# Tema 3. Segmentación avanzada y predicción de saltos

Departamento de Ingeniería y Tecnología de Computadores

# Índice

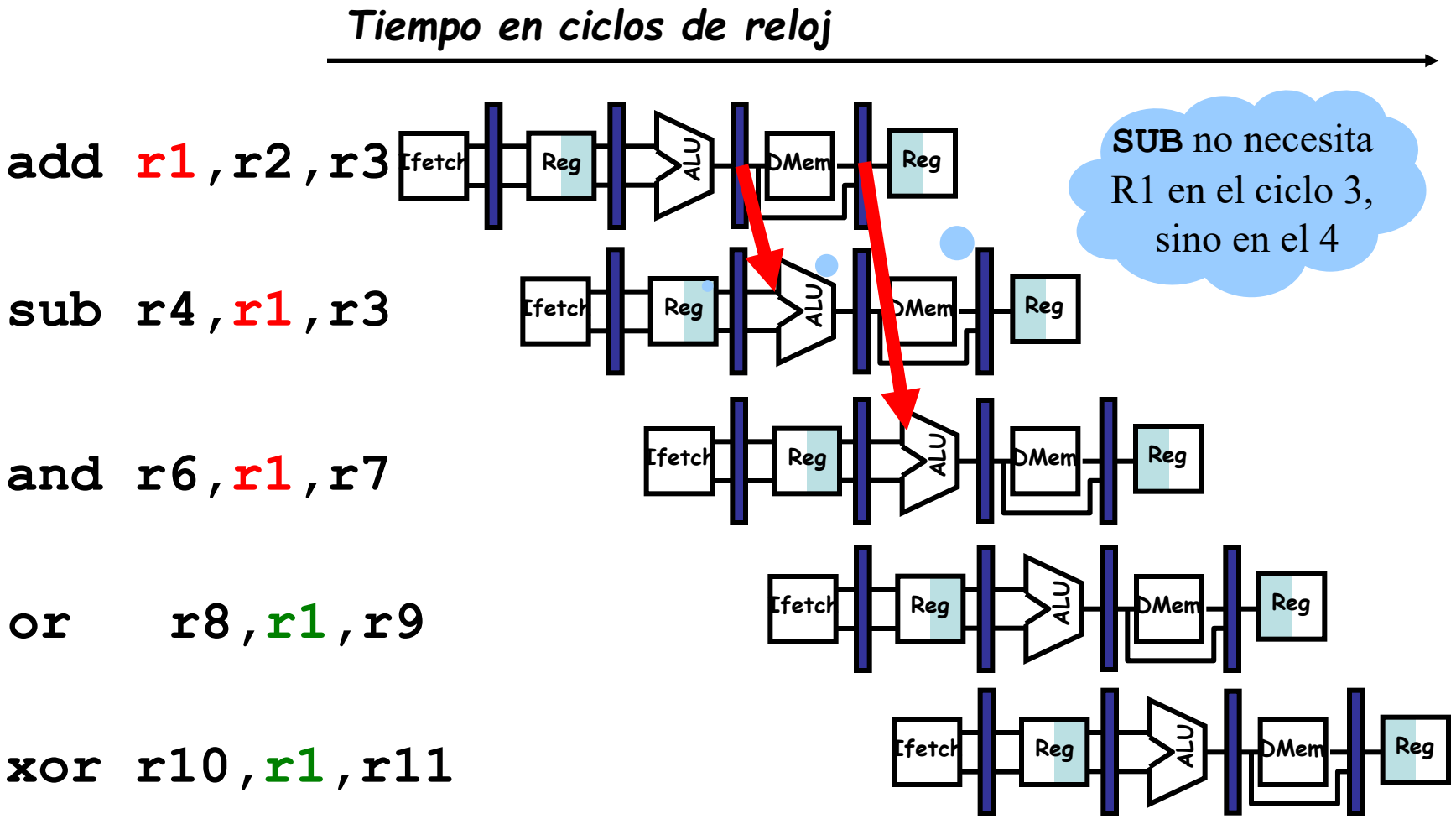
- Mitigación de los riesgos de datos
  - Adelantamientos
- Mitigación de los riesgos de control
  - Predicción de saltos estática
  - Predicción de saltos dinámica
- Excepciones
- Segmentación para punto flotante
  - Riesgos, adelantamientos y excepciones
- Emisión de múltiples instrucciones

# Mitigación de los riesgos de datos

- En el Tema 2 se estudió la forma de **resolver los riesgos de datos** para evitar resultados erróneos mediante:
  - Una **solución HW**: insertando ciclos de parada
  - Una **solución SW**: el compilador inserta instrucciones NOP
- Sin embargo, en ambos casos se produce una **degradación del rendimiento** (debido a ciclos de parada o aparición de instrucciones adicionales)
- Veamos ahora **otra solución HW** que no solo resuelve los riesgos de datos sino que además lo hace **afectando lo menos** posible al rendimiento
- Esta técnica HW se denomina **adelantamiento** y consiste en que el trasvase de información entre dos instrucciones dependientes **no se haga a través del banco de registros** sino desde la **UF productora** del dato hasta la **UF consumidora** del dato

# Mitigación de los riesgos de datos

- **Adelantamiento**: técnica hardware para minimizar el impacto negativo de los riesgos de datos (**RAW**)



# Mitigación de los riesgos de datos

- La técnica del **adelantamiento** puede **generalizarse** para que cualquier unidad funcional pueda obtener un operando directamente del registro de segmentación en el que pudiera encontrarse

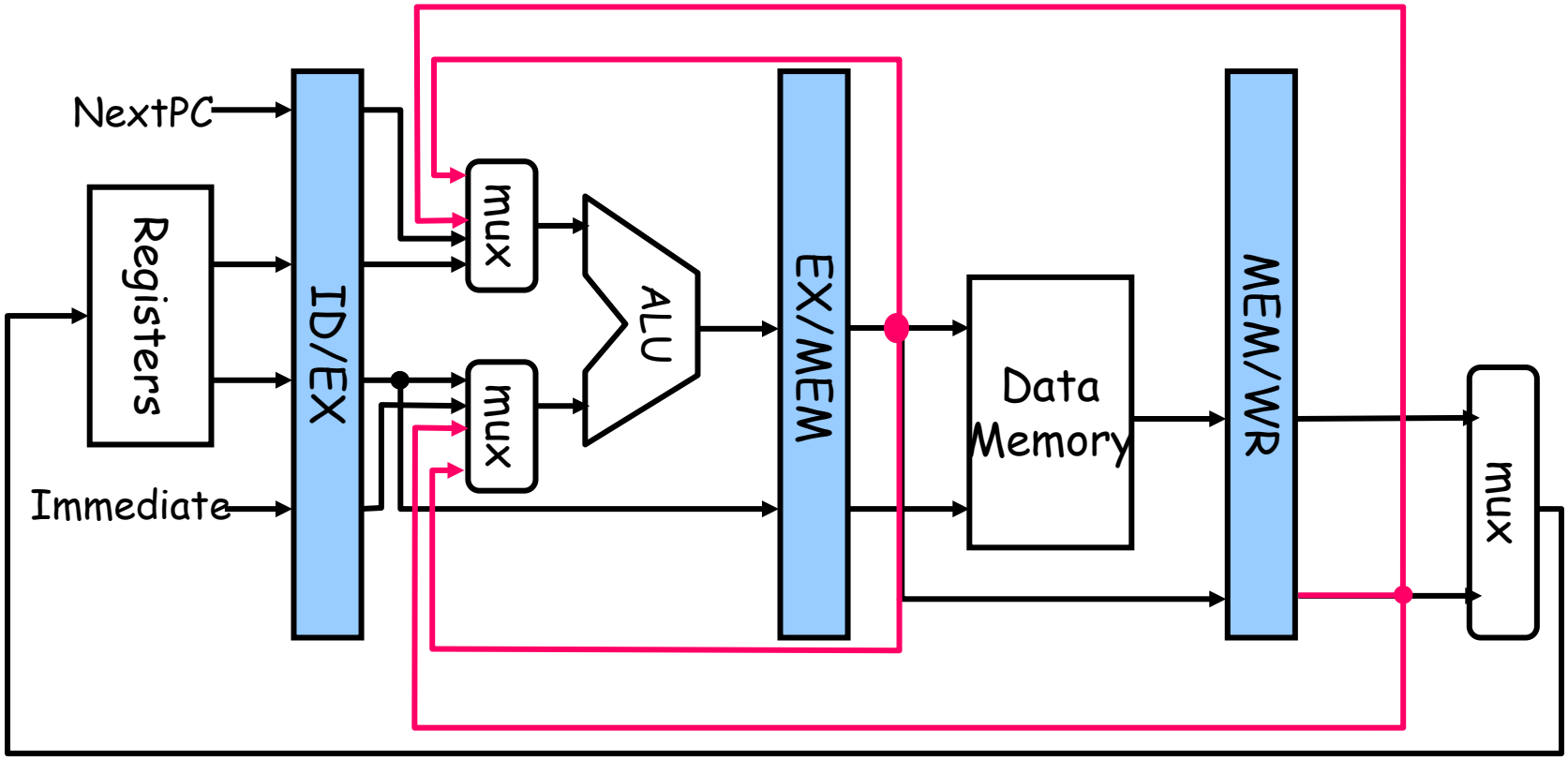
- **Ejemplo**: Para la siguiente secuencia de instrucciones:

```
ADD   R1, R2, R3
LW    R4, 0(R1)
SW    12(R1), R4
ADDI  R4, R4, -1
BNEZ  R4, Loop
```

¿Qué adelantamientos habría que realizar y a qué unidades funcionales afectarían?

# Mitigación de los riesgos de datos

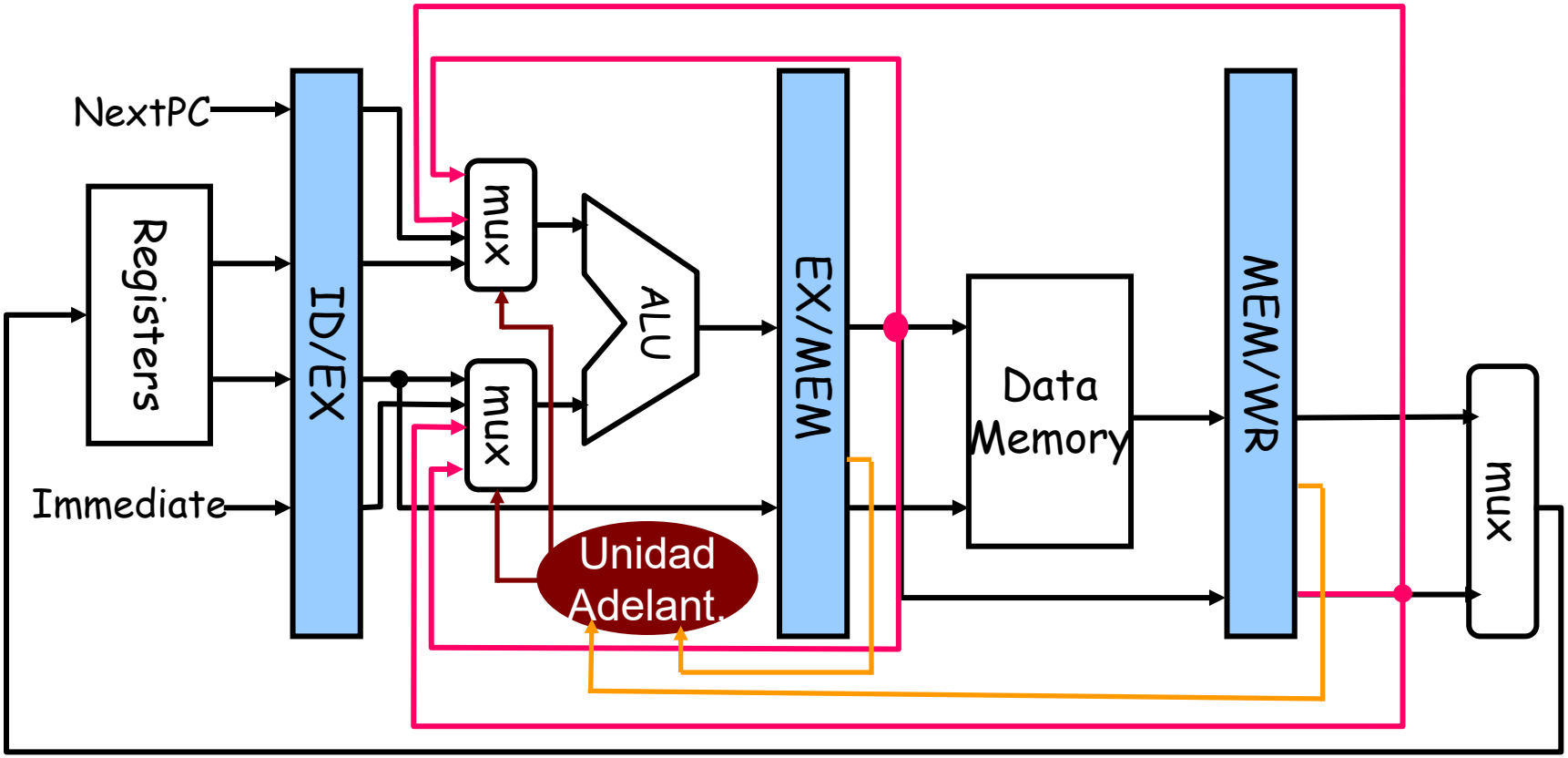
- Para que se puedan hacer **adelantamientos** necesitamos tener una circuitería adicional
- **Ejemplo**: para implementar adelantamientos a la ALU:




- Esto hace que aparezcan nuevas líneas de control y/o se modifiquen otras ya existentes

# Mitigación de los riesgos de datos

- Implementación de adelantamientos para la ALU:



# Mitigación de los riesgos de datos

- ¿Qué chequeos haría la **Unidad de Adelantamiento**?

