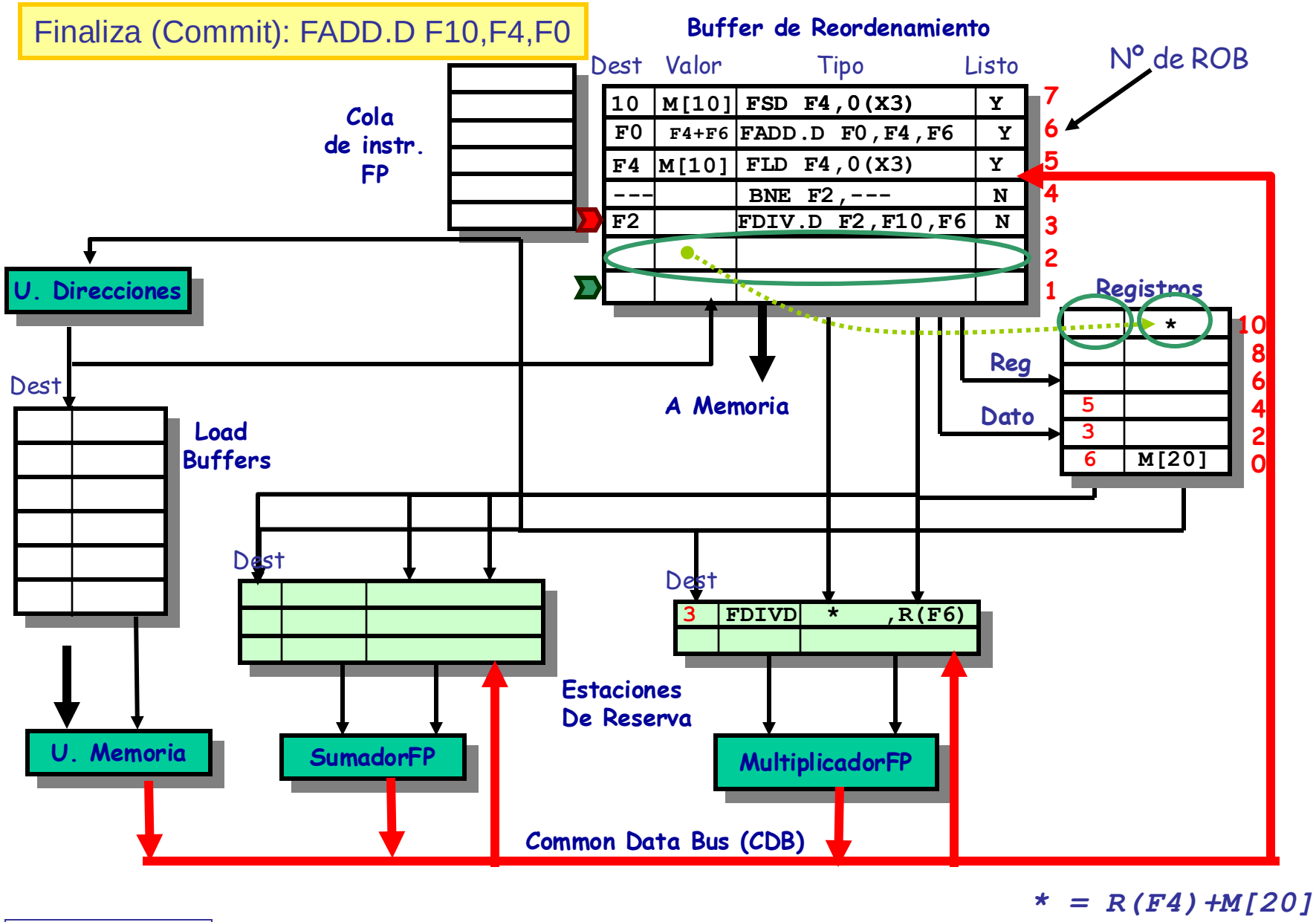
Especulación: Ejemplo

Finaliza (Commit): FLD F0,10(X2)



