❖Tema 2. ILP, PLANIFICACIÓN DINÁMICA, PREDICCIÓN Y ESPECULACIÓN

➢ 2.1 Paralelismo a Nivel de Instrucción (ILP)

El paralelismo a nivel de instrucciones consiste en conseguir ejecutar en un computador diferentes instrucciones al mismo tiempo que sean próximas en el código.

Una estrategia que se estudió en Estructura de Computadores consistía en la *segmentación* de la ruta interna del procesador.

Bloque básico: es una secuencia de código sin saltos. Un solo punto de entrada y salida.

Nuestro objetivo es explotar al máximo el paralelismo entre bloques básicos, ya sea internamente o externamente. Recordemos las dependencias que pueden existir entre instrucciones cercanas, que pueden ser de datos o de control (saltos).

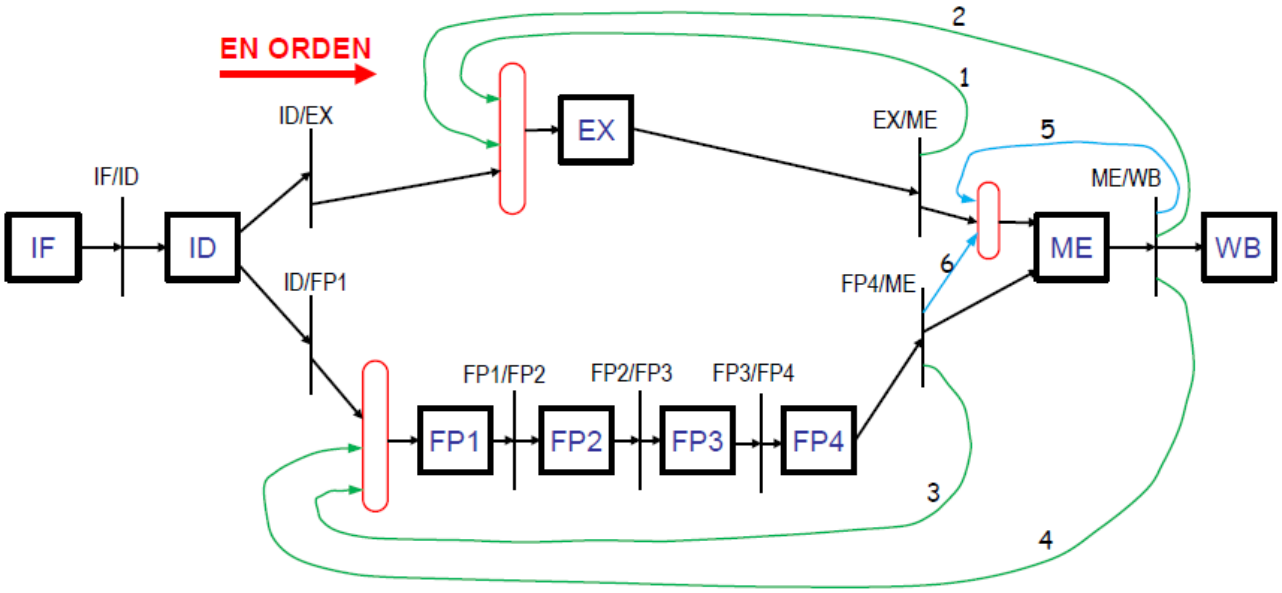
Dependencias entre instrucciones $\begin{cases} LDE \\ EDE \\ EDL \\ Control \end{cases}$

➢ 2.2 Técnicas Software para explotar ILP

Vamos a estudiar algunas técnicas básicas de mejora a partir de un procesador segmentado muy similar al estudiado en EC.

Tenemos un procesador segmentado con unidades funcionales variadas, posibilidad de paradas en ID y anticipación de operandos. Las instrucciones siempre se ejecutan en orden.

- Lanzamiento a ejecución en ORDEN. Detección de riesgos en ID. Paradas en ID.
- Saltos resueltos en ID.
- EXE con varias Unidades Funcionales. Finalización fuera de orden.
- Banco de Registros separado de INT y FLOAT.



Red de anticipación de operandos (by-pass)

1: Resultado de op int
2: Res. de op int o load INT
3: Res. de op FP
4: Resultado de op FP o load FP
5: Res. de op int, load INT → store int
Res. de op FP, load FP → store FP
6: Res. de op FP → store FP

Recordemos que en la ruta de datos segmentada existen un conjunto de registros intermedios que almacenan la información de las instrucciones entre etapas (representados en la figura por una línea negra vertical entre las etapas).