Adelantamiento **EX**:

si (EX/MEM.RegWrite
  y (EX/MEM.Rd<>0)
  y (EX/MEM.Rd=ID/EX.Rs))
anticiparA=10

si (EX/MEM.RegWrite
  y (EX/MEM.Rd<>0)
  y (EX/MEM.Rd=ID/EX.Rt))
anticiparB=10

Adelantamiento **MEM**:

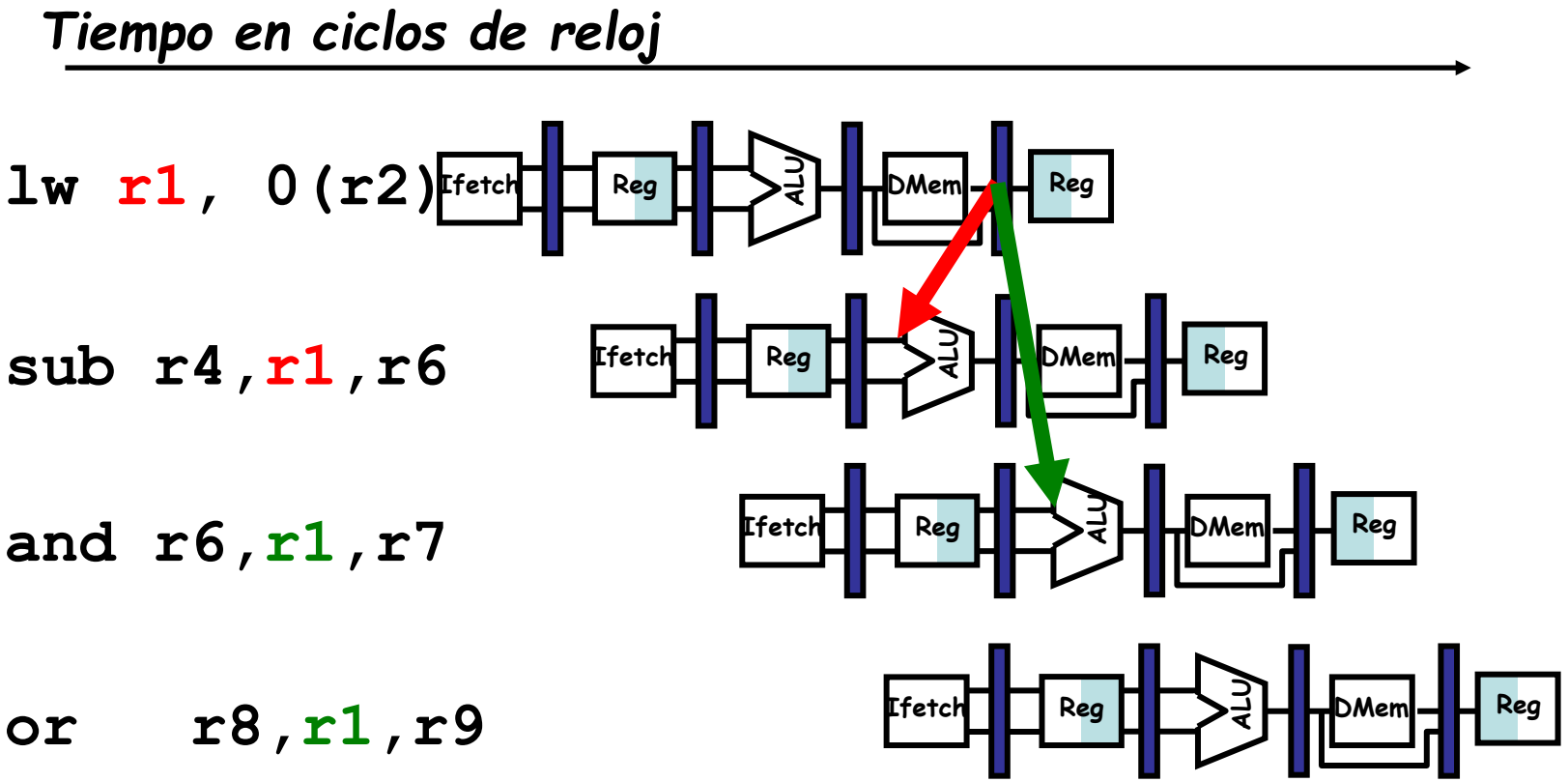
si (MEM/WB.RegWrite
  y (MEM/WB.Rd<>0)
  y (MEM/WB.Rd=ID/EX.Rs))
anticiparA=01

si (MEM/WB.RegWrite
  y (MEM/WB.Rd<>0)
  y (MEM/WB.Rd=ID/EX.Rt))
anticiparB=01

- Lógicamente, se haría primero el test de adelantamiento en EX, y sólo si este resultase negativo, lo comprobaría en MEM

# Mitigación de los riesgos de datos

- **OJO**: la técnica de adelantamiento en el DLX no es capaz de resolver todos los riesgos de datos que puedan aparecer:




- En DLX, las instrucciones de carga tienen un retraso que **no se puede eliminar totalmente con los adelantamientos** pero sí que queda muy mitigado (*solo **1 ciclo de parada***)

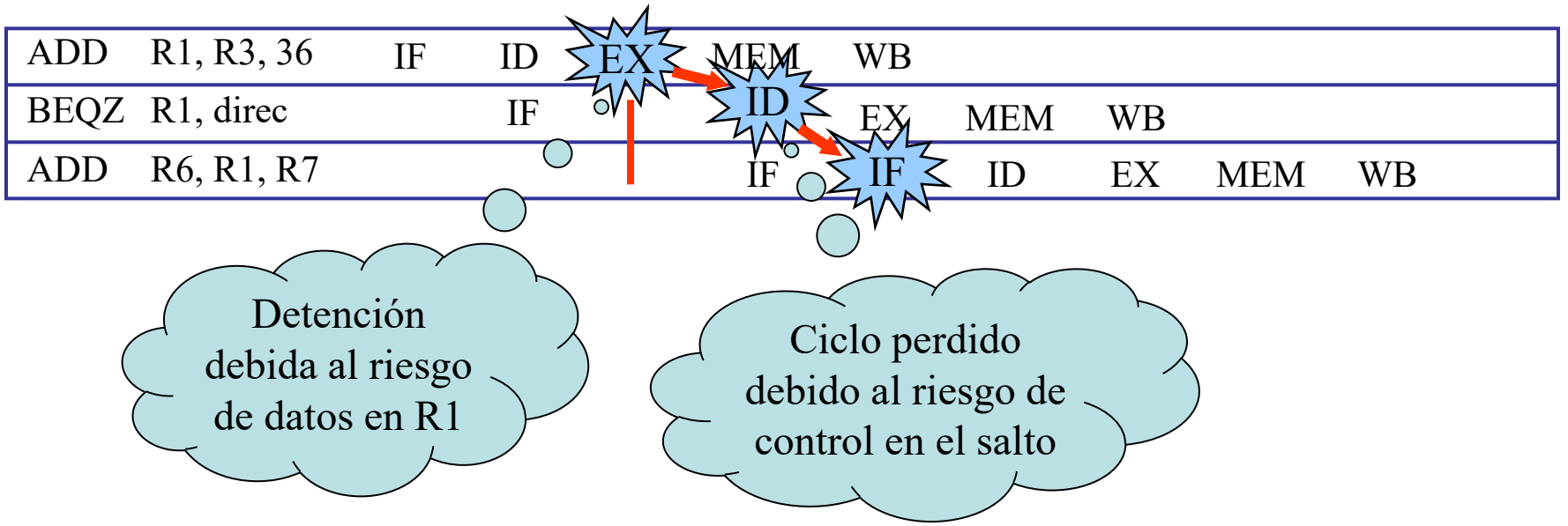
# Índice

# Mitigación de los riesgos de control

- La presencia de **instrucciones de salto** exige resolver los riesgos de control que surgen en el cauce
- Conforme tratamos de explotar más ILP las **dependencias de control** se convierten en una **severa limitación** del rendimiento, especialmente en aquellos **cauces con mayor número de etapas**
- En el **Tema 2** se vio la técnica conservadora de **detener el cauce** hasta conocer el resultado del salto. Pero ésta lleva asociada una fuerte degradación del rendimiento
- Con el fin de mejorar el rendimiento y **mitigar la penalización** de los saltos, los procesadores modernos recurren a técnicas de **predicción de saltos:**
  - Técnicas de **predicción estáticas**
    - **No** miran el **comportamiento** del salto
  - Técnicas de **predicción dinámicas**
    - Usan hardware para predecir el resultado de un salto en tiempo de ejecución gracias a que los saltos siguen **patrones predecibles**

# Mitigación de los riesgos de control

- En DLX la **penalización de los saltos** no es muy importante al tener un cauce de tan solo 5 etapas
- En el **Tema 2** se vio cómo se podía optimizar el camino de datos para que esta penalización fuera de **tan solo 1 ciclo**:
  - Pero **mover** la unidad detectora de ceros a **ID** tiene como efecto secundario la aparición de un **riesgo de datos nuevo**
  - Incluso con **adelantamiento** una instrucción ALU seguida por un salto cuya condición comprueba lo calculado en la instrucción ALU tendrá **1 ciclo extra de detención** que se sumará al ciclo de penalización del salto (*total **2 ciclos** de parada*)



# Predicción de saltos estática

**Predecir estáticamente como no tomado** (*not-taken*)

- Tratar todos los saltos como si no se fueran a tomar hasta que se resuelvan, procediendo con las instrucciones del camino NT
  - En caso de que el salto **sea tomado** se han de **descartar** todas las instrucciones **ejecutadas especulativamente**
  - En **DLX** es sencillo de implementar: seguir emitiendo instrucciones del camino NT y si resulta después que el salto era tomado, convertir a NOP las instrucciones buscadas del camino NT (observar que las instrucciones del camino NT nunca llegarán a escribir en los registros o en memoria)

**Predecir estáticamente como tomado** (*taken*):

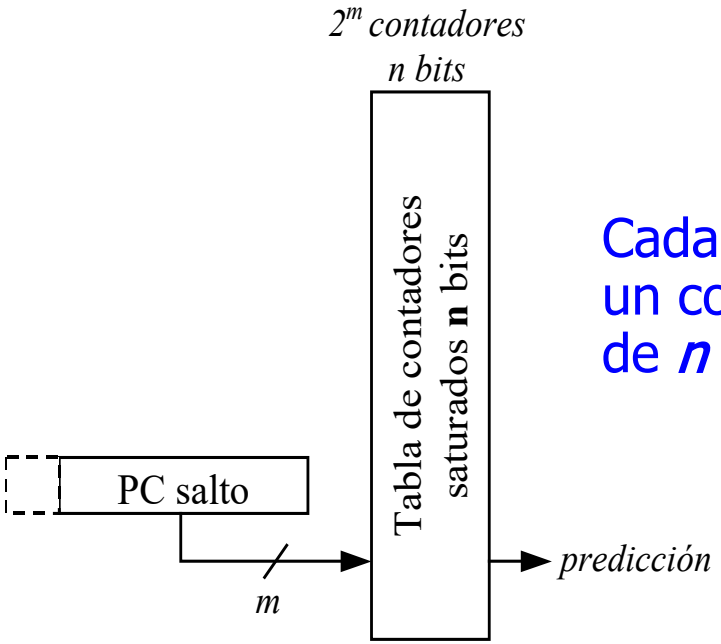
- Tratar todos los saltos como si se fueran a tomar:
  - En cuanto se tenga la dirección del salto se procede por ahí
  - Como antes, en caso de que el salto **sea no tomado** se ha de **deshacer** lo hecho
  - En **DLX** esta alternativa no tiene sentido, pues vamos a saber la dirección de salto al mismo tiempo que el resultado de la condición