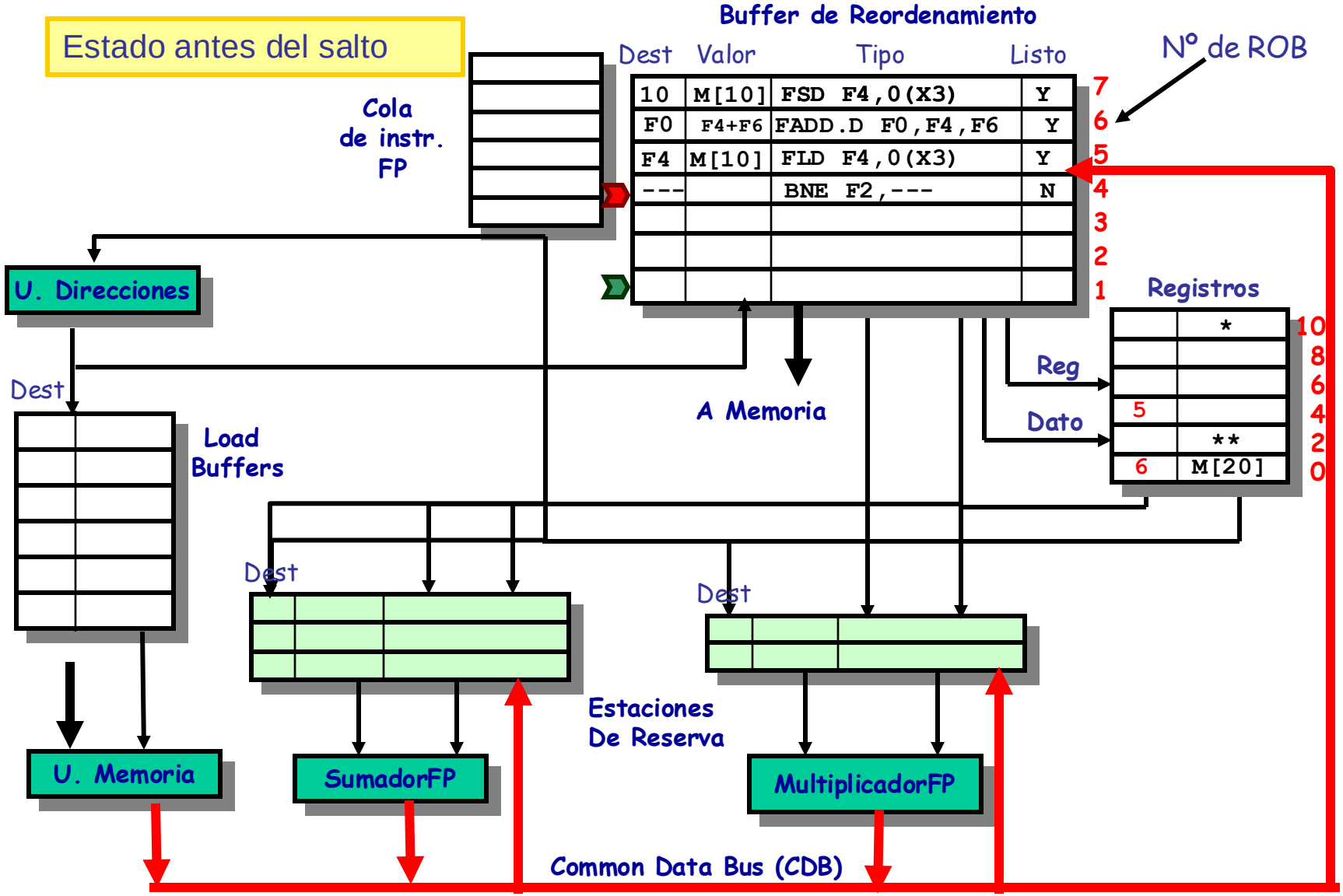
Especulación: Ejemplo

Finaliza (Commit): FADD.D F10,F4,F0



* = R(F4)+M[20]