Técnicas de Mejora:

1. Reordenación de instrucciones.
2. Loop Unrolling.
3. Software "pipelining".

- **2.2.1 Reordenación de instrucciones**

Una de las posibilidades de mejora del rendimiento, que ya se empezó a estudiar en EC, consiste en reordenar el código en ensamblador para evitar lo máximo posible las dependencias de datos. También se pueden usar los saltos retardados para que las dependencias de control tengan un menor impacto.

Es importante recordar que la reordenación de instrucciones se hace en *tiempo de compilación*, antes de lanzar el código a ejecutar.

Código maquina MIPS

```
Loop: L.D      F0,0(R1)    ; F0← elemento array
      ADD.D    F4,F0,F2    ; sumar escalar en F2
      S.D      F4,0(R1)    ; almacenar resultado
      DADDIU   R1,R1,#-8   ; decrementar puntero
      BNE      R1,R2,Loop  ; repetir si R1 ≠ R2
```

```
Loop: L.D      F0,0(R1)
      DADDIU   R1,R1,#-8
      ADD.D    F4,F0,F2
      Espera
      BNE      R1,R2,Loop
      S.D      F4,8(R1)
```

- **2.2.2 Loop Unrolling**

Tenemos el bucle del apartado anterior que se tiene que ejecutar 1000 veces.

Código maquina MIPS

```
Loop: L.D     F0,0(R1)    ; F0← elemento array
      ADD.D   F4,F0,F2    ; sumar escalar en F2
      S.D     F4,0(R1)    ; almacenar resultado
      DADDIU  R1,R1,#-8   ; decrementar puntero
      BNE     R1,R2,Loop  ; repetir si R1 ≠ R2
```

```
Loop: L.D     F0,0(R1)
      ADD.D   F4,F0,F2
      S.D     F4,0(R1)
      L.D     F6,-8(R1)
      ADD.D   F8,F6,F2
      S.D     F8,-8(R1)
      L.D     F10,-16(R1)
      ADD.D   F12,F10,F2
      S.D     F12,-16(R1)
      L.D     F14,-24(R1)
      ADD.D   F16,F14,F2
      S.D     F16,-24(R1)
      DADDIU  R1,R1,#-32
      BNE     R1,R2,Loop
```

Podemos desenrollar un cierto número de vueltas para aumentar el rendimiento. Pongamos que en vez de hacer 1000 saltos hacemos sólo 250, desenrollamos 4 vueltas.

Sería recomendable, además de desenrollarlo, que pudiésemos reordenar las instrucciones porque el bucle desenrollado tiene todavía muchas dependencias de datos.

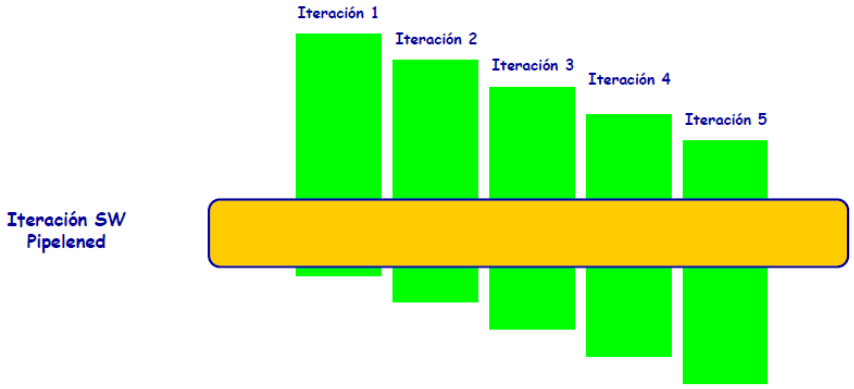
❑ Desenrollado + Planificación

```
Loop:   L.D      F0,0(R1)
        L.D      F6,-8(R1)
        L.D      F10,-16(R1)
        L.D      F14,-24(R1)
        ADD.D    F4,F0,F2
        ADD.D    F8,F6,F2
        ADD.D    F12,F10,F2
        ADD.D    F16,F14,F2
        S.D      F4,0(R1)
        S.D      F8,-8(R1)
        DADDIU   R1,R1,#-32
        S.D      F12,16(R1)      ; 16-32= -16
        BNE      R1,R2,Loop
        S.D      F16,8(R1)       ;  8-32 = -24
```