# Predicción de saltos dinámica

- Técnicas de **predicción dinámicas**
  - Pretenden predecir el resultado de un salto condicional (*tomado* o *no-tomado*) en tiempo de ejecución
  - Obtienen muy buenas tasas de acierto gracias a que los saltos siguen **patrones de comportamiento predecibles**, generalmente basados en su historia
  - A continuación se estudiarán los siguientes predictores dinámicos de saltos:
    - Predictor de saltos basado en contadores saturados de *n* bits
    - Predictor de saltos con correlación (o de contexto)
    - Buffer de destino de saltos (*Branch Target Buffer* o BTB)

# Predictor basado en contadores saturados

- Es el esquema dinámico de predicción de saltos más simple
- Usa **contadores saturados** de *n* bits para predecir la condición de cada salto (Tomado o No Tomado)
- Estos contadores se organizan en una tabla indexada con los bits menos significativos del PC de la instrucción de salto




Cada entrada contiene un contador saturado de ***n*** bits

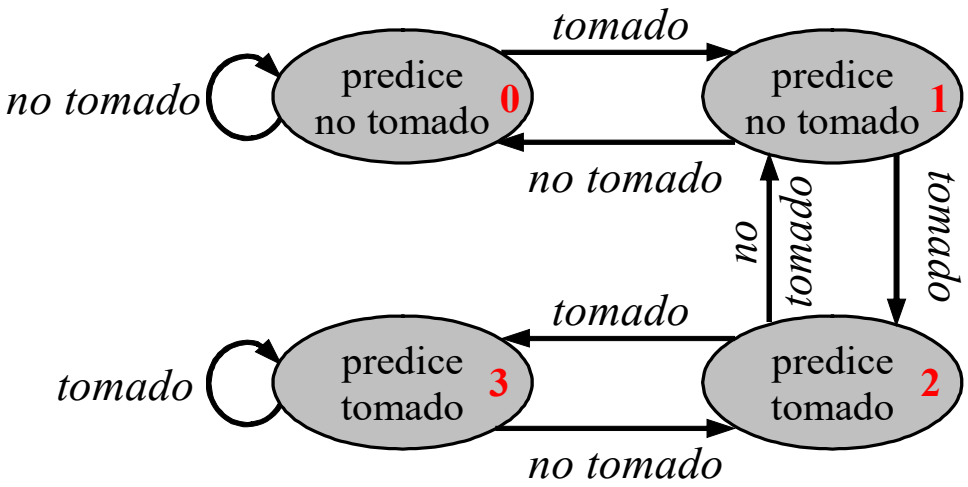
# Predictor con contadores saturados de 1 bit

- Cada contador contiene el último resultado de la instrucción de salto con la que está asociada: 0 si el salto no fue tomado, o 1 si el salto fue tomado
- Después de cada salto el contador se actualiza si fuera necesario

- **Problemas**
  - Se producen interferencias (*aliasing*) entre saltos diferentes que tengan en común los ***m*** bits menos significativos del **PC**
  - Este sencillo esquema de predicción de 1 bit **falla dos veces en cada bucle**, en lugar de una vez
- **Soluciones**
  - *Aliasing*: Tener más entradas en la tabla de contadores
  - *Fallos*: Utilizar más de un bit en la predicción

# Predictor con contadores saturados de 2 bits

- Usa contadores saturados de **2 bits** para cada salto




- El contador se debe actualizar tras ejecutar cada salto
  - Se incrementa si el salto es tomado y se decrementa en caso contrario
  - Al ser contadores saturados, el valor máximo al que puede llegar es 3, y el mínimo es 0

# Predictor con contadores saturados de *n* bits

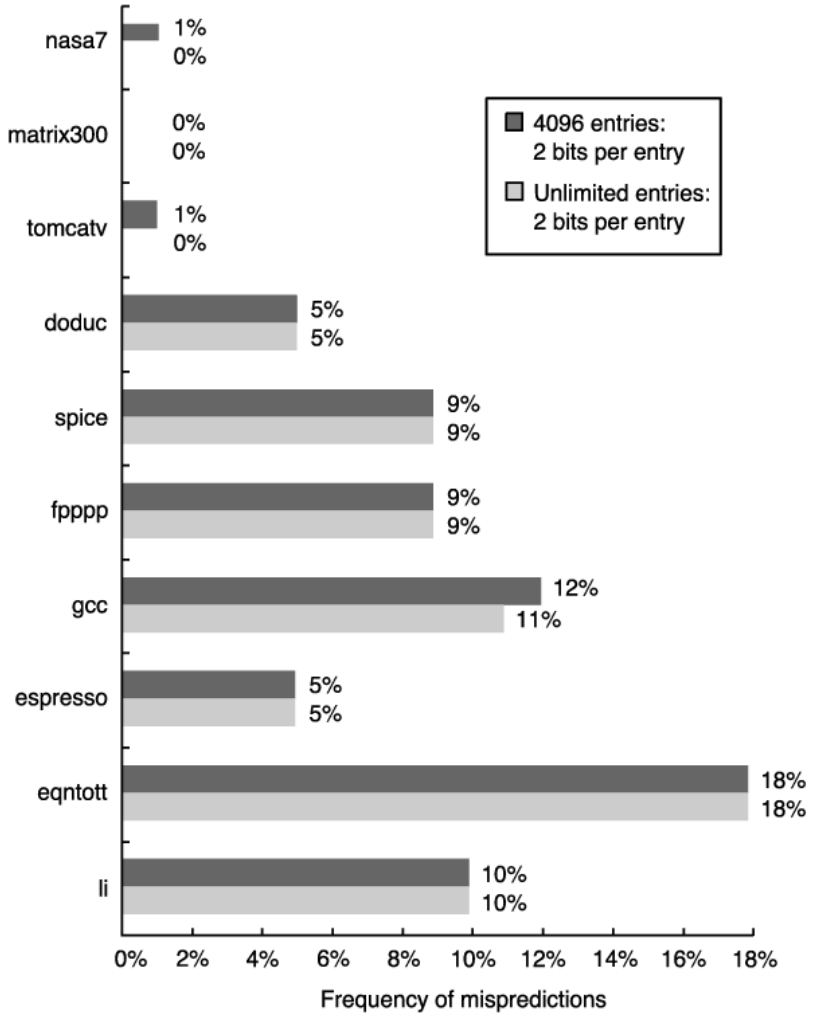
- Se puede **generalizar** este predictor a que use **contadores saturados de *n*-bits**

- En este caso el autómata asocia la **mitad** de los valores del contador al estado **Tomado** y la otra mitad a **No Tomado**
  - Si el salto se toma, se **incrementa** el contador
  - Si no se toma, se **decrementa**
  - Al ser un **contador de saturación**, no da la vuelta

  - Si el primer bit (más significativo) del contador es **1** → se predice **tomado**
  - Si es **0** → se predice **no tomado**

# Predictor con contadores saturados: resultados

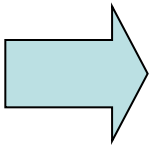
- ¿Qué precisión puede esperarse de un predictor con contadores saturados de 2 bits?
  - Veamos uno de 4096 entradas frente a uno ilimitado
- Podemos ver que lo que limita la precisión del predictor **no es el número de entradas**
- *¿Qué podemos hacer para mejorar el resultado?*



# Predictores con correlación

- **Objetivo:** usar información de **otros** saltos además de la historia del salto a predecir
  → **Predictores con correlación (o de contexto)**
- **Ejemplo**:

```
if (d==0)
    d=1;
if (d==1)
```



```
        BNEZ R1,L1      ;branch b1 (d!=0)
        ADDI R1,R0,#1   ;d==0, así que d=1
L1:     SUBI R3,R1,#1
        BNEZ R3,L2      ;branch b2 (d!=1)
        ...
L2:
```

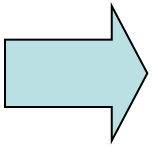
| Initial value of d | d==0? | b1 | Value of d before b2 | d==1? | b2 |
|---|---|---|---|---|---|
| 0 | Yes | Not taken | 1 | Yes | Not taken |
| 1 | No | Taken | 1 | Yes | Not taken |
| 2 | No | Taken | 2 | No | Taken |

Se observa que el comportamiento de **b2** depende del de **b1**

# Predictores con correlación

- Veamos qué ocurriría si **d** alterna los valores 0 y 2 para un predictor basado en contadores saturados de **1 bit** inicializado a **Not-Taken**

```
if (d==0)
    d=1;
if (d==1)
```



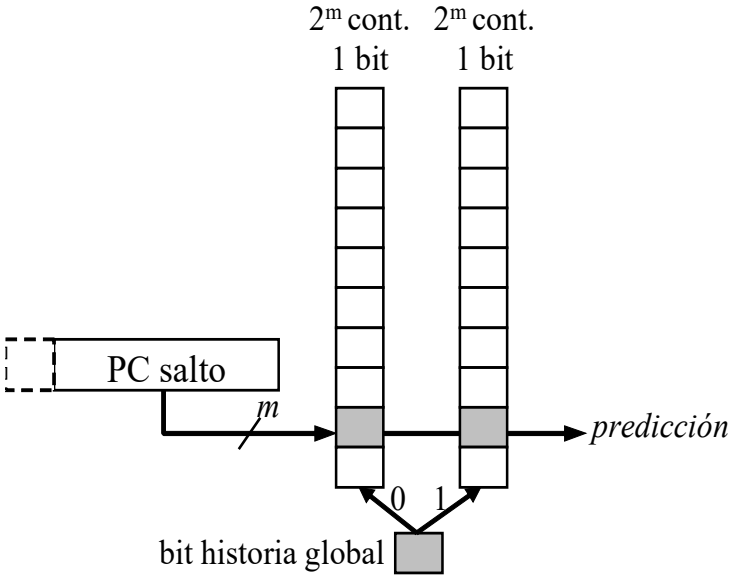
```
        BNEZ R1,L1      ;branch b1 (d!=0)
        ADDI R1,R0,#1   ;d==0, así que d=1
L1:     SUBI R3,R1,#1
        BNEZ R3,L2      ;branch b2 (d!=1)
        ...
L2:
```

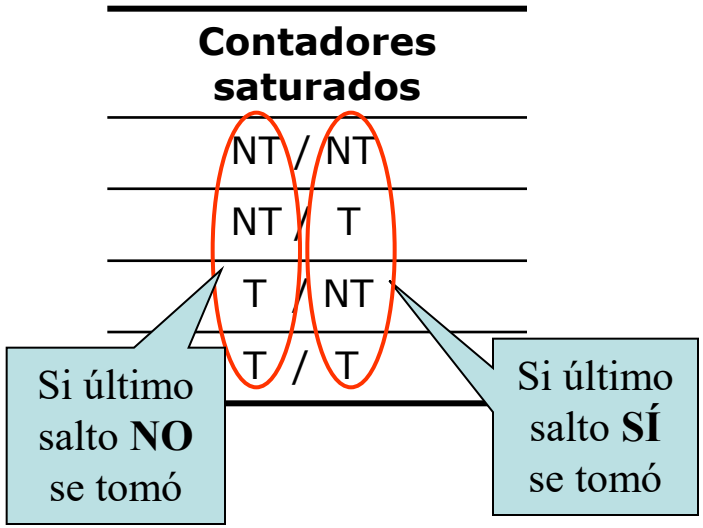
| d=? | b1 prediction | b1 action | New b1 prediction | b2 prediction | b2 action | New b2 prediction |
|---|---|---|---|---|---|---|
| 2 | NT | T | T | NT | T | T |
| 0 | T | NT | NT | T | NT | NT |
| 2 | NT | T | T | NT | T | T |
| 0 | T | NT | NT | T | NT | NT |

¡¡Todos los saltos se equivocan en su predicción!!