Especulación: Ejemplo

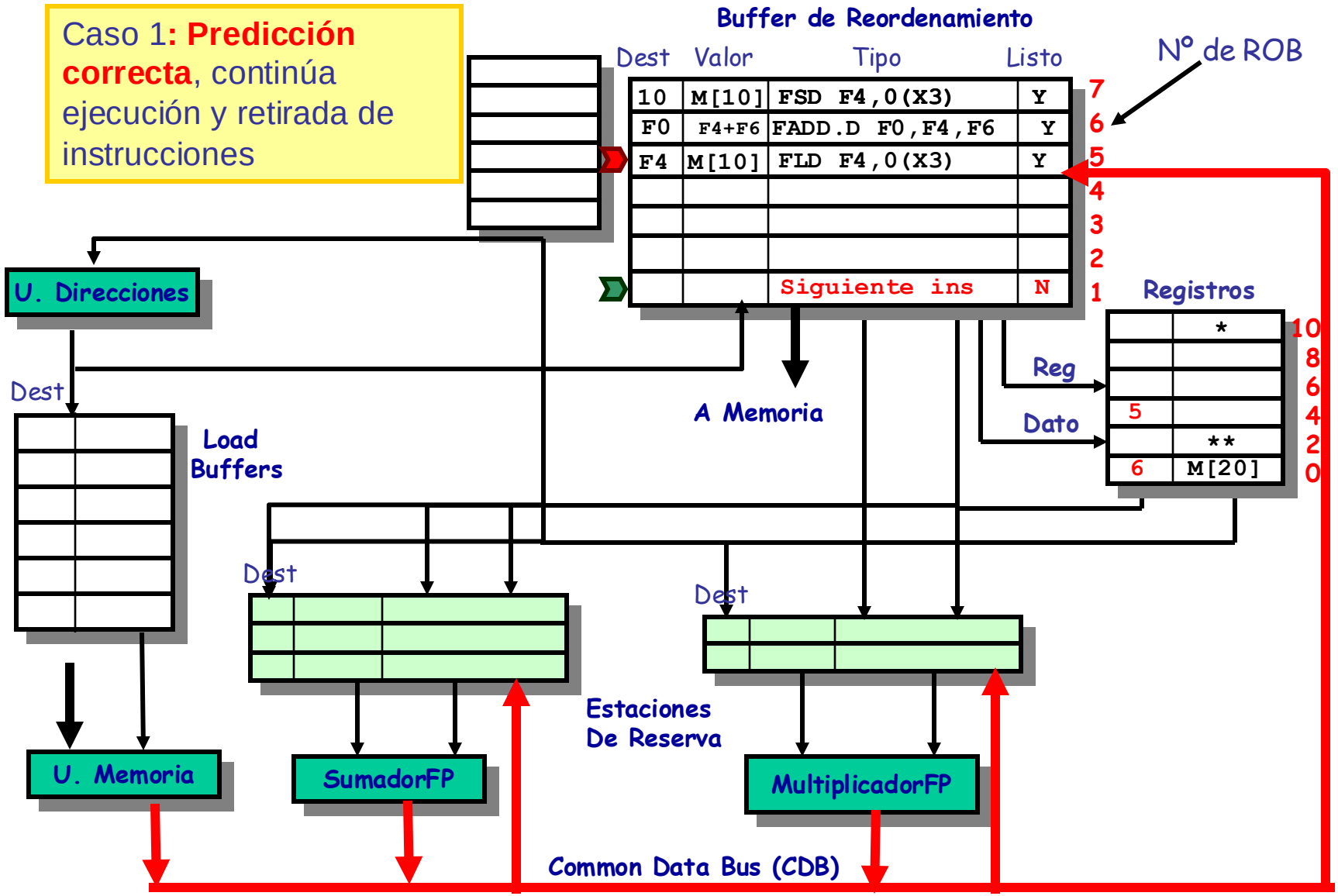


* = R(F4)+M[20]

