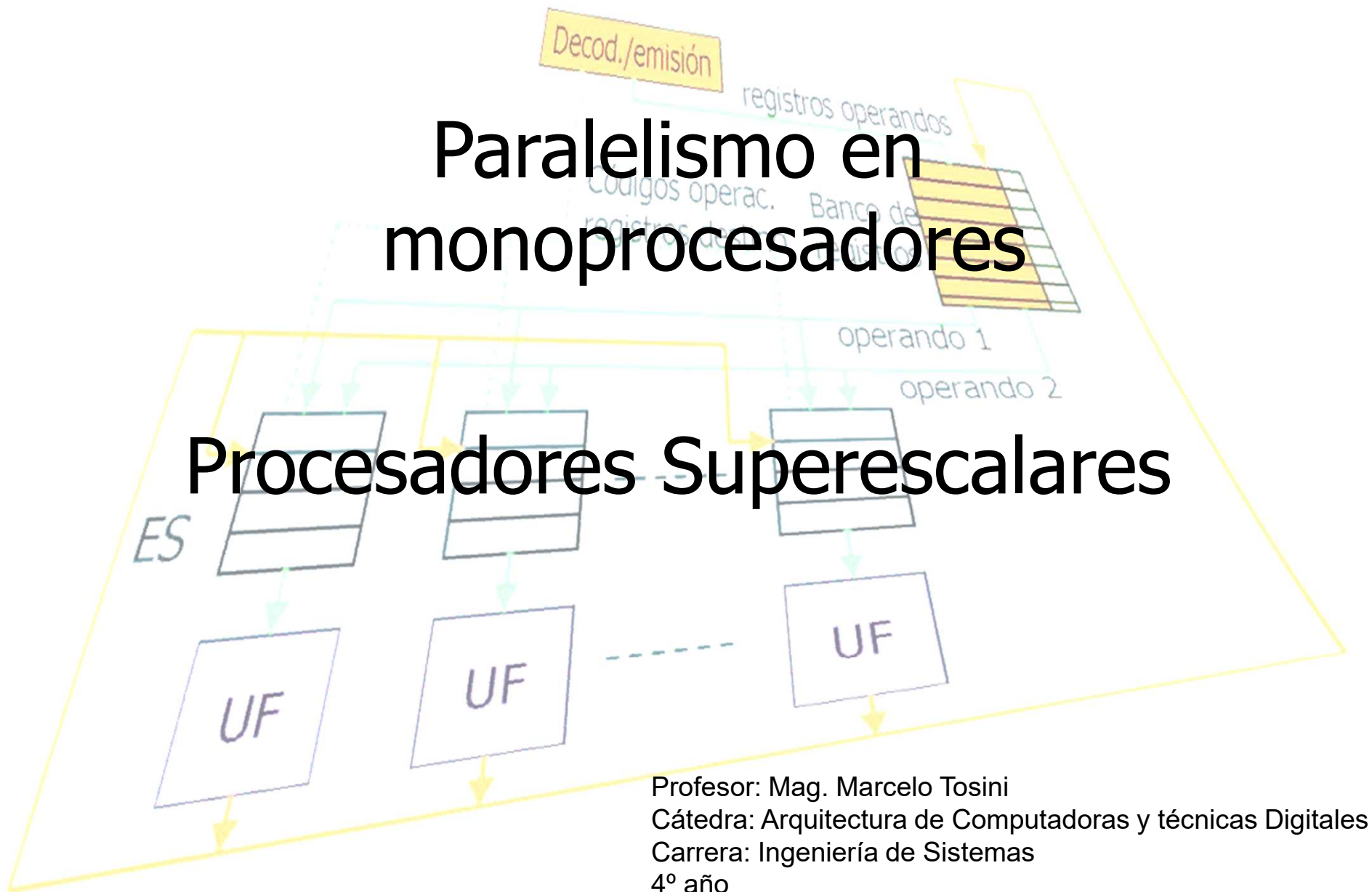


# Paralelismo en monoprocesadores

## Procesadores Superescalares



Profesor: Mag. Marcelo Tosini  
Cátedra: Arquitectura de Computadoras y técnicas Digitales  
Carrera: Ingeniería de Sistemas  
4º año

# Procesadores Superescalares

---

- **Descripción estructural:** Arquitectura y componentes de las distintas etapas de un procesador superescalar
- **Descripción funcional:** Funcionamiento de un procesador superescalar y descripción de las estructuras que permiten la normal ejecución del flujo de instrucciones de un programa

# Introducción

---

La técnica de segmentación de circuitos aplicada a la microarquitectura de un procesador es efectiva pero presenta algunas limitaciones que degradan el rendimiento.

Una solución superescalar permite obtener rendimientos mucho mayores que la solución puramente escalar a costa de un compromiso de diseño mas grande

Los pipelines superescalares traspasan los limites de la emisión simple de instrucciones de un pipeline escalar con rutas de datos mas elaboradas que trabajan con varias instrucciones a la vez

Para lograr lo anterior incorporan múltiples unidades funcionales que aumentan la concurrencia de ejecución de instrucciones

Además, los pipelines superescalares permiten la ejecución de instrucciones en un orden distinto al impuesto por el programa original

# Limitaciones de los pipelines escalares

---

Un procesador escalar se caracteriza por un pipeline lineal de  $k$  etapas

Todas las instrucciones, sin importar su tipo, atraviesan las mismas etapas del pipeline.

A excepción de las instrucciones frenadas por cuestiones de conflictos todas las instrucciones permanecen en cada etapa por un ciclo de reloj

Estos pipelines rígidos tienen 3 limitaciones destacadas:

1. El máximo rendimiento está limitado a una instrucción por ciclo
2. La unificación de todos los tipos en un solo pipeline genera un diseño ineficiente
3. El frenado de un pipeline rígido induce burbujas innecesarias con la consiguiente pérdida de ciclos de reloj

# rendimiento de los pipelines escalares

(limitación 1: Máximo rendimiento limitado a una instrucción por ciclo)

El rendimiento de un pipeline escalar

$$\text{Rendimiento} = \frac{1}{\text{recuento\_inst}} * \frac{\text{inst}}{\text{ciclo}} * \frac{1}{\text{tiempo\_ciclo}} = \frac{\text{IPC} * \text{frecuencia}}{\text{recuento\_inst}}$$

puede incrementarse aumentando el número de instrucciones por ciclo o la frecuencia o decrementando el número de instrucciones ejecutadas

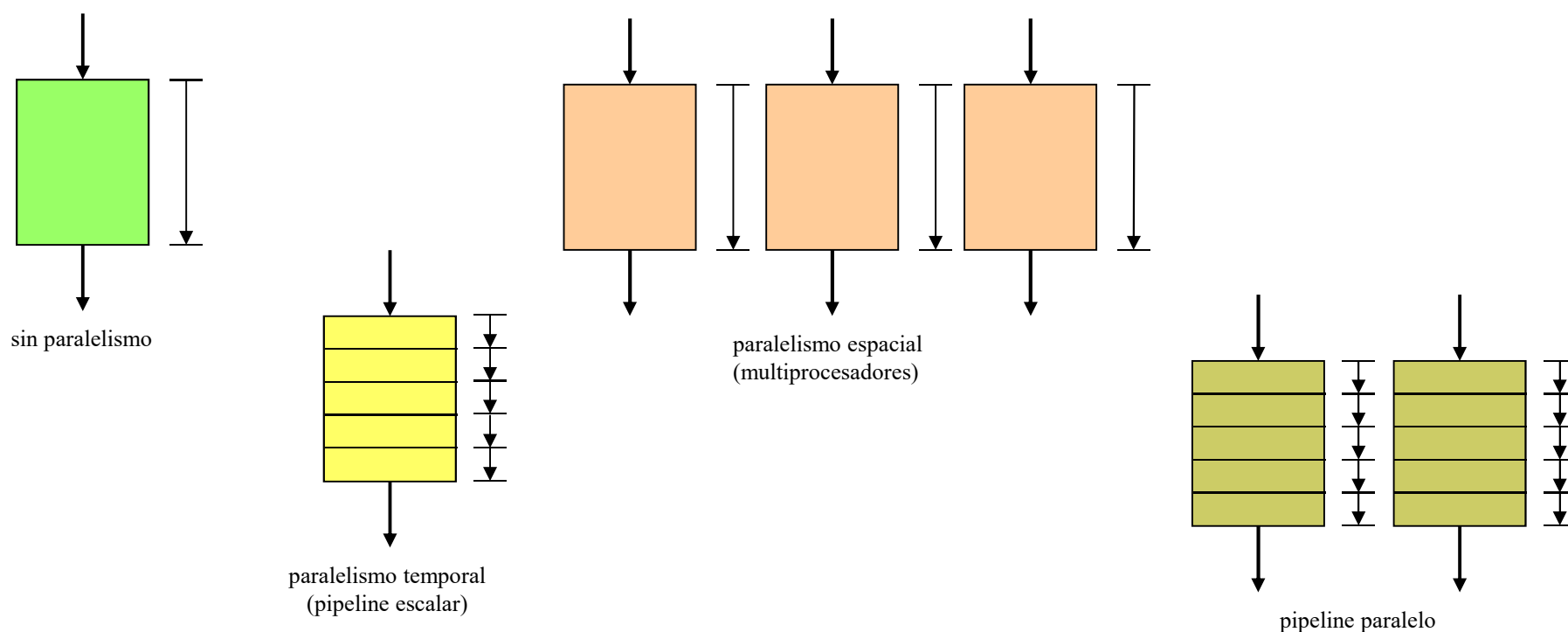
consideraciones:

- La frecuencia puede aumentarse incrementando la profundidad del pipeline reduciendo, en consecuencia el tiempo de ciclo
- Un aumento en la profundidad tiene la desventaja adicional de perdida excesiva de ciclos ante dependencias
- Un pipeline escalar solo puede iniciar la ejecución de (como máximo) una sola instrucción por ciclo de reloj: IPC=1

# Pipelines paralelos

Lograr un  $IPC > 1$  requiere que el procesador sea capaz de iniciar el proceso de más de una instrucción en cada ciclo

Esto requiere incrementar el ancho del pipeline de modo que sea capaz de tener mas de una instrucción en cada etapa a cada instante



# Pipelines diversificados (especializados)

(limitación 2: Unificación de tipos genera diseño ineficiente)

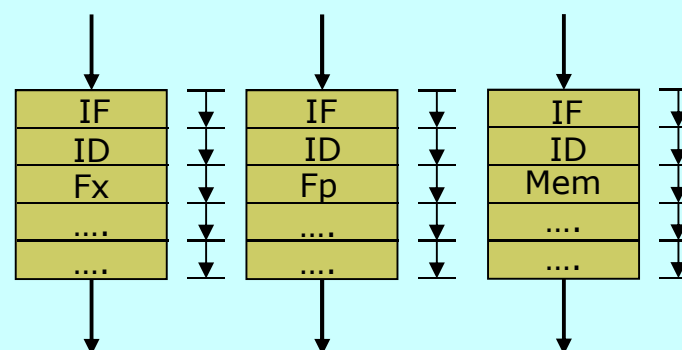


1 ciclo   1 ciclo   > ciclo >> ciclo   1 ciclo

El concepto de pipeline paralelo remite a la simple replicación espacial del hardware de un pipeline escalar

Un pipeline escalar clásico se compone de etapas con funcionalidad y tiempos de cálculo diferentes

Una mejora al diseño es la separación de las operaciones diferentes en distintas unidades funcionales especializadas, de modo que entonces cada unidad funcional es ahora mas simple y mas rápida



pipeline diversificado

# Pipelines estáticos y dinámicos

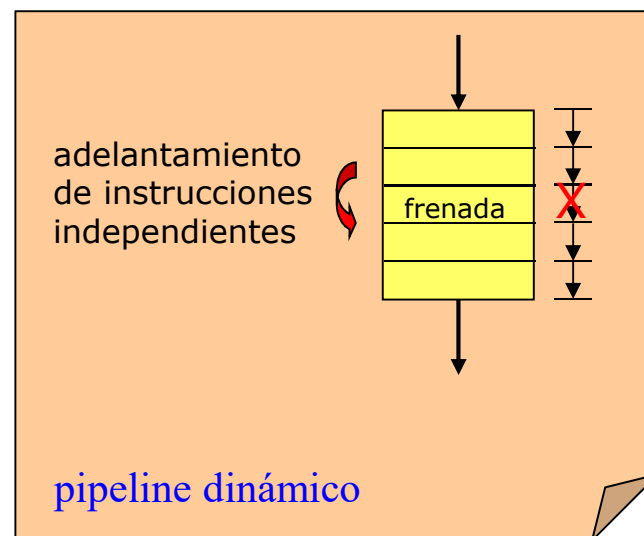
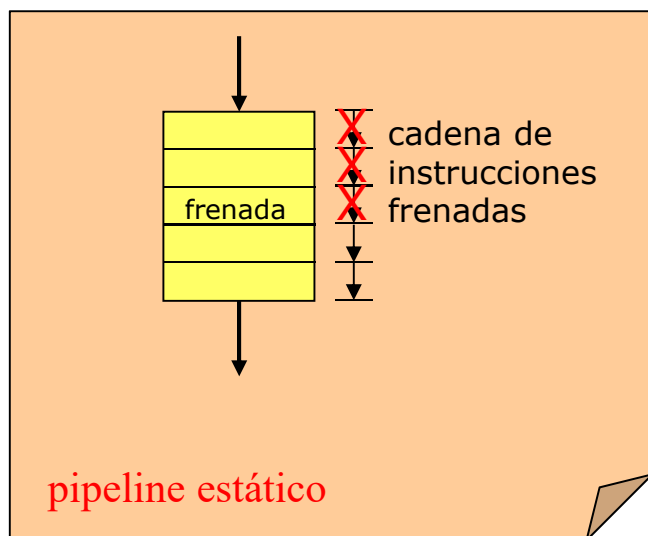
(limitación 3: Frenado induce burbujas con pérdida de ciclos de reloj)

Un pipeline escalar es rígido ya que las instrucciones entran y avanzan en él paso a paso y atravesando todas las etapas

Las instrucciones entran al pipeline de acuerdo al orden impuesto por el programa (*en orden*)

En un **pipeline estático** una instrucción frenada en una etapa  $i$  frena la ejecución de todas las instrucciones en etapas previas (1, 2, ...,  $i-1$ )

Una solución es permitir que las instrucciones frenadas que no tengan ningún tipo de dependencia puedan adelantarse a la instrucción frenada. Este tipo de pipeline se denomina **pipeline dinámico** y pueden realizar ejecución *fuera de orden*





# Paso de un pipeline escalar a superescalar

Estructural

Los pipelines superescalares son descendientes naturales de los escalares que solucionan en mayor o menor medida las 3 limitaciones de estos

Los pipelines superescalares...

- **son pipelines paralelos:**

pueden iniciar la ejecución de varias instrucciones en cada ciclo de reloj

- **son pipelines diversificados:**

ya que emplean múltiples unidades funcionales heterogéneas

- **son pipelines dinámicos:**

poseen la capacidad de reordenar la secuencia de instrucciones impuesta por el compilador a fin de aumentar la eficiencia

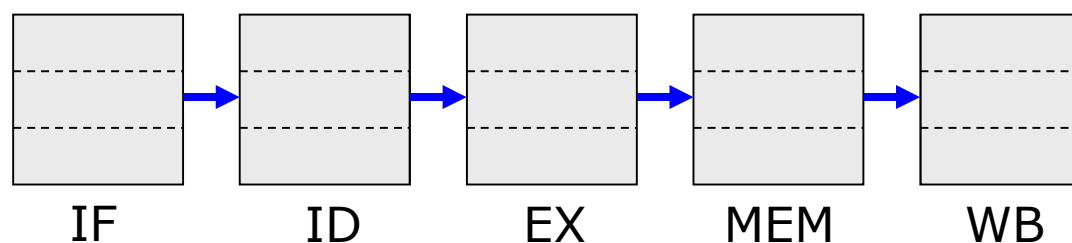
# pipelines superescalares

un pipeline escalar de  $k$  etapas puede procesar hasta  $k$  instrucciones simultáneamente

Su rendimiento se compara con procesadores sin paralelismo y se espera que la mejor aceleración obtenible sea del orden de  $k$

un pipeline superescalar usualmente se compara con la versión escalar y es determinado por el ancho del mismo o grado de paralelismo espacial

Un pipeline de ancho  $s$  puede procesar hasta  $s$  instrucciones simultáneas en cada una de sus etapas, alcanzando, en el mejor caso, una aceleración de  $s$



Pipeline paralelo de ancho  $s = 3$

# pipelines diversificados

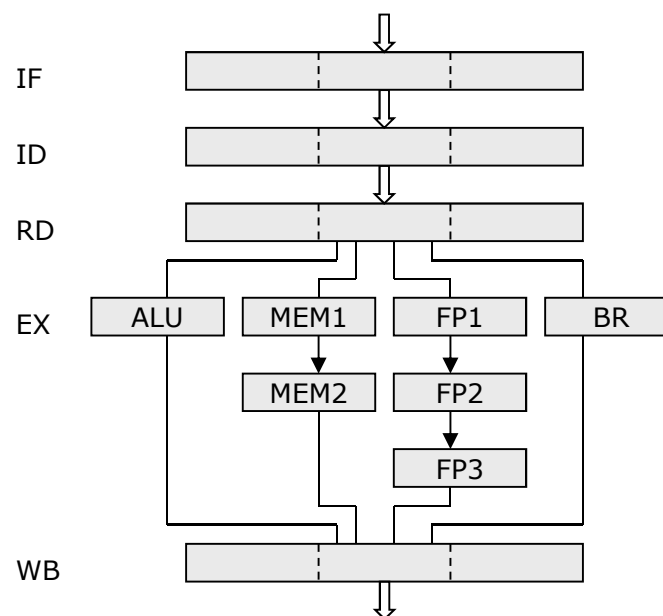
Los recursos de hardware necesarios para la ejecución de diferentes operaciones pueden variar significativamente

Tipos de operaciones: Punto fijo, Punto flotante, acceso a memoria

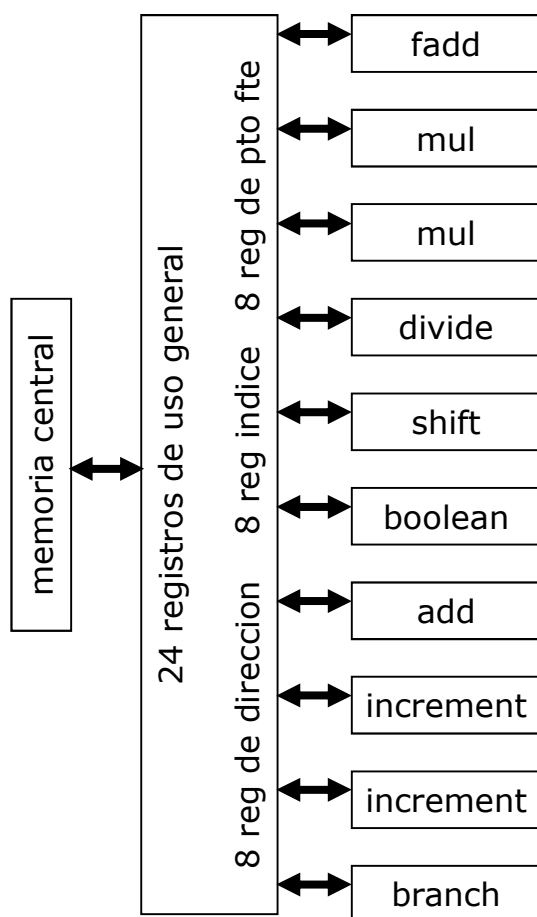
**Pipelines diversificados: implementación de las diferentes operaciones en distintas unidades funcionales**

Ventajas:

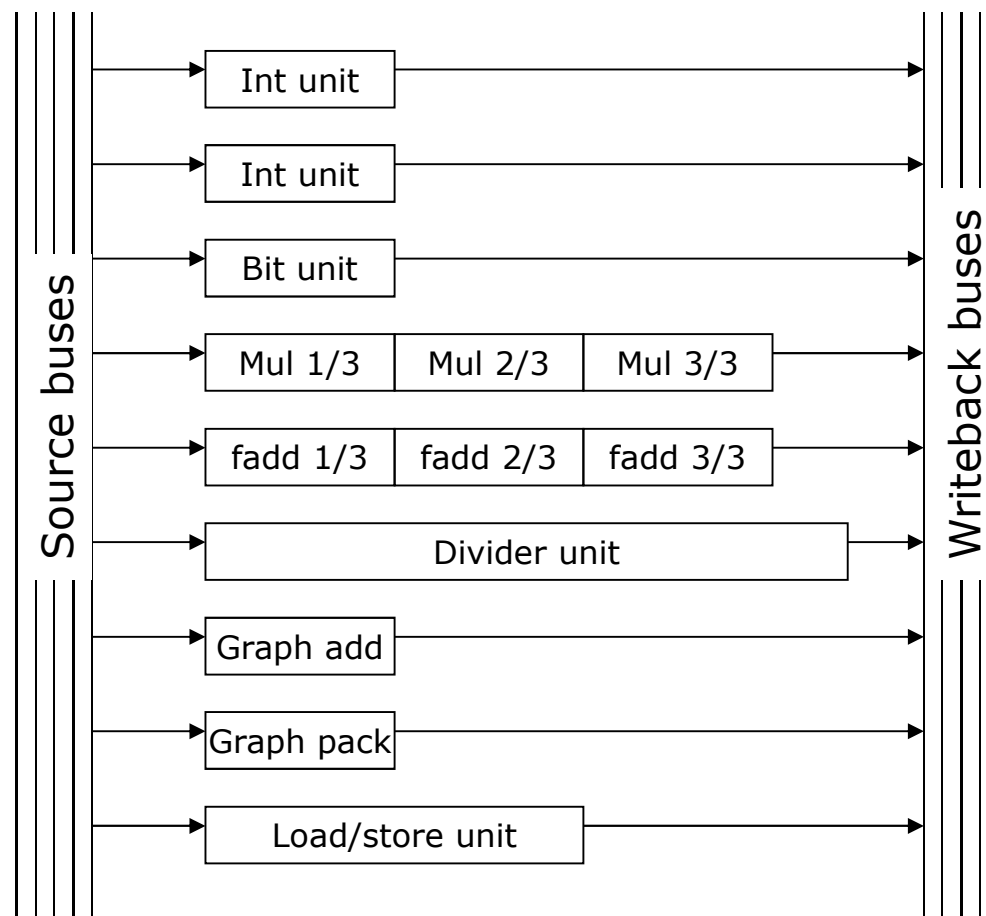
- cada pipe puede ser especializado en un tipo particular de operación
- cada sub-pipe de ejecución puede tener el número justo de etapas
- instrucciones de un sub-pipe no se frenan por dependencias en otros pipes



# pipelines diversificados (ejemplos)

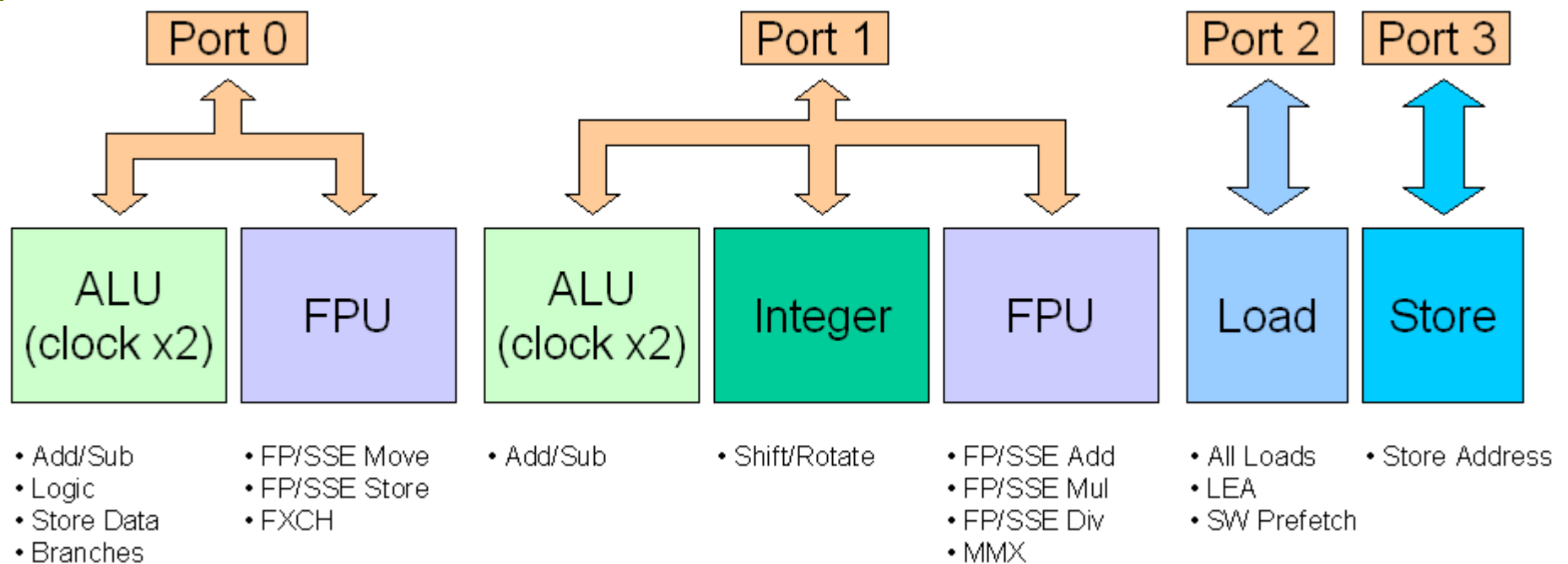


CDC 6600 (1964)



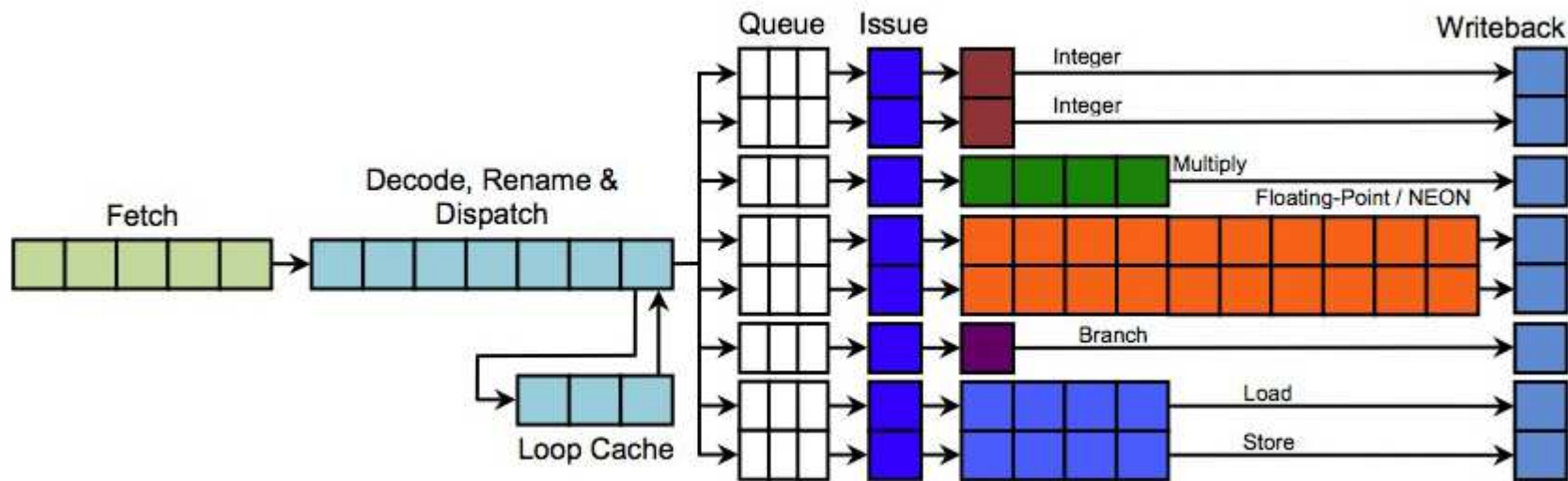
Motorola 88110 (1982)

# pipelines diversificados (ejemplos)



Pentium 4 (2005)

# pipelines diversificados (ejemplos)



ARM Cortex-A15 (2010)

# pipelines dinámicos

## Pipeline rígido:

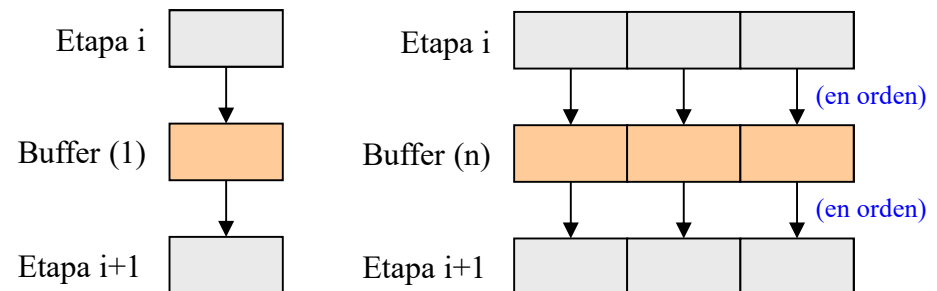
- 2 etapas  $i$  e  $i+1$  se separan y conectan por medio de un buffer simple (1 instr.)
- el buffer se activa en cada ciclo de reloj
- En caso de conflictos (stall de etapa  $i+1$ ) el buffer no se actualiza
- En caso de frenado todas las etapas  $1, \dots, i+1$  se frenan
- Las instr. entran y salen del buffer en el orden del programa

## Pipeline paralelo:

- 2 etapas  $i$  e  $i+1$  se separan y conectan por medio de un buffer de  $n$  instr
- En caso de conflicto de 1 sola instr. el buffer no se actualiza
- En caso de frenado todas las etapas  $1, \dots, i+1$  se frenan
- Las instr. entran y salen del buffer en el orden del programa
- Cada entrada del buffer esta conectada a cada etapa  $i$  y cada salida a una etapa  $i+1$