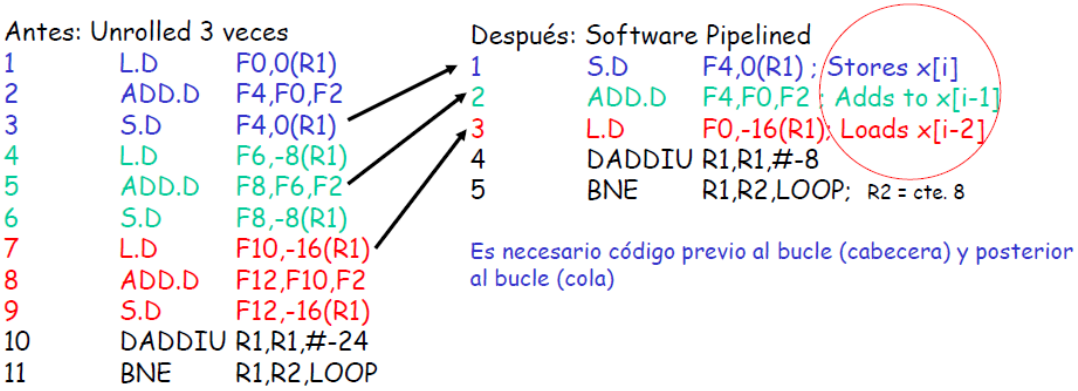
- ✓ Mover S.D después de DADDIU: ojo al valor de R1
- ✓ 14 instrucciones lazadas en 14 ciclos: 3.5 ciclos por iteración
- ✓ CPI = 1
- ✓ Más registros (Imprescindible !!)

- **2.2.3 Software "pipelining".**

¿Y si las diferentes iteraciones de un bucle no tuviesen nada que ver una con la otra?



Podríamos mezclar operaciones de vueltas diferentes.



Conclusiones de las técnicas SW para ILP

- Loop Unrolling
  - Bloque grande para planificar
  - Reduce el numero de saltos
  - Incrementa el tamaño del código
  - Tiene que incluir iteraciones extra
  - Presión sobre el uso de registros
- Software Pipelining
  - No hay dependencias en el cuerpo del bucle
  - No reduce el numero de saltos
  - Necesita inicio y finalización especial