# Predictores con correlación

- Veamos cómo se organiza un **predictor con correlación** que usa contadores saturados de **1 bit** y **1 bit de historia global**
  - Con 1 bit de historia global habrá **2 tablas de contadores saturados**. Así que cada salto tiene asociado **dos contadores**
    - Uno se usará si **el último salto NO se tomó**
    - El otro se usará si **el último salto SÍ se tomó**
  - La elección se basa en el **bit de historia global**



# Predictores con correlación

- Analicemos el comportamiento del predictor anterior con contadores saturados de **1 bit** y **1 bit de historia global**
  - Este tipo de predictor se denomina **(1,1)** porque utiliza el último salto para elegir entre un par de contadores de 1 bit

| d=? | b1 prediction | b1 action | New b1 prediction | b2 prediction | b2 action | New b2 prediction |
|---|---|---|---|---|---|---|
| 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 |

- Únicamente falla en la primera iteración cuando **d** vale 2
- Debido a los valores que toma **d**, el predictor se adapta al comportamiento del programa
  - Si **d** tomara otros valores, la adaptación sería diferente

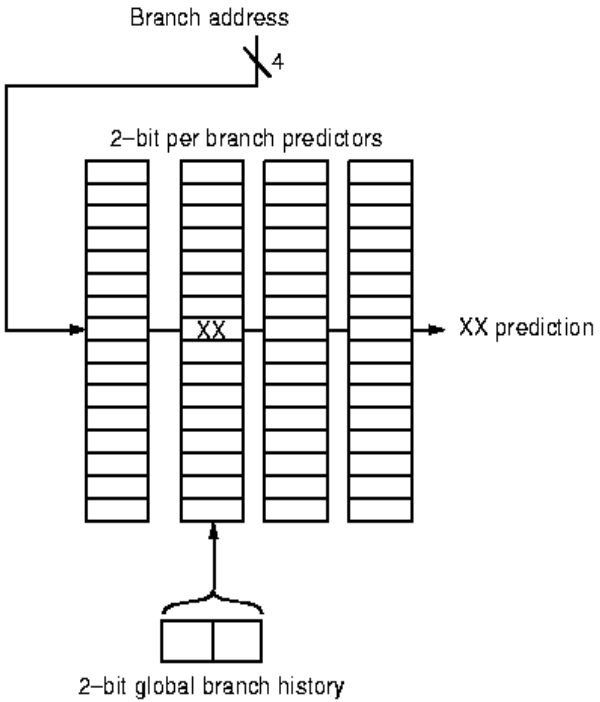
# Predictores con correlación

- Estos predictores se puede generalizar a un predictor **(m,n)** donde se usa el comportamiento de los últimos **m** saltos para seleccionar una de las **$2^m$** tablas de contadores saturados de **n** bits
  - ***Nota***: Si **m=0**, tenemos un predictor **sin historia global**, denominado **(0,n)** que se corresponde a un predictor básico únicamente basado en contadores saturados de **n** bits (como los vistos en el apartado previo)
- Un predictor **(m,n)** ocupa:

  **($2^m \times$ n $\times$ Número de entradas) bits**
- La historia global de los últimos **m** saltos se almacenan en un registro de desplazamiento de **m** bits

# Predictores con correlación

- **Ejemplo:** Predictor (2,2)




4 tablas con $2^4$ entradas cada una = 64 entradas

**PC del salto:** para indexar las tablas

**Historia global (2 bits):** para seleccionar una de las 4 tablas

- Este predictor (2,2) ocupará: $2^2 \times 2 \times 16 =$ **128 bits**

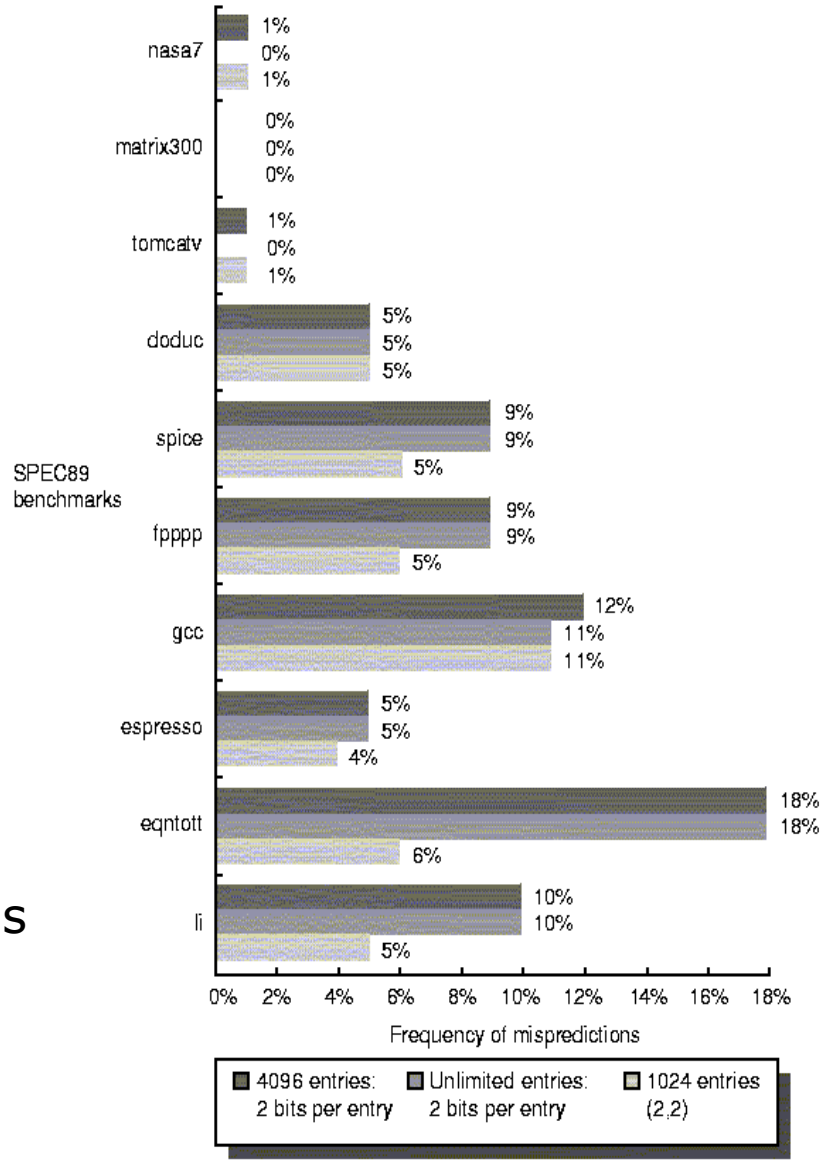
# Predictores con correlación

- Para cuantificar el efecto de usar información de correlación vamos a comparar un **(0,2)** y un **(2,2)** de igual tamaño
- Suponiendo un tamaño de **8 Kbits**:
  - ¿cuántas entradas debe tener cada uno de ellos?

    El (0,2) → 4096 entradas

    El (2,2) → 1024 entradas

- El **(2,2)** de 1024 entradas
  - supera al de **(0,2)** de 4096 entradas
  - Y al **(0,2)** de tamaño ilimitado!!



# Buffer de destino de saltos (BTB)

- Para reducir la penalización de los saltos en nuestro **DLX** a solo **0 ciclos** hemos de saber en la **etapa IF**:
  - ¿Es la instrucción que se acaba de buscar un salto?
  - Y en caso afirmativo, ¿será tomado o no?
  - Y si es tomado, ¿cuál es la dirección de la siguiente instrucción?

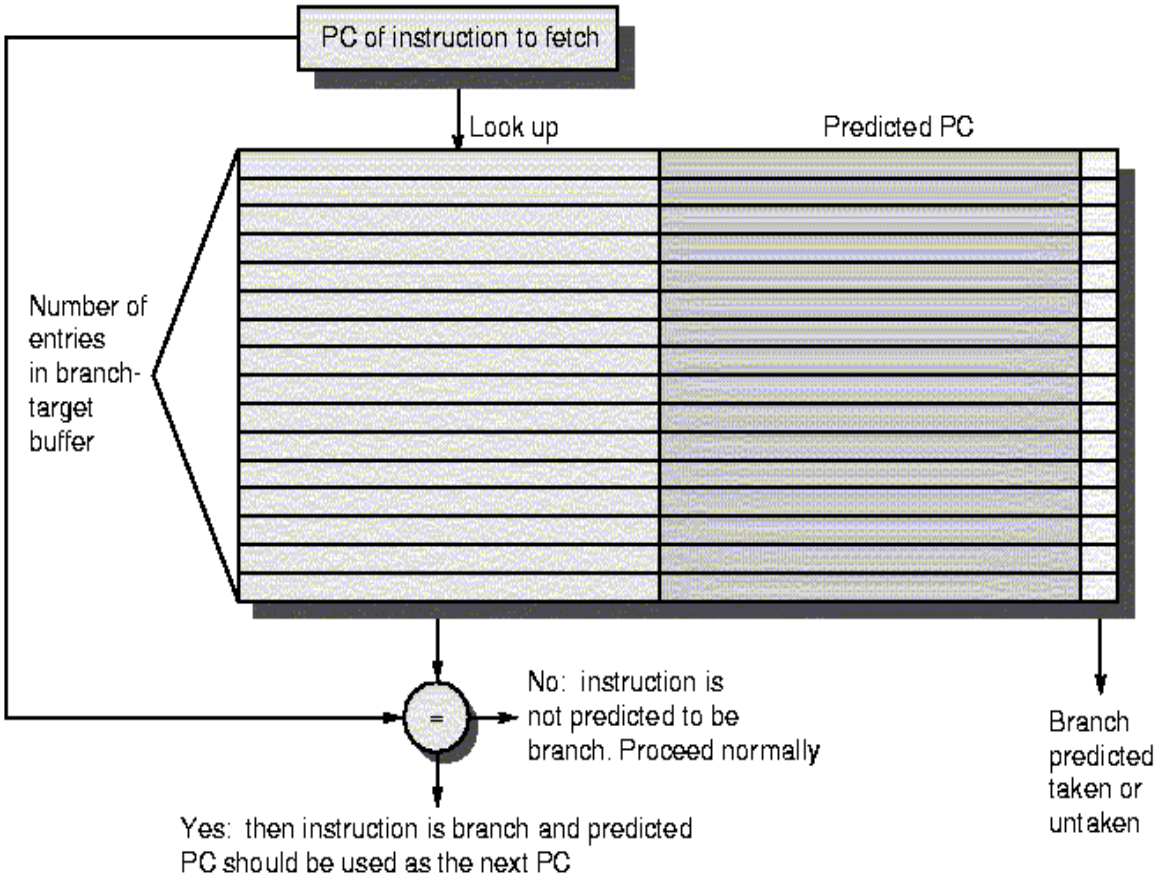
- **Buffer de Destino de Saltos** (*Branch Target Buffer,* **BTB**)
  - **Caché** que proporciona la dirección de la siguiente instrucción que sigue a un salto
  - Se accede en la etapa **IF** usando el **PC** actual, y si hay acierto de BTB suponemos que se trata de un **salto** y el **BTB** proporciona el **siguiente PC**
    - Puede ser el **PC secuencial** (PC+4) o el **PC destino del salto**
  - A diferencia de los predictores anteriores, el **BTB usa etiquetas** (parte del PC) para evitar interferencias