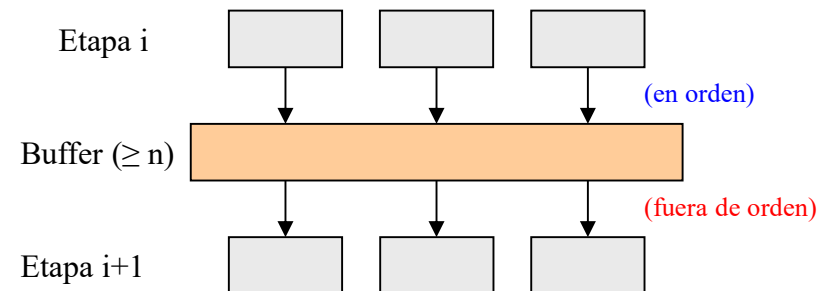
## Pipeline superescalar:

- Uso de buffers multientrada complejos
- Las instr. entran al buffer a cualquier posición del mismo
- Las instr. salen del buffer en cualquier orden con la única condición de tener todos sus operandos completos



Pipeline con  
buffer simple

Pipeline con  
buffer múltiple



Pipeline con  
buffer con reordenamiento

# pipeline paralelo diversificado

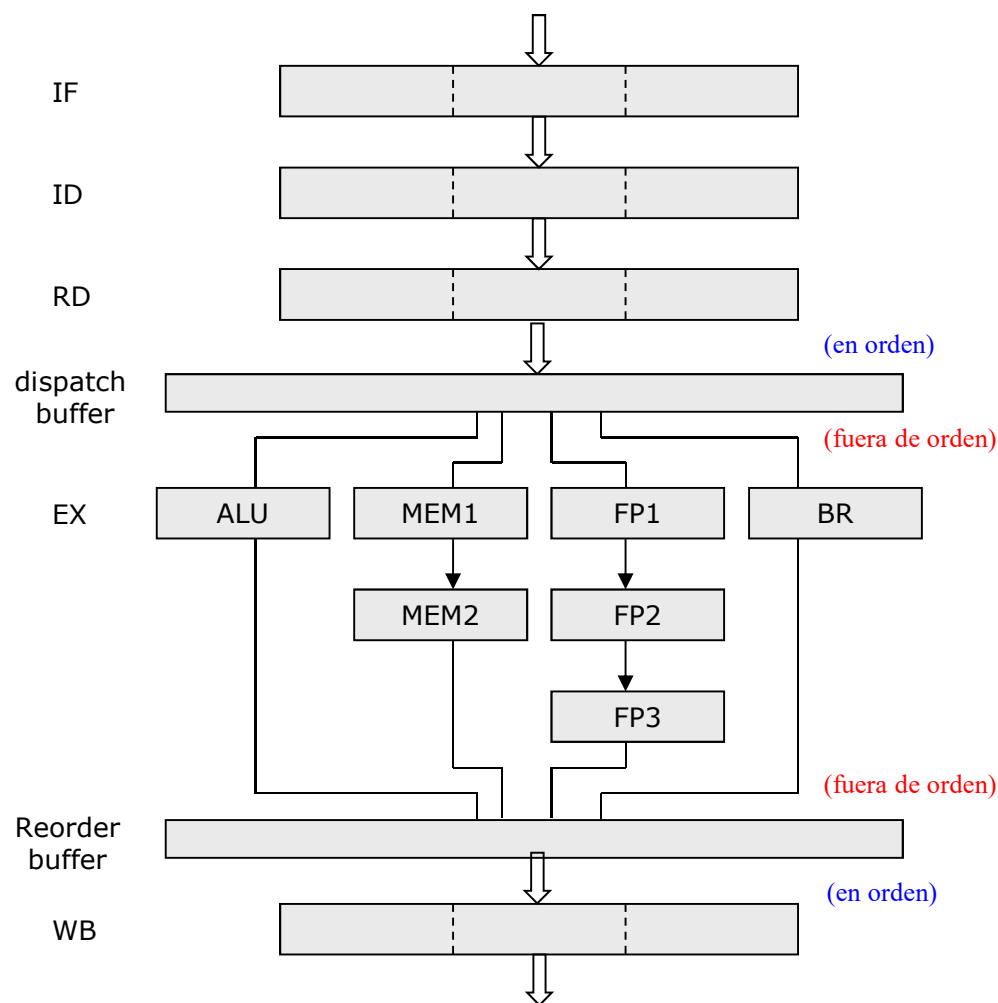
Pipeline segmentado de entre 5 y 8 etapas (según tipo de instrucción)

las primeras 3 etapas de ancho  $s = 3$  y ejecución en orden

La etapa de ejecución soporta hasta 4 rutas independientes (Fx, Fp, Mem y Branch)

En la etapa de ejecución puede haber hasta 7 instrucciones ejecutándose

La ejecución es en desorden, por lo tanto es necesario un buffer adicional para volver a ordenar las instrucciones según el orden original del programa

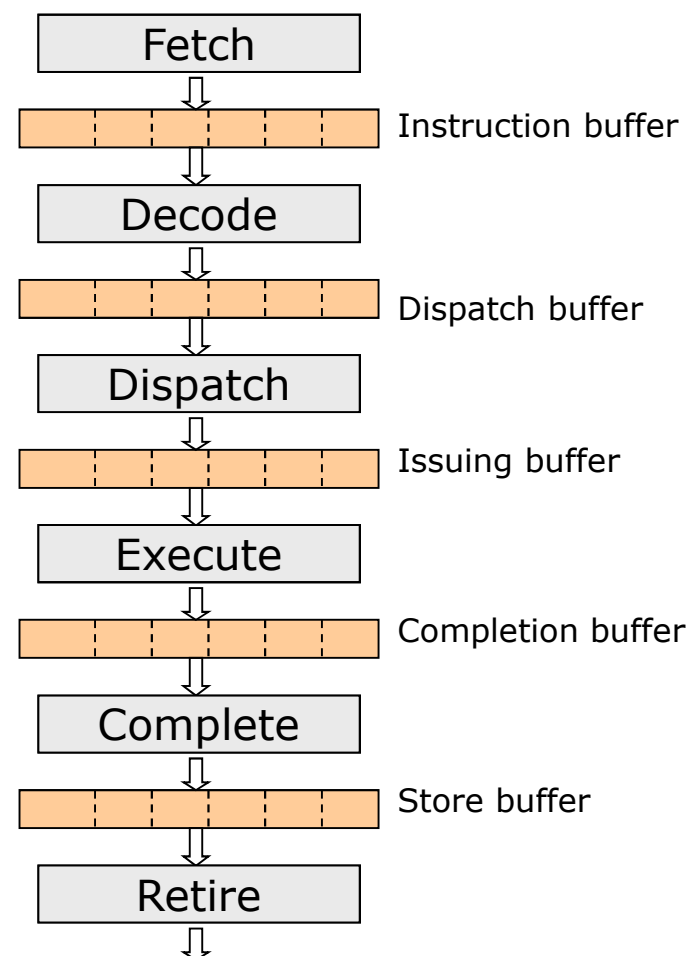




# Organización básica de un superescalar

## Organización genérica de un pipeline Superescalar de 6 etapas:

- **Fetch:** lectura de múltiples instrucciones
- **Decode:** decodificación de múltiples instrucciones
- **Dispatch:** distribuye instrucciones a las diferentes unidades funcionales
- **Execute:** incluye múltiples unidades funcionales especializadas de latencia variable
- **Complete:** reordena las instrucciones y asegura la actualización *en orden* del estado de la máquina
- **Retire:** salida de la instrucción del procesador



# Captura de instrucciones (fetching)

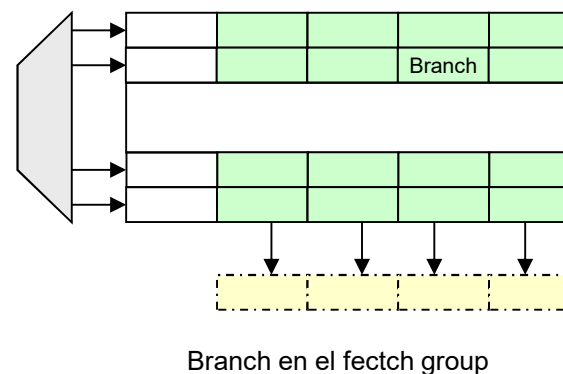
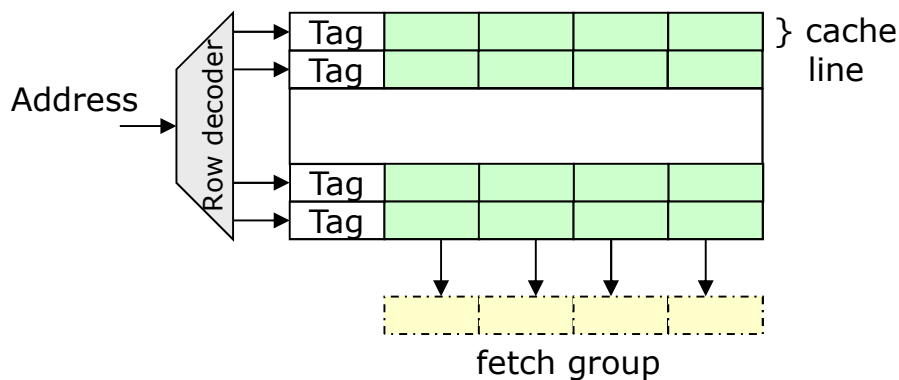
Un pipeline superescalar de grado  $s$  debe ser capaz de leer  $s$  instrucciones de la I-cache (Instruction caché) en cada ciclo de reloj (*fetch group*).

Para mayor rendimiento la caché debe organizarse de modo que

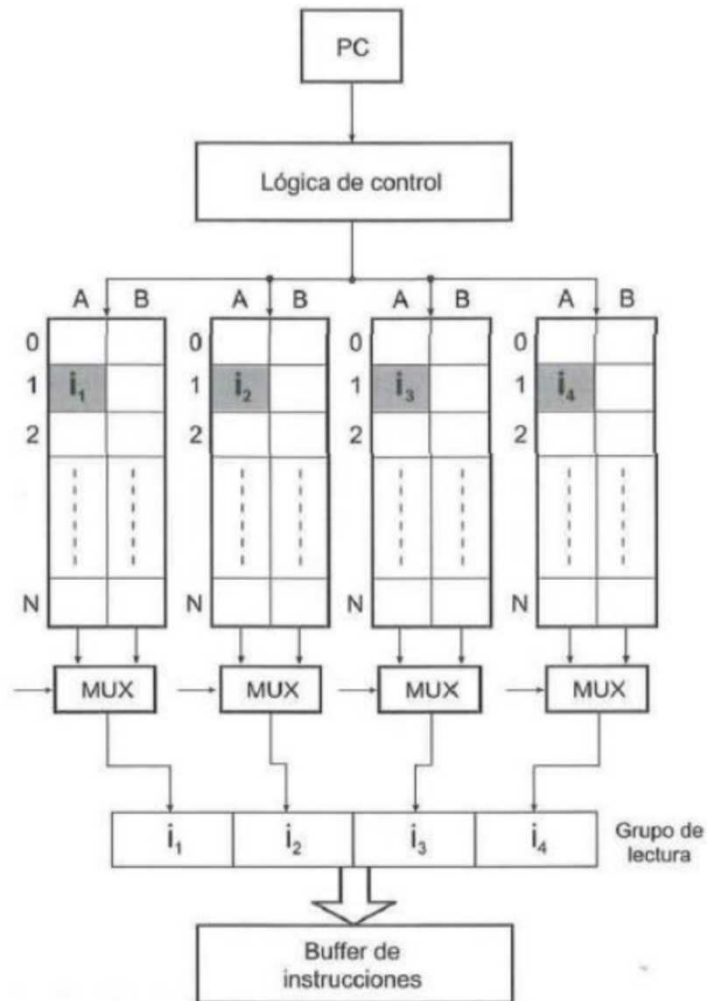
- \* cada fila de caché tenga  $s$  instrucciones y
- \* cada fila pueda leerse en un solo ciclo de reloj

Problemas potenciales:

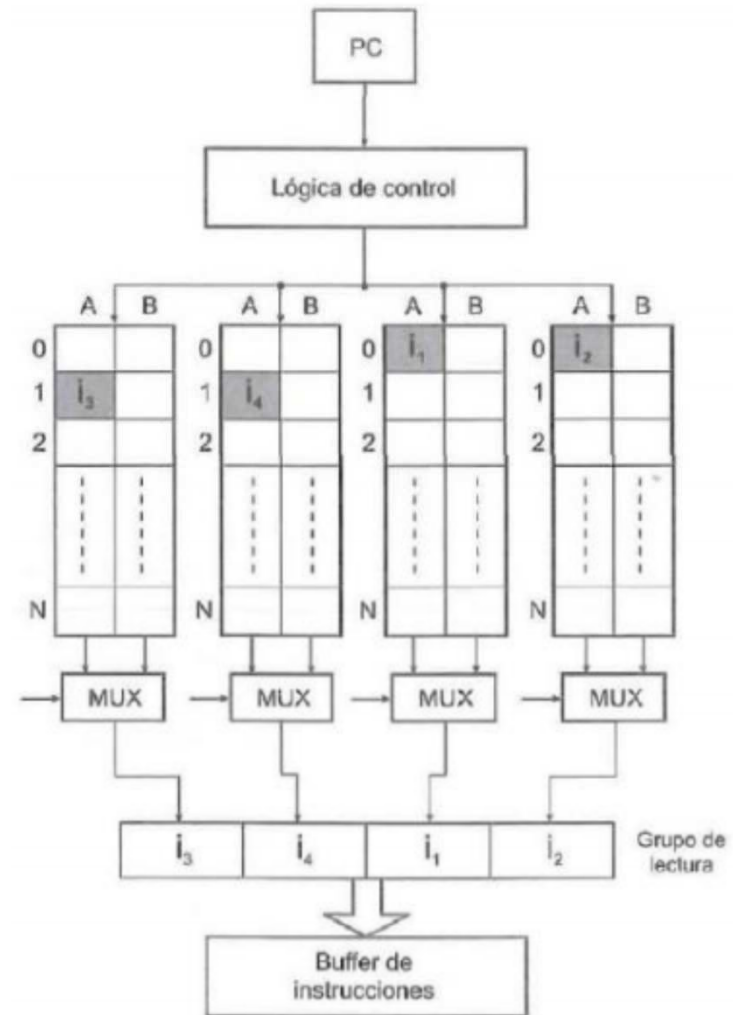
- *fetch group* y *Caché row* desalineadas
- Presencia de instrucciones de salto en el *fetch group*



# fetch group y Caché row desalineadas



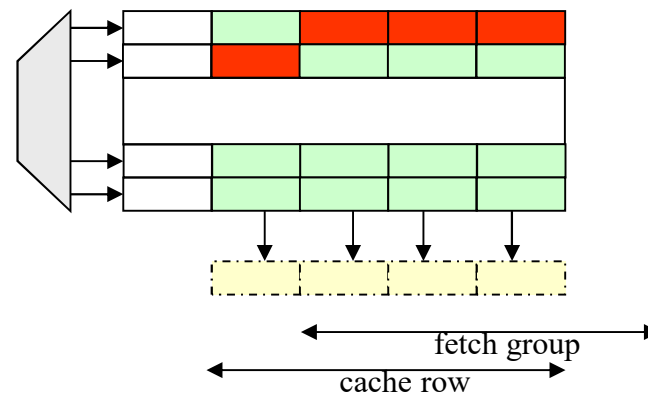
Alineadas



NO Alineadas

# fetch group desalineado (fetching)

Problema: El comienzo del fetch group y la cache row no son coincidentes



## 2 soluciones factibles:

### 1.- ordenamiento estático en tiempo de compilación:

- El compilador conoce la organización de la I-cache y genera las direcciones de salto de las instrucciones alineadas al comienzo de las filas de la cache.
- 2 desventajas:
  - desperdicio de espacios de memoria de código
  - código objeto generado dependiente de una organización de cache particular

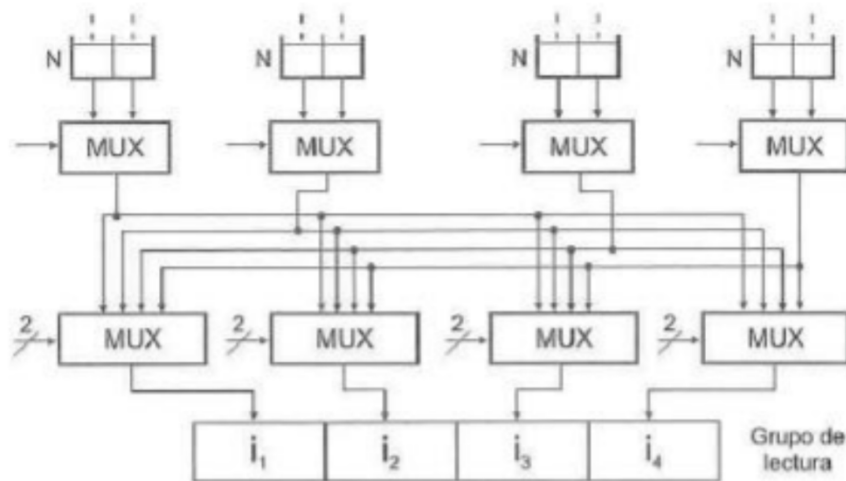
### 2.- Uso de hardware especializado que resuelve el problema en tiempo de ejecución

# fetch group y Caché row desalineadas

---

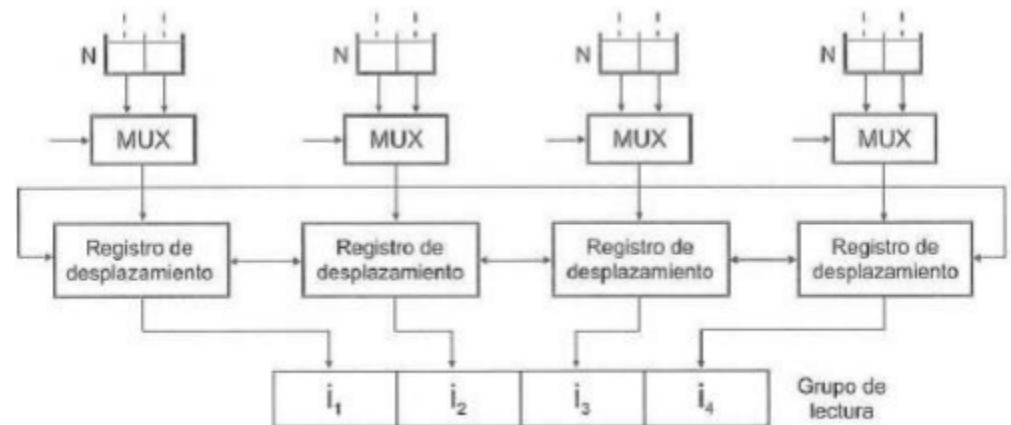
- Posibles soluciones
  - **Red de alineamiento:** Consiste en reubicar las salidas de la caché mediante multiplexores que conducen las instrucciones leídas a su posición correcta dentro del grupo de lectura.
  - **Red de desplazamiento:** Recurre a registros de desplazamiento para mover las instrucciones.
  - **Cola de prefetching o prelectura:** Técnica utilizada para minimizar el impacto de los fallos de lectura en la caché de instrucciones.

# fetch group y Caché row desalineadas



Multiplexado

Registros de desplazamiento



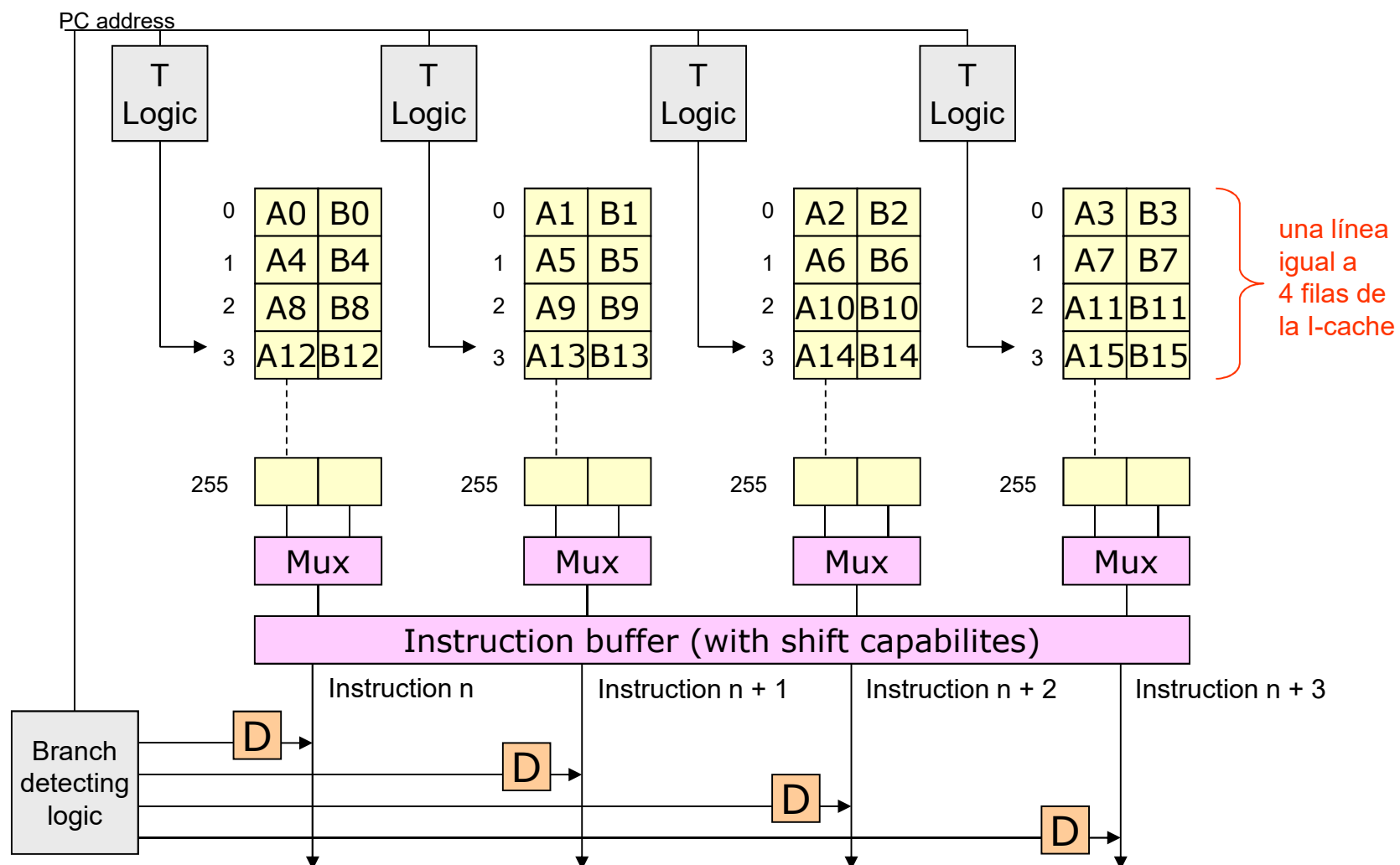
# Ejemplo IBM RS/6000

---

## La RS/6000 usa una I-cache

- Asociativa por conjuntos de 2 vías
- Con un ancho de línea de 32 instrucciones (128 bytes)
- Cada fila de la caché almacena 4 conjuntos asociativos de 2 instr c/u
- Cada línea de cache se compone de 4 filas de cache
- El arreglo físico de la I-cache esta realmente formado de 4 sub-arreglos (0, 1, 2, 3) accesibles en paralelo
- 1 instrucción puede ser accedida de cada sub-arreglo en cada acceso a la I-cache
- Las direcciones de instrucción están distribuidas con un interleave de 2

# Ejemplo IBM RS/6000





# Decodificación de instrucciones (decoding)

---

Estructural

## Tareas:

- Identificación de las instrucciones individuales
- Determinación de los tipos de instrucción
- Detección de dependencias entre las instrucciones dentro del *fetch group*

## dependiente de:

- El juego de instrucciones del procesador (Instruction Set Architecture (ISA))
- El ancho (s) del pipeline paralelo

# Decodificación de instrucciones (decoding)

---

Estructural

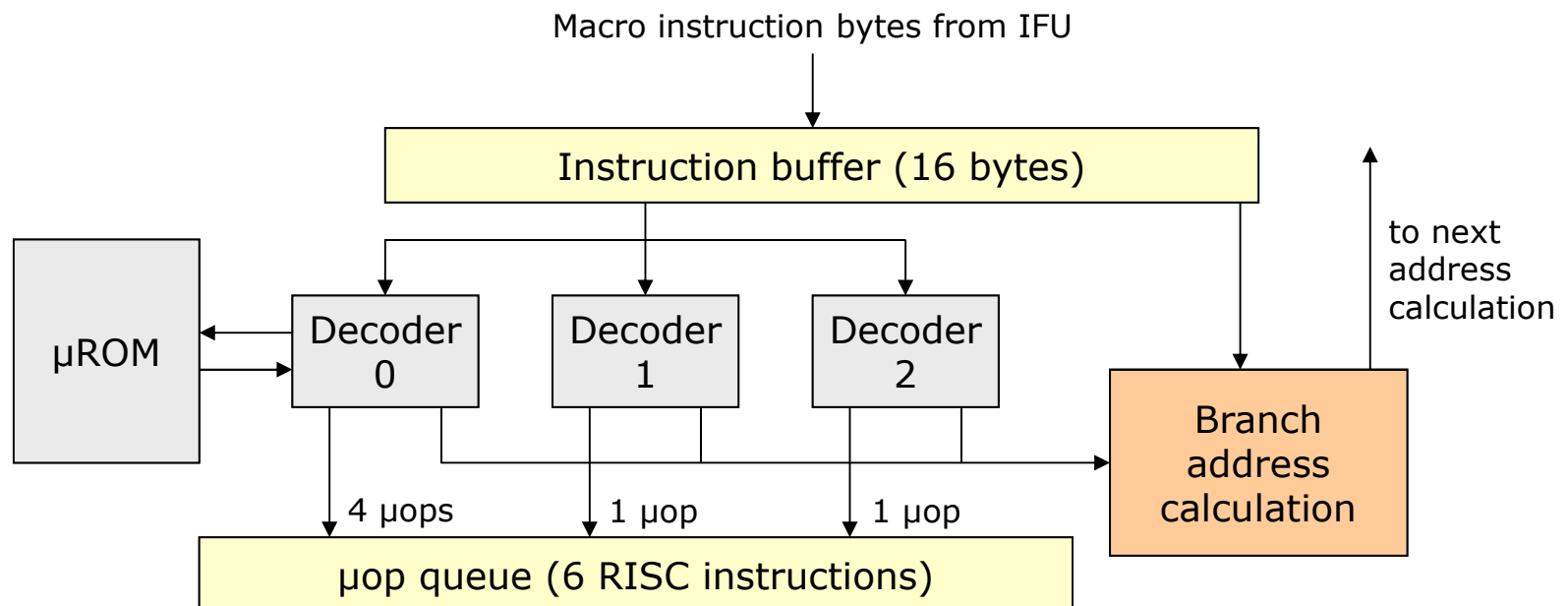
- Decodificación RISC
  - Clases de instrucciones
  - Tipos de recursos requeridos (registros)
  - Branch target address
  - Añade hasta 4 bits por instrucción
- Decodificación CISC
  - Más complejidad. Puede usar varias etapas del pipeline
  - Separación entre instrucciones
  - Localización de códigos de operación y operandos

# Decodificación de instrucciones (decoding)

Estructural

## Unidad de decodificación del Pentium P6

- la I-cache entrega 16 bytes a la cola de instrucciones
- 3 decodificadores trabajan en paralelo para decodificar instr del buffer
- Decoder 0 puede decodificar todas las instrucciones IA32
- Decoder 1 y 2 sólo decodifican instrucciones IA32 simples (reg to reg)



# Decodificación de instrucciones (decoding)

Estructural

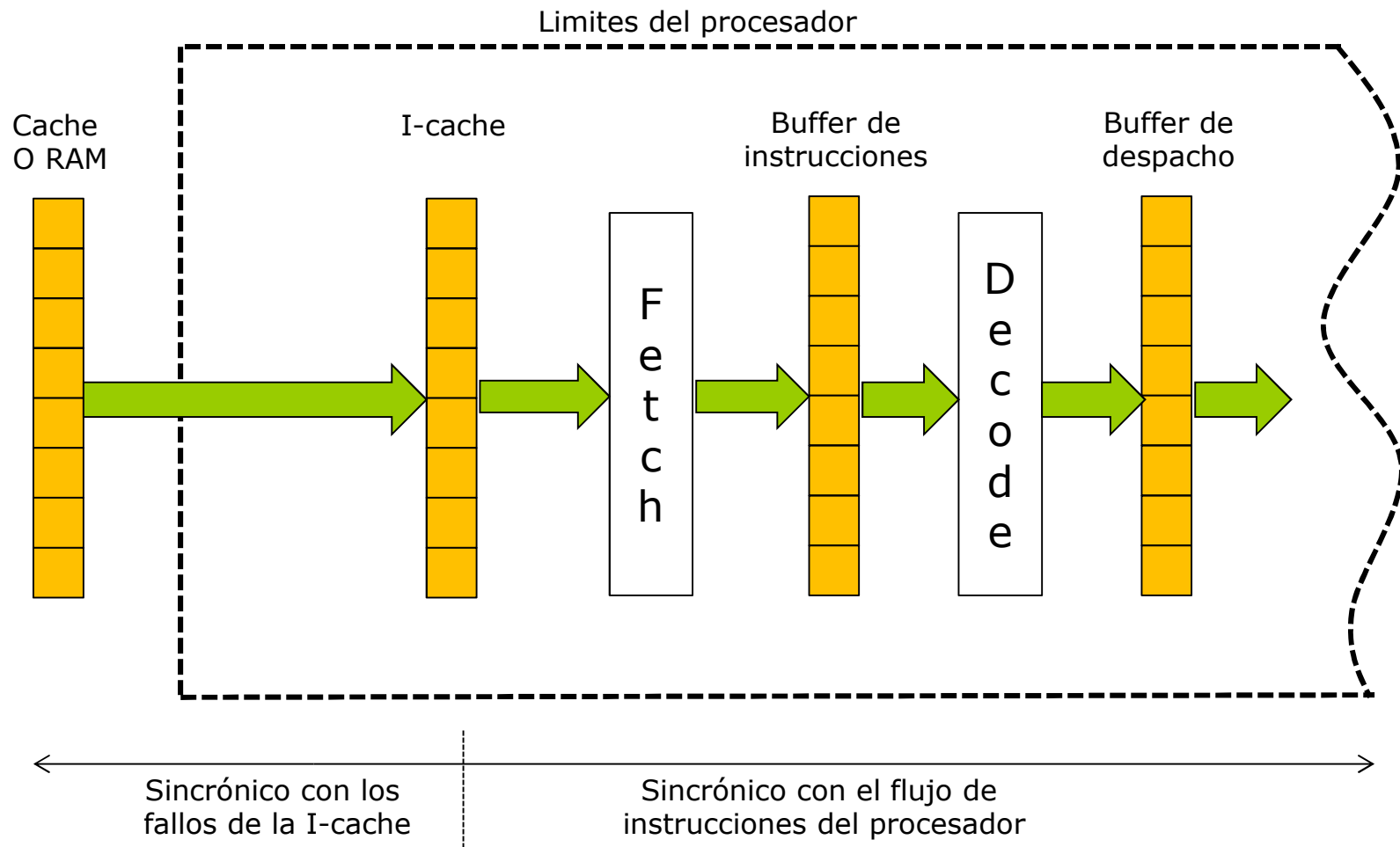
## Problemas:

- Para muchos superescalares con juego de instr CISC, el HW de decodificación puede ser muy complejo y requiere varias etapas de pipeline.
- Cuando el número de etapas de decodificación se incrementa, aumenta también la penalización de salto en términos de ciclos de reloj perdidos.