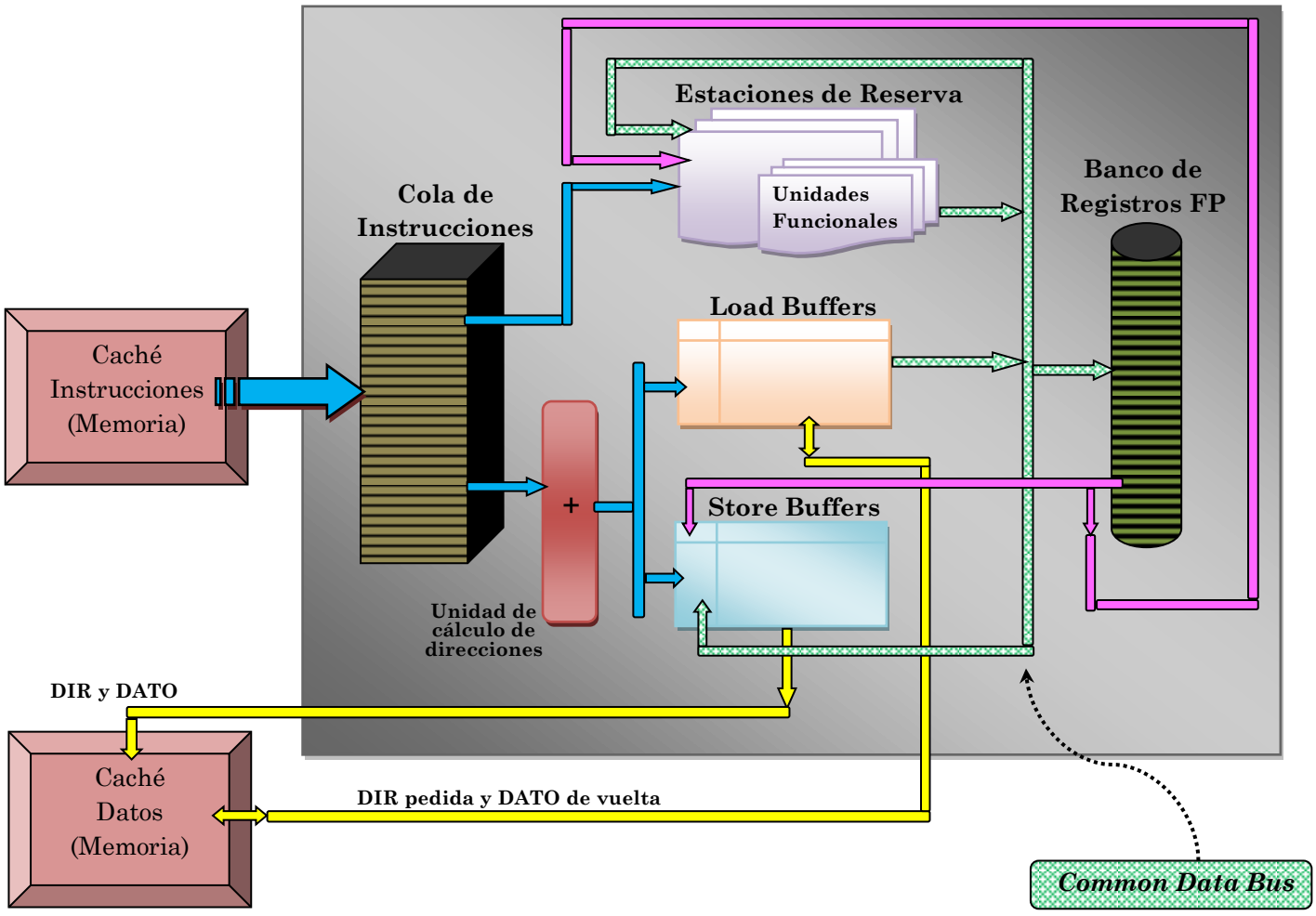
➢ 2.3 Planificación dinámica. Algoritmo de Tomasulo.

¿ Y si el código se lanza a ejecución sin ser reordenado ni planificado, pero la propia ruta de datos decide en qué orden se ejecutan las instrucciones en tiempo de ejecución (no en compilación) para minimizar los riesgos de datos y control? Eso es la planificación dinámica.

El algoritmo de Tomasulo **sin especulación** presenta las siguientes características:

I. Lanzamiento de instrucciones en orden.
II. Ejecución fuera de orden.
III. Finalización fuera de orden.
IV. No existen riesgos EDE ni EDL por renombrado de registros.
V. Los riesgos LDE se atienden con paradas en la ejecución.
VI. Para atender los riesgos de control se necesita ESPECULACIÓN. (de momento no tendremos saltos)

Tomasulo SIN Especulación



(*) El CDB solo puede tener volcado un dato en cada ciclo, por lo tanto se debe arbitrar su uso por parte de todos los componentes de la ruta.

Consideraciones importantes en el estudio de Tomasulo para este documento

1. Se omite la fase de Fetch donde las instrucciones viajan desde memoria a la cola de instrucciones. Empezaremos la ejecución de las instrucciones en la fase de Issue.
2. No se ven reflejadas las operaciones enteras ni sus unidades funcionales. El uso de los registros de enteros no se muestra en la ruta (por ejemplo en el cálculo realizado por la unidad de obtención de direcciones). Sólo lanzaremos operaciones aritmético-lógicas de Punto Flotante.

- **2.3.1 Etapas de una instrucción.**

  - **ISSUE:** se toma la instrucción primera de la cola de instrucciones y se envía a las estaciones de reserva (ER) o a los buffers (dependiendo del tipo). Las operaciones A-L se envían a las ER, los LOAD a un buffer de Load y los STORE a un buffer de Store. El número de ER y de buffers es limitado, por lo tanto si no es posible disponer de hueco, la realización del Issue se detiene y la cola se para hasta que la instrucción pueda disponer de un hueco que le corresponda por su tipo.

    Las instrucciones de LOAD & STORE pueden requerir de un cálculo de dirección basado en un inmediato y un registro de enteros. Si se necesita este cálculo se realiza en la etapa de Issue por medio de la unidad de cálculo de direcciones. Cuando la instrucción es enviada al buffer correspondiente la dirección efectiva ya está calculada.