** = (R(F4) + M[20])/ [F6]

Especulación: Ejemplo

Caso 1: **Predicción correcta**, continúa ejecución y retirada de instrucciones

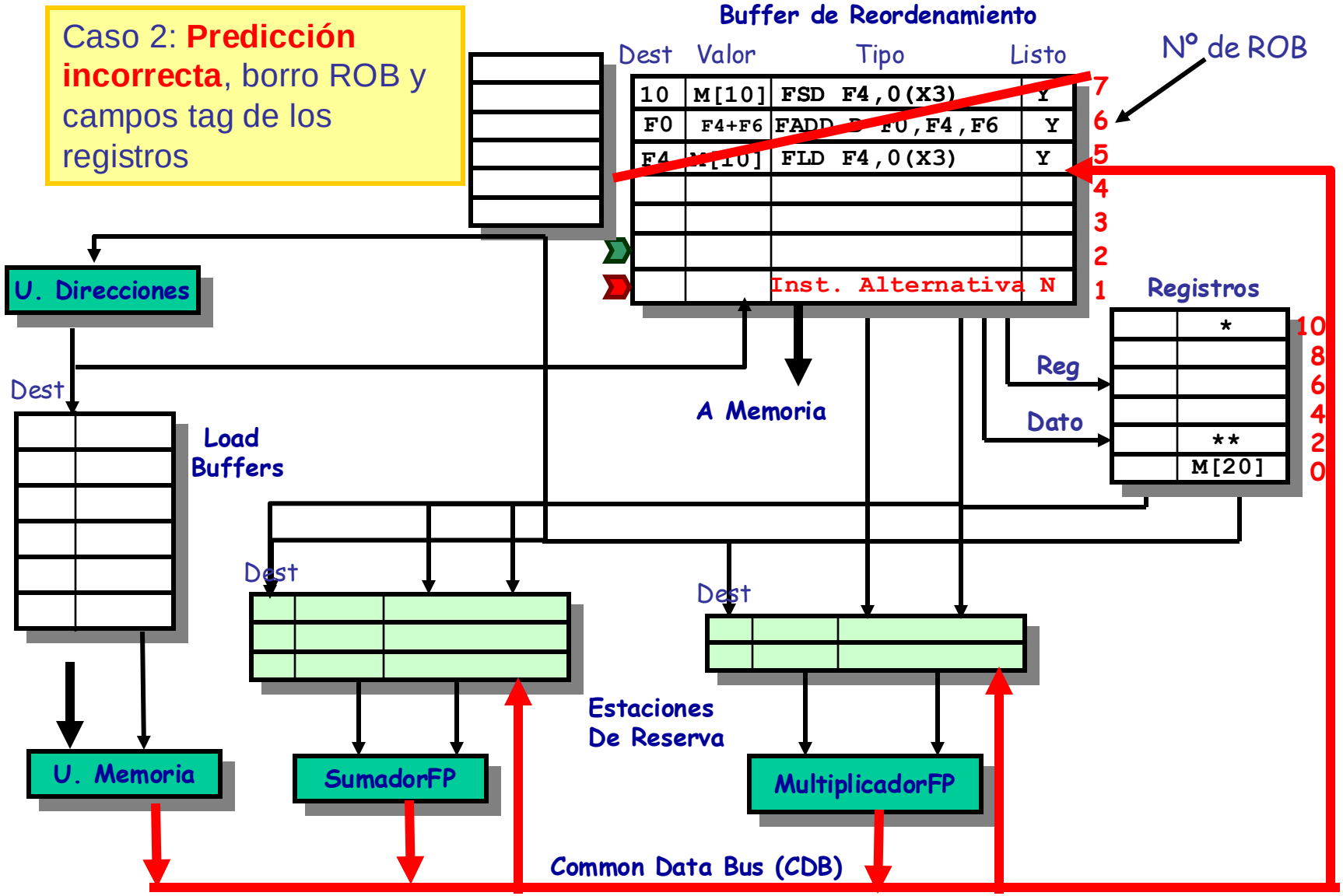


* = R(F4)+M[20]

** = (R(F4) + M[20]) / [F6]

Especulación: Ejemplo

Caso 2: **Predicción incorrecta**, borro ROB y campos tag de los registros



* = R(F4)+M[20]

** = (R(F4) + M[20])/ [F6]

Especulación: Saltos e interrupciones

- El ROB permite recuperarse de saltos mal predichos e implementar un modelo de excepciones precisas
- Si una instrucción de salto bien predicha llega a cabecera de ROB =>
  - Eliminarla de ROB
- Si una instrucción de salto mal predicha llega a cabecera de ROB =>
  - Borrar contenido del ROB, ER y Buffer de Load.
  - Borrar marcas (campo "Nº de ROB)" de todos los registros.
  - Buscar instrucción correcta.
- Si una instrucción genera una interrupción =>
  - Registrar la petición en el ROB
  - Si la instrucción llega a la cabecera del ROB (no especulada), entonces reconocer la interrupción.
  - Cualquier instrucción anterior habrá finalizado. Por tanto, ninguna instrucción anterior puede provocar una excepción.

Especulación: Riesgos a través de memoria

- Riesgos EDE y EDL: no pueden aparecer dado que la actualización de memoria se hace en orden.
  - Esperar hasta que la instrucción ST se halle en la cabecera de ROB => Todos los LD y ST anteriores se han completado.

- Riesgos LDE: Podrían producirse si un LD accede a la posición de memoria A, habiendo en el ROB un ST previo que almacena el resultado en A. Se evitan mediante el siguiente mecanismo:
  - Un LD no ejecuta el acceso a memoria si hay un ST previo en el ROB con la misma dirección de memoria.
  - Tampoco se ejecuta el LD si está pendiente el cálculo de la dirección efectiva de algún ST del ROB