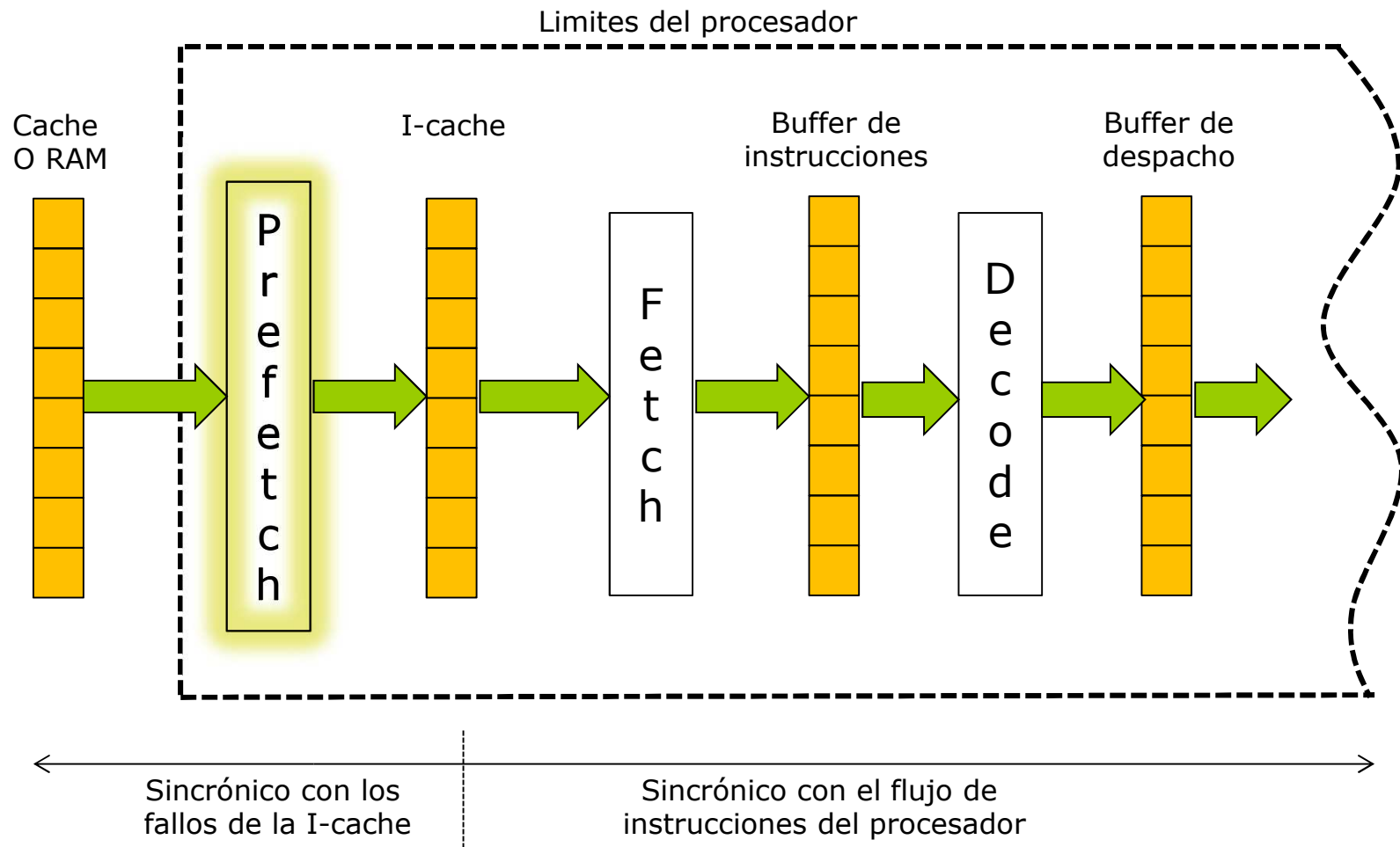
## Solución:

- Uso de técnica de *Predecodificación*, que mueve parte de la tarea de decodificación antes de la I-cache.
- Cuando ocurre un fallo de la I-cache, una nueva línea de cache es traída desde la memoria. Las instrucciones en esta línea son parcialmente decodificadas antes de almacenar la línea en la I-cache.
- A la línea original se le agrega cierta información adicional que ayuda a la decodificación posterior

# Predecodificación de instrucciones



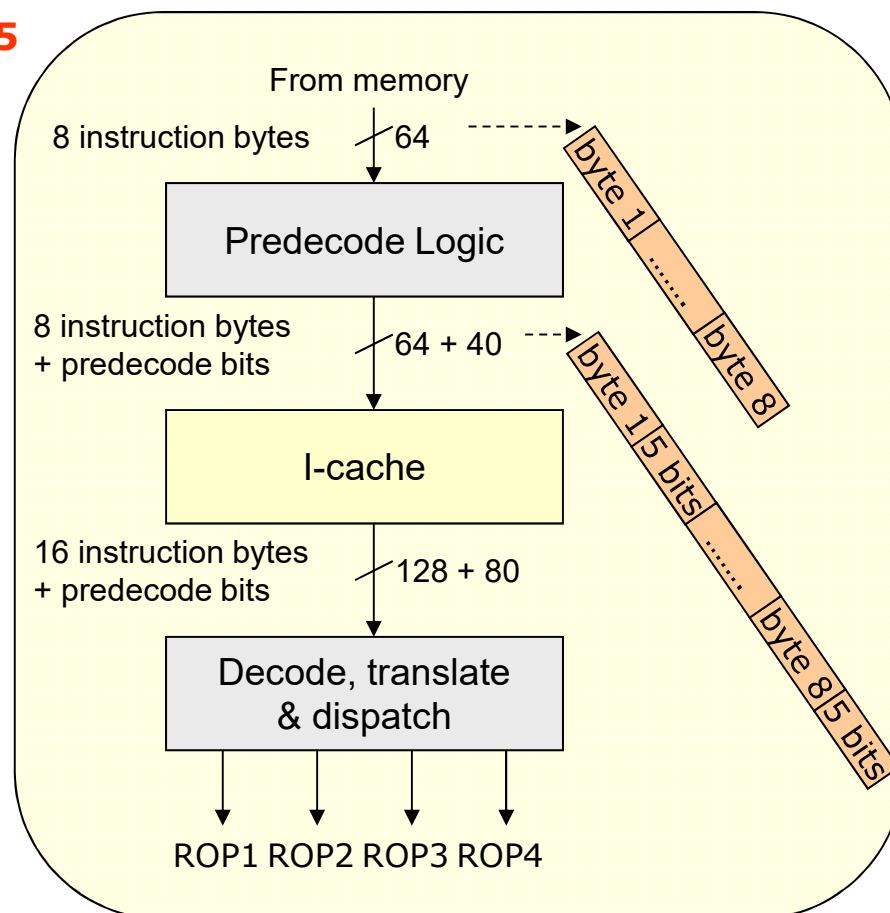
# Predecodificación de instrucciones



# Ejemplo

## Mecanismo de predecodificación del AMD K5

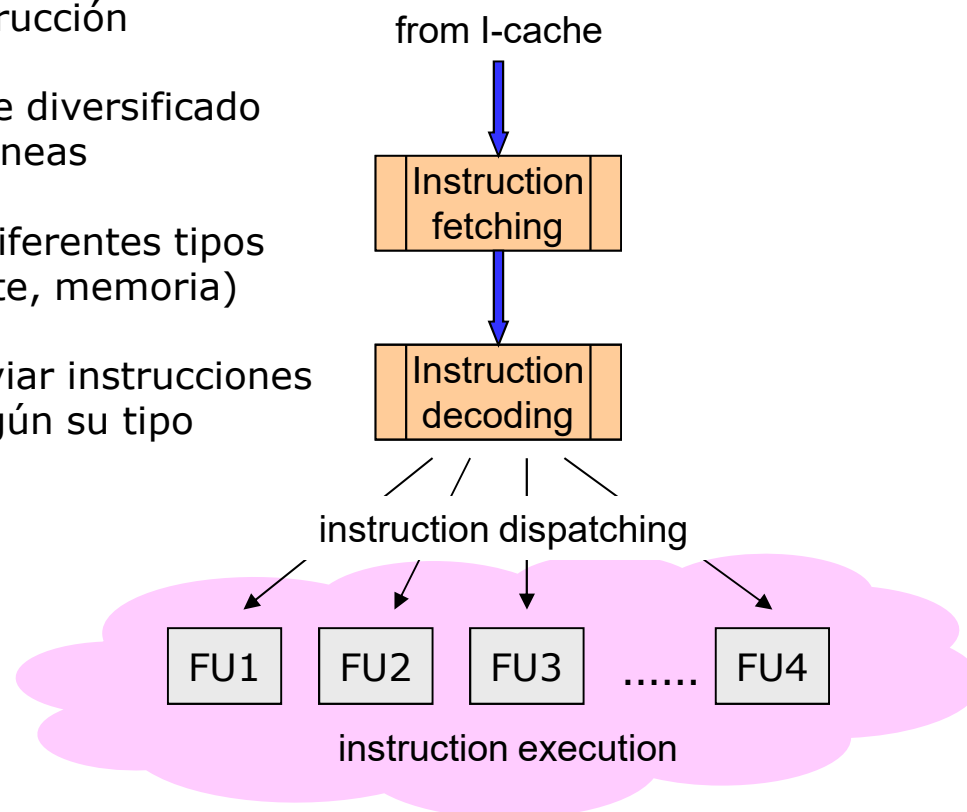
- Predecodificación agresiva del set IA32 antes de guardar las instrucciones leídas en I-cache
- 8 bytes de instrucciones en cada ciclo de bus
- la predecodificación agrega 5 bits por byte
  - inicio/fin de la instrucción
  - número de ROPs de la instrucción
  - ubicación de Codop y operandos
- Cada ciclo de predecodificación almacena 104 (64+40) bits en la I-cache
- el tamaño de línea de la I-cache es de 16 bytes de instrucciones + 80 bits



ROP: RISC Operation en terminología AMD

# Despacho de instrucciones (dispatch)

- En un procesador escalar todas las instr atraviesan el único pipeline sin importar el tipo de instrucción
- Un procesador superescalar es un pipeline diversificado con varias unidades funcionales heterogéneas
- Distintas unidades funcionales ejecutan diferentes tipos de instrucciones (punto fijo, punto flotante, memoria)
- La unidad de despacho se encarga de enviar instrucciones a la unidad funcional correspondiente según su tipo

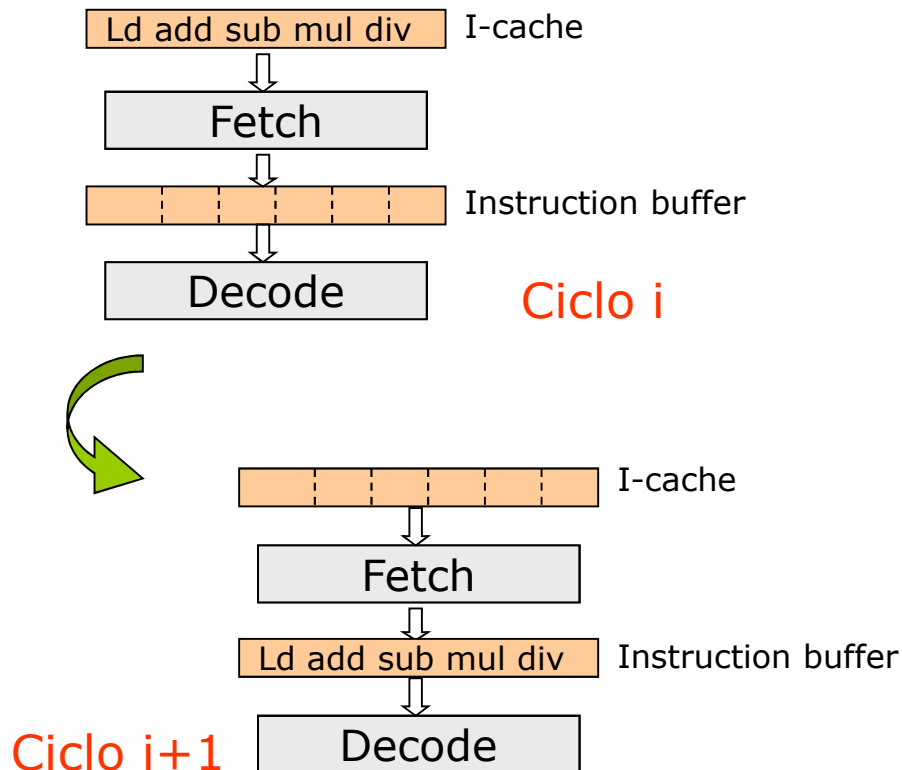




# Despacho de instrucciones (dispatch)

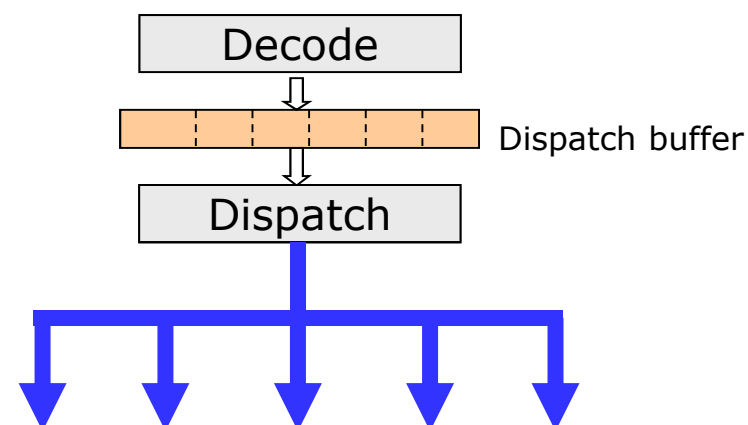
## Modo centralizado:

- Todas las instrucciones avanzan juntas entre etapas



## Modo distribuido:

- Las instrucciones se distribuyen en diferentes unidades funcionales según su tipo
- Se necesita una unidad encargada del despacho



# Despacho de instrucciones (acceso a operandos)

Estructural

## En pipeline escalar:

- El acceso a operandos se realiza en la etapa de decodificación.
- Si algún operando no está actualizado, se frena la instrucción dependiente hasta que el operando este disponible.
- El frenado de una instrucción frena las siguientes en el pipeline
- El uso de técnicas como bypassing combinadas con estrategias de compilación elimina el frenado

## En pipeline superescalar:

- Algunos operandos pueden no estar disponibles al momento de decodificación de una instr.
- Una instrucción puede depender de instrucciones previas que aún no han terminado su ejecución.
- No es posible aplicar técnicas de bypassing.

## Solución:

- Agregado de un buffer entre la decodificación y ejecución de instrucciones
- Las instrucciones decodificadas se almacenan en el buffer hasta que tengan todos sus operandos
- Cuando una instrucción tiene todos sus operandos se extrae del buffer y se ejecuta
- Dicho buffer actúa como una estación de reserva (reservation station – Tomasulo,1967)

# Políticas de emisión

---

- Tres políticas mas usuales
  - Emisión en orden y finalización en orden
    - Las instrucciones se emiten en orden secuencial y los resultados se escriben en ese mismo orden
  - Emisión en orden y finalización desordenada
    - Se decodifican instr. hasta el punto de dependencia o conflicto
    - Usada en los RISC escalares con instr. que necesitan varios ciclos
    - Maximiza el grado de paralelismo del procesador
    - Sensible a las dependencias de salida (WAR y WAW)
  - Emisión desordenada y finalización desordenada
    - Desacopla etapas de decodificación-ejecución: **ventana de instrucciones (estaciones de reserva)**
    - La emisión para ejecución se hace desde la ventana de ejecución
    - Sensible a las depedencias WAR y WAW

# Ejemplo de políticas de emisión

---

- Cauce superescalar
- Capaz de captar y decodificar 2 instrucciones a la vez
- 3 unidades funcionales independientes
- 2 copias de la etapa de escritura (bancos de registros auxiliares)

Fragmento de programa de 6 instrucciones  
con las siguientes restricciones:

- I1 necesita 2 ciclos de ejecución
- I3 e I4 compiten por la misma unidad funcional
- I5 depende del resultado de I4
- I5 e I6 compiten por la misma unidad funcional

# Emisión y finalización en orden

---

Decode		Execute			Write		Cycle
11	12						1
13	14	11	12				2
13	14	11					3
	14			13	11	12	4
15	16			14			5
	16		15		13	14	6
			16				7
					15	16	8

# Emisión en orden y finalización desordenada

Decode		Execute			Write		Cycle
11	12						1
13	14	11	12				2
	14	11		13	12		3
15	16			14	11	13	4
	16		15		14		5
			16		15		6
					16		7

Al terminar la ejecución de I2 se puede ejecutar I3, aún antes de que acabe I1

# Emisión desordenada y finalización desordenada

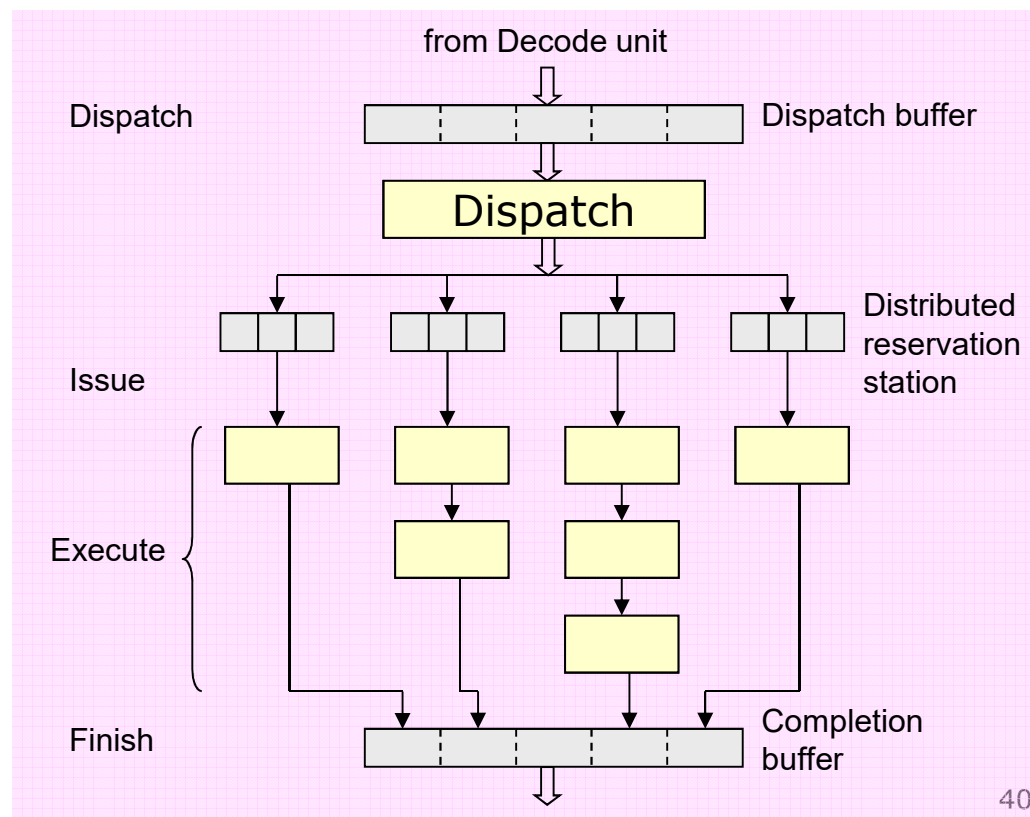
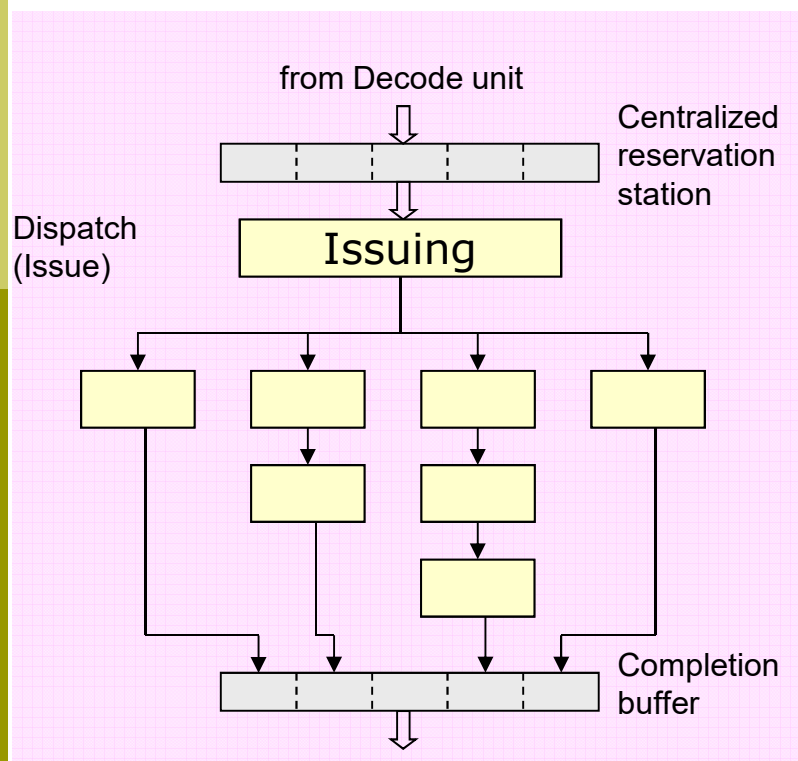
Decode		Window	Execute			Write		Cycle
11	12							1
13	14	11,12	11	12				2
15	16	13,14	11		13	12		3
		14,15,16		16	14	11	13	4
		15		15		14	16	5
						15		6

La instrucción I6 puede emitirse para ejecución antes que I5, ya que I6 no depende del resultado de I4 (aún no finalizada)

# Estaciones de reserva

## 2 tipos de estaciones de reserva:

- Estación de reserva centralizada
  - Un buffer simple a la entrada de la unidad de despacho
- Estación de reserva distribuida
  - Varios buffer individuales a la salida de la unidad de despacho



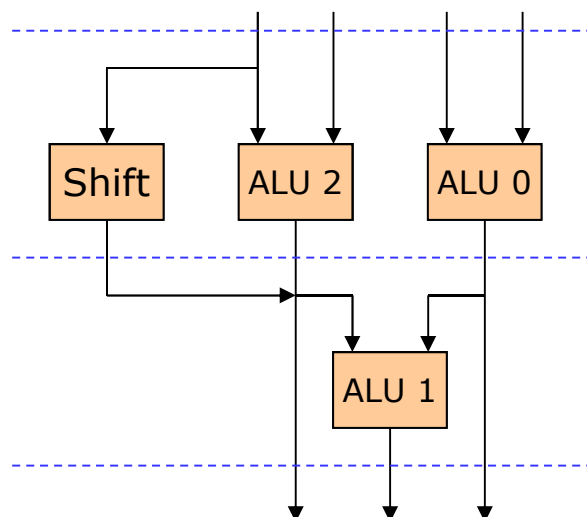
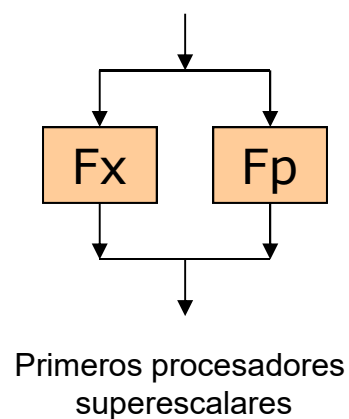


# Notas sobre estaciones de reserva

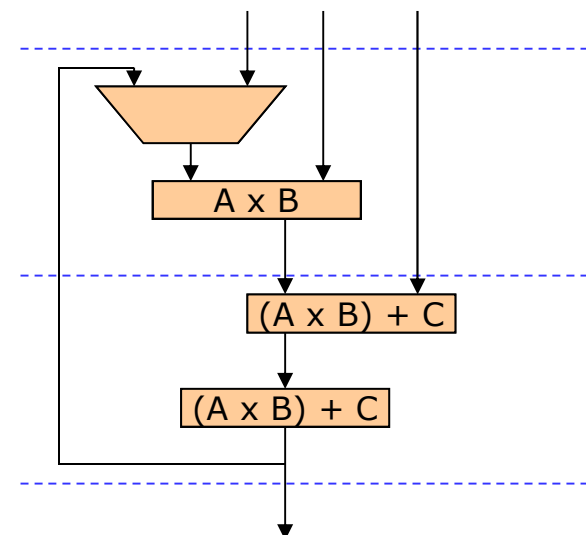
- Algunas arquitecturas presentan híbridos de los dos casos extremos  
**(Clustered reservation station)**
- En una arquitectura híbrida hay varias estaciones de reserva y cada una de ellas alimenta a varias unidades funcionales
- Ventajas y desventajas de una estación centralizada:
  - ↑• Todos los tipos de instr comparten la misma estación
  - ↑• Mejor utilización de las entradas de la estación
  - ↓• Diseño de hardware más complejo
  - ↓• Control centralizado y acceso multipuerto
- Ventajas y desventajas de una estación distribuida:
  - ↑• Puertos de salida de un solo puerto
  - ↓• Fragmentación interna
  - ↓• Utilización mas baja que las centralizadas
  - ↓• Si se llenan producen frenados de las instrucciones del tipo que manejan
  - ↑• Diseño y manejo mas simple

# Ejecución de instrucciones (Execute)

Tendencia actual: pipelines mas paralelos y diversificados  
(mas unidades funcionales mas especializadas)



Unidad de enteros  
T1 super SPARC



Unidad de punto flotante  
IBM RS/6000

# Ejecución de instrucciones (Execute)

---

¿Qué tipos de unidades funcionales debe usarse en un superescalar?

¿Cuál es la mejor combinación de tipos de unidades funcionales?