# Buffer de destino de saltos (BTB)

- Si el PC se encuentra en las etiquetas
  → acierto de BTB
  → **es un salto**

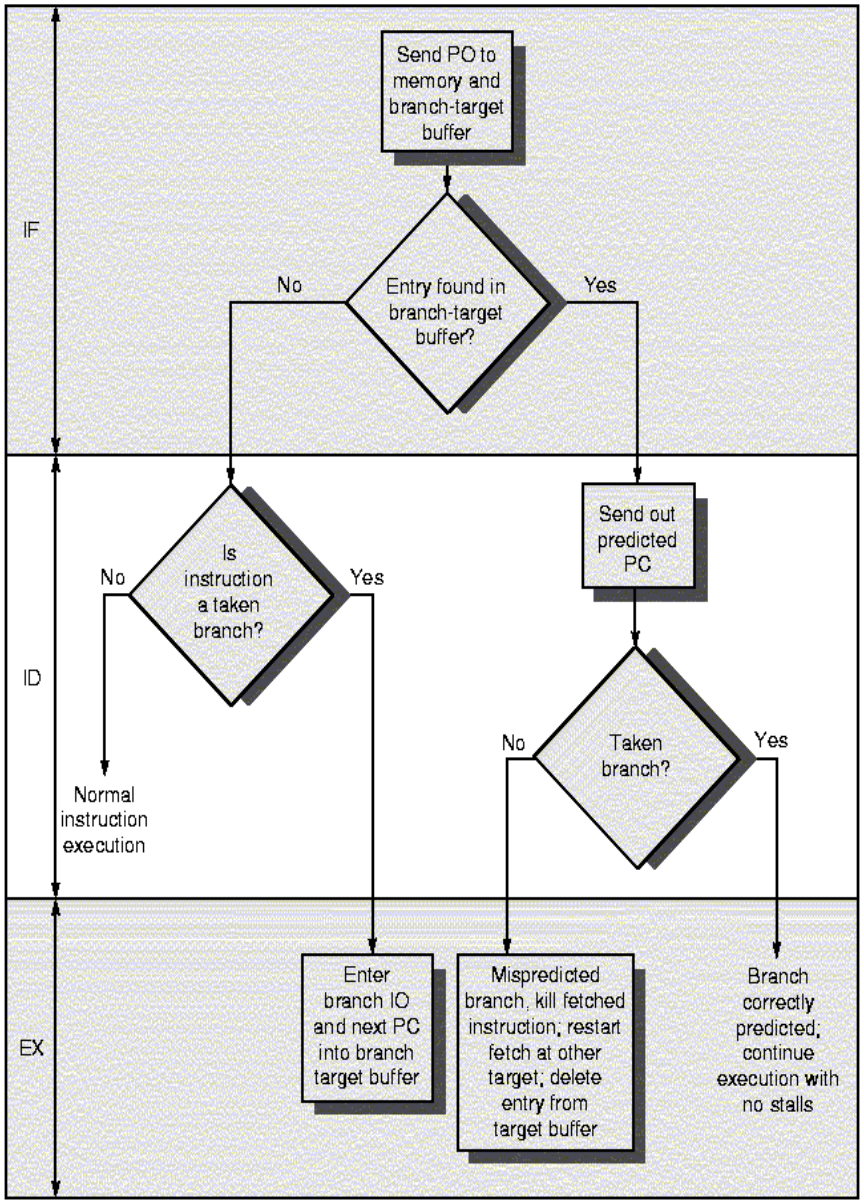
- El segundo campo proporciona el **PC siguiente** (secuencial ó destino)

- En un tercer campo **opcional** se podría tener información de predicción (ej. 1 ó 2 bits de historia)



# BTB (que solo almacena saltos tomados)

- **Comportamiento**
  - Si encontramos el PC en la BTB, predecimos que la instrucción será un **salto tomado**. Si en el siguiente ciclo vemos que no es un salto tomado, lo quitamos del BTB.
  - Si no la encontramos, usamos el PC secuencial (PC+4) y si resulta que es un salto tomado, actualizamos el BTB para futuras ocasiones.
- **Casos que pueden ocurrir**
  - **No hay penalización:**
    - Si la instrucción SÍ está en la BTB y la predicción es correcta (salto tomado)
    - Si la instrucción NO está en la BTB y resulta ser un salto no tomado (u otra instrucción)
  - **Sí hay penalización:**
    - Si la instrucción SÍ está en la BTB pero la predicción es incorrecta (salto no tomado)
    - Si la instrucción NO está en la BTB y resulta ser un saldo tomado



# BTB (que solo almacena saltos tomados)

- **Ejemplo**
  - Determinar la penalización media de un **BTB**, suponiendo una penalización para fallos en la predicción de **2 ciclos**. Suponer una tasa de aciertos en el BTB de 80%, una tasa de aciertos en la predicción de 90% y una frecuencia de saltos tomados de 60%.

**Solución**

Penalización por salto = acierto en BTB $\times$ predicciones incorrectas $\times$ 2 +
fallo en BTB $\times$ saltos tomados $\times$ 2

Penalización por salto = $(0.80 \times 0.10 \times 2) + (0.20 \times 0.60 \times 2)$ = **0.4 ciclos**

# Índice

# Excepciones

- Llamamos **excepción** al cambio inesperado en el flujo de control proveniente de una causa interna o externa, y si esa causa es externa se puede llamar también **interrupción**
- Ejemplos de excepciones:
  - Solicitud de un dispositivo de E/S
  - Fallo de página
  - Acceso a memoria no alineado
  - Violación de la protección de memoria
  - Breakpoints, etc.
- Estas situaciones son más difíciles de manejar en las máquinas segmentadas debido a la superposición de instrucciones, ya que esto hace más difícil saber cuándo una instrucción puede cambiar de forma segura el estado de la máquina
- Las excepciones:
  - suelen suceder en medio de la ejecución de una instrucción
  - deben ser capaces de guardar el estado, atender la excepción, restaurar el estado y "recomenzar" el programa original

# Excepciones en DLX

- Los problemas que se pueden presentar en cada etapa son:

| Etapa del Cauce | Posibles excepciones |
|---|---|
| IF | Fallo de página en la búsqueda de la instrucción<br>Acceso a memoria mal alineado<br>Violación de la protección de memoria |
| ID | Código de operación indefinido o ilegal |
| EX | Excepción aritmética |
| MEM | Fallo de página en la búsqueda de datos<br>Acceso a memoria mal alineado<br>Violación de la protección de memoria |
| WB | Ninguna |

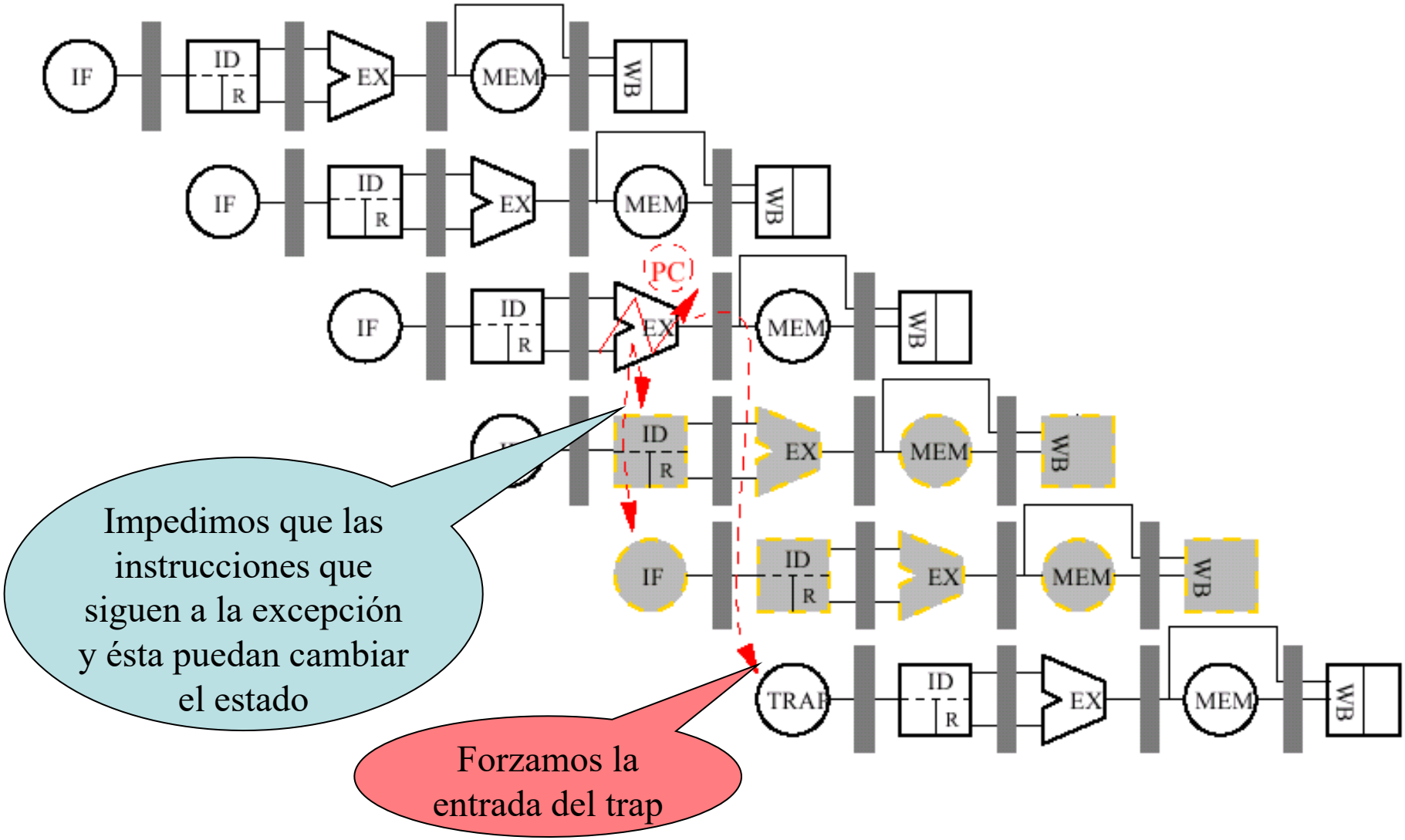
- Las excepciones más difíciles suceden en la etapa EX ó MEM de nuestro pipeline y deben poder reiniciarse
- Un fallo de página debe ser **reiniciable** y necesita la intervención del SO

# Excepciones en DLX

- En DLX podemos tratar las excepciones de la siguiente forma:
  - Forzamos la entrada de una instrucción **TRAP** en IF
  - Impedimos que las instrucciones que siguen a la causante de la excepción y ésta misma **puedan cambiar el estado de la máquina**
  - El SO guarda el PC de la instrucción que causó el fallo. Lo usará para volver de la interrupción
- Si se cumplen estas condiciones, decimos que el *pipeline* implementa **"excepciones precisas"**