  - **EJECUCIÓN:** tanto los buffers como las ER actúan de forma independiente resolviendo sus tareas propias.

    Las ER envían la información de la operación y los operandos (si es que éstos están disponibles) a las unidades funcionales. Mientras la operación se realiza, la ER queda ocupada por la instrucción. Una vez termina la ejecución se pasa a write.

    Los buffers de LOAD ejecutan la búsqueda del dato en memoria, tardan un cierto número de ciclos y después pasan a write. Sólo hay un puerto de memoria, debe estar libre.

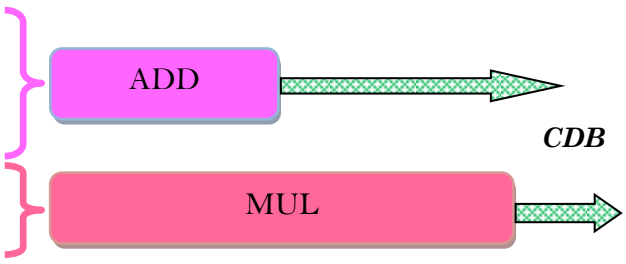
    Los buffers de STORE esperan a que su información esté actualizada. En la propia fase de ejecución copia el dato sobre la memoria y después no realizará fase de write. El buffer es liberado en el último ciclo de la ejecución. Se considera terminada la instrucción.

  - **WRITE:** los STORE no tienen fase de write, no hacen absolutamente nada así que no ocupan el CDB. Sólo puede haber una instrucción que realice su fase de write al mismo tiempo.

    Tanto los LOAD como las A-L escriben sus resultados en el CDB y liberan los recursos que estuviesen usando (buffers o ER). El CDB debe ser gestionado puesto que hay muchos componentes que pueden querer volcar su información a la vez.

- **2.3.2 Estaciones de Reserva y Unidades Funcionales.**

UF	Busy	Op	V_j	V_k	Q_j	Q_k
Add1						
Add2						
Add3						
Mul1						
Mul2						



El procesador dispone de una serie de Unidades Funcionales para operaciones en Punto Flotante. Normalmente suele tener un par, las unidades funcionales pueden hacer normalmente varias operaciones, se debe especificar en los enunciados. En general:

ADD (suma y resta) MUL (multiplicación y división)

Si una unidad funcional tiene 3 ER asociadas quiere decir que puede almacenar 3 Instrucciones del tipo a la vez. Por ejemplo podría tener 2 sumas y una resta, pero esto no quiere decir que las esté realizando a la vez. Cuando una instrucción A-L realiza su ISSUE toma los valores del banco de registros que pueda con valor actual o adquiere su Q_i (ver banco de registros de FP).

Si la UF está segmentada puede ir realizando los diferentes ciclos de latencia de sus instrucciones de forma secuencial (tiene su pipeline interno). Si no está segmentada, se debe esperar a que la instrucción en ejecución termine para lanzar la siguiente.

Todas las ER tienen una serie de etiquetas que sirven para gestionar correctamente las operaciones:

UF → Es el nombre de la ER, con tipo de UF y número, se identifica así para el CDB de forma unívoca.

Busy → Se indica si la ER está conteniendo una instrucción o no.

OP → Indica cuál de las operaciones tiene que realizar la UF (suele haber un par de opciones).

V_j → Dato válido de primer operando, se escribe si está disponible. Si $Q_j = 0$ es válido.

V_k → Dato válido de segundo operando, se escribe si está disponible. Si $Q_k = 0$ es válido.

Q_j → Indica de dónde se debe coger un dato aún no disponible para ponerlo en V_j lo antes posible.

Q_k → Indica de dónde se debe coger un dato aún no disponible para ponerlo en V_k lo antes posible.

Una operación no puede dar comienzo hasta que sus dos datos V_j y V_k estén disponibles. Comenzará la ejecución **el ciclo después** de tener todos los datos. Se puede observar cómo no hay un etiquetado acerca de dónde va a parar el resultado, la razón es porque el CDB recibe el resultado de, por ejemplo *Add2*, y todos los componentes que estuviesen esperando ese dato en sus correspondientes Q_x lo pueden obtener a la vez.