Estadísticamente un programa tiene:

- 40% de instrucciones aritméticas
- 20% de instrucciones de salto
- 40% de instrucciones de acceso a memoria (load/store)

Procesadores actuales:

- 4 unidades aritméticas (Fx y Fp)
- 1 unidad de salto
- 1 unidad de acceso a memoria (load/store)

El desbalanceo es preferible ya que 4 unidades de memoria requieren que la cache de datos sea multipuerto (+ costo)

Solución: Uso de múltiples bancos de memoria simulando una caché multipuerto real

# Finalización y retiro de instrucciones (completion and retiring)

Estructural

Una instrucción se considera completa cuando

- **Termina** su ejecución y
- **Actualiza** el estado de la máquina

2 casos alternativos:

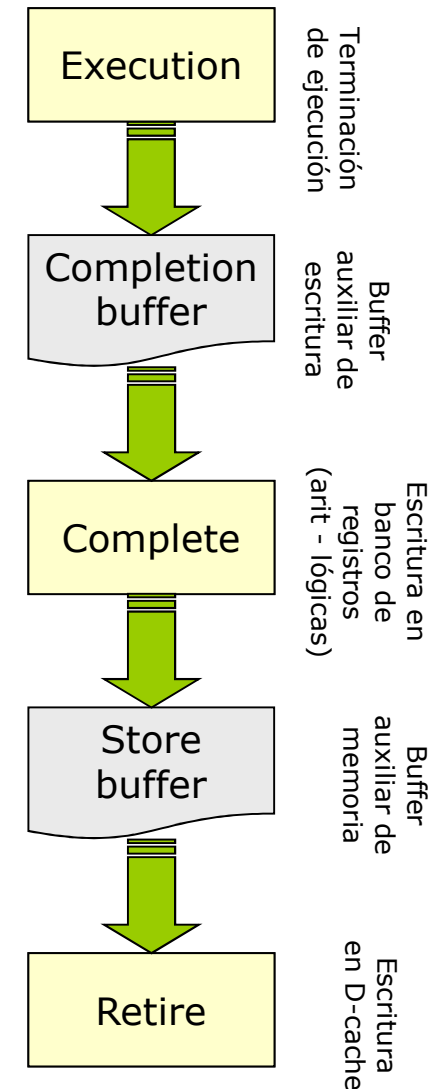
- Instrucciones aritmético-lógicas o *Load* de memoria
  - Cuando una instrucción termina su ejecución sus resultados se almacenan en buffers temporales
  - Cuando una instrucción es completada sus resultados se escriben en los registros de la arquitectura
- Instrucción *Store* en memoria (D-cache)
  - En instrucciones de memoria (Store) el dato no necesariamente se escribe en la D-cache al salir del *Completion buffer* sino que se almacena en un *buffer temporal de memoria* y se escribe en la D-cache en el siguiente periodo libre de bus

## Instruction Completion

Actualiza el estado de la máquina

## Instruction Retiring

Actualiza el estado de la memoria



# Estructura del buffer de reordenamiento

---

- Buffer circular con punteros *head* y *tail*
- Las instrucciones se escriben en el ROB estrictamente en el orden del programa
  - Cada vez que se despacha una instrucción se crea una nueva entrada en el ROB
- Estado de la instrucción
  - Emitida, en ejecución, terminada
- Una instrucción puede retirarse (Completion) si y solo si
  - Ha terminado
  - Todas las instrucciones previas han sido retiradas
  - No se ha ejecutado de forma especulativa

# Estructura del buffer de reordenamiento

## Status:

- Issued (I)
  - En espera en alguna estación de reserva
- Execution (X)
  - En ejecución en alguna unidad funcional
- Finished (F)
  - En ROB a la espera de su *Completion*

## Instrucción

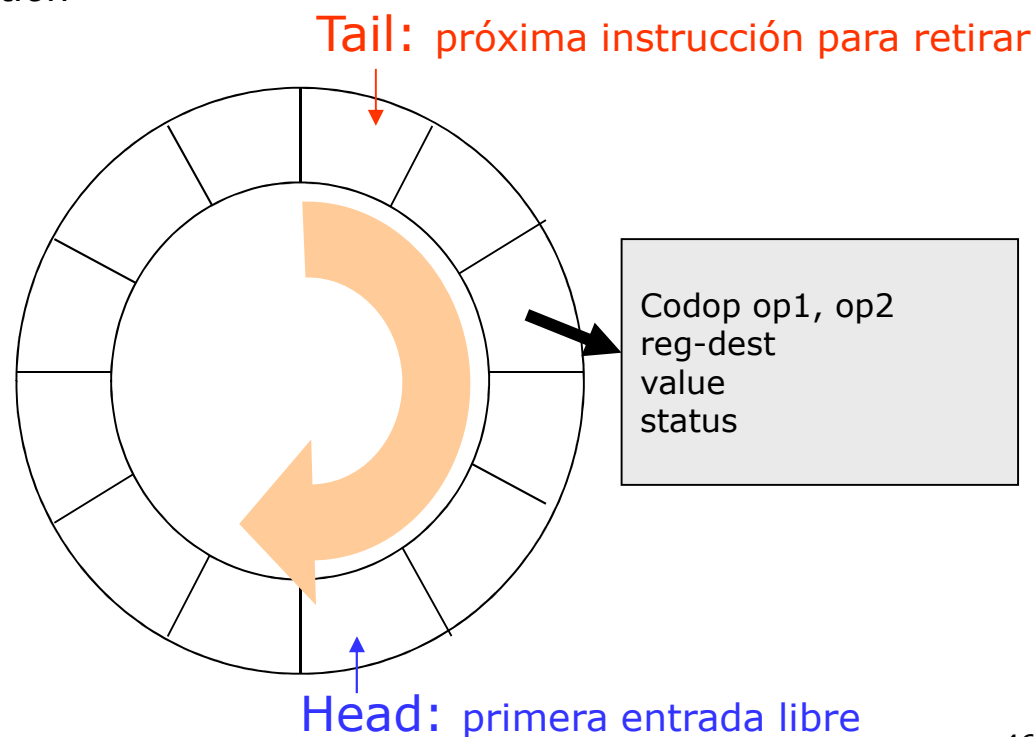
*Codop op1, op2*

## Reg-dest

*registro de escritura de la inst.*

## Value

*valor del resultado de la inst.*



# ¿Para qué sirve el ROB?

El despacho y las diferentes latencias de las instrucciones generan ejecución fuera de orden.

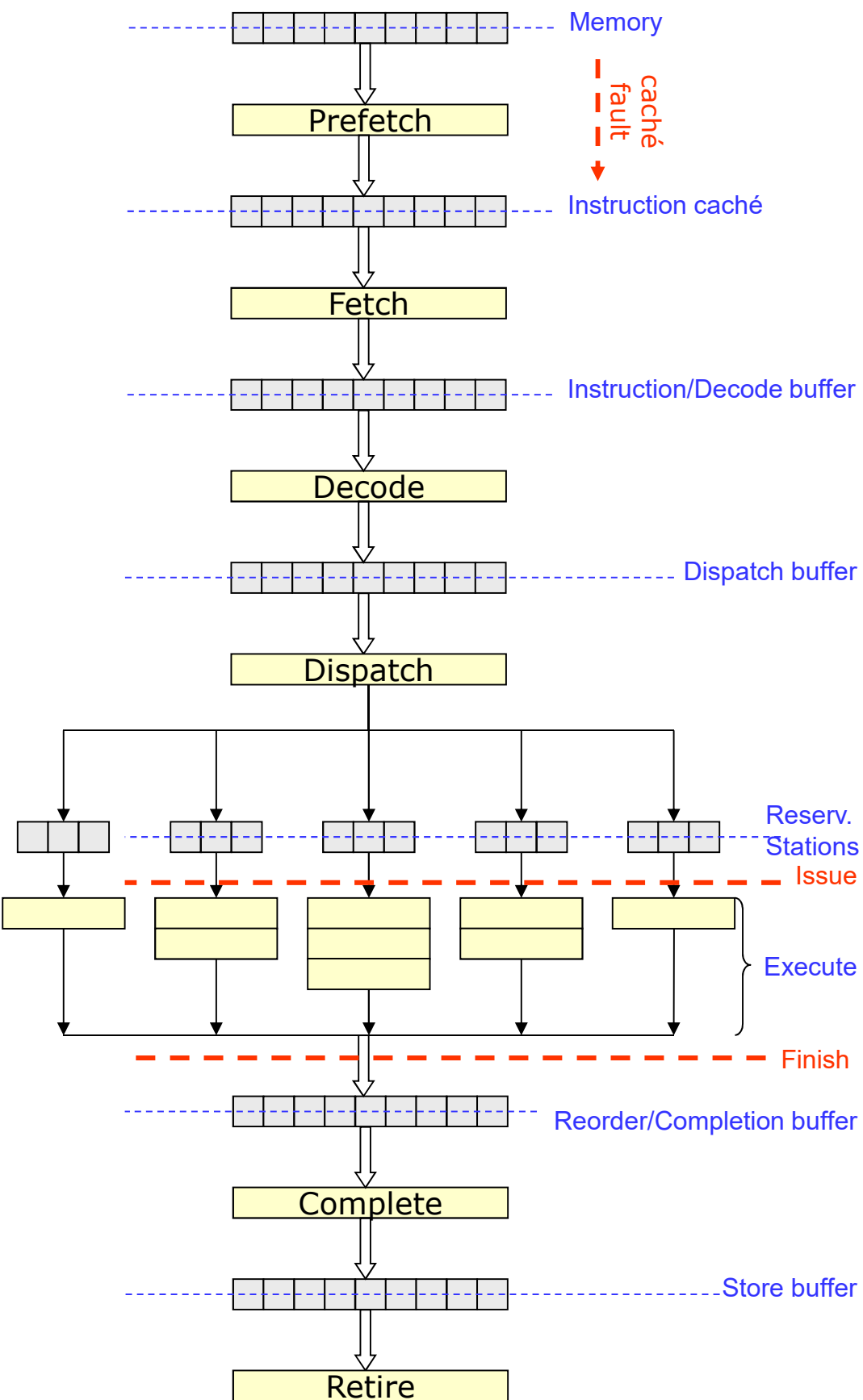
Desde el punto de vista de las dependencias de datos **no importa** que las instrucciones terminen fuera de orden

Si es importante la consistencia secuencial para las dependencias de control

- Saltos condicionales especulativos
  - En un salto especulativo se cargan instrucciones de una vía de ejecución que puede no ser la correcta, entonces, algunas instrucciones deben anularse
- Interrupciones
  - Rupturas de secuencia de ejecución normal causadas por eventos externos como dispositivos de I/O
  - Cuando ocurre una INT, el programa debe suspenderse para atenderla
  - Se detiene el *Fetching* y se termina de ejecutar las instr aun en el pipeline
- Excepciones
  - Inducidas por la ejecución de alguna inst del programa (evento anormal)
  - La ejecución del programa debe suspenderse en el estado de la inst anterior a la que produce la excepción.
  - se debe asegurar que las inst anteriores en la secuencia del programa se terminen y las posteriores se anulen

# Arquitectura completa de un pipeline superescalar

Estructural



In order

Out of order

In order



# Procesadores Superescalares

---

- **Descripción estructural:** Arquitectura y componentes de las distintas etapas de un procesador superescalar

- **Descripción funcional:** Funcionamiento de un procesador superescalar y descripción de las estructuras que permiten la normal ejecución del flujo de instrucciones de un programa

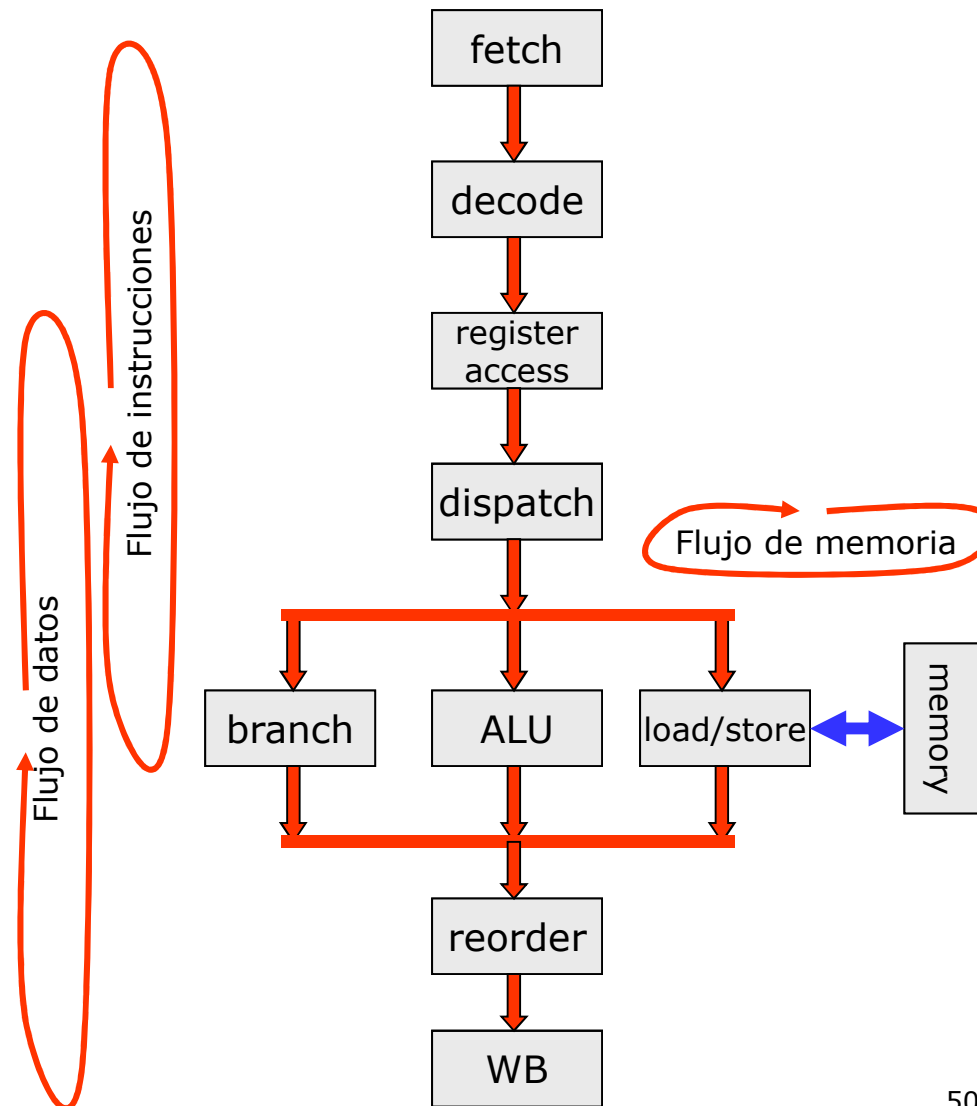
# Funcionamiento de un superescalar


3 flujos principales:

- Flujo de instrucciones
- Flujo de datos
- Flujo de memoria

Asociados con los 3 grandes tipos de instrucciones:  
saltos, ALU, memoria

**Objetivo:** maximizar el volumen de instrucciones procesadas en los 3 flujos





---

## Manejo del flujo de instrucciones

# Manejo del flujo de instrucciones

Uso de los  
ciclos perdidos



Salto  
retrasado

Reordenamiento  
de instrucciones

Procesamiento de salto  
condicional no resuelto



Bloqueo



Especulación



Multivía

Procesamiento  
de saltos

eliminación  
de saltos



Instrucciones  
predicadas

# Manejo del flujo de instrucciones

En flujo secuencial:

- máximo rendimiento
- todas las etapas llenas

Cuando hay salto:

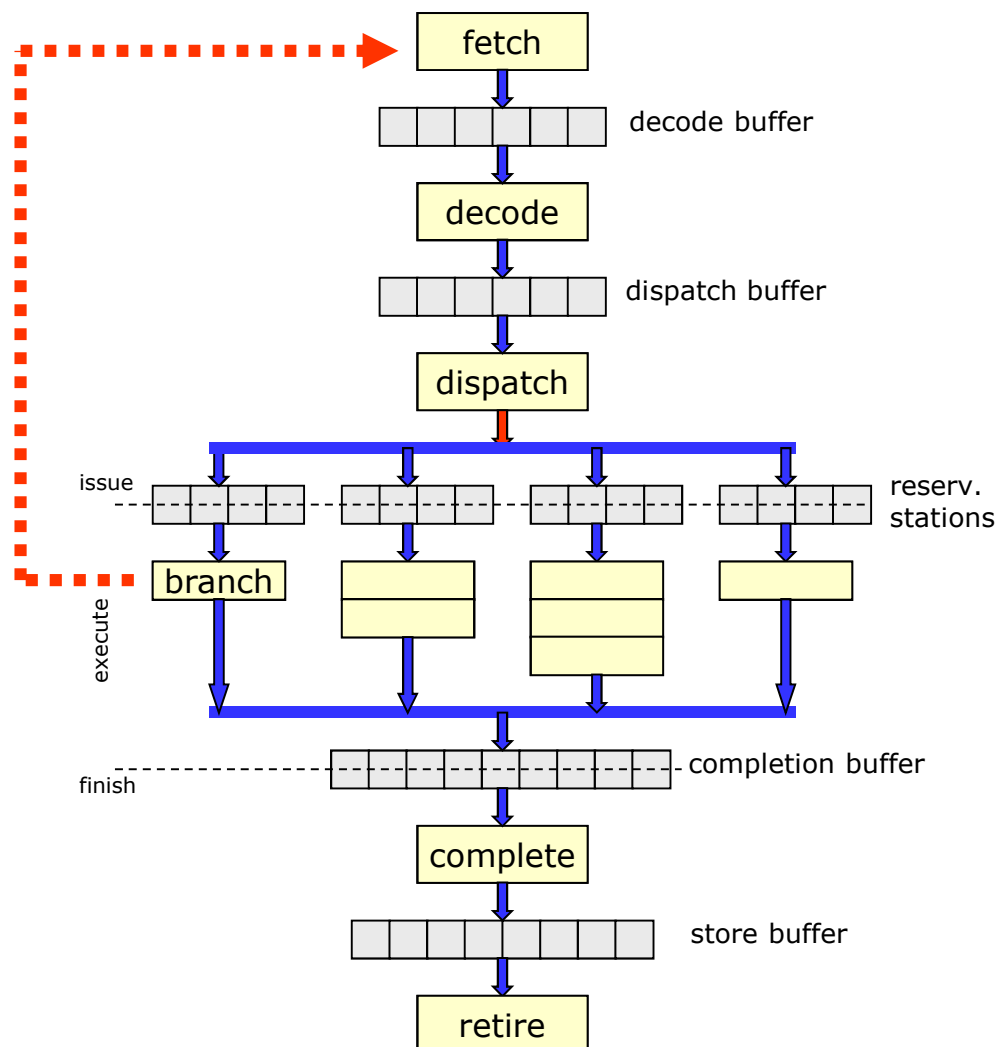
- se pierden 3 (n) ciclos
- penalización de  $3 * \text{ancho pipeline}$

En saltos incondicionales

- se debe esperar el cálculo del *target*

En saltos condicionales

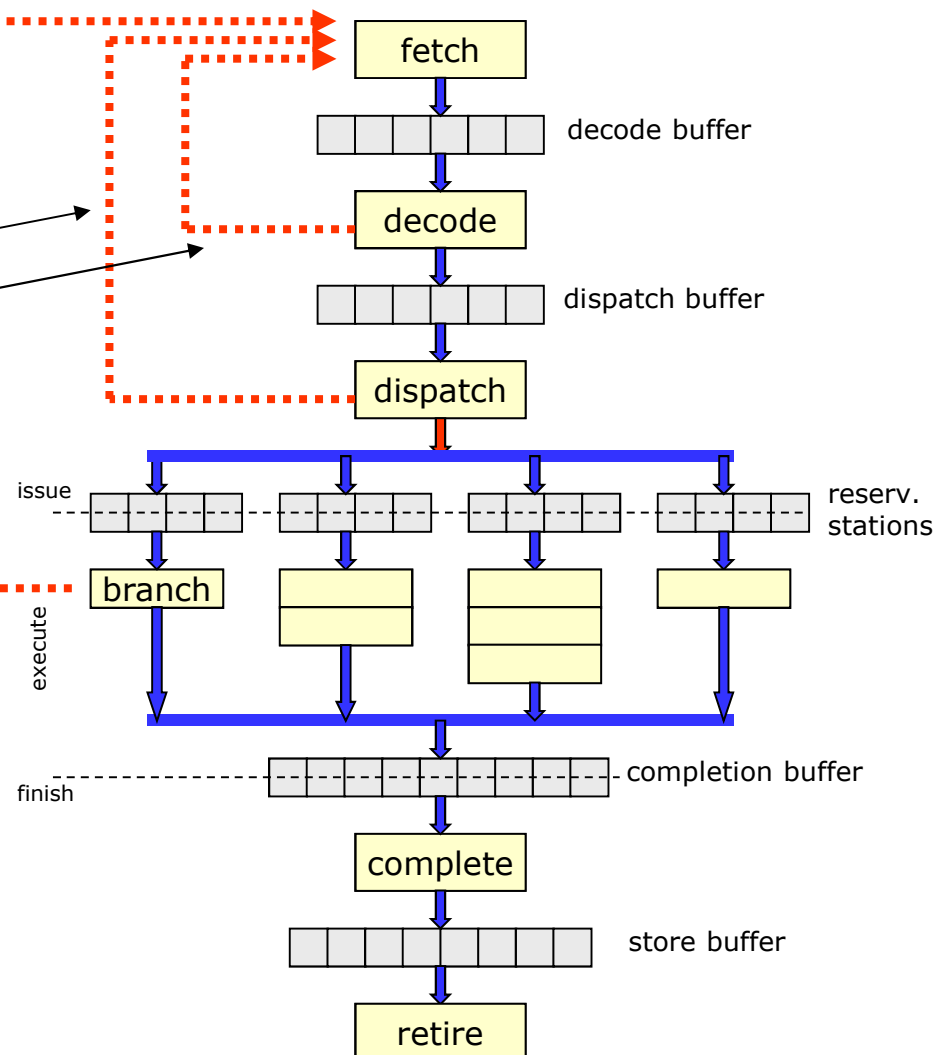
- se debe calcular *target*
- se debe calcular *condición*



# Manejo del flujo de instrucciones

En saltos **incondicionales** los ciclos de frenado dependen del modo de direccionamiento utilizado

- $\text{target} = \text{Registro} + \text{offset}$
- $\text{target} = \text{registro}$
- $\text{relativo al PC}$



# Manejo del flujo de instrucciones

En saltos **condicionales** los ciclos de frenado también dependen de la forma de cálculo de la condición

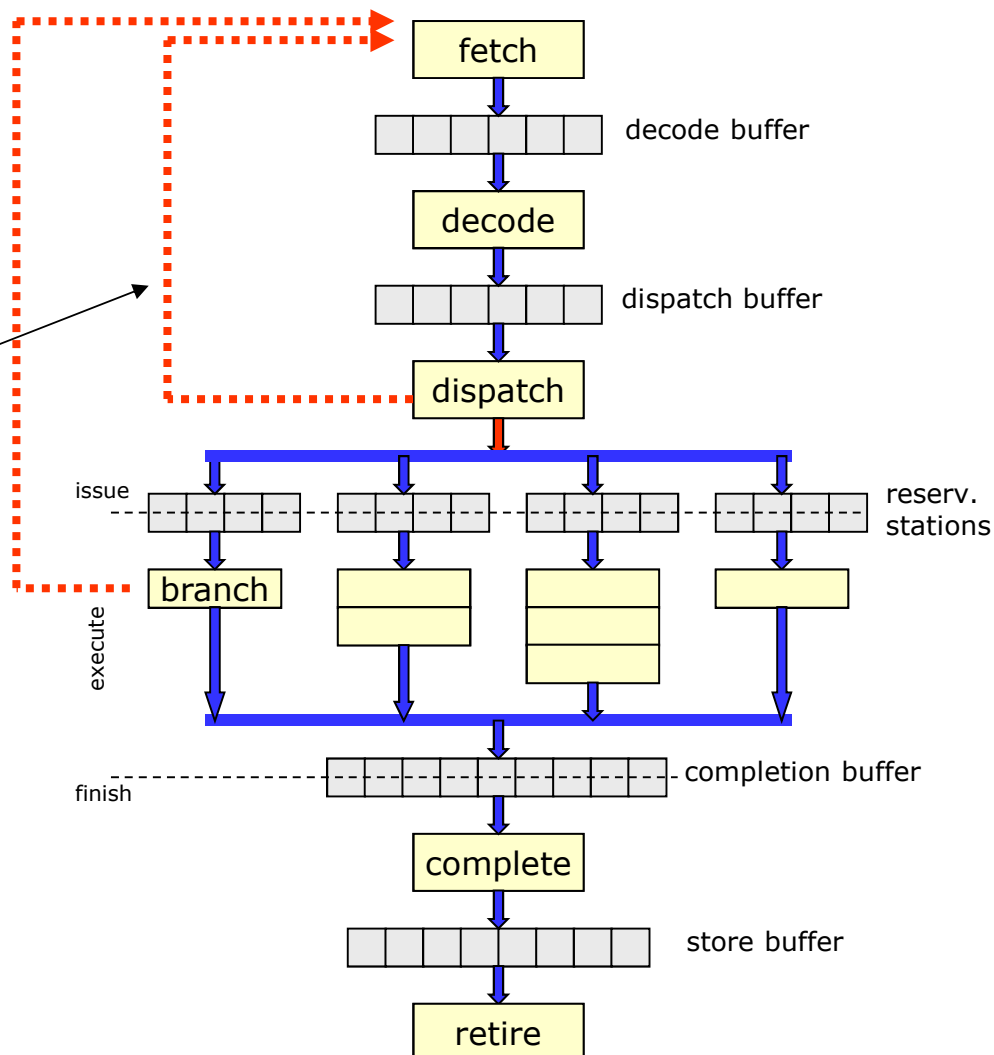
- Comparación de registros en la propia instrucción de salto

*Beq r1, r2, dir\_salto*

- testeo de bits de estado

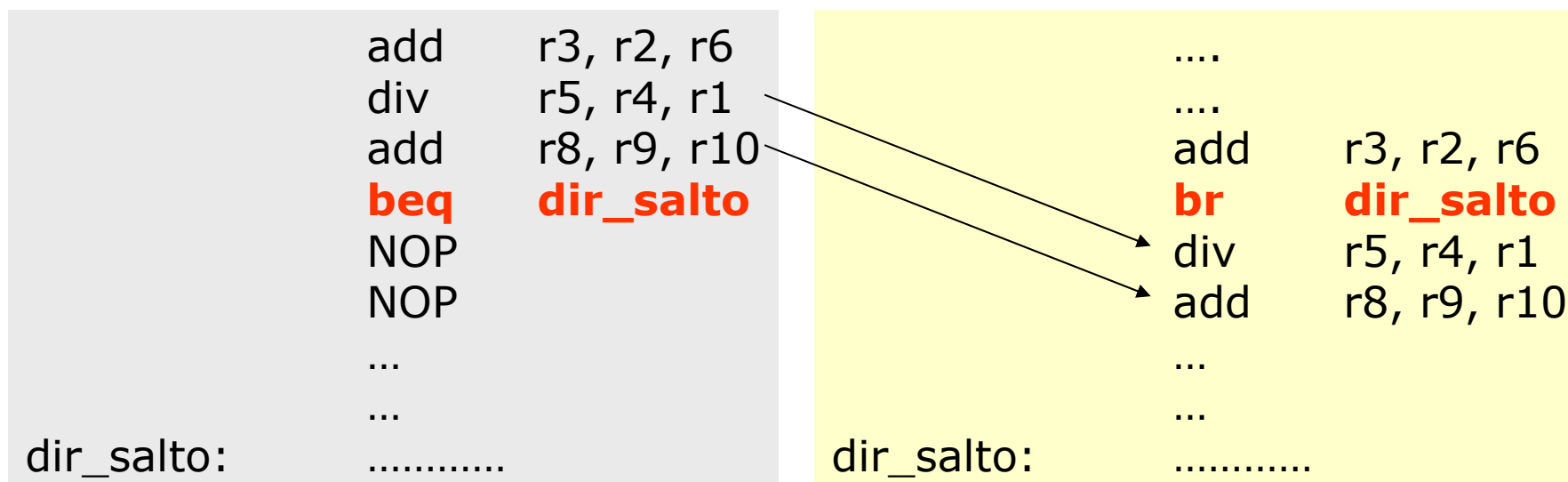
*Beqz dir\_salto*

Desventajas del testeo de estado:  
Es un concepto secuencial, entonces surgen problemas al paralelizar las instrucciones



# Salto retrasado

Migración de código en tiempo de compilación a fin de rellenar las etapas frenadas por la ejecución de una posible vía de salto tomado



La migración de código debe ser incondicional. Esto es, la condición a ser testeada por el salto no debe ser generada por las instrucciones migradas



# Técnicas de predicción de saltos

---

Algunos estudios experimentales demostraron que las instrucciones de salto condicional son bastante predecibles

## Cuando se encuentra una instrucción de salto:

- **PREDICCIÓN:** Se especula cual es la siguiente instrucción a ejecutar
- Comienzan a ejecutarse instrucciones especulativamente
- En algún momento se chequea la condición de salto

## Si se acierta en la especulación:

- Se continúa la ejecución

## Si no se acierta en la especulación:

- Se eliminan las instrucciones ejecutadas especulativamente
- Se buscan las instrucciones correctas

# Técnicas de predicción de saltos

---

**Las técnicas de predicción de saltos involucran 2 aspectos:**

- **Especulación de la dirección de salto:**

se presupone una dirección de salto durante la fase de fetch a fin de poder cargar la nueva instrucción en el ciclo siguiente (sin pérdida de ciclos).

Usualmente la dirección predicha es la anterior dirección de salto

- **Especulación de la condición de salto:**

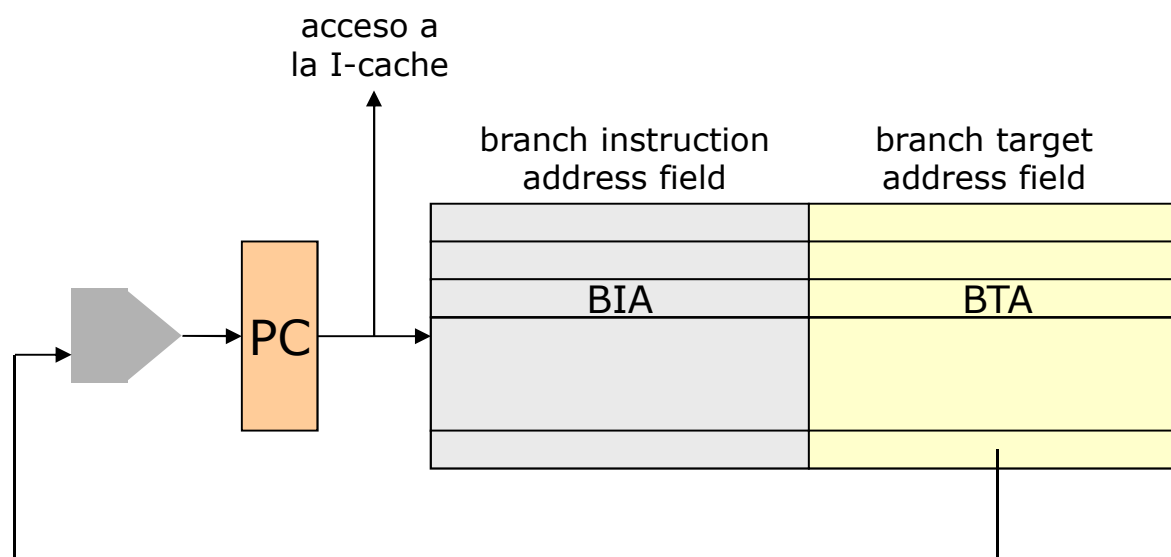
se presupone un resultado de la condición de salto.

- Salto no tomado (SNT): continuar por la vía secuencial
- Salto tomado (ST): ir a la instrucción de la dirección de destino del salto

# Especulación de la dirección de salto

## Uso de un buffer de direcciones de salto (BTB – Branch Target Buffer)

1. Se usa el PC para acceder a la I-caché a buscar la siguiente instrucción
2. Al mismo tiempo con el PC se accede al BTB. Si la dirección del PC corresponde a una instrucción de salto (campo BIA), el buffer devuelve la dirección de salto a la que previamente salto la instrucción branch (BTA).
3. la dirección BTA se carga en el PC para que en el siguiente ciclo se acceda a la instrucción a la que salta el branch (si se predijo como ST)
4. Paralelamente la instrucción branch leída desde la I-caché se ejecuta para validar o no la dirección especulada vía BTB



# Especulación en la condición de salto

---

- **Predicción fija**

- Siempre se hace la misma predicción: Salto tomado o salto no tomado (vía secuencial)

(LA MAS SIMPLE PERO POCO EFECTIVA!!)