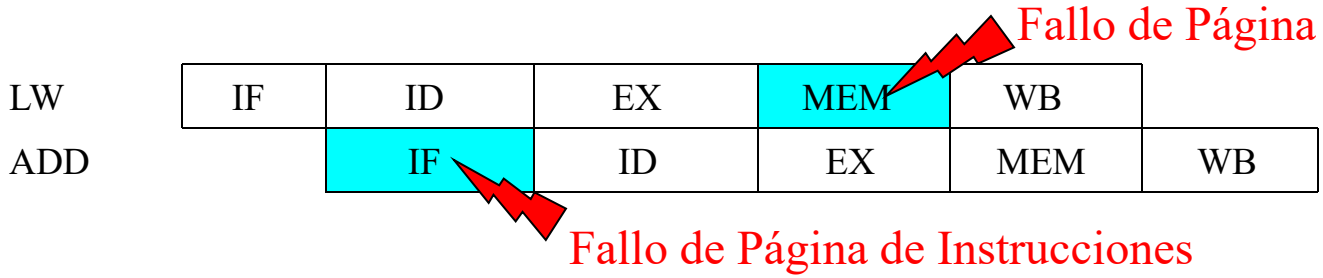
# Excepciones en DLX

**Ejemplo.** Excepción en la etapa EX



# Excepciones en DLX

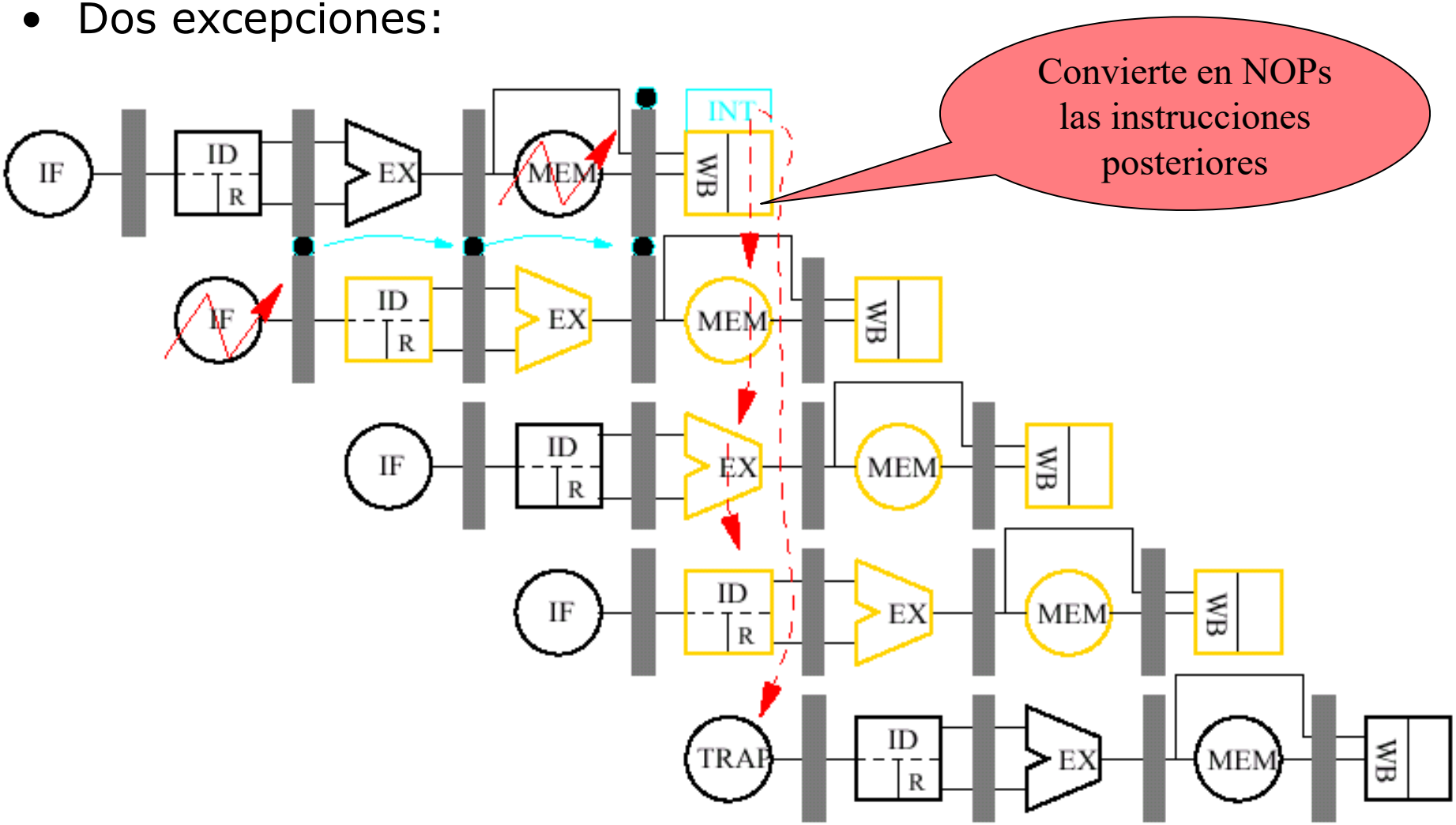
- Una instrucción puede provocar una excepción antes de que la cause otra instrucción anterior




- Dos posibles soluciones:
  - No tener excepciones precisas y tratar las excepciones en el orden de aparición
  - Tratar las excepciones según el orden de programa:
    - Cada instrucción que entra en el pipeline tiene asociado un registro con tantos bits como etapas en las que se pueden originar excepciones
    - Si se produce una excepción se pone a 1 el bit de la etapa correspondiente y se convierte en NOP la instrucción
    - En la última etapa se mira si ocurrió excepción (alguno de los bits está a 1)

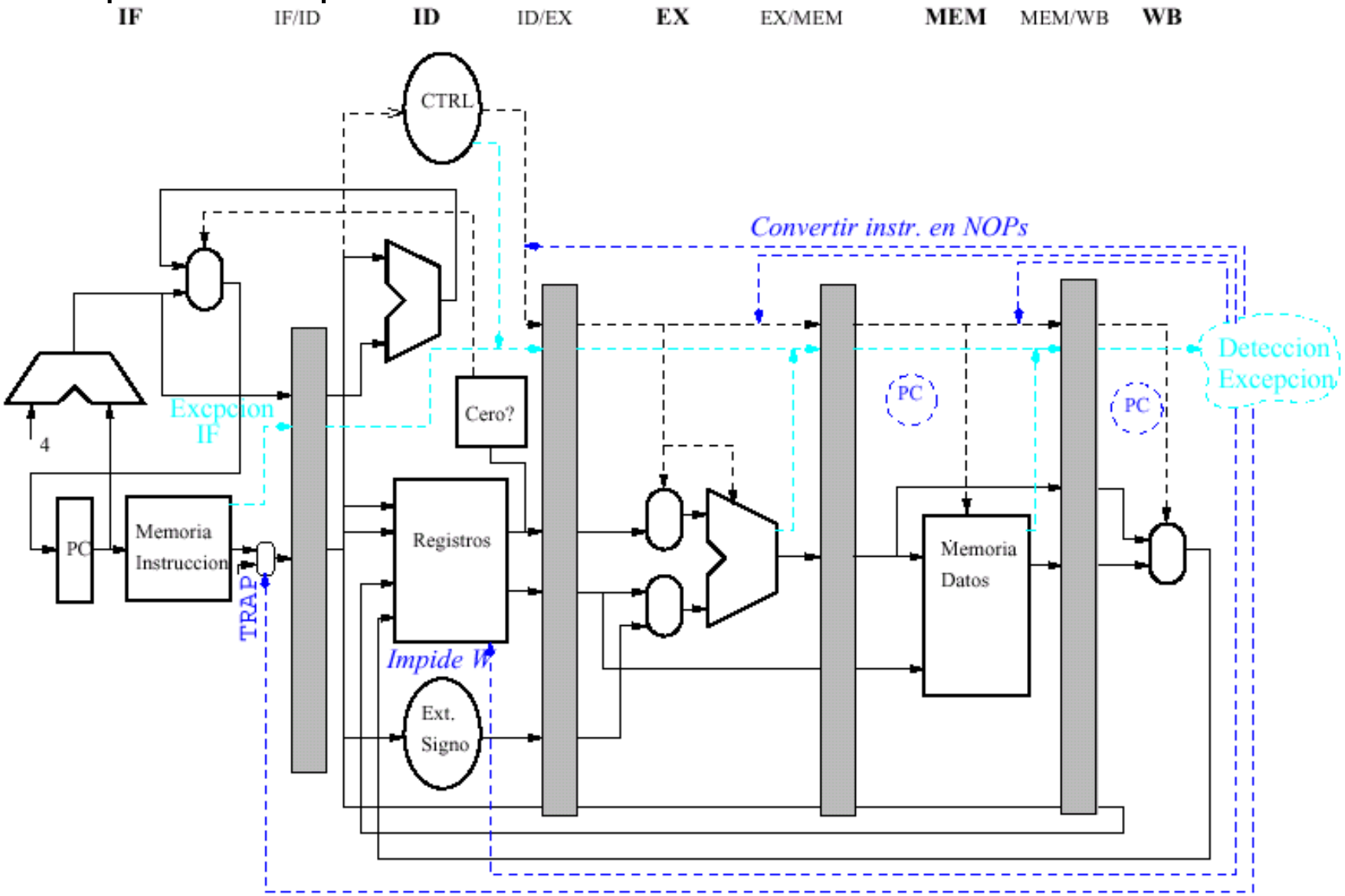
# Excepciones en DLX

- Dos excepciones:



# Excepciones en DLX

- Una posible implementación:



# Índice

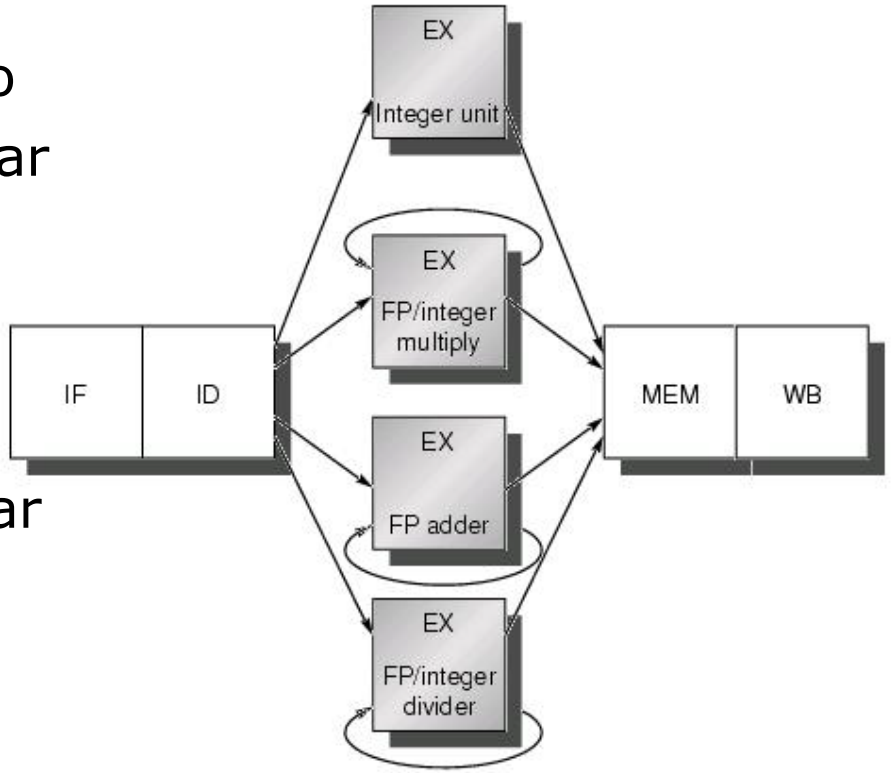
# Segmentación para punto flotante

- No podemos exigir que todas las operaciones de punto flotante de DLX **se completen en un ciclo de reloj**, pues obligaría a:
  - tener un reloj muy lento
  - implementaciones muy costosas en las unidades de coma flotante
- Las instrucciones de punto flotante necesitan **varios ciclos** en la etapa de ejecución
- Podemos verlo como si las operaciones en PF repitieran varias veces la etapa EX y además tuviéramos varias unidades funcionales de PF
- Consideramos una versión de DLX con 4 unidades funcionales:
  - La unidad funcional principal de enteros, que maneja las cargas y almacenamientos, las operaciones ALU enteras y los saltos
  - Multiplicador entero y PF
  - Sumador PF que se encarga de sumas, restas y conversiones PF
  - Divisor de enteros y PF

# Segmentación para punto flotante

- Definimos la **latencia** de una unidad funcional como el número de ciclos requerido para completar su operación

- El **tiempo de iniciación** es el número de ciclos que deben pasar entre la emisión de dos operaciones del mismo tipo

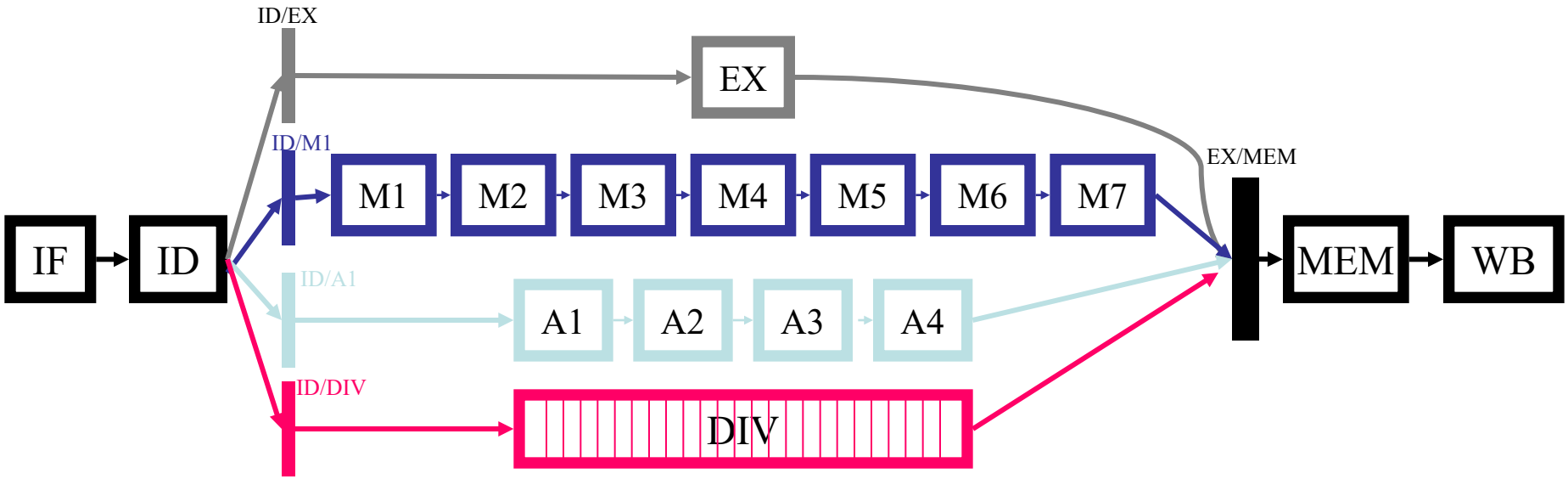



| Unidad Funcional | Latencia | Intervalo de Iniciación |
|---|---|---|
| ALU de enteros | 1 | 1 |
| Memoria de datos (cargas enteras y PF) | 1 | 1 |
| Sumador PF | 4 | 1 |
| Multiplicador PF (también de enteros) | 7 | 1 |
| Divisor PF (también de enteros y raíz cuadrada PF) | 25 | 25 |