- **2.3.3 Buffers de Load y de Store.**

	Busy	DIR	Valor
Load1			
Load2			
Load3			
Load4			
Load5			

Funcionan muy parecido a las estaciones de reserva. Cuando reciben una instrucción se ponen en Busy y se pasan ejecución.

Los *LoadX* no necesitan esperar a ningún dato, simplemente vale con que la memoria esté libre. Copia en *Valor* lo leído de memoria al acabar su EXE.

Los *StoreX* sí necesitan que el dato V_i esté disponible para comenzar su ejecución. La etiqueta Q_i funciona igual que en las ER.

	Busy	DIR	V_i	Q_i
Store1				
Store2				
Store3				

En los enunciados nos indicarán que los buffers tienen un tiempo de latencia, esto es el tiempo que tardan en acceder a memoria para realizar su operación (siempre que esté todo listo).

Un procesador solo tiene una unidad de LOAD y una unidad de STORE, esto quiere decir que solo se puede gestionar una petición a la vez. Sin embargo esta unidad puede estar **segmentada** lo que nos permite lanzar a ejecución una detrás de otra sin esperar toda la latencia.

- **2.3.4 Banco de Registros de Punto Flotante.**

	F0	F2	F4	F6	F8	F10	F12	F14	F16	F18	F20
Q_i											
V_i											

Tiene un funcionamiento sencillo, el valor real del registro es V_i , pero puede estar obsoleto.

Si la etiqueta $Q_i = 0$ entonces el valor es correcto.
Si la etiqueta $Q_i \neq 0$ entonces se debe actualizar con el resultado de la ER o LOAD que se indique.

➢ 2.4 Predicción de Saltos

Cuando se detecta una instrucción de salto condicional →

- Se predice el camino del salto (TOMADO o NO TOMADO)
- Si la predicción es de tomado, se debe calcular la dirección lo antes posible.
- La ejecución continúa de forma **especulativa** hasta que se resuelva la condición.

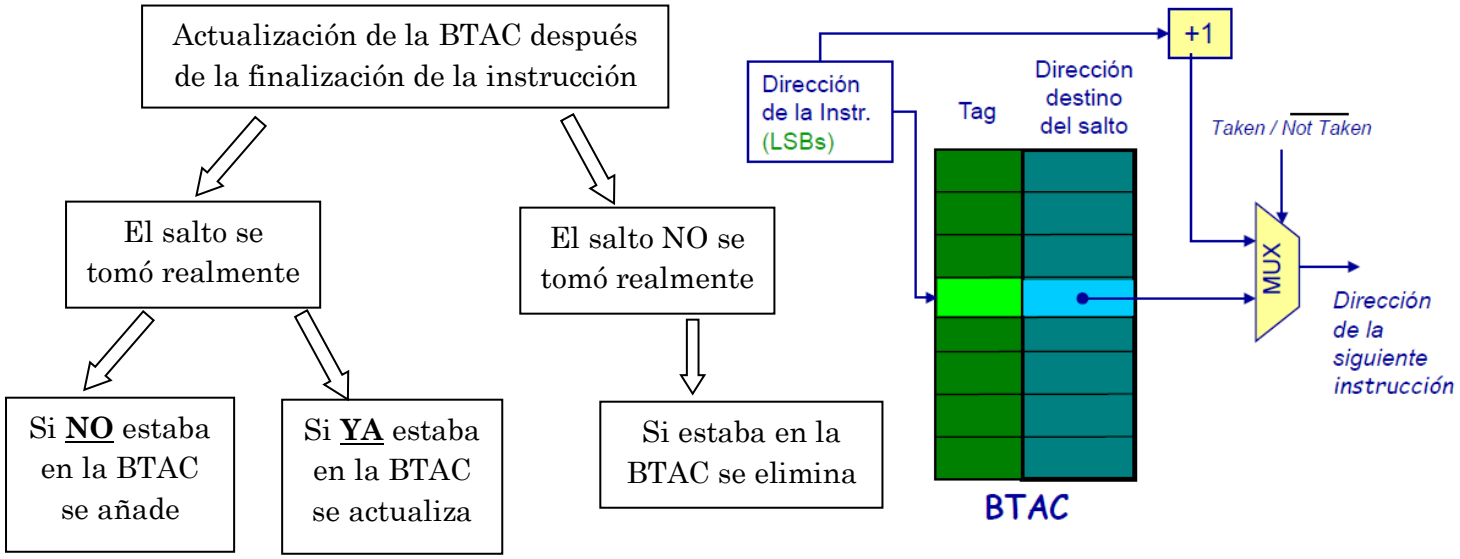
Cuando se resuelve la condición →

- Si la predicción fue correcta, se continúa.
- Si la predicción fue errónea se eliminan las instrucciones especulativas y se toma el camino correcto.