- **Predicción verdadera**

- La predicción cambia para diferentes saltos o diferentes ejecuciones del mismo salto

- Dos tipos:

- Predicción **estática**: basada en el análisis del código  
(EN TIEMPO DE COMPILACIÓN!!)

- Predicción **dinámica**: basada en la historia del salto  
(EN TIEMPO DE EJECUCIÓN!!)

# Predicción estática

---

- ✓ Predicción basada en el código de operación
  - Para algunas instrucciones de salto se predice como ST y para otras como SNT
  - MC88110 , PowerPC 603 – con saltos relativos a ciertos registros
- ✓ Predicción basada en el desplazamiento
  - Si desplazamiento  $< 0$ , ST, si  $\geq 0$ , SNT
- ✓ Predicción dirigida por el compilador
  - Se usa un bit adicional en el código de operación de la instrucción
  - El compilador analiza el código para determinar si el salto en cuestión debe realizar predicción de ST o SNT
  - Ejemplo:
    - branch relacionado con loops : ST
    - branch relacionado con IF : SNT

# Predicción dinámica

---

- La predicción se hace según la historia del salto

***La historia es una buena guía para predecir el futuro***

- Esquema básico: un salto tomado en las últimas ejecuciones se predice como verdadero en la siguiente
  - Mayor rendimiento que la predicción estática
  - Mas complejo para implementar
- Mejor comportamiento en la predicción de saltos asociados a *loops*

# Predicción dinámica

## 2 técnicas principales:

- **Explícita: Bits de historia**

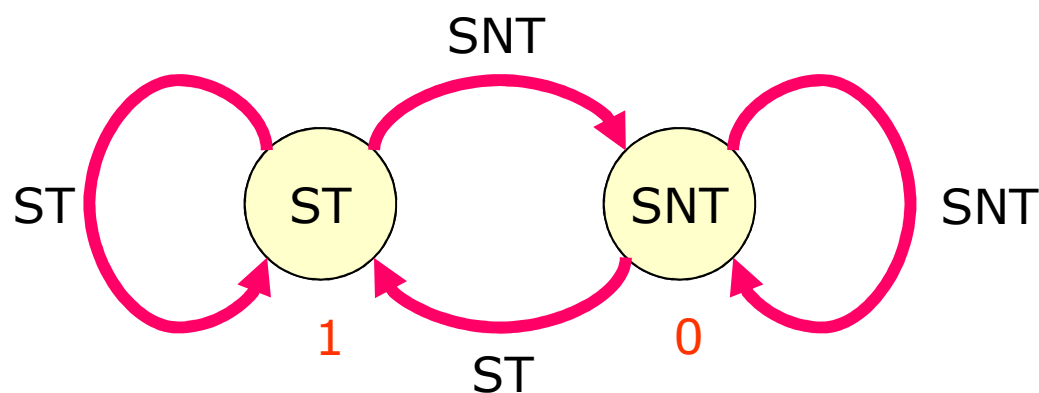
- La historia del salto se marca explícitamente usando cierta cantidad de bits de historia
- Usualmente 2 bits (UltraSparc, R10000, Pentium, PowerPC)
- Los bits de historia representan la probabilidad de que el salto se efectúe en la siguiente ejecución

- **Implícita**

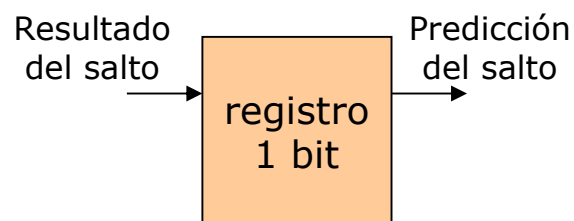
- La historia del salto se indica implícitamente mediante la presencia de una entrada en un buffer de destino de saltos
  - Si el salto fue tomado la última vez, se agrega al buffer
  - Si el salto NO fue tomado la última vez, se quita del buffer

# Predicción dinámica de 1 bit

Se usa 1 bit para cada salto indicando si en la última ejecución fue tomado o no tomado



estado actual	resultado del salto	nuevo estado
0	0	0
0	1	1
1	0	0
1	1	1



1-bit saturation counter



# Predicción dinámica de 1 bit

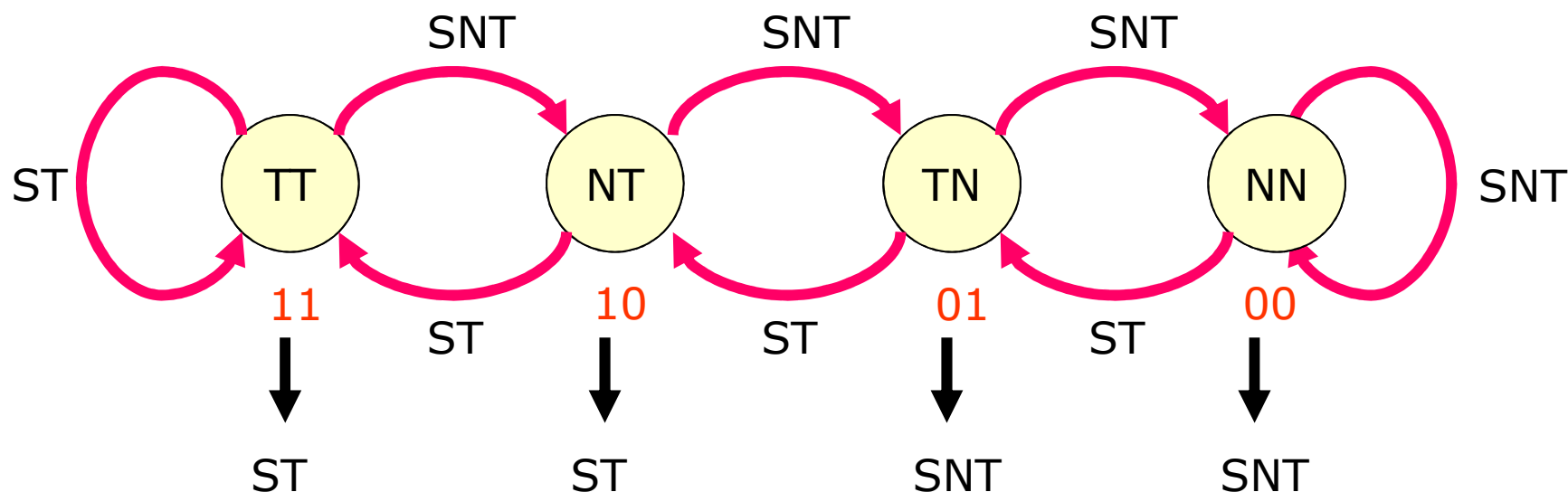
Comportamiento en *loops* anidados: cuando el salto no es efectivo se predice incorrectamente 2 veces

```
lazo_1: .....
        .....
lazo_2: .....
        ...
        ...
        beqz r1, lazo_2
        beqz r2, lazo_1
```

lazo_1	lazo_2	predicho	real	nueva predicción
1	1	SNT	ST	ST
	2	ST	ST	ST
	3	ST	ST	ST
	.....			
	última	<b>ST</b>	<b>SNT</b>	SNT
2	1	<b>SNT</b>	<b>ST</b>	ST
	2	ST	ST	ST
	.....			

# Predicción dinámica de 2 bits

Con dos bits se puede almacenar la historia de las 2 últimas ejecuciones del salto



Se puede implementar con un contador ascendente/descendente de 2 bits con saturación en 11 y 00 (2-bit saturation counter)

TT: Tomado/Tomado  
 NT: No tomado/Tomado  
 TN: Tomado/No tomado  
 NN: No tomado/No tomado

1X : predecir salto tomado  
 0X : predecir salto NO tomado

# Predicción dinámica de 2 bits

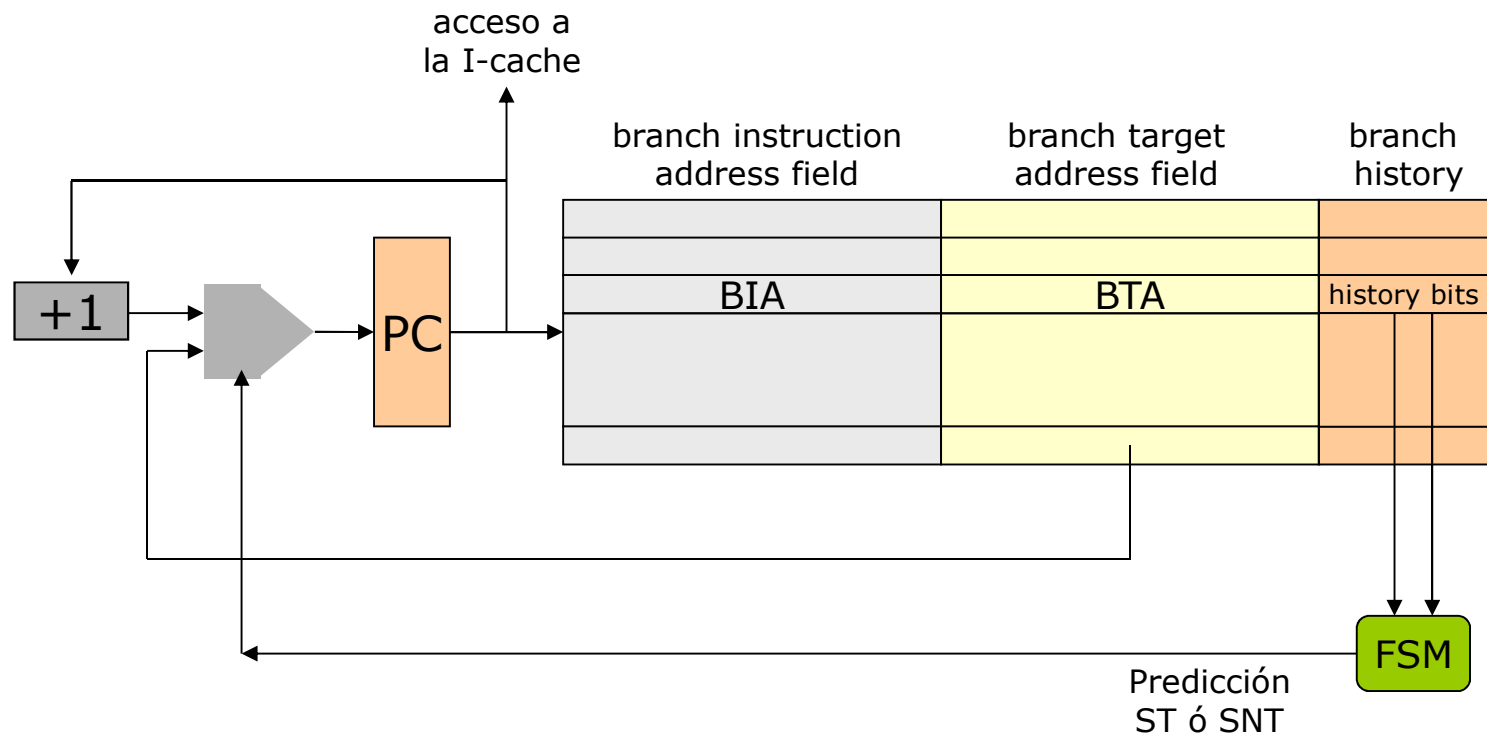
Comportamiento en *loops* anidados: cuando el salto no es efectivo se predice incorrectamente **1 vez**

```
lazo_1: .....
        .....
lazo_2: .....
        ...
        ...
        beqz r1, lazo_2
        beqz r2, lazo_1
```

lazo_1	lazo_2	predicho	real	nueva predicción
1	1	SNT	ST	SNT
	2	SNT	ST	ST
	3	ST	ST	ST
	.....			
	última	<b>ST</b>	<b>SNT</b>	ST
2	1	ST	ST	ST
	2	ST	ST	ST
	.....			

# Implementación de la predicción con bits de historia

Funcional



- Por SNT se continúa la ejecución secuencial y se carga PC con PC + 1
- Por ST se carga en PC la BTA

# Efectividad de la predicción con bits de historia

Funcional

1 bit de historia	Recuerda la dirección tomada la última vez (ST ó SNT)	82,5% a 96,2
2 bits de historia	Predice la dirección a partir de las últimas 2 veces (ST ó SNT)	86% a 97%
3 bits de historia	Predice la dirección a partir de las últimas 3 veces (ST ó SNT)	88,3% a 97%

El incremento de efectividad en las predicciones empieza a ser mínimo a partir de 2 bits de historia. Por lo tanto, el uso de contadores de 2 bits es la mejor opción con relación costo/rendimiento.

# Predicción dinámica Implícita

---

- **2 esquemas:**
  - BTAC (Branch Target Address Cache):
  - BTIC (Branch Target Instruction Cache):
- **Se basan en el uso de una caché adicional que contiene:**
  - Entradas para los saltos más recientes (idealmente para todos)
  - Entradas sólo para los saltos tomados (ST)
  - Si el salto tiene una entrada en la caché entonces se predice como tomado
  - Los saltos no tomados (SNT) no están en la cache o se sacan si es que estaban
- **Similar a un predictor de 1 bit**

# Implementación de BTAC y BTIC

**caché BTAC:** almacena dirección del salto y dirección de la instrucción de destino predicha (como un BTB)

branch address	target address
1000	1004 ó 2000

**caché BTIC:** Básicamente un buffer llenado directamente desde la I-caché con la(s) instrucción(es) que siguen al salto en el orden normal del programa. Permite implementar saltos incondicionales en CERO ciclos

branch address	target address	target instruction
1000	1004 ó 2000	Addi r8,..

1000	Beqz r5, 2000
1004	Addi r8, r1, 20
2000	Load r1, r2, r3

# Recuperación de predicciones erróneas

Cualquier técnica especulativa de salto requiere de un mecanismo de recuperación ante fallos en la predicción:

- Eliminación de las instrucciones de la ruta errónea
- carga y ejecución de las instrucciones de la ruta correcta

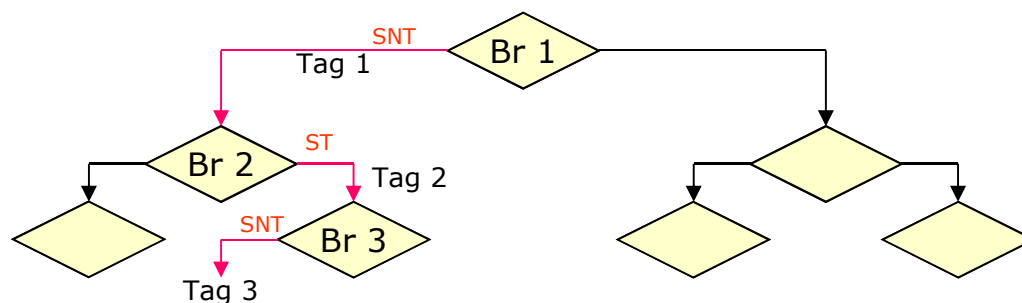
La ejecución de instrucciones de una ruta especulativa puede dar lugar a la aparición de nuevas instrucciones de salto que serán especulados como el primero aún antes de que la condición de este se haya resuelto

En un momento dado puede haber en el procesador instrucciones ejecutadas **NO especulativamente** mezcladas con varios niveles de instrucciones ejecutadas **especulativamente**



# Recuperación de predicciones erróneas

Para identificar las instrucciones en ejecución especulativa de diferentes saltos se usan rótulos (Tags) asociados a cada salto



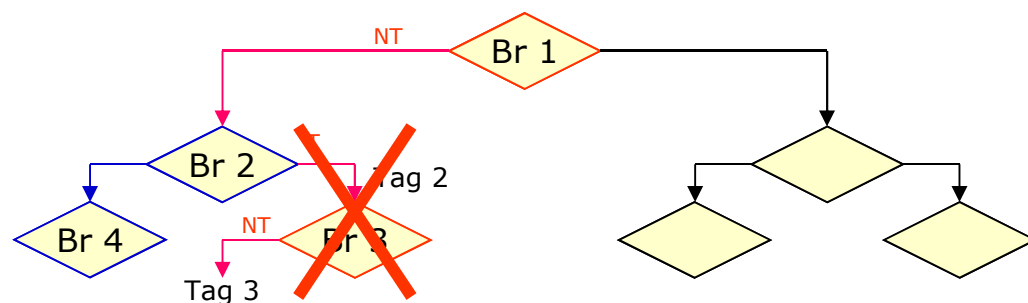
Una instrucción rotulada indica que es especulativa y el rótulo la asocia a un nivel de salto especulativo

Cuando se especula, la dirección de todas las instrucciones de salto, al igual que las direcciones de salto tomado y salto no tomado se deben almacenar para el caso de que sea necesaria una recuperación

# Recuperación de predicciones erróneas

Cuando se termina la ejecución real del salto, se puede validar el mismo:

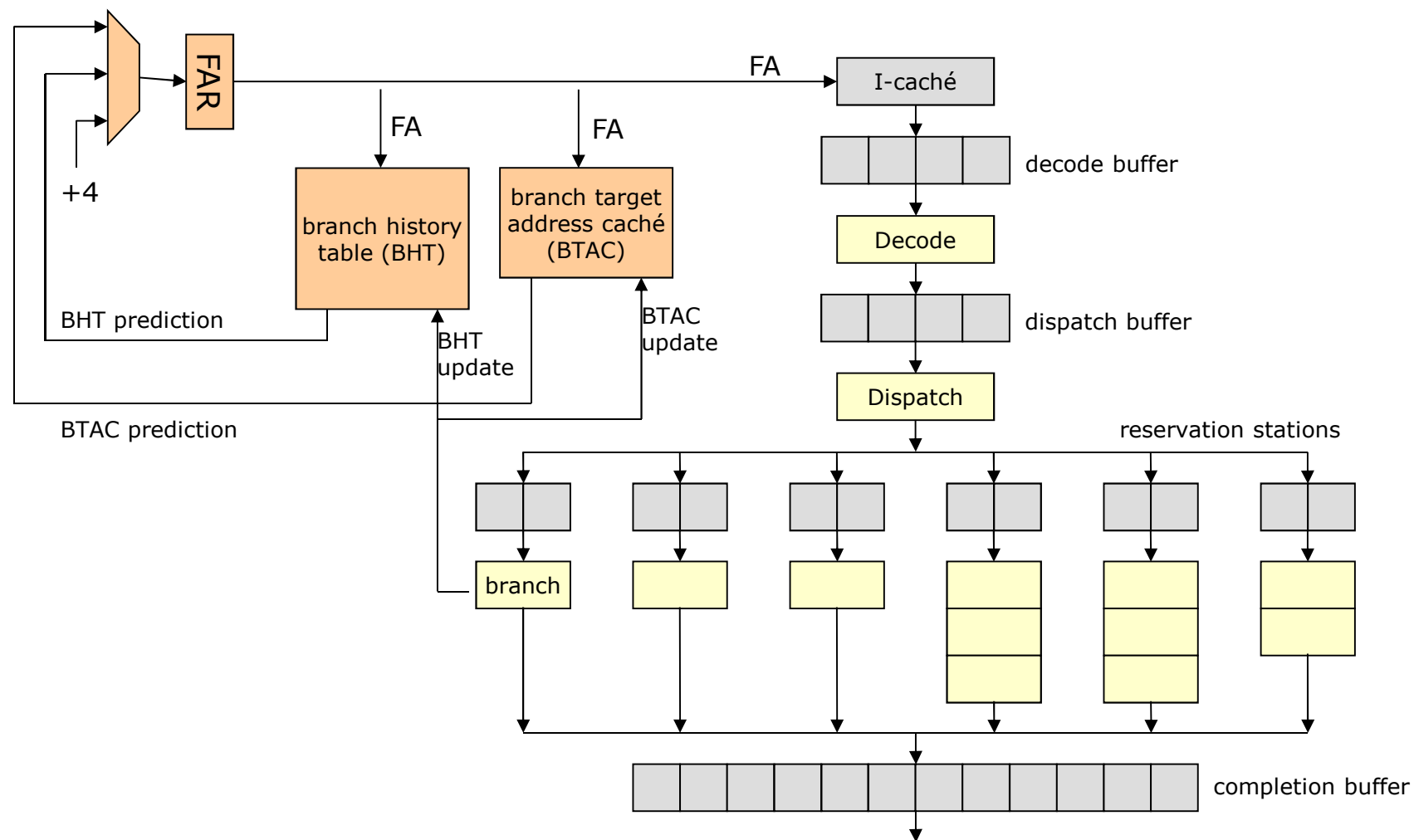
- Si la predicción coincide con el resultado evaluado, se puede eliminar el tag de ese salto de las instrucciones correspondientes
- Si la predicción no coincide, se eliminan las instrucciones asociadas a ese tag (tag n) y las de los tags subsiguientes (tag n+i)



- branch 1 termina su ejecución y se predijo correctamente, por lo que el tag 1 de las instrucciones asociadas se elimina
- Branch 2 se predijo incorrectamente por lo que se eliminan de la ruta de ejecución todas las instrucciones con tag 2 y 3
- Un nuevo branch (4) se predice y a partir de ese punto las instrucciones se rotularán con un tag 4....

Las instrucciones erróneas que están aún en los buffers de decodificación, despacho y estaciones de reserva se invalidan. Las instrucciones erróneas en el ROB se retiran

# Ejemplo: PowerPC 604



# Ejemplo: PowerPC 604

- Superescalar de grado 4 capaz de recuperar, decodificar y despachar hasta 4 instrucciones simultáneas en cada ciclo de máquina
  - 2 buffers separados para soporte de predicciones de salto:
    - BTAC: caché asociativa de 64 entradas que almacena las direcciones de salto
    - BHT: caché de acceso directo de 512 entradas que almacena un predictor de 2 bits de historia para cada salto
- 
- Ambas tablas se acceden durante la etapa de fetching (ciclo i).
  - BTAC responde en 1 ciclo si existe un salto en la dirección aportada por el PC
  - BHT responde en 2 ciclos para la misma búsqueda
  - Si BTAC responde positivamente, se carga el PC con la dirección BTAC (en ciclo i+1)
  - En ciclo i+2 BHT dispone de la predicción por bits de historia para la dirección:
    - Si predice salto tomado (igual que BTAC) se continúa tal como predijo BTAC
    - Si predice SNT, entonces se anula la predicción de BTAC y se carga el PC con la dirección secuencial

# Técnicas avanzadas de predicción:

## Predicción en 2 niveles

Funcional

Historia Global

### Técnicas anteriores tienen limitaciones:

- Porcentaje de fallas cercano al 10%. NO aceptable para altas prestaciones
- La predicción se hace en función de una historia limitada del salto que se está analizando
- NO tiene en cuenta la dinámica de saltos globales que se están ejecutando en un momento dado

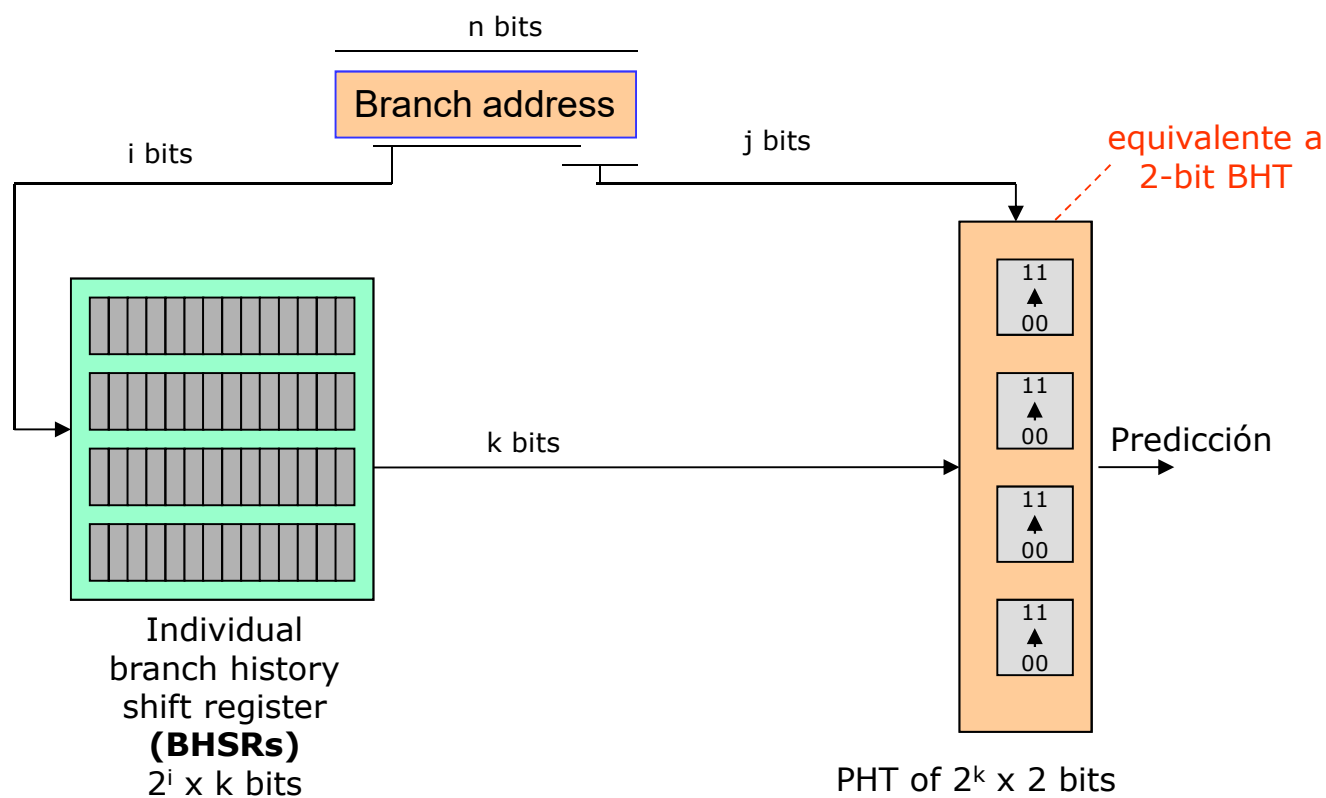
### Predicción de 2 niveles

- Supera en teoría el 95% de aciertos
- Adaptable a contextos dinámicos (saltos por loop, if, etc)
- Usa un conjunto de tablas de historia (Pattern history Table – PHT)
- El contexto de salto es determinado por un patrón de saltos recientemente ejecutados (Branch History Shift Register – BHSR)
- El BHSR se usa para indexar la PHT y obtener una entrada que contiene cierto número de bits de historia (Ej. saturation counter)

# Predictores Locales

## Predictores locales (L-Shared)

- $2^i$  registros individuales: Local BHSRs de  $k$  bits (historia de cada salto  $k$  veces en el pasado)
- Shared PHT: puesto que no se usan los  $n$  bits de la dirección de salto sino sólo  $j$  puede haber *aliasing*
- Memoriza  $2^i$  contextos de  $k$  saltos anteriores



# Predictores Globales

---

**utilizan información correspondiente al comportamiento reciente de otras instrucciones de salto distintas**

Se describen mediante el par

$(G,L)$

C: número de saltos globales a evaluar

L: número de bits del predictor local

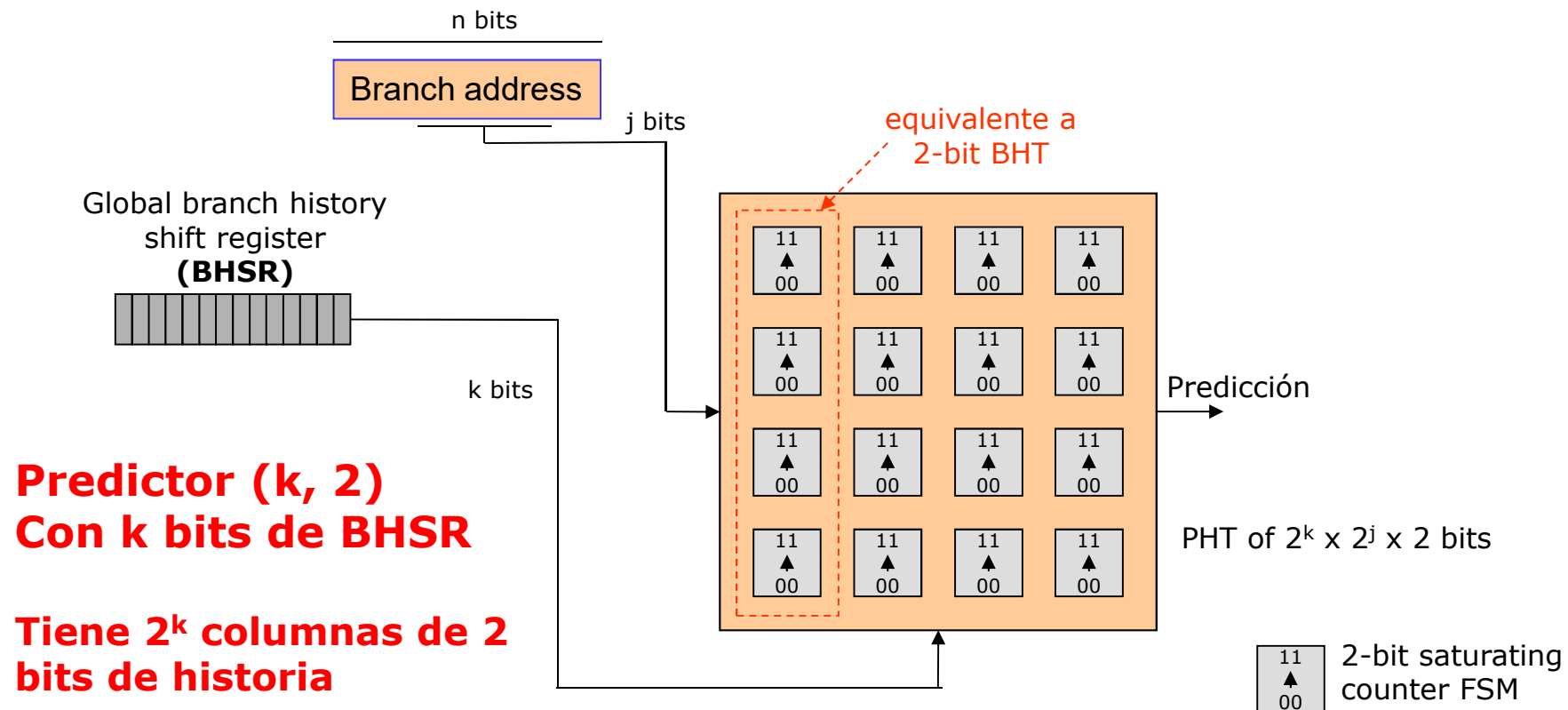
Predictores locales vistos hasta ahora:

- $(0, 1)$  : predictor de 1 bit
- $(0, 2)$  : predictor de 2 bits

# Predicción en 2 niveles

## G-Shared (Global)

- 1 único registro: Global BHSR de **k** bits
- Shared PHT: puesto que no se usan los **n** bits de la dirección de salto sino sólo **j** puede haber *aliasing*
- Memoriza un único contexto de **k** saltos anteriores

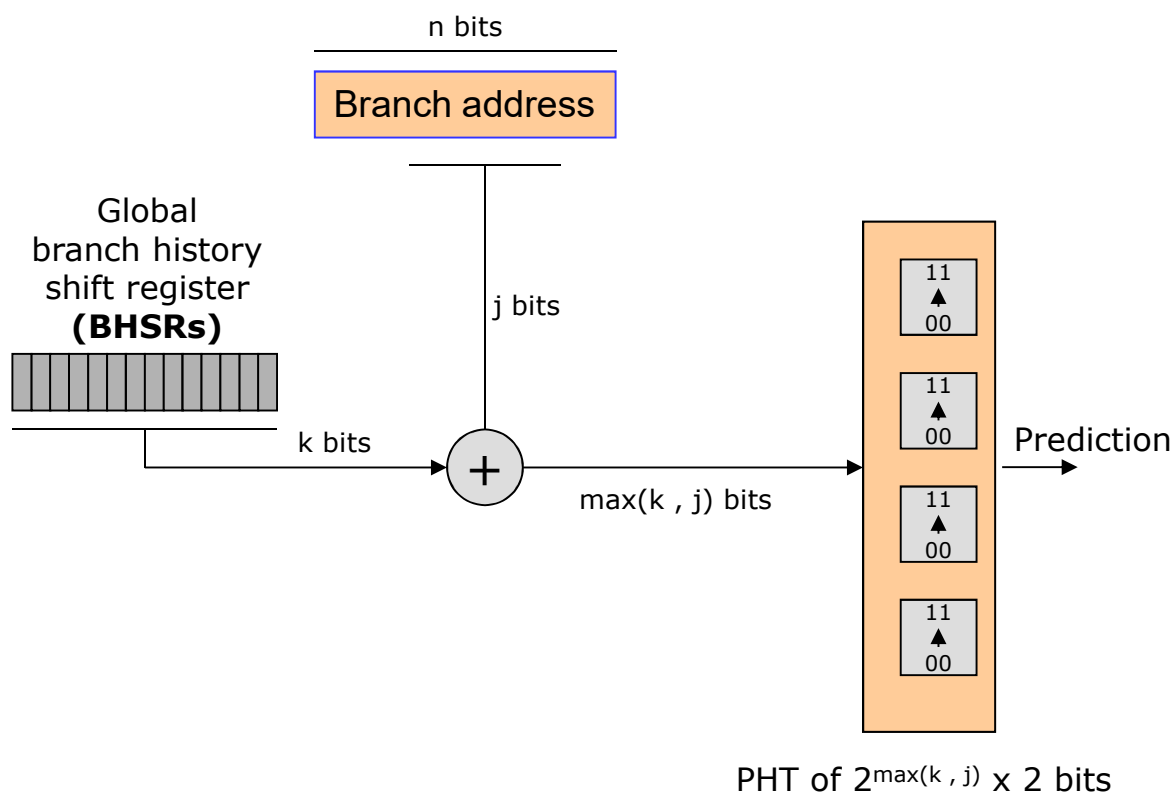





# Predicción en 2 niveles

gshare correlated branch predictor – McFarling, 1993:

- $j$  bits de la dirección de salto se usan como función de hashing (bitwise XOR) con  $k$  bits del Global-BHSR. Los  $\max(k, j)$  bits resultantes se usan para indexar una PHT de predictores de 2 bits de historia



 2-bit saturating counter FSM

# Predicción Híbrida (tournament predictors)

## 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

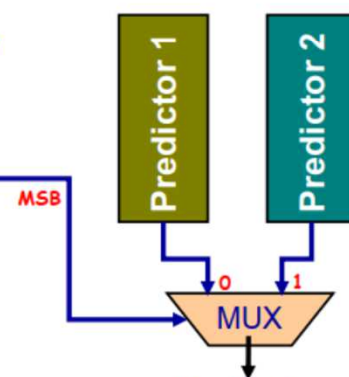
## Implementación del mecanismo de selección

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

Instrucción captada

Dirección

Tabla de Selección



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

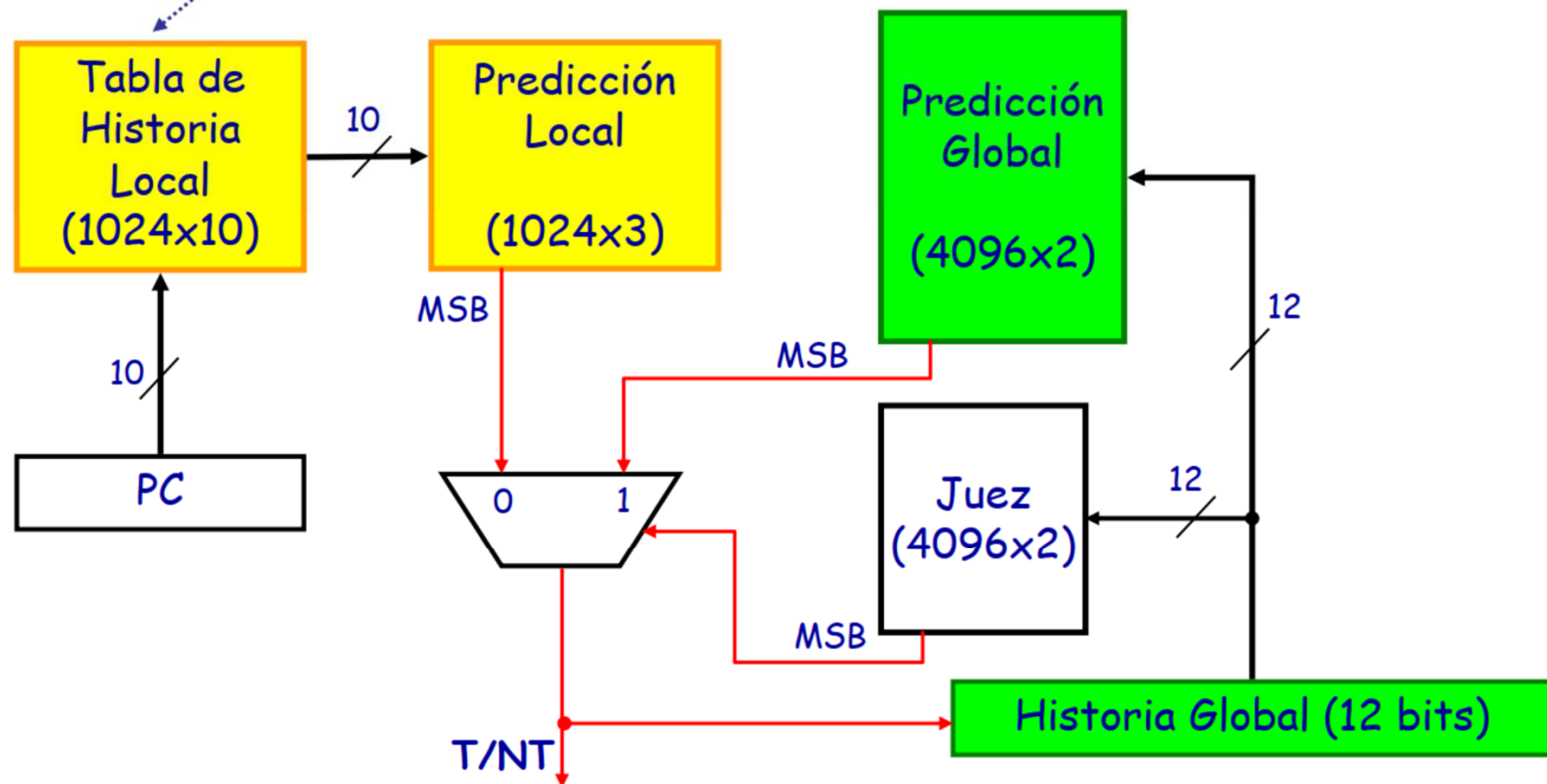
- Si P2 acierta más que P1  
⇒ *Cont* aumenta
- Si P1 acierta más que P2  
⇒ *Cont* disminuye

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

# Ejemplo: Tournament predictor del Alpha 21264

Funcional

Comportamiento de las 10 últimas  
ejecuciones de 1024 saltos



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

# Otros predictores actuales

---

- El predictor TAGE (TAGged GEometric length predictor) se basa en varias tablas predictoras indexadas a través de funciones independientes de la historia de saltos global y la dirección del PC.
- Predictores neuronales
  - Perceptrones
  - Backpropagation neural networks
  - LVQ (Learning vector quantization)
- Algoritmos genéticos

Manejo del flujo de datos  
(entre registros)

# Flujo de datos entre registros

- Técnicas tendientes a optimizar la ejecución de instrucciones del tipo aritmético-lógicas (ALU) en el núcleo superescalar del procesador
- Las instrucciones tipo ALU realizan las tareas “reales” del programa. Las instrucciones de salto y de memoria cumplen los roles de proveer nuevas instrucciones o datos (respectivamente) a este flujo de trabajo
- En una arquitectura tipo load/store las instrucciones tipo ALU especifican operaciones a ser realizadas entre operandos almacenados en registros

Formato típico de operación:

$$R_i = F(R_j, R_k)$$

con

$R_i$ : registro destino

$R_j$  y  $R_k$ : operandos fuente

$F$ : operación a realizar

Si  $R_j$  ó  $R_k$  no están disponibles

Dependencia de datos (RAW)

Si  $F$  no está disponible

Dependencia estructural (recursos)

Si  $R_i$  no está disponible

Dependencias falsas (WAR o WAW)

# Tratamiento de dependencias falsas

---

Dependencias de salida y anti-dependencias debidas a reuso de registros

Generadas por el compilador al asignar registros del procesador a variables del programa

Escritura de un registro: **DEFINICIÓN**

Lectura de un registro: **USO**

Luego de una **definición** puede haber varios **usos** de un registro

**RANGO DE VIDA** de un valor: duración entre la **definición** y el último **uso** de un registro

Dependencias WAR y WAW se producen cuando se **redefine** un registro durante un **rango de vida** previo

# Tratamiento de dependencias falsas

---

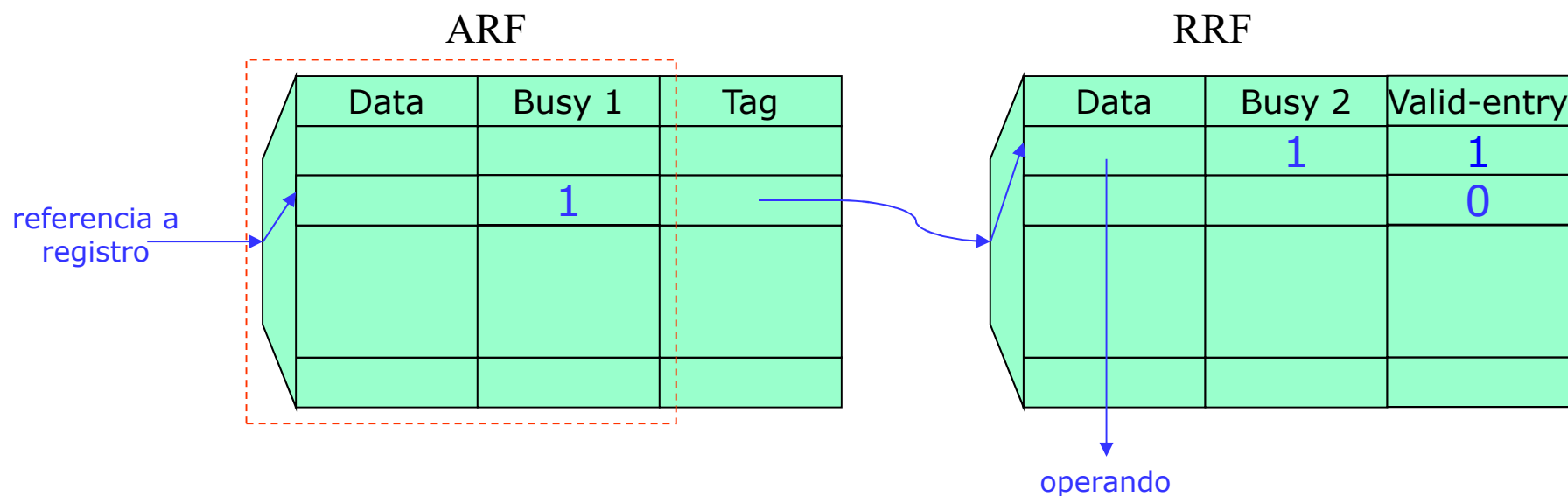
Técnicas para tratar dependencias falsas:

- **Frenado:** detener la instrucción que redefine el registro hasta que finalice el rango de vida previo
- **Renombre de registros:** Asignación dinámica de distintos nombres a las múltiples definiciones de un registro de la arquitectura
  - Uso de **banco de renombre** (*Rename Register File - RRF*) separado del **banco de registros** de la arquitectura (*Architected Register File - ARF*)
  - **Banco de renombre** (RRF) implementado como **parte del buffer de reordenamiento** (ROB)
  - **Pooled Registers:** Implementa el RRF y el ARF juntos en un mismo banco
  - **Map Table:** Usada en la FPU del IBM RS/6000



# Renombre de registros

## RRF separado del ARF

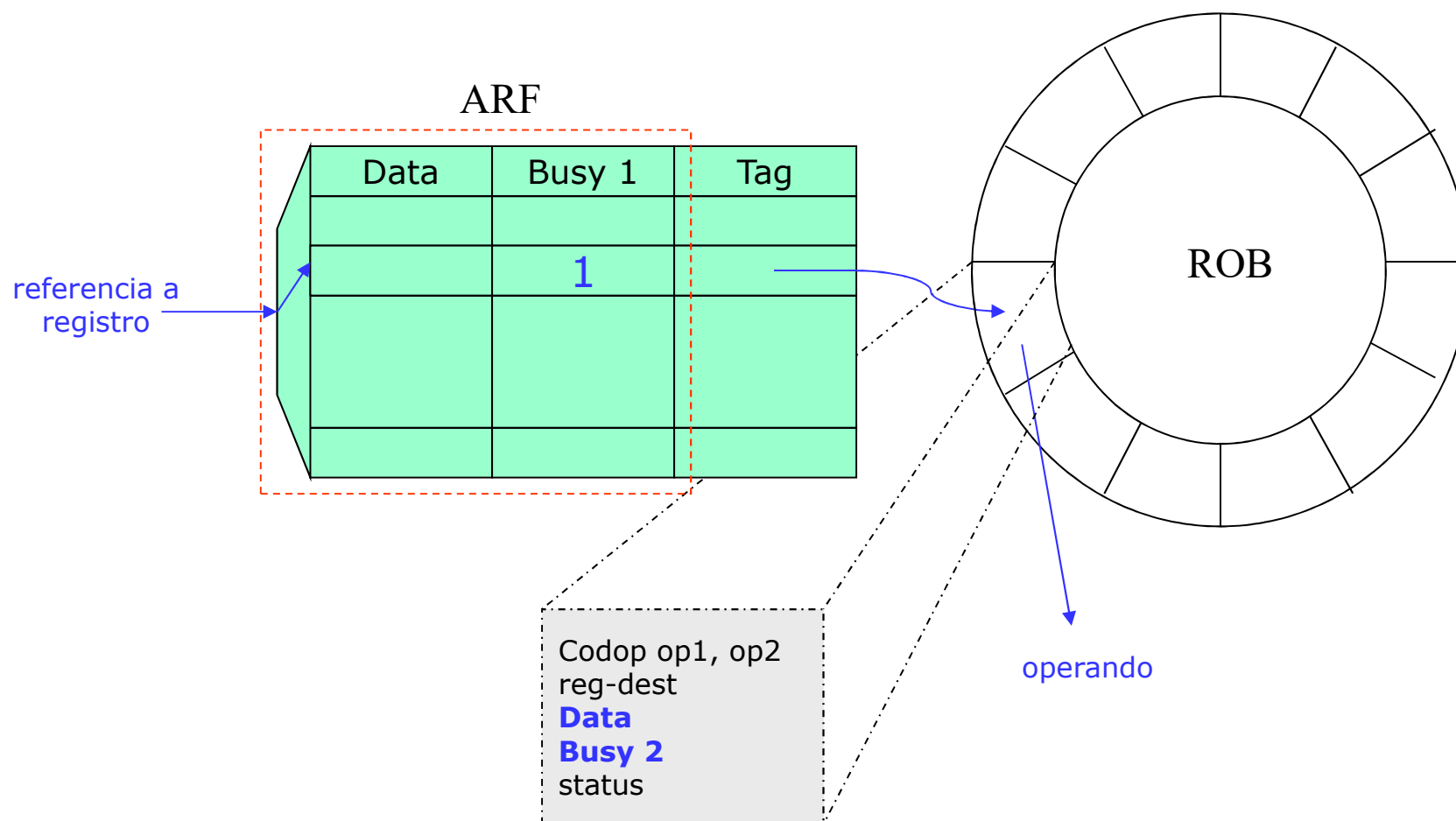


**Busy = 1** : El registro está ocupado y no puede ser utilizado ya que alguna instrucción en ejecución cambiará su valor

**0** : El valor de registro está disponible y puede ser leído como operando

# Renombre de registros

## RRF dentro del ROB



# Renombre de registros

---

## Comparación de las técnicas ARF-RRF y RRF-ROB

- ↓ • Ambas técnicas requieren el agregado al ARF de una columna adicional para direccionar (Tag)
- ↑ • En ARF-RRF ambas tablas pueden estar juntas dentro del chip
- ↓ • En RRF-ROB la tabla RRF está al final del pipeline (en ROB) y ARF al principio (en Decode o dispatch)
- ↓ • En RRF-ROB todas las entradas del ROB tienen soporte para renombre aunque no siempre se usan (Ej: saltos, Stores, etc)
- ↓ • En ARF-RRF son necesarios puertos adicionales desde las ALUs hasta la tabla RRF. En ROB ya existe la conexión de datos entre ALUs y ROB

# Funcionamiento del renombre de registros

Funcional

El renombre de registros involucra 3 tareas:

## 1. Lectura de operandos

- En etapa de Decode o Dispatch
- Lectura de operandos fuente de la instrucción
- Procedimiento:
  - se lee Reg desde ARF
    - Si **Busy 1** = 0, entonces se lee el **Data** desde ARF
    - Si **Busy 1** = 1, el valor **Data** de ARF no es válido, entonces con **Tag** se accede al RRF:
      - Si **Busy 2** = 0, entonces se lee el **Data** (instr terminó Execute pero aún está en el ROB)
      - Si **Busy 2** = 1, la instr aún no ha sido ejecutada, entonces se guarda el **Tag** en la estación de reserva

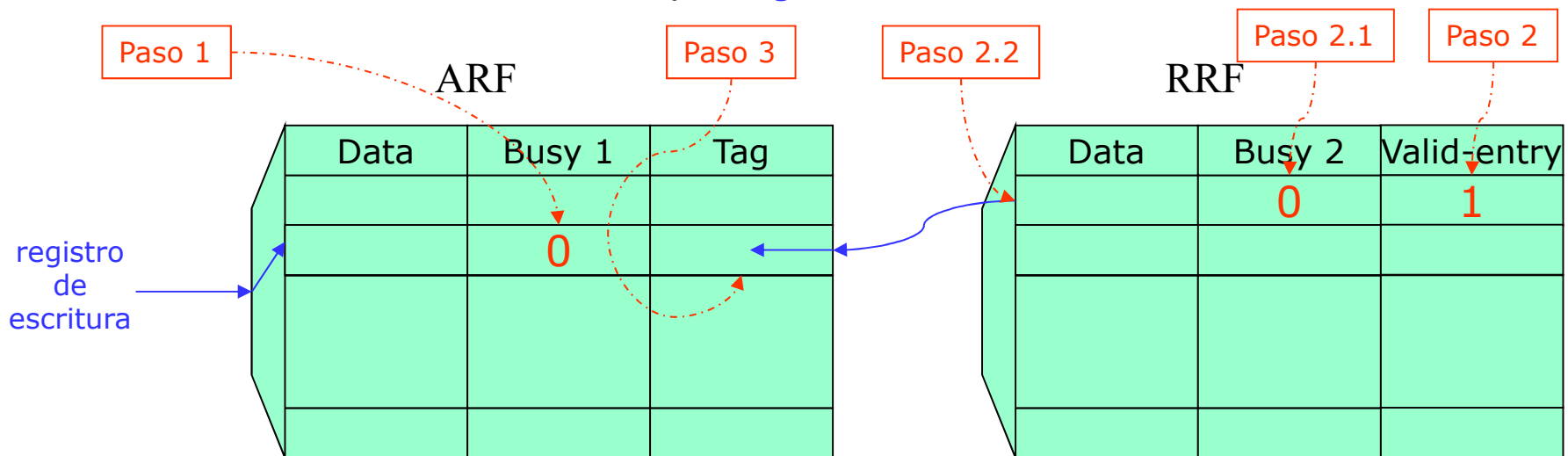
2. Alocación de registro destino
3. Actualización de registro

# Funcionamiento del renombre de registros

Funcional

## Alocación de registro destino:

- Se realiza en etapa de Decode o dispatch
- 3 pasos:
  1. Poner **Busy 1** = 1 para el registro que escribirá la instrucción
  2. Obtener un registro libre de la RRF y marcarlo como usado (**Valid-entry** = 1)
    - Poner su bit **Busy 2** = 1 (no executed yet)
    - Obtener el **Tag** correspondiente (índice en RRF)
  - Actualizar el campo **Tag** de la ARF

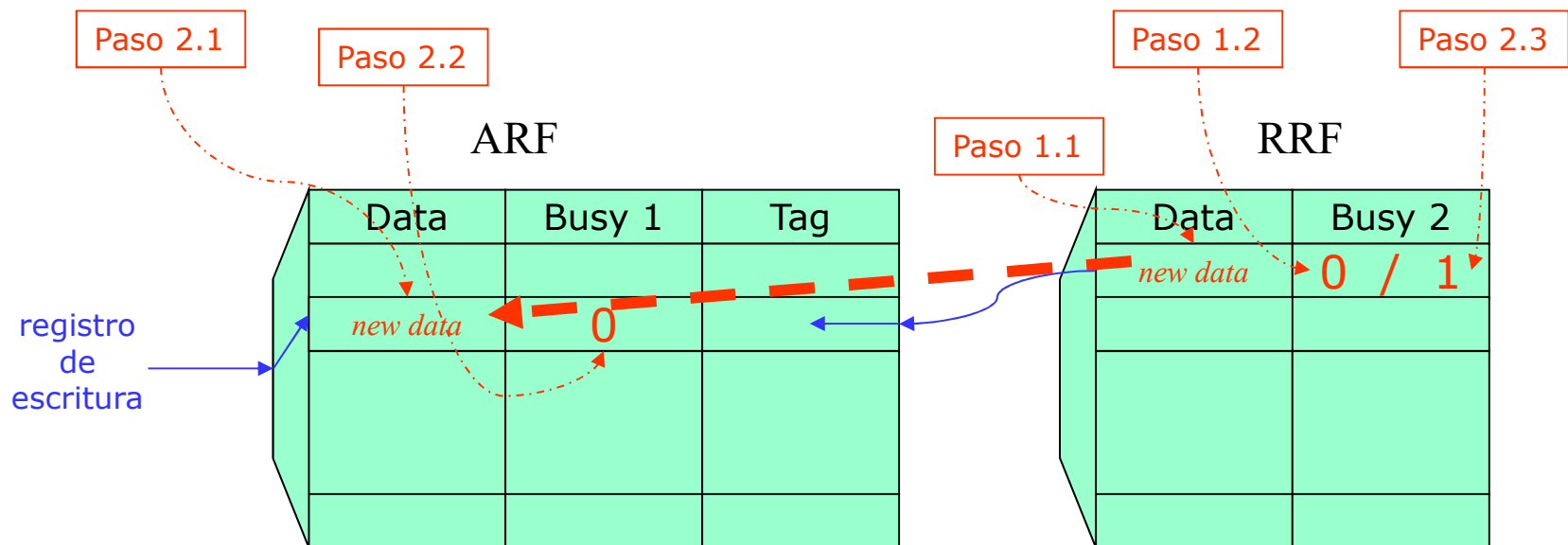


# Funcionamiento del renombre de registros

Funcional

## Actualización de registro (Update):

- Se realiza en etapa Complete
- 2 pasos:
  1. Cuando la instr termina (finish),
    - 1.1 Actualizar resultado en RRF[Tag]
    - 1.2 Poner **Busy 2** = 0
  2. Cuando la instr se completa (Complete),
    - 2.1 Actualizar Data RRF[Tag] al ARF correspondiente
    - 2.2 Poner **Busy 1** = 0
    - 2.3 Poner **Busy 2** = 1



# Renombre de registros

---

## Pooled Registers:

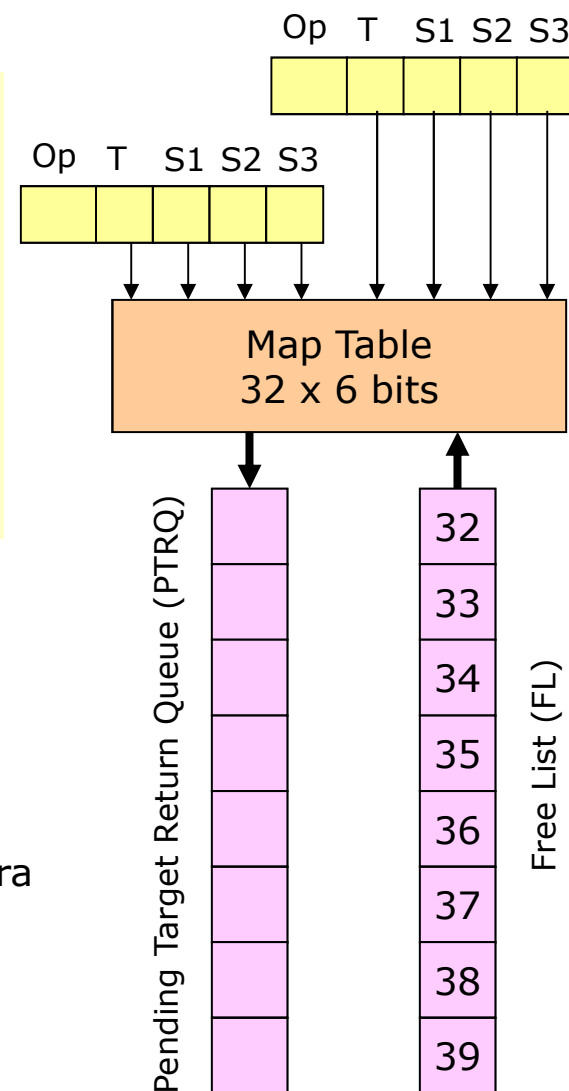
- Una única estructura física de registros que sirven como ARF y RRF
- Cualquier entrada se designa como RRF o ARF con un bit adicional
- ↑ • Ventaja: En la fase de Update no se necesita realizar copia desde RRF a ARF, basta con cambiar el bit de tipo y eliminar instancias anteriores del registro
- ↓ • Desventaja: Extrema complejidad de Hardware
- ↓ • Desventaja: Al momento de Swaping de tarea, cuesta identificar los registros ARF a fin de salvar el estado del procesador

# Renombre de registros

## Map Table (IBM RS/6000):

- 40 registros físicos que incluyen 32 de arquitectura
- La tabla de mapeo tiene 32 entradas de 6 bits c/u
- Cada entrada representa a un reg. de la arquitectura y referencia un registro físico
- FL es una cola que contiene los reg. físicos NO usados
- PTRQ contiene la lista de registros que fueron usados para renombrar registros de la arquitectura y han sido posteriormente re-renombrados en la tabla de mapeo
- Registros físicos en la PTRQ son devueltos a la FL luego de su último uso por parte de una instrucción

- ↑ • Forma más agresiva de renombre de registros
- ↓ • Alta complejidad de hardware
- ↓ • Es difícil detectar cuando un registro de la PTRQ se usa por última vez
- ↑ • No necesita actualización de los registros de arquitectura al final del ciclo de una instrucción (Complete stage)
- ↑ • Detección simple de los registros físicos asignados a registros de la arquitectura en caso de excepciones (Map Table)





# Renombre de registros

## Otra variante de tabla de mapeo

- Banco de M registros físicos que mapean N registros de arq ( $M > N$ )
- Funciona como buffer circular
  - **Valid:**
    - 0 : entrada sin uso
    - 1 : entrada usada
  - **Reg des:**
    - registro de arq al cual se asignó el registro físico
  - **value :**
    - el valor del registro
  - **Busy :**
    - validación del dato
    - 0 : valor disponible
    - 1 : valor no actualizado
  - **Last :**
    - indica la última definición de un registro de arq

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	0				
4	0				
.	.				
.	.				
.	.				
.	0				