La unidad de división en PF no está segmentada

# Segmentación para punto flotante

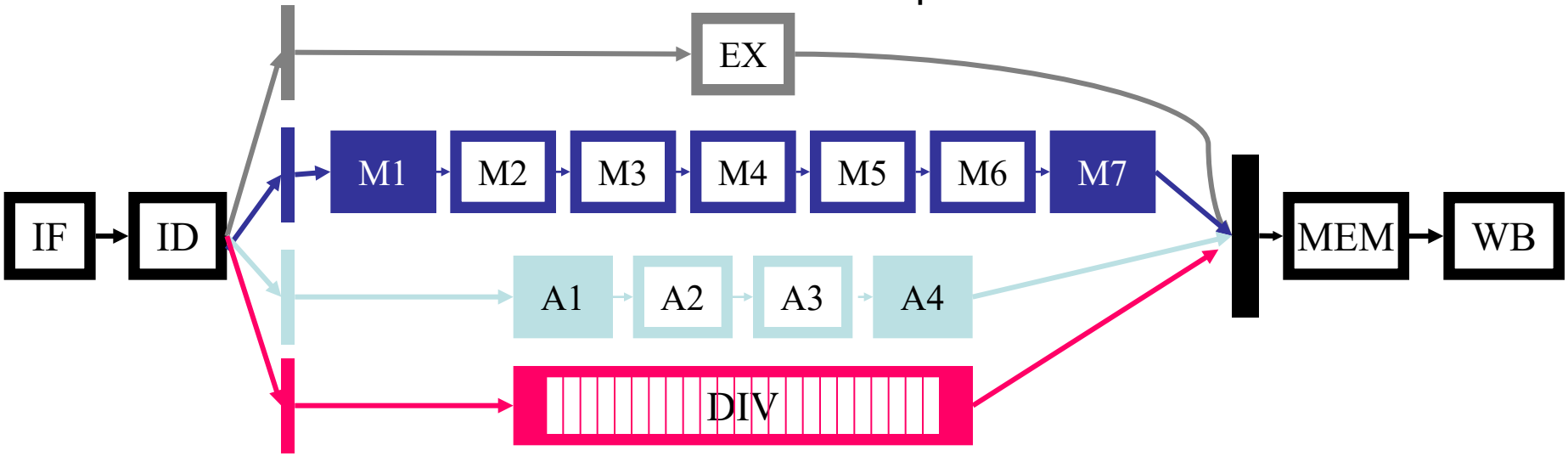
- Tendríamos algo así:




- Necesitamos poner ***pipeline registers*** entre las **etapas** de las **Unidades Funcionales de PF**
- También debemos extender el registro ID/EX para que ahora sea: ID/EX, ID/A1, ID/M1, ID/DIV
- Sólo nos hace falta un registro EX/MEM pues únicamente podremos tener una instrucción a la vez entrando en dicha etapa

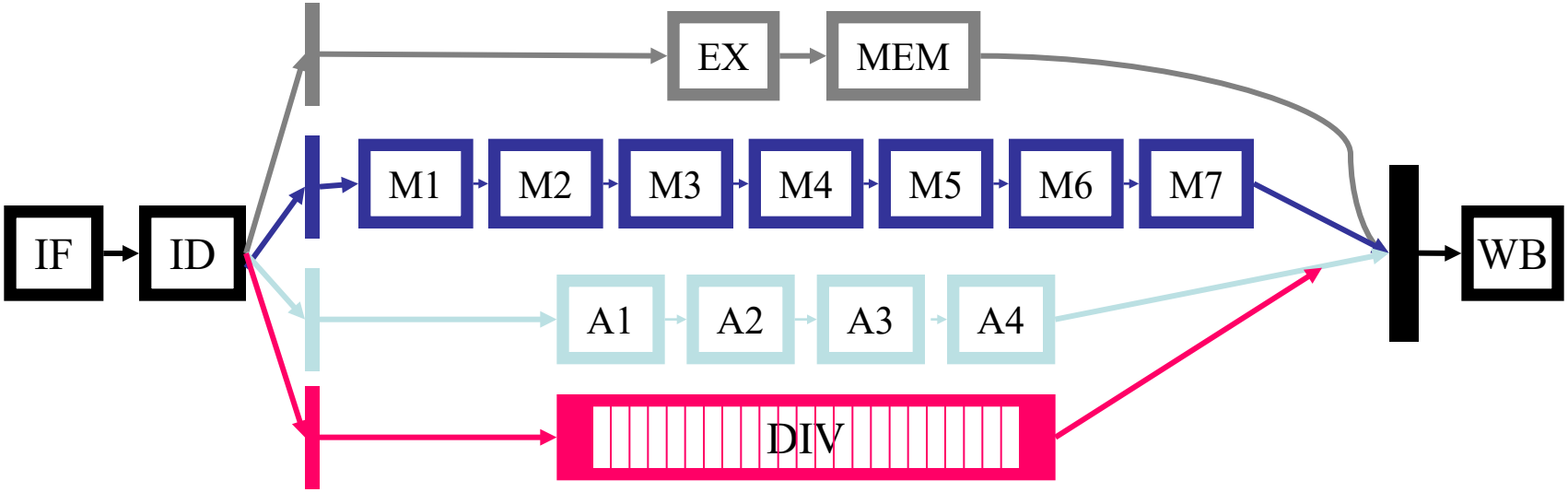
# Segmentación para punto flotante

- **Adelantamientos** en el nuevo cauce:
  - Al igual que hacíamos con el cauce de enteros, también aquí podemos implementar adelantamientos detectando si el registro destino en algún *latch* intermedio EX/MEM, A4/MEM, M7/MEM, D/MEM o MEM/WB es un registro fuente de una instrucción PF
  - Si es necesario adelantar el dato PF debe habilitarse el multiplexor correspondiente para que la entrada se coja del registro intermedio de la instrucción que aún no ha completado
  - Los adelantamientos deben hacerse **hacia** la primera etapa de la unidad funcional **desde** la última etapa de la unidad funcional



# Segmentación para punto flotante

- El hecho de que solamente las instrucciones enteras accedan a memoria nos permitiría modificar el esquema del procesador, dejándolo de esta forma:




- La mayoría de procesadores actuales tienen una estructura similar a ésta, teniendo el acceso a memoria en la ruta de enteros

# Riesgos en punto flotante

- El solapamiento en la ejecución de instrucciones cuyos tiempos de ejecución difieren, y el que algunas unidades funcionales no estén completamente segmentadas, crea varias complicaciones:
  - Pueden aparecer **riesgos estructurales** en el acceso a la UF de la división:

| Instrucción | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ……. | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DIVD F2,F4,F6 | IF | ID | D1 | D2 | D3 | D4 | D5 | …….. | D24 | D25 | WB | | | | | |
| DIVD F8,F10,F12 | | IF | ID | **Riesgo estructural por la unidad funcional de la división** | | | | | | | D1 | D2 | D3 | D4 | D5 | D6 |
| SD 0(R2),F2 | | | IF | | | | | | | | ID | EX | Mem | WB | | |

  - Debido a que cada instrucción tiene diferente número de ciclos de ejecución, podría haber varias instrucciones que quieran **escribir a la vez** su resultado final en el banco de registros:

| Instrucción | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MULTD F2,F4,F6 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | WB | |
| ……. | | | | | | | | | | | |
| ……. | | | | | | | | | | | |
| ADDD F8,F4,F6 | | | | IF | ID | A1 | A2 | A3 | A4 | WB | |

**Riesgo estructural por el acceso al banco de registros**

# Riesgos en punto flotante

- Puede ocurrir que las instrucciones **terminen en distinto orden** a como fueron emitidas, causando esto posibles problemas con las excepciones
- Pueden aparecer **riesgos WAW**:

| Instrucción | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MULTD F2,F6,F8 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | WB | |
| ADDD F2,F4,F6 | | IF | ID | A1 | A2 | A3 | A4 | WB | | | |


- Debido a que las operaciones tienen latencias mayores, va a haber más **detenciones** debidas a los riesgos RAW:

| Instrucción | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LD F4, 0(R2) | IF | ID | EX | Mem | WB | | | | | | | | | | | | | |
| MULTD F0,F4,F6 | | IF | ID | | **M1** | M2 | M3 | M4 | M5 | M6 | M7 | WB | | | | | | |
| ADDD F2,F0,F8 | | | IF | | ID | | | | | | | **A1** | A2 | A3 | A4 | WB | | |
| SD 0(R2),F2 | | | | | IF | | | | | | | ID | | | EX | Mem | WB | |

## Solución:

- Extender la unidad de detección de riesgos para prevenir o eliminar los riesgos que introducen las unidades funcionales de punto flotante

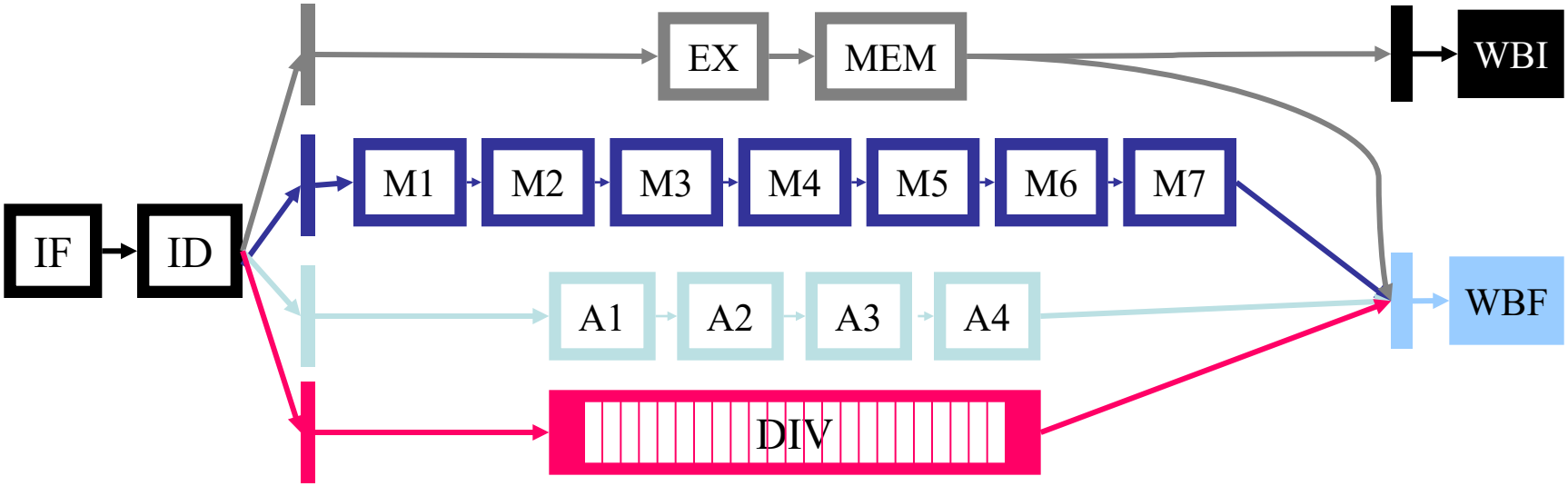
# Riesgos en punto flotante

## Bancos de registros independientes para enteros y PF

- Sólo las cargas o almacenamientos de PF y los movimientos entre registros PF y enteros involucran a los dos bancos de registros en una misma instrucción, con lo que sólo en estas instrucciones se pueden producir riesgos entre ambos bancos de registros
- **Ventajas**:
  - No se producen riesgos estructurales entre las operaciones enteras
  - Se duplica el número de registros sin complicar la lógica de decodificación, ni el tiempo de acceso, ni añadir más bits al formato de la instrucción
  - Se duplica el ancho de banda de registros sin añadir más puertos
- **Desventajas**:
  - A veces hay que transferir información entre los dos bancos de registros
  - Se limita a priori el número de registros de cada tipo