# Renombre de registros

mul r2, r0, r1

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	0				
4	0				
.	.				
.	.				
.	.				
.	0				

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	0				
.	.				
.	.				
.	.				
.	0				

# Renombre de registros

mul r2, r0, r1

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	0				
.	.				
.	.				
.	.				
.	0				

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	0				
.	.				
.	.				
.	.				
.	0				

r0=0 r1=10

# Renombre de registros

add r3, r1, r2

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	0				
.	.				
.	.				
.	.				
.	0				

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	1	3		0	1
5	.				
.	.				
.	.				
.	0				

# Renombre de registros

add r3, r1, r2

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	0				
.	.				
.	.				
.	.				
.	0				

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	1	3		0	1
5	.				
.	.				
.	.				
.	0				

¿r2? r1=10

# Renombre de registros

sub r2, r0, r1

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	1
4	1	3		0	1
5	.				
.	.				
.	.				
.	0				

→

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	0
4	1	3		0	1
5	1	2		0	1
.	.				
.	.				
.	0				

# Renombre de registros

termina mul r2, r0, r1

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2		0	0
4	1	3		0	1
→ 5	1	2		0	1
.	.				
.	.				
.	0				

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2	0	1	0
4	1	3		0	1
→ 5	1	2		0	1
.	.				
.	.				
.	0				

# Renombre de registros

Se retira instrucción con resultado en posición 0

	valid	reg. des.	value	busy	last
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2	0	1	0
4	1	3		0	1
5	1	2		0	1
.	.				
.	.				
.	0				

	valid	reg. des.	value	busy	last
0	0	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	1
3	1	2	0	1	0
4	1	3		0	1
5	1	2		0	1
.	.				
.	.				
.	0				



# Dependencias verdaderas de datos

---

Las dependencias de datos verdaderas (RAW) obligan a serializar la ejecución de instrucciones debido a la relación “*productor-consumidor*” entre instrucciones que prducen resultados y otras que los usan como operandos de entrada

Aún cuando estas dependencias no se pueden eliminar, es necesario proveer al procesador de mecanismos para detectar las mismas y serializar las instrucciones que tienen este tipo de conflicto

$r1 = r2 * r3$   
 $r5 = r1 + r8$

# Dependencias verdaderas de datos

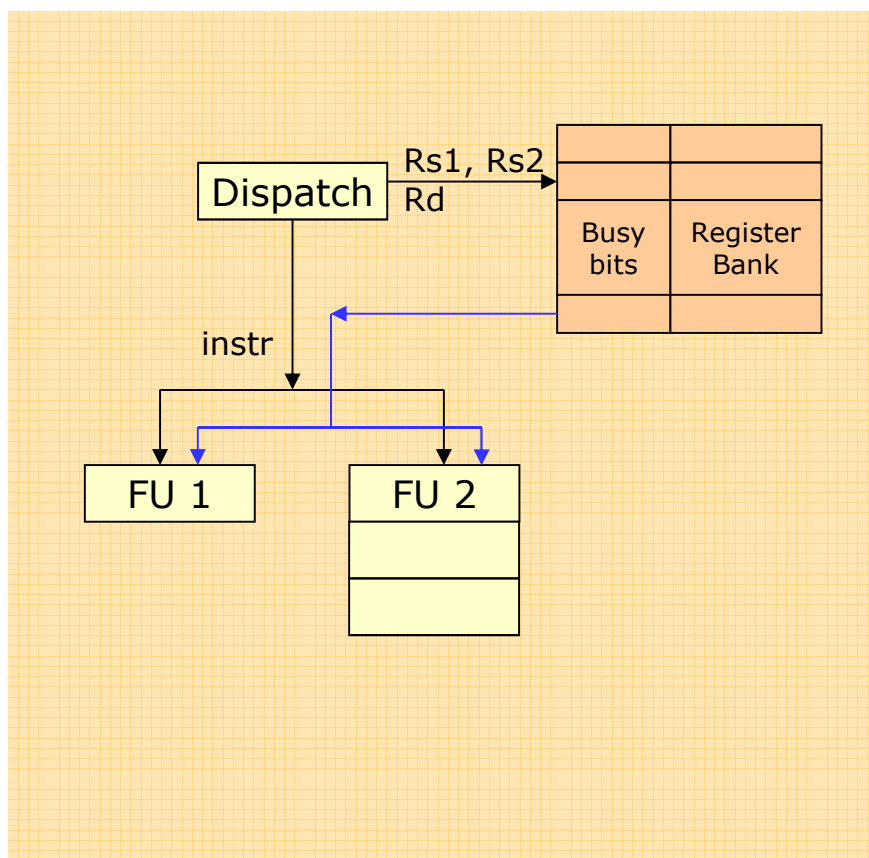
---

## Un poco de historia... Algoritmo de Tomasulo

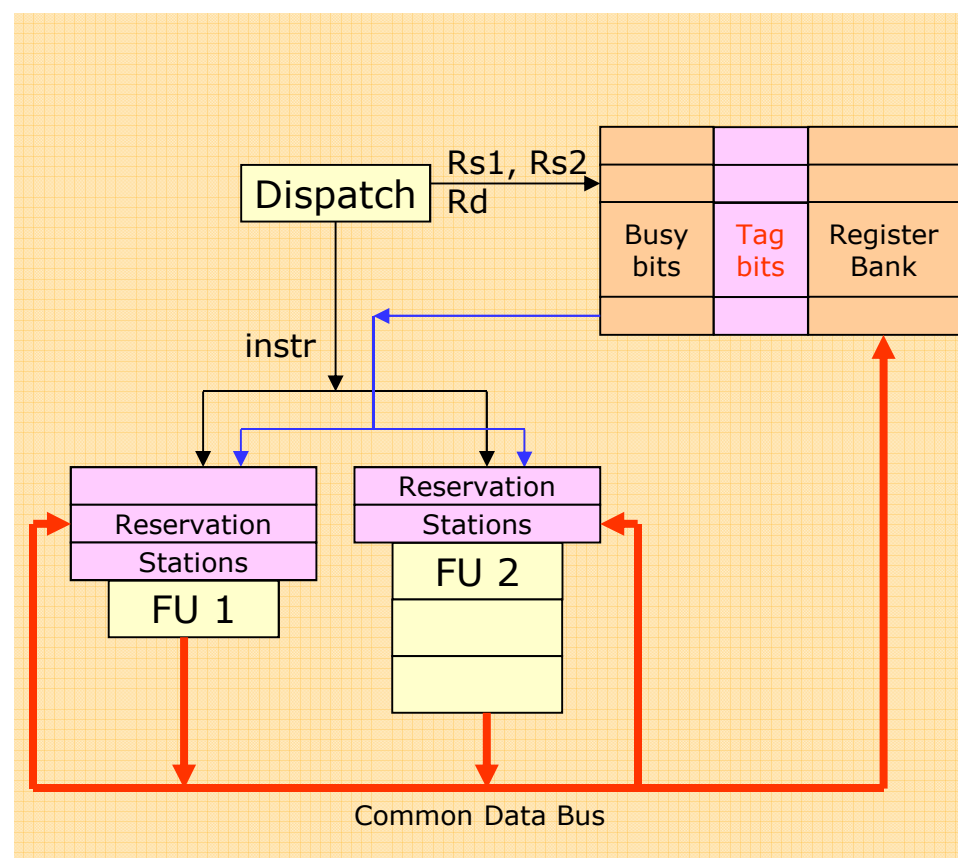
- Técnica precursora de los algoritmos modernos de detección de dependencias de datos verdaderas (RAW)
- Propuesto por Tomasulo en 1967 para la FPU del IBM 360/91
- No sólo detecta dependencias RAW sino que elimina dependencias falsas (WAR y WAW) como efecto colateral
- Se basa en la adición de tres mecanismos a la ALU clásica:
  - Unidades de reserva (*reservation stations*)
  - Bus de datos común (*common data bus - CDB*)
  - etiquetas de registros (*register tags*)
- La versión original no soporta tratamiento de excepciones con mantenimiento de la consistencia secuencial

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo



ALU clásica

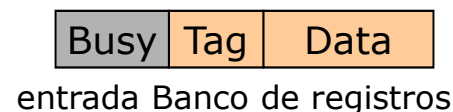
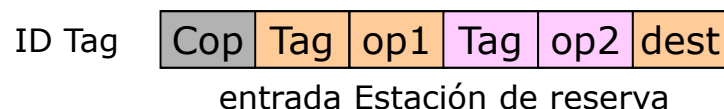


ALU de Tomasulo

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo

- Las estaciones de reserva se pueden ver como unidades funcionales virtuales que reciben una instrucción para ejecutar y al cabo de cierto número de ciclos se genera un resultado (en realidad el resultado lo generará la unidad funcional asociada a la estación)
- Cada generador de futuros resultados se identifica con un Tag (cada estación de reserva)
- Si un registro no está disponible (está no-valid), su valor se reemplaza por el tag del generador de dicho valor
- Cuando una unidad funcional genera un resultado, lo almacena en la ER de la instrucción y ésta lo envía por el CDB a todos los posibles receptores (banco de registros, estaciones de reserva, etc) que tengan dicho tag



# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Implementación

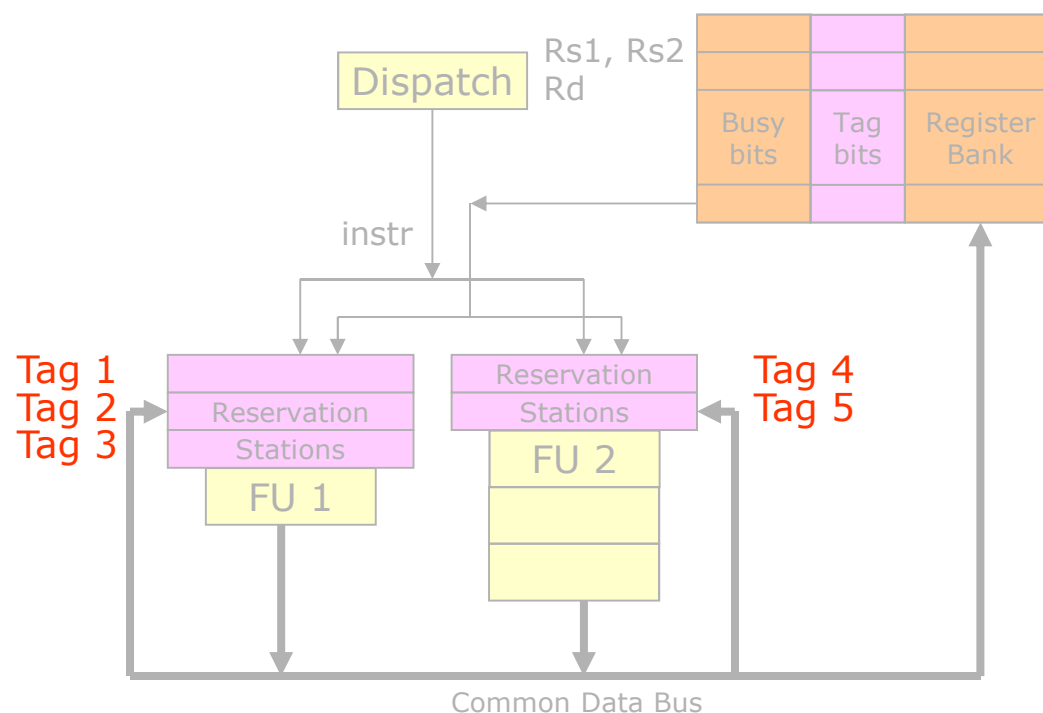
Una entrada de estación de reserva con Tag = 0 para algún operando se asume que contiene el valor del dato.

Si el Tag es  $\neq 0$ , el valor lo generará la instrucción que está en la estación de reserva identificada con dicho Tag

	op1		op2		
add	0	500	2	---	r4

El op1 tiene Tag 0 por lo que su valor es 500.

El op2 tiene Tag 2, por lo que el resultado lo generará la instrucción en la estación de reserva 2



# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Ejemplo

- Suponer una etapa de ejecución con soporte para instrucciones de 2 operandos fuente y una referencia a registro destino
- La unidad de despacho emite 2 instrucciones por ciclo
- Hay 2 unidades funcionales:
  - **Adder** : Con una estación de reserva de 3 entradas (con Tags 1, 2 y 3)  
2 ciclos de reloj de latencia
  - **Mult** : Con una estación de reserva de 2 entradas (con Tags 4 y 5)  
3 ciclos de reloj de latencia
- Una instrucción puede iniciar su ejecución en el mismo ciclo en que es despachada a la estación de reserva
- Las unidades funcionales pueden adelantar sus resultados en su último ciclo de ejecución a los distintos destinos que los necesiten
- Cualquier unidad que recibe un operando puede iniciar su ejecución en el siguiente ciclo de reloj

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Ejemplo

S1 : add r4, r0, r8  
S2 : mul r2, r0, r4  
S3 : add r4, r4, r8  
S4 : mul r8, r4, r2

### Ciclo 1 Despacho de S1 y S2

	Tag	Op 1	Tag	Op 2
S1 1	0	6,0	0	7,8
- 2				
- 3				

S1 Adder

Functional unit 1

	Tag	Op 1	Tag	Op 2
S2 4	0	6,0	1	---
- 5				

Mult

Functional unit 2

	Busy	Tag	Data
0			6,0
2	1	4	3,5
4	1	1	10,0
8			7,8

Register Bank

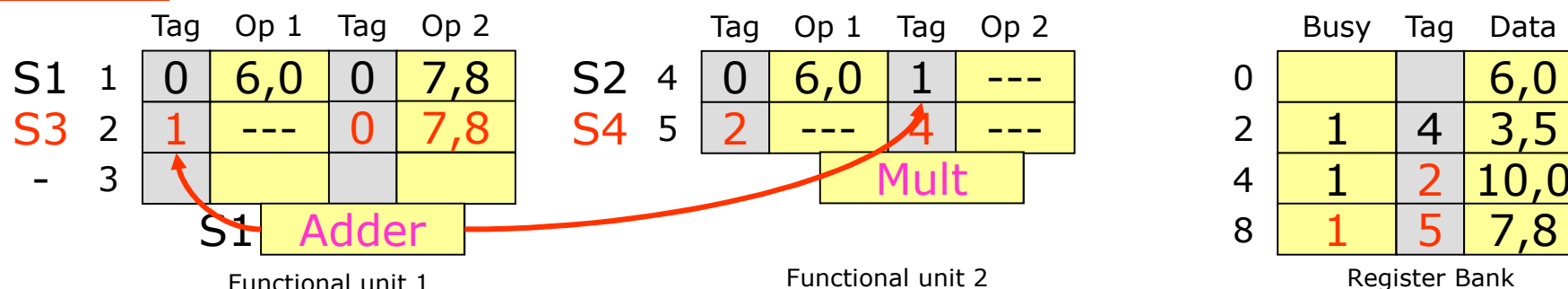
- Se despachan las instrucciones S1 y S2 a las estaciones de reserva 1 y 4
- Los registros destino de esas instrucciones son r4 y r2
- Se setean a "1" los bits Busy de r4 y r2
- Como S1 es despachada a la estación 1, el Tag de r4 se pone en 1
- Como S2 es despachada a la estación 4, el Tag de r2 se pone en 4
- S1 tiene sus 2 operandos entonces comienza a ejecutarse en este ciclo
- S2 debe esperar el resultado de S1 (r4) entonces op 2 se setea con Tag 1

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Ejemplo

S1 : add r4, r0, r8  
S2 : mul r2, r0, r4  
S3 : add r4, r4, r8  
S4 : mul r8, r4, r2

### Ciclo 2 Despacho de S3 y S4



- Se despachan las instrucciones **S3** y **S4** a las estaciones de reserva **2** y **5**
- **S3** necesita del resultado de **S1** así que se almacena con un Tag = **1**
- **S4** depende de **S3** y de **S2**, entonces se almacena con Tags **2** y **4**
- **S3** actualiza **r4** por lo que su Tag en el banco de registros se actualiza a **2**
- **S4** actualiza **r8** por lo que su Tag en el banco de registros se pone a **5** y su bit **Busy** se pone a **1**
- **S1** en su último ciclo de ejecución, manda el token **<r4,13,8>** por el **CDB**



# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Ejemplo

S1 : add r4, r0, r8  
S2 : mul r2, r0, r4  
S3 : add r4, r4, r8  
S4 : mul r8, r4, r2

### Ciclo 3

	Tag	Op 1	Tag	Op 2
- 1				
S3 2	0	13,8	0	7,8
- 3				

S3 Adder

Functional unit 1

	Tag	Op 1	Tag	Op 2
S2 4	0	6,0	0	13,8
S4 5	2	---	4	---

S2 Mult

Functional unit 2

	Busy	Tag	Data
0			6,0
2	1	4	3,5
4	1	2	10,0
8	1	5	7,8

Register Bank

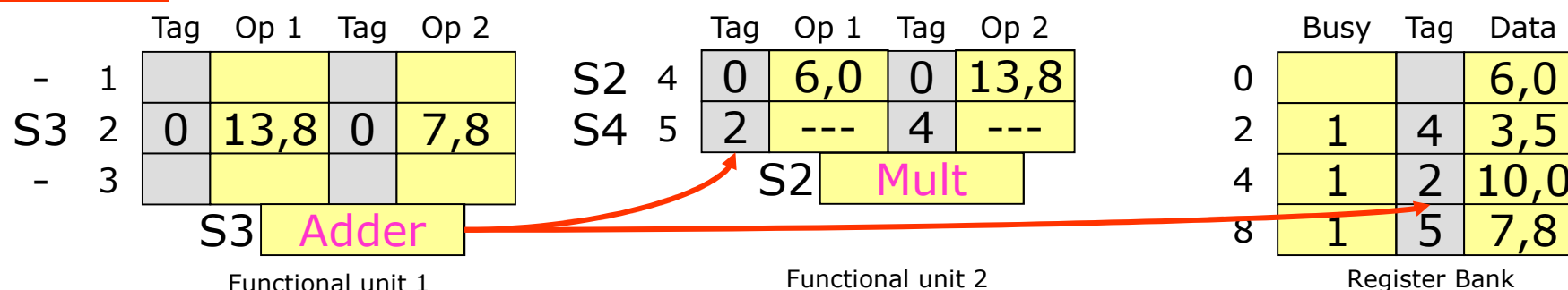
- Se actualizó el **operando 1** de S3 y el **operando 2** de S2 ambos a **13,8**
- Se libera la estación de reserva 1
- Comienza la ejecución de S3 en el **Adder** y de S2 en el **Mult**

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Ejemplo

S1 : add r4, r0, r8  
S2 : mul r2, r0, r4  
S3 : add r4, r4, r8  
S4 : mul r8, r4, r2

### Ciclo 4



- Último ciclo de ejecución de S3 en el Adder por lo que envía el token  $\langle r4, 21, 6 \rangle$  al CDB

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Ejemplo

S1 : add r4, r0, r8  
S2 : mul r2, r0, r4  
S3 : add r4, r4, r8  
S4 : mul r8, r4, r2

### Ciclo 5

	Tag	Op 1	Tag	Op 2
- 1				
- 2				
- 3				

Adder

Functional unit 1

	Tag	Op 1	Tag	Op 2
S2 4	0	6,0	0	13,8
S4 5	0	21,6	4	-

S2 Mult

Functional unit 2

	Busy	Tag	Data
0			6,0
2	1	4	3,5
4			21,6
8	1	5	7,8

Register Bank

- El token  $\langle r4, 21,6 \rangle$  del CDB actualizó el operando 1 de la estación de reserva 5
- El token  $\langle r4, 21,6 \rangle$  del CDB actualizó el registro 4 del banco de registros de modo que su bit **Busy** se pone en 0 (resultado actualizado)
- Último ciclo de ejecución de S2 en el Mult por lo que envía el token  $\langle r2, 82,8 \rangle$  al CDB

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo: Ejemplo

S1 : add r4, r0, r8  
S2 : mul r2, r0, r4  
S3 : add r4, r4, r8  
S4 : mul r8, r4, r2

### Ciclo 6

	Tag	Op 1	Tag	Op 2
- 1				
- 2				
- 3				

Adder

Functional unit 1

	Tag	Op 1	Tag	Op 2
- 4				
S4 5	0	21,6	0	82,8

S4 Mult

Functional unit 2

	Busy	Tag	Data
0			6,0
2			82,8
4			21,6
8	1	5	7,8

Register Bank

- El token **<r2, 82,8>** del **CDB** actualizó el registro 2 del banco de registros de modo que su bit **Busy** se pone en **0** (resultado actualizado)
- Comienza la ejecución de **S4** en el **Mult**

# Dependencias verdaderas de datos

---

## Algoritmo de Tomasulo

### Tratamiento de RAW

Mediante el uso de Tags que permiten visualizar mas registros virtuales:  
banco de registros + estaciones de reserva

### Eliminación de WAR

Mediante la capacidad de almacenar el resultado de la ejecución en la estación de reserva de la instrucción que lo generó para ser usado por las instrucciones dependientes

### Eliminación de WAW

Solamente la última instrucción en orden del programa escribe en el banco de registros

# Dependencias verdaderas de datos

---

## Algoritmo de Tomasulo

### Desventajas:

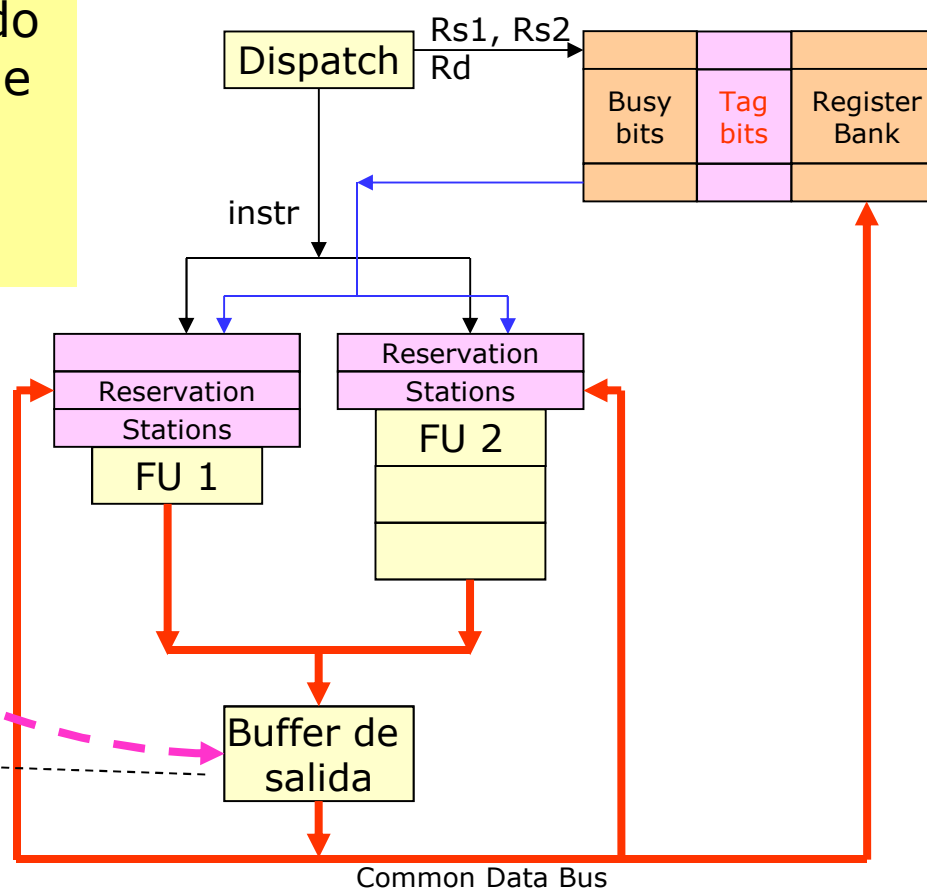
- 1 Desperdicio de las estaciones de reserva ya que son usadas como buffers temporales para almacenar el resultado de la instrucción hasta el final del rango de vida del valor
- 2 No soporta tratamiento preciso de excepciones

# Dependencias verdaderas de datos

## Algoritmo de Tomasulo

**Solución de desventaja 1:** Agregado de un buffer temporal de salida que almacena los resultados liberando la estación de reserva correspondiente

Tag	Rd	value



# Dependencias verdaderas de datos

## Algoritmo de Tomasulo

Desventaja 2: El algoritmo no soporta tratamiento preciso de excepciones

Al final del segundo ciclo S1 debería haber actualizado el contenido de r4 pero cuando se despachó S3 al principio del segundo ciclo cambió el Tag de r4 de 1 a 2

Si se produce una excepción (por ejemplo de S2 en el ciclo 3) no podrá ser posible recuperar el estado exacto del procesador ya que r4 no representa el resultado de haber ejecutado S1

Estado del banco de registros según Tomasulo

	Busy	Tag	Data
0			6,0
2	1	4	3,5
4	1	2	10,0
8	1	5	7,8

Register Bank

Estado del banco de registros con consistencia secuencial

	Busy	Tag	Data
0			6,0
2	1	4	3,5
4	1	2	13,8
8	1	5	7,8

Register Bank

Erroneo

Correcto



# Dependencias verdaderas de datos

## Algoritmo de Tomasulo

### Solución 2

Uso del buffer temporal de salida para almacenar el orden de emisión de las instrucciones, a fin de actualizar el banco de registros en orden del programa cuando termina cada instrucción

Precursor del buffer de reordenamiento de los actuales procesadores

Soluciona la inconsistencia temporal del banco de registros ante excepciones a costa de introducir una inconsistencia potencial dentro del mismo banco entre el tag y el valor

Estado del banco de registros con consistencia secuencial

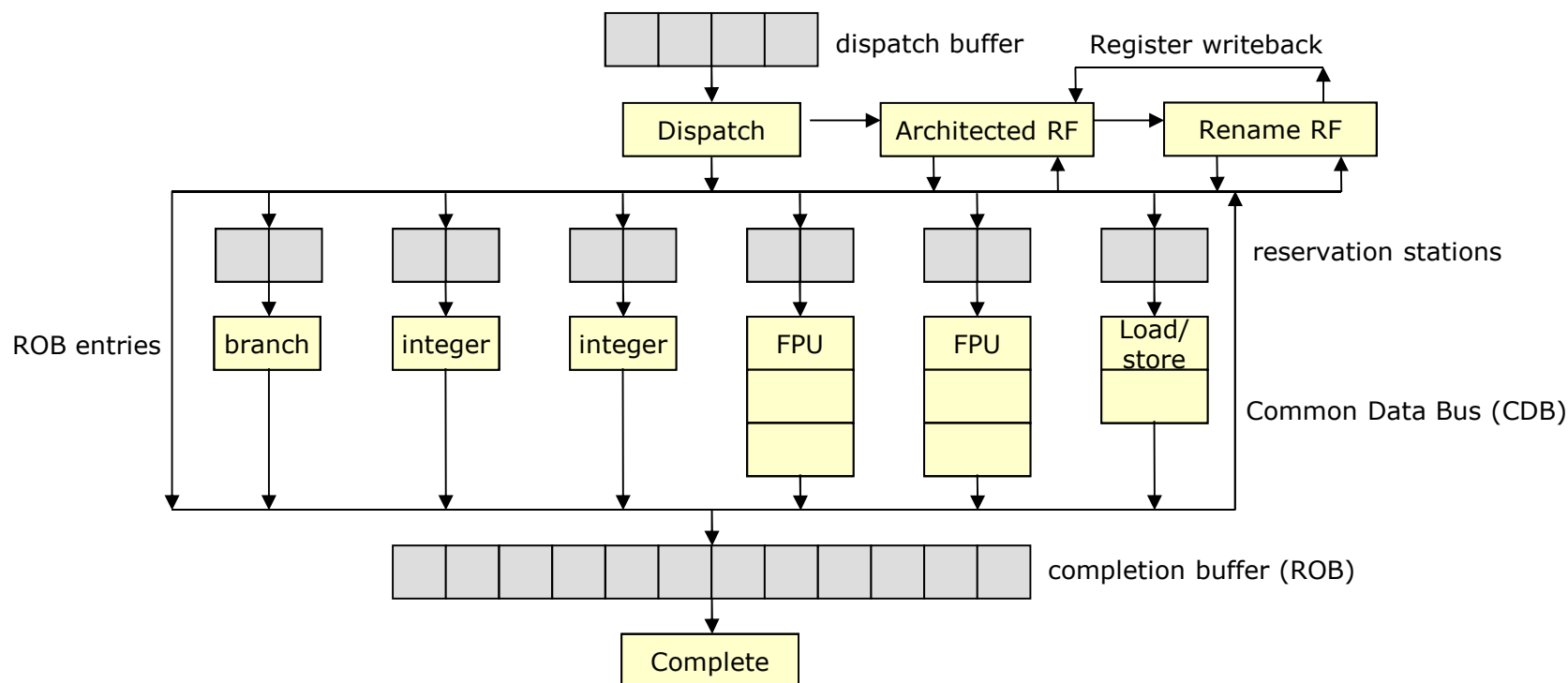
	Busy	Tag	Data
0			6,0
2	1	4	3,5
4	1	2	13,8
8	1	5	7,8

Register Bank

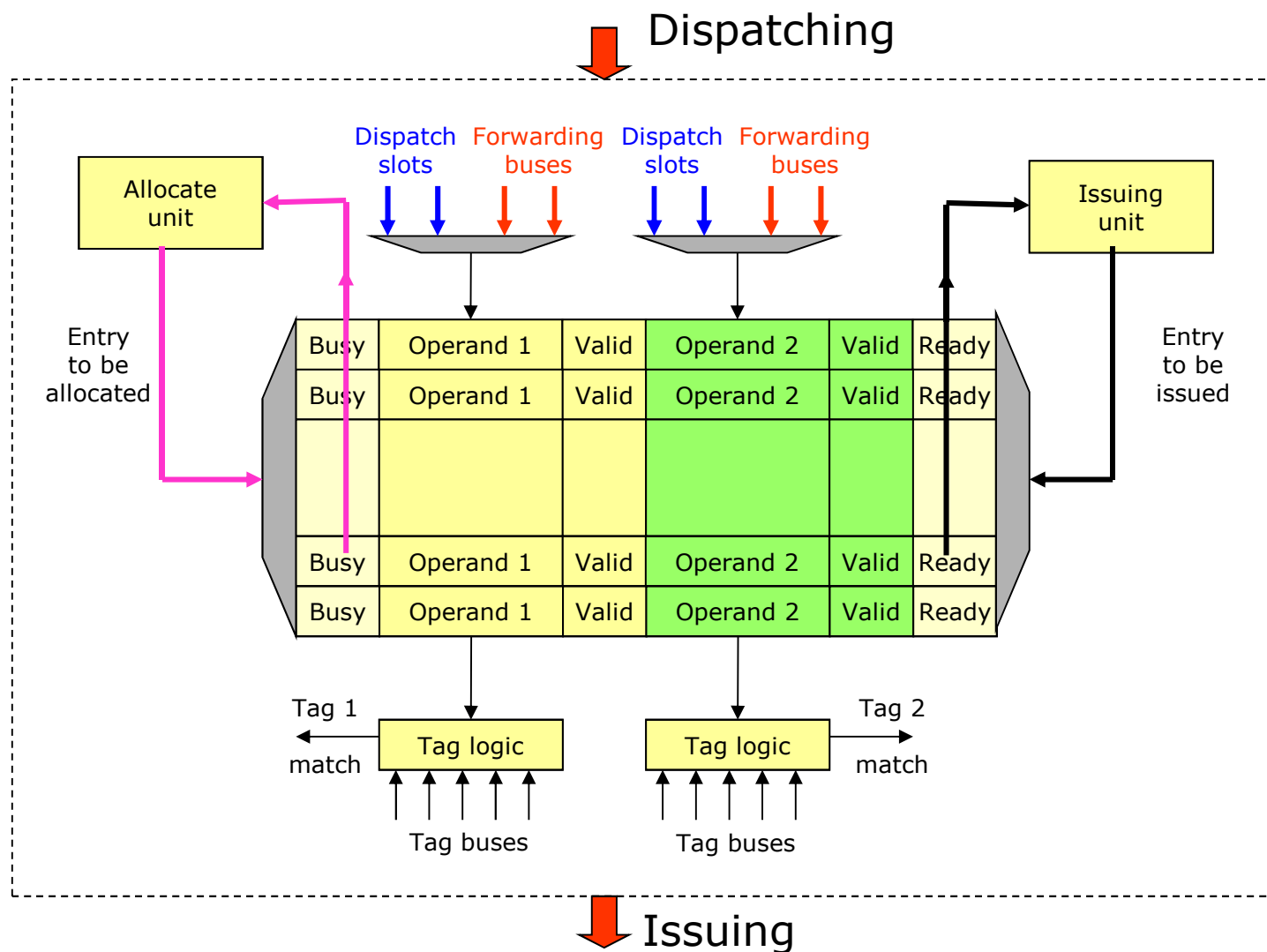
13,8 es el resultado de la instrucción de tag 1 (S1) pero en el registro 4 el campo Tag tiene un 2 (S3)

# Núcleo dinámico de ejecución

Segmento de ejecución “fuera de orden” de un procesador superescalar

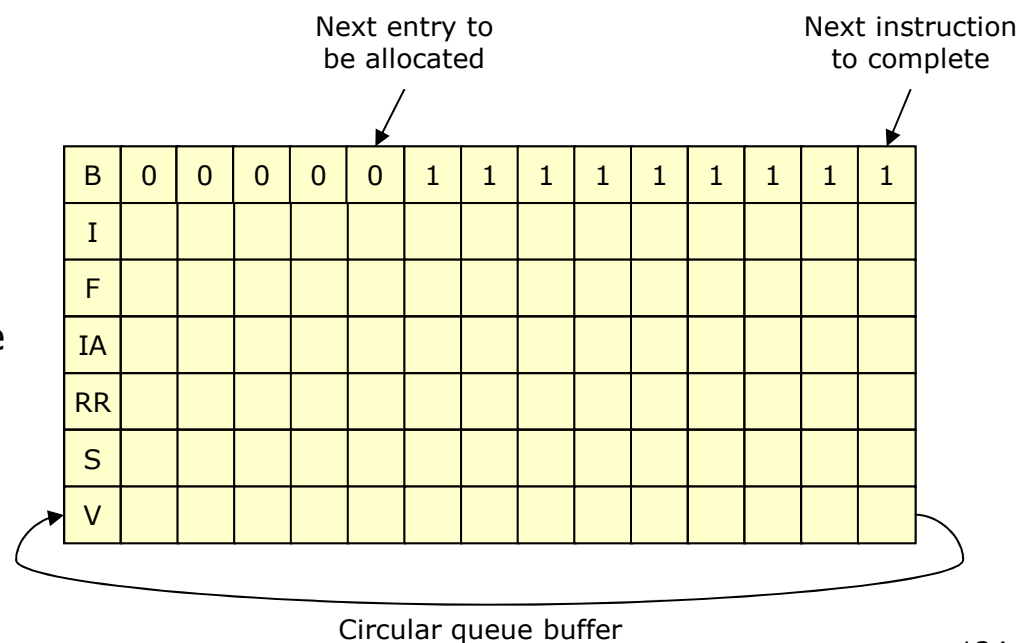


# Estaciones de reserva



# Buffer de reordenamiento (ROB)

- **Busy (B):** La entrada correspondiente del ROB está siendo usada
- **Issued (I):** La instrucción está en ejecución en alguna unidad funcional
- **Finished (F):** La instrucción terminó la ejecución y espera a ser completada  
 Issued = Finished = 0: la instrucción está en espera en alguna estación de reserva
- **Instruction address (IA):** Dirección de la instrucción en el I-buffer
- **Rename register (RR):** Registro de renombre (si se usa RRF-ROB)
- **Speculative (S):** Indicador de ejecución especulativa de la instrucción (Puede contener además bits indicando el nivel de salto especulativo )
- **Valid (V):** En caso de instrucción especulativa cuya rama resulte incorrecta la instrucción se marca como NO válida



# Estrategias de acceso a operandos

El administrador de instrucciones (dynamic instruction scheduler) maneja el flujo de operandos hacia las instrucciones en espera, el flujo de resultados desde las unidades funcionales y la activación (wake-up) de instrucciones en las estaciones de reserva

Según la política de acceso a los operandos hay 2 alternativas de implementación

## Acceso a operandos durante el despacho a las estaciones de reserva (Data captured Scheduling)

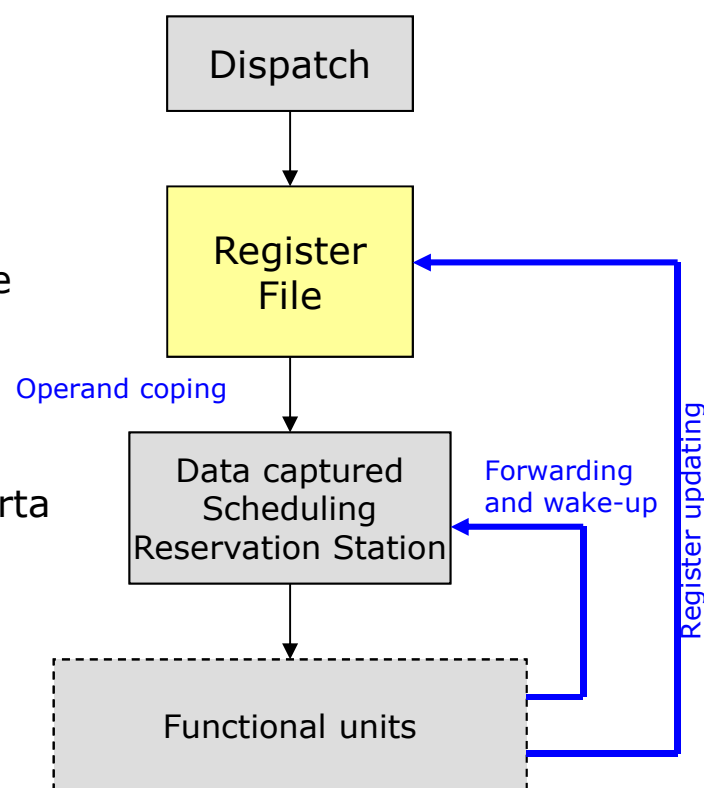
- Primeros procesadores superescalares
- Los operandos disponibles en el despacho se almacenan en la estación de reserva
- Si el operando no está disponible, se almacena un Tag

## Acceso a los operandos durante la emisión a las unidades funcionales (Non data captured Scheduling)

- Política más moderna, adoptada por algunos procesadores actuales
- Todos los operandos son accedidos durante la emisión a las unidades de ejecución

# Data captured Scheduling

- Durante el despacho a las estaciones de reserva se accede al banco de registros para recuperar operandos  
Si el operando no está disponible se utiliza un **Tag**
- Las instrucciones permanecen en la estación de reserva hasta tener todos sus operandos
- Al finalizar la ejecución de una instrucción el resultado se usa para actualizar el banco de registros y además se envía un token (**<Tag, Data>**) a las estaciones de reserva para actualizar los campos de **Tag** faltantes
- La **Issuing Unit** analiza todas las instrucciones y despierta aquellas que tienen todos sus operandos

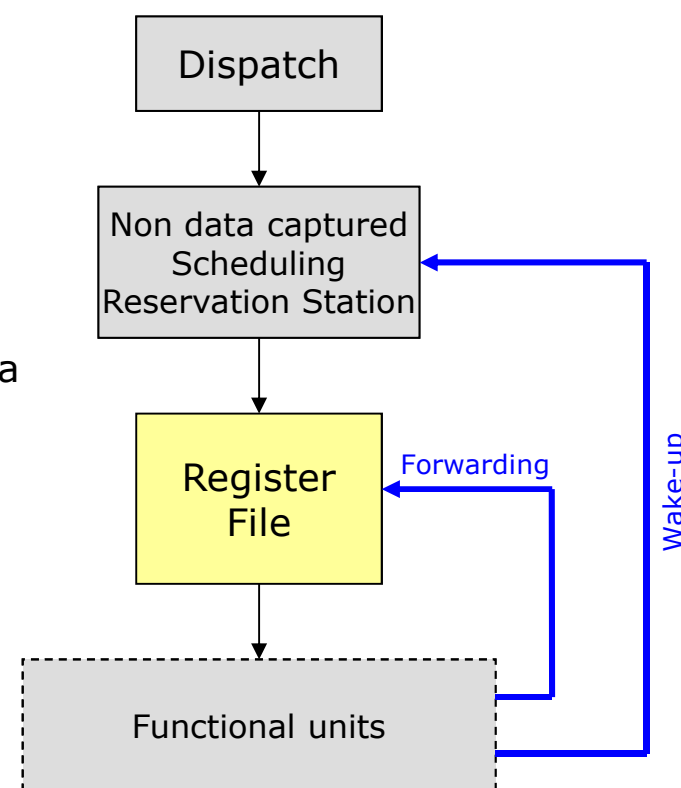


# Non data captured scheduling

- Durante el despacho a las estaciones de reserva **NO** se accede al banco de registros para recuperar operandos  
Para todos los operandos se utilizan **Tags**
- Al finalizar la ejecución de una instrucción el resultado se usa para actualizar el banco de registros (forwarding) y además se envía el **Tag** a las estaciones de reserva para actualizar la disponibilidad de operandos
- La **Issuing Unit** analiza todas las instrucciones y despierta aquellas que tienen todos sus **Tags** actualizados
- La instrucción activa accede al banco de registros durante la emisión a una unidad funcional para recuperar los operandos (que ya están disponibles)

## Ventaja:

- El dato se actualiza en un solo lugar
- Los buses hacia las estaciones de reserva son mas chicos (sólo el Tag)



## Manejo del flujo de memoria



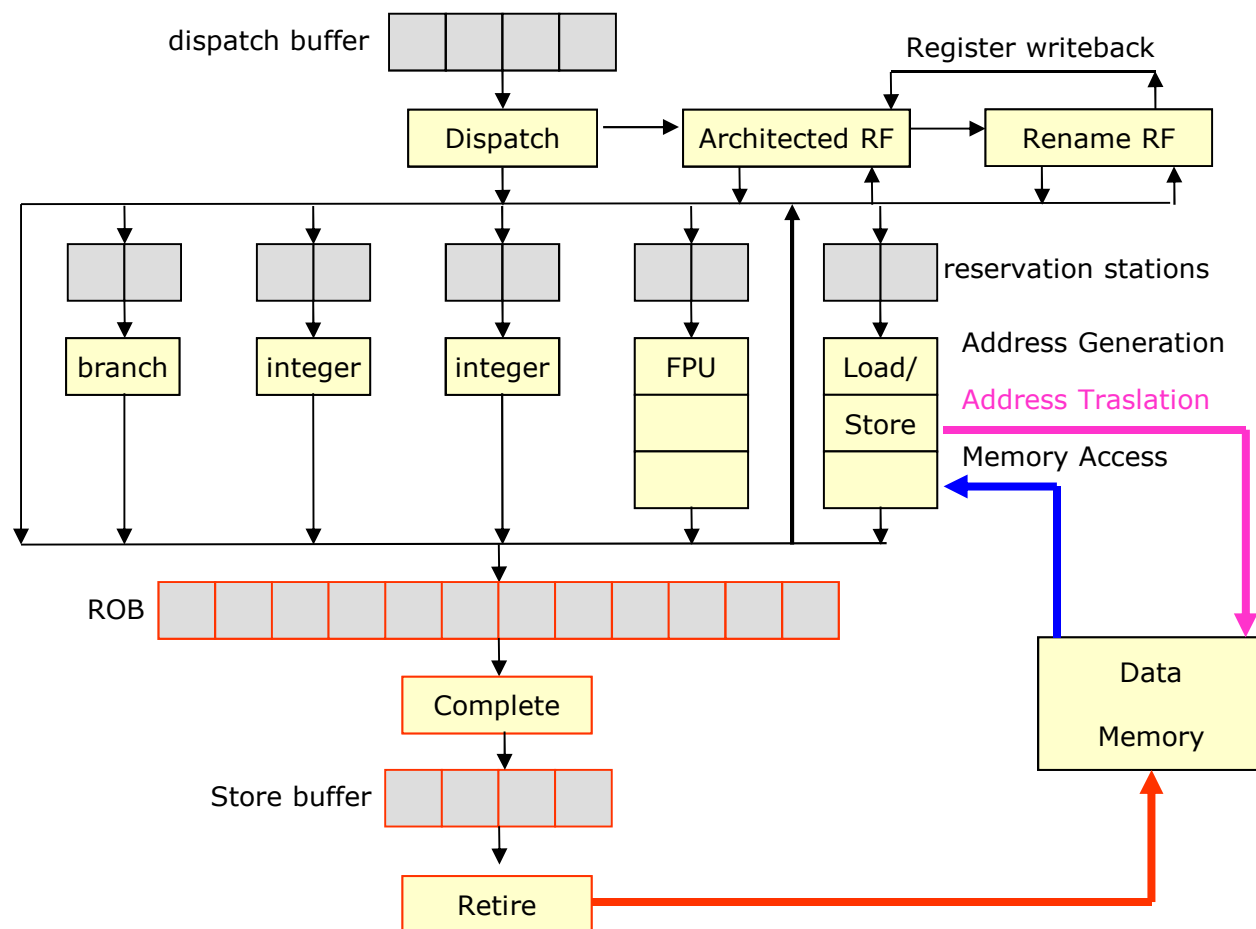
# Etapas del acceso a memoria

---

La ejecución de las instrucciones de acceso a memoria involucra tres etapas:

- **Generación de dirección:** La dirección completa de memoria no está en la instrucción. Usualmente hay que calcularla a partir de otros valores (inmediato + registro, etc)
- **Traslación de dirección:** Usualmente la dirección se expresa en un entorno de memoria virtual. La translación realiza un mapeo entre memoria virtual y memoria física. Usualmente se usa una TLB (translation Lookaside Buffer)
- **Acceso a memoria:** Acceso a una palabra de la memoria
  - **LOADs:** Se lee una palabra de memoria y se almacena en un registro
  - **STOREs:** Una palabra contenida en registro se escribe en la dirección calculada de memoria

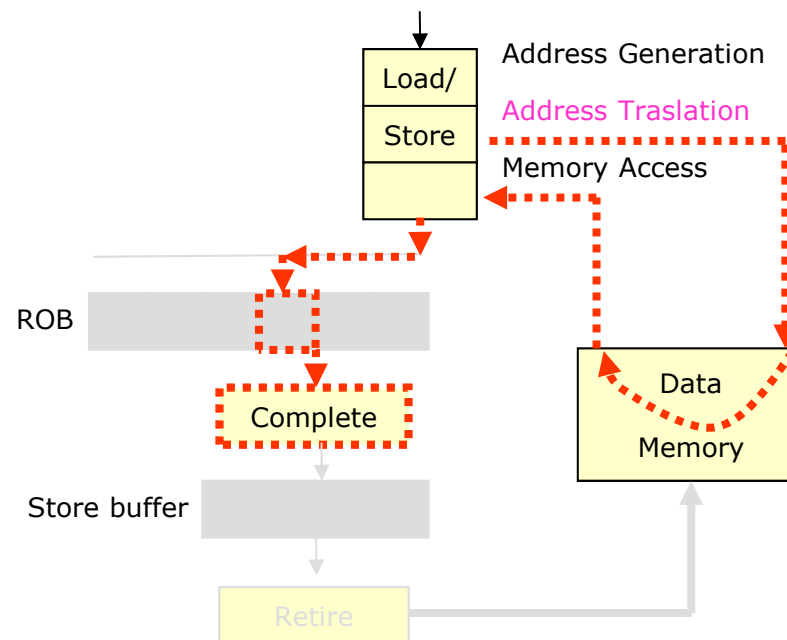
# Etapas del acceso a memoria



# Ciclos de acceso para lectura

- 1º ciclo: Address Generation
- 2º ciclo: Address traslation
- 3º ciclo: Memory access
  - Acceso a una cache de datos
  - Si el acceso es un *hit*, la lectura se realiza en un ciclo de reloj
  - Si el acceso es un *miss*, se produce un fallo de memoria y se frena el pipe de memoria

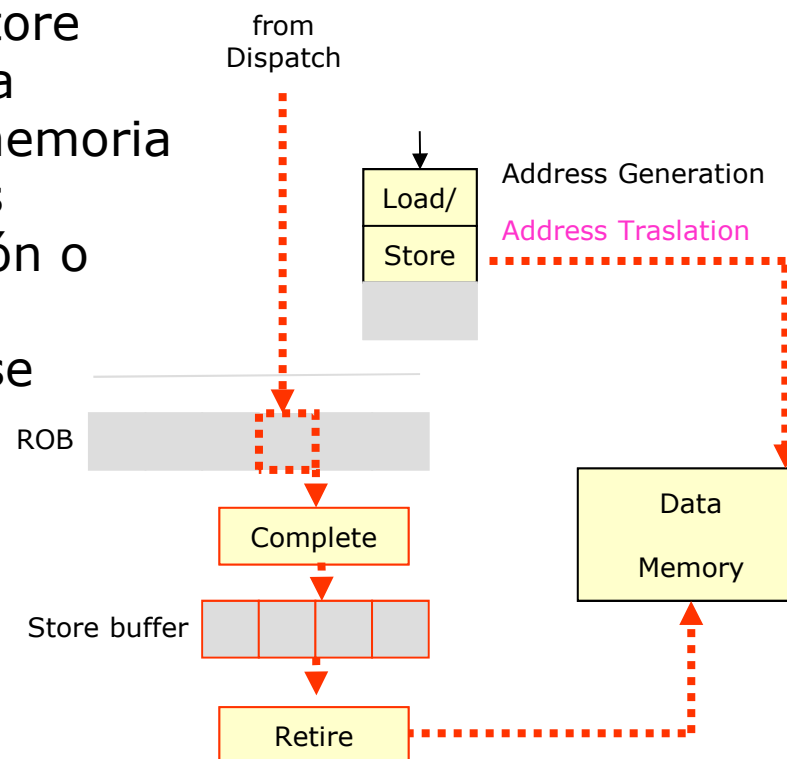
Una instrucción Load se considera terminada (finished) cuando finaliza el 3º ciclo, aún cuando no haya sido completada



# Ciclos de acceso para escritura

- 1º y 2º ciclos iguales al Load
- 3º ciclo:
  - El registro origen se almacena en el ROB junto con la instrucción Store
  - Cuando la instrucción se completa se realiza la escritura real en la memoria
  - Lo anterior previene de escrituras anticipadas si ocurre una excepción o un salto mal predicho
  - En caso de que no deba ejecutarse se borra del ROB sin efectos colaterales

Una instrucción Store se considera terminada (finished) cuando finaliza el **2º** ciclo, luego de que se realice una traslación exitosa



# Mantenimiento de la consistencia de memoria

---

Funcional

Lecturas fuera de orden no perjudican la consistencia de memoria

Escrituras fuera de orden producen dependencias WAW y WAR al acceder a las mismas posiciones de memoria

A fin de mantener la consistencia de memoria se debe secuenciar los Store. Al menos no necesariamente su ejecución pero si su finalización

La serialización de escrituras asegura:

- Recuperación segura de excepciones y otras rupturas de secuencia
- Consistencia en modelos multiprocesador

# Mantenimiento de la consistencia de memoria

Funcional

¿Cómo asegurar la serialización de los Store sin comprometer la extracción de paralelismo?

Buffer de escritura (Store Buffer):

- Almacena secuencialmente (orden de programa) los requerimientos de escritura a memoria
- La unidad de retiro lee operaciones de escritura a memoria y las lleva adelante cuando el bus de memoria está libre
- La prioridad en el uso del bus es siempre para las lecturas
- Una instrucción Store puede haber terminado su ejecución y puede estar completada, pero aún su escritura a memoria puede no haber sido realizada si está en el Store buffer
- Si ocurre una excepción los requerimientos de escritura en el Store buffer no son eliminados pues corresponden a Stores ya completos

# Otras técnicas avanzadas de manejo de memoria

---

Funcional

La ejecución fuera de orden de las instrucciones **Load** es la principal fuente de aumento de rendimiento

Las instrucciones **Load** son, frecuentemente, el inicio de las cadenas de dependencias

La ejecución prematura de las instrucciones **Load** permitirá el adelanto de ejecución de todas las demás instrucciones del programa


Meta principal:  
Adelantar la ejecución de las instrucciones **Load**  
sin violar dependencias de datos

# Otras técnicas avanzadas de manejo de memoria

Funcional

## Load Bypassing


Permite la ejecución de un **Load** antes que los **Store** que la preceden siempre y cuando no haya dependencias entre las direcciones de memoria del Load con las de los Store que adelanta



```
.....  
SW  r10, 100(r0)  
.....  
SW  r11, 200(r0)  
.....  
LW  r12, 300(r0)
```

## Load Forwarding

Si un **Load** tiene la misma dirección de memoria que un **Store** previo (dependencia RAW), el load puede cargarse con el valor que almacenará el store



```
.....  
SW  r10, 100(r0)  
.....  
SW  r11, 200(r0)  
.....  
LW  r12, 100(r0)
```



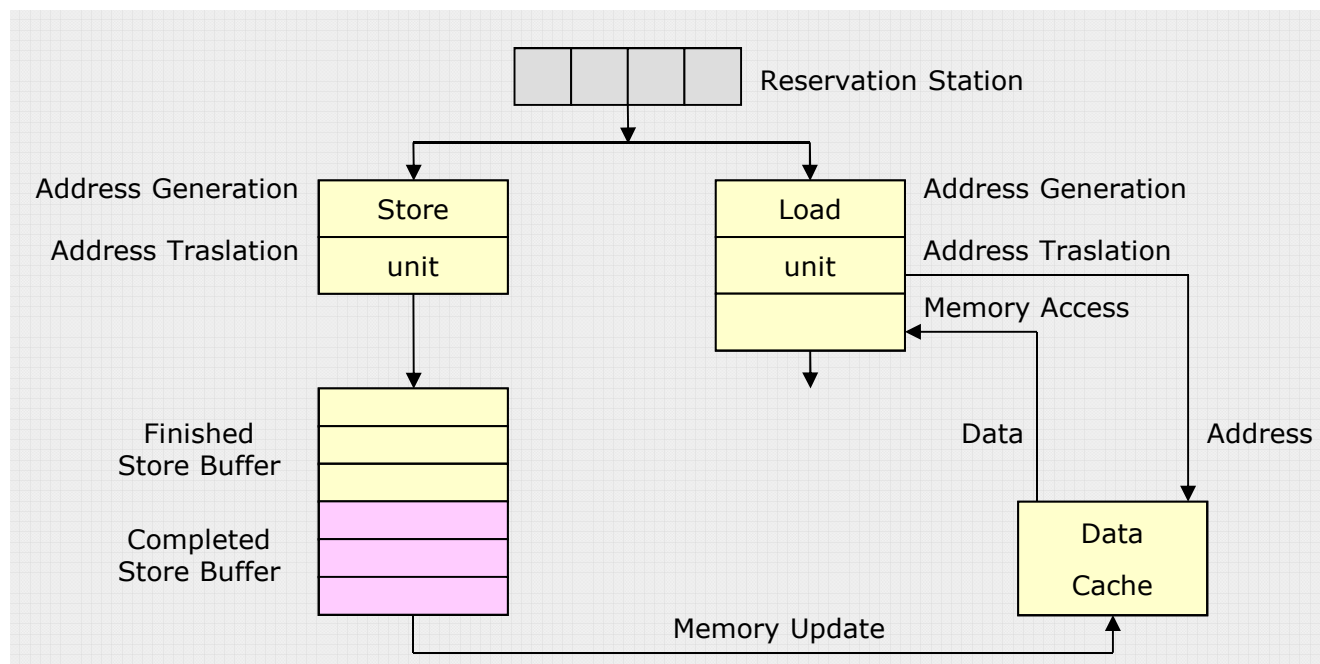
# Etapa Load/Store estándar

Store buffer de 2 partes:

- **Finished: Stores** que terminaron su ejecución pero que aún no están arquitecturalmente completas
- **Completed: Stores** completos arquitecturalmente esperando actualizar la memoria

Un **Store** en la parte *finished* del buffer puede ser especulativo y ser eliminado en caso de mala predicción

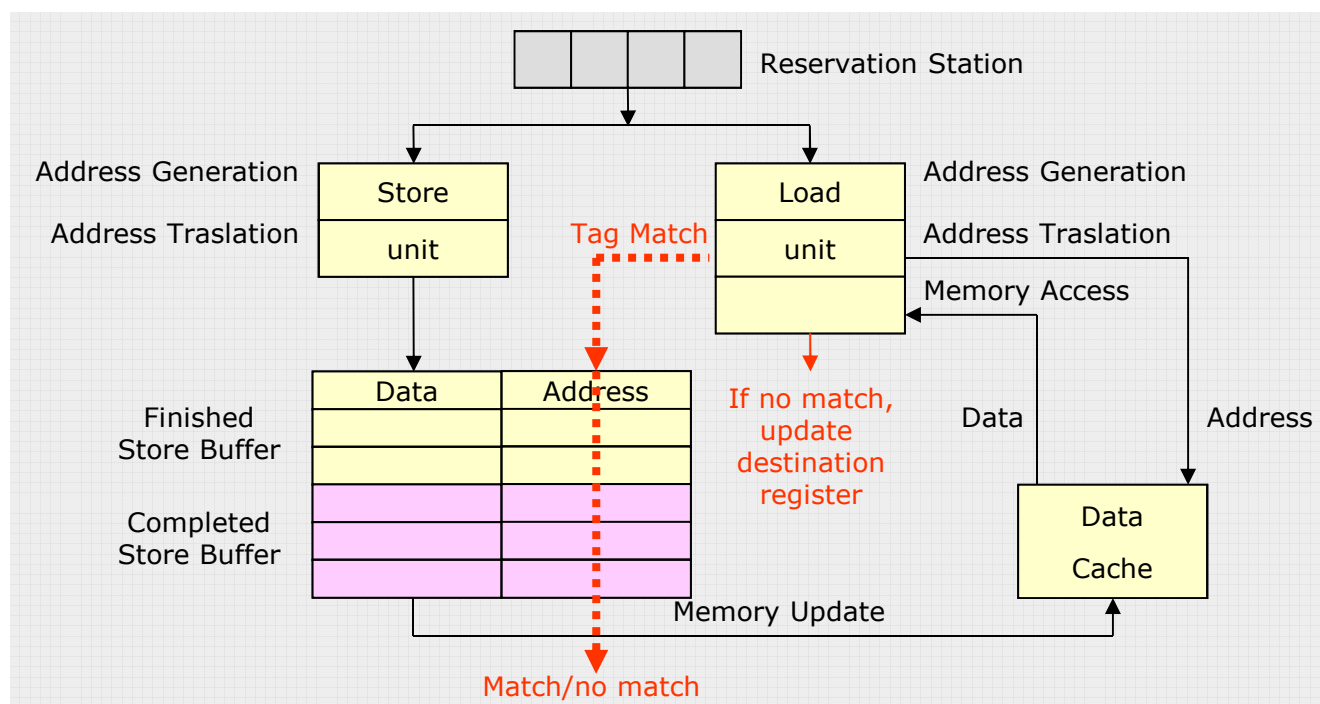
Un **Store** en la parte *Completed* del buffer no es eliminable y se completa siempre



# Load Bypassing

Se compara la dirección del **Load** con las direcciones del *Store buffer*.  
 Si no hay coincidencias, el **Load** no entra en conflicto con ningún **Store** y puede ser adelantado  
 Si hay conflicto, el **Load** es frenado y devuelto a la estación de reserva

**Desventaja:** La parte de dirección del Store Buffer debe implementarse como una memoria asociativa para una búsqueda rápida de coincidencias



# Load Forwarding

Se compara la dirección del **Load** con las direcciones del *Store buffer*.  
Si hay coincidencia con alguna dirección, se copia el dato del **Store** en el registro de renombre del **Load**

**Desventaja:** De haber varias coincidencias se debe usar la del **Store** mas reciente (el último que actualizará la posición de memoria que leerá el **Load**)