# Riesgos en punto flotante

- El cauce podría verse así:



# Riesgos en punto flotante

## Riesgo estructural por escritura en banco de registros PF

- Si asumimos que el banco de registros PF tiene un puerto de escritura, una secuencia de operaciones en PF, o cargas de PF junto con operaciones PF, pueden causar conflictos en el uso del puerto de escritura del banco de registros PF

| Instrucción | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | **10** | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MULTD F0,F4,F6 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | **WB** | |
| ..... | | IF | ID | EX | Mem | WB | | | | | |
| ..... | | | IF | ID | EX | Mem | WB | | | | |
| ADDD F2,F4,F6 | | | | IF | ID | A1 | A2 | A3 | A4 | **WB** | |
| ..... | | | | | IF | ID | EX | Mem | WB | | |
| LD F2, 0(R2) | | | | | | IF | ID | EX | Mem | **WB** | |
| | | | | | | | | | | | |

- Con un único puerto de escritura en el banco de registros PF, la máquina debe **"serializar"** la finalización de las instrucciones

# Riesgos en punto flotante

- Podríamos **incrementar el número de puertos**, pero esta opción sería poco utilizada y complica mucho el hardware del banco de registros
- Para tratar este riesgo hay que detectar en qué ciclo de reloj va a usar el puerto de escritura del banco de registro cada instrucción y si varias instrucciones coinciden, **sólo emitir una de ellas** y detener el resto:
  - Puede implementarse mediante un **vector de bits de reserva** del ciclo para acceso al recurso
  - Al entrar la instrucción mira si estará libre el recurso (su bit a 0) y lo pondrá a 1 reservándolo. Si está ya a 1 nos detenemos un ciclo. Cada ciclo el vector se desplaza a la izquierda un bit

# Riesgos en punto flotante

## Riesgos WAW

- Los riesgos WAW sólo ocurren cuando se ejecuta una instrucción "inútil"
- Aunque un buen compilador nunca generaría dos escrituras en el mismo registro sin lecturas intermedias, puede haber situaciones inesperadas:
  - Entre una instrucción del programa y otra de la rutina de atención a una excepción.

# Riesgos en punto flotante

- Hay dos posibles formas de manejar los **riesgos WAW**:
  - Detectarlos en **ID** e insertar ciclos de parada:

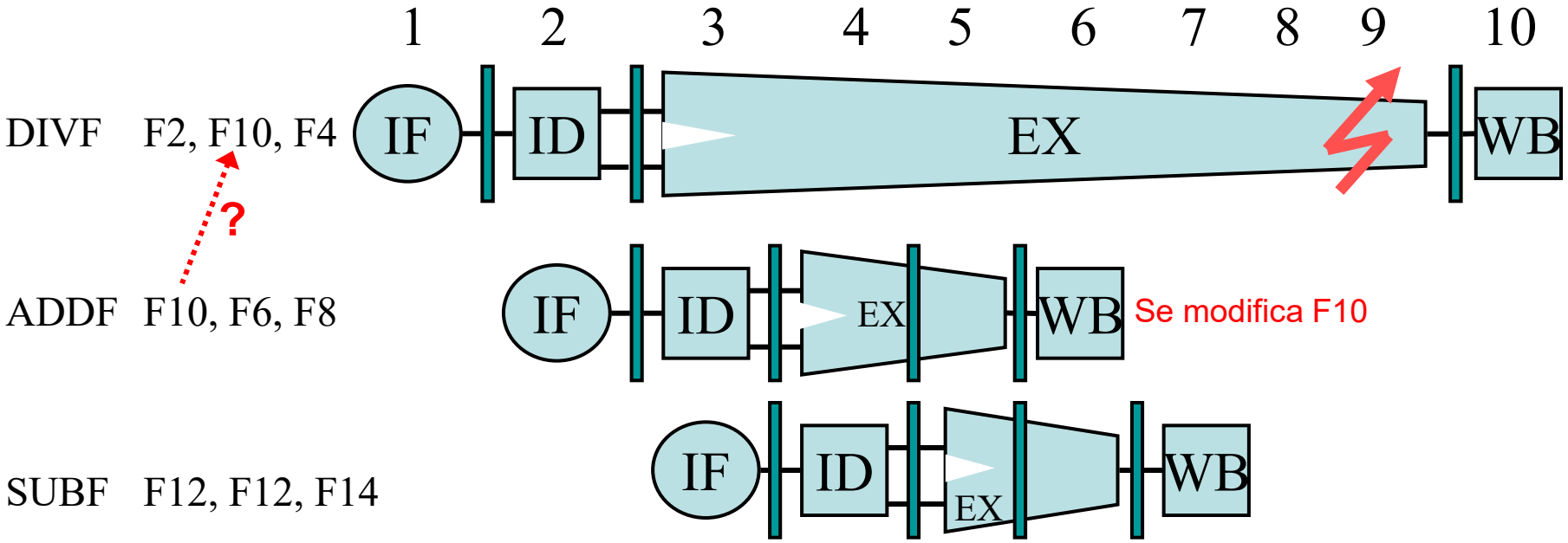
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MULTD **F0**, F2, F4 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | **WB** | |
| ADDD **F0**, F10,12 | | IF | ID | **ID** | **ID** | **ID** | A1 | A2 | A3 | A4 | **WB** |

  - Detectarlo en ID y "anular" la primera instrucción convirtiéndola en burbuja con el fin de que no llegue a escribir su resultado:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| MULTD **F0**, F2, F4 | IF | ID | ~~M1~~ | ~~M2~~ | ~~M3~~ | ~~M4~~ | ~~M5~~ | ~~M6~~ | ~~M7~~ | ~~**WB**~~ |
| ADDD **F0**, F10,12 | | IF | ID | A1 | A2 | A3 | A4 | **WB** | | |

# Riesgos en punto flotante

- Surge el problema de que una instrucción posterior puede terminar antes que otra anterior: esto da lugar a tener **excepciones imprecisas** en lugar de excepciones precisas




- El estado de la máquina podría quedar alterado. Ocurre porque estamos dejando que las instrucciones terminen en diferente orden a como fueron emitidas: **terminación fuera de orden**

# Riesgos en punto flotante

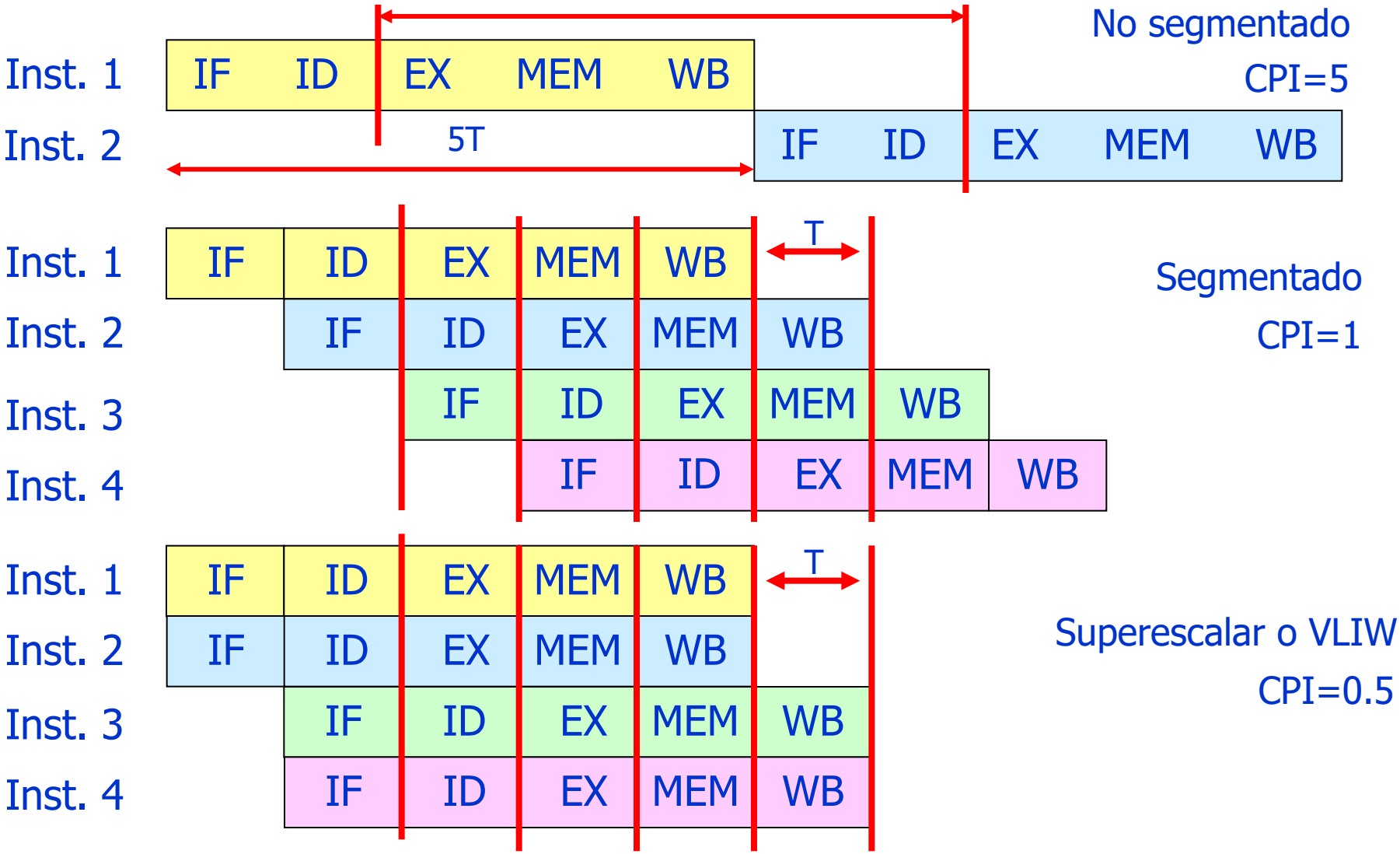
- Podemos ignorar el problema y conformarnos con **excepciones imprecisas**
  - Opción típica en los años 70 y 80
  - Nuestro procesador DLX implementa esta opción
- Hoy en día, sin embargo, la mayoría de procesadores impiden que una instrucción cambie el estado de la máquina si hay instrucciones previas que aún no han acabado:
  - Para ello las instrucciones realizan sus escrituras en una cola en orden (conocida como *ReOrder Buffer,* ROB) y de ahí se pasan al banco de registros (o memoria) en orden de programa
  - Se verá en 3º en la asignatura de AOC

# Índice

# Emisión de múltiples instrucciones

- Hasta ahora, el objetivo de la segmentación ha sido el de conseguir el **CPI = 1** (es decir, el CPI ideal)
  - ¿Por qué limitar a una instrucción por ciclo?
- Nuevo objetivo: **CPI < 1** (ó IPC > 1)
  - **SOLUCIÓN**: Lanzar y ejecutar simultáneamente **varias instrucciones** por ciclo
  - Permite mejorar aún más el rendimiento ⇒ más ILP
- ¿Tenemos recursos?
  - Habrá más presión sobre la memoria y registros
  - Habrá más posibilidad de los riesgos (estructurales) y dependencias (datos y control)
    - Técnicas para resolver dependencias de datos (ejecución fuera de orden)
    - Técnicas para resolver dependencias de control (especulación)
  - Necesario más área de silicio ⇒ mayor consumo energético

# Emisión de múltiples instrucciones



# Emisión de múltiples instrucciones

- Este tipo de procesadores se denominan **Procesadores Superescalares**
  - Se estudiarán en la asignatura de 3º (AOC)
- **Características de los superescalares**
  - Lanzan un número variable de instrucciones por ciclo (de 0 a 12) *¿tiene sentido emitir más?*
  - Planificación
    - **Estática** por el compilador
    - **Dinámica** por el hardware del procesador
  - Reglas de **ejecución**
    - **En orden**: usado en las primeras generaciones (Alpha 21164, Sun UltraSPARC III, …)
    - **Fuera de orden + especulación**: Intel (Kaby Lake, Sunny Cove), IBM (Power9, Power10), AMD (Zen series)