■ Acciones importantes que debe tomar el Hardware de Predicción
- Calcular las direcciones de salto lo antes posible.
- Resolver la condición de salto lo antes posible.
- Ejecutar una predicción sobre la instrucción de bifurcación. (*Algoritmos de predicción*).
- Eliminar instrucciones añadidas a la ruta de forma especulativa en caso de fallo de predicción.

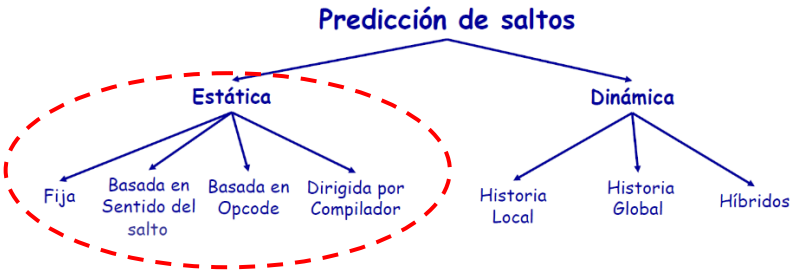
• 2.4.1 Branch Target Addres Cache (BTAC)

Éste es un dispositivo que nos aporta rapidez a la hora del cálculo de las direcciones de salto (independientemente del algoritmo de predicción). Es una Caché de direcciones de los últimos saltos tomados. Se accede a ella mediante la dirección de la instrucción de salto.



La BTAC es una caché, por lo tanto está sujeta a las políticas de emplazamiento y reemplazamiento vistas en Estructura de Computadores. Directa, asociativa o asociativa por conjuntos.

- **2.4.2 Predicción Estática**



Las predicciones estáticas no son muy sofisticadas. Tienen unas reglas fijas que se aplican al programa siempre.

Son fáciles de implementar, pero su tasa de acierto no es muy elevada.

- Fija: siempre se toma la misma decisión al encontrarse un salto.
  Always Taken
  Always Not Taken

- Basada en sentido del salto: si el salto es hacia atrás se toma, si es hacia delante no se toma.

- Basada en OPCode: según el código de la instrucción decidimos si salta o no.

- Basada en el Compilador: al realizarse la compilación se añade a las instrucciones un bit de predicción estático según algún criterio (especificado por el programador, o por la historia pasada de las ejecuciones o por el tipo de estructura que contiene el salto).

- **2.4.3 Predicción Dinámica**

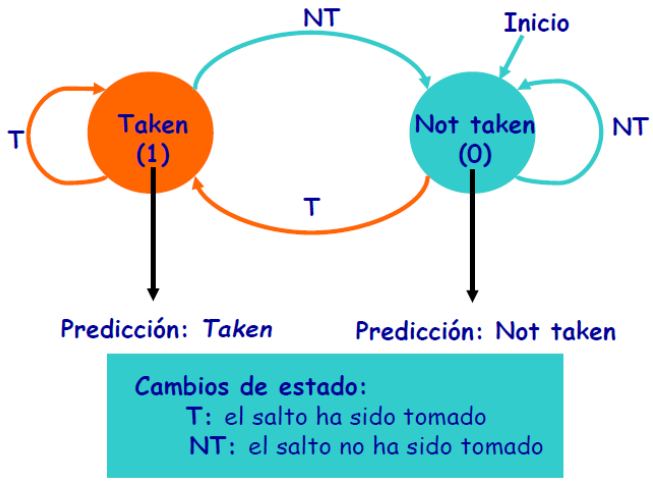
Esta sección es mucho más interesante que la anterior. La predicción se realiza en tiempo de ejecución observando el comportamiento de los saltos que preceden al salto que está siendo evaluado (Historia).

Vamos a estudiar predictores dinámicos en función de su estructura:

- Predictor de 1 bit
- Predictor de 2 bits (bimodal)
- Predictor de dos niveles.
- Predictores híbridos
  ↳ *Tournament Predictor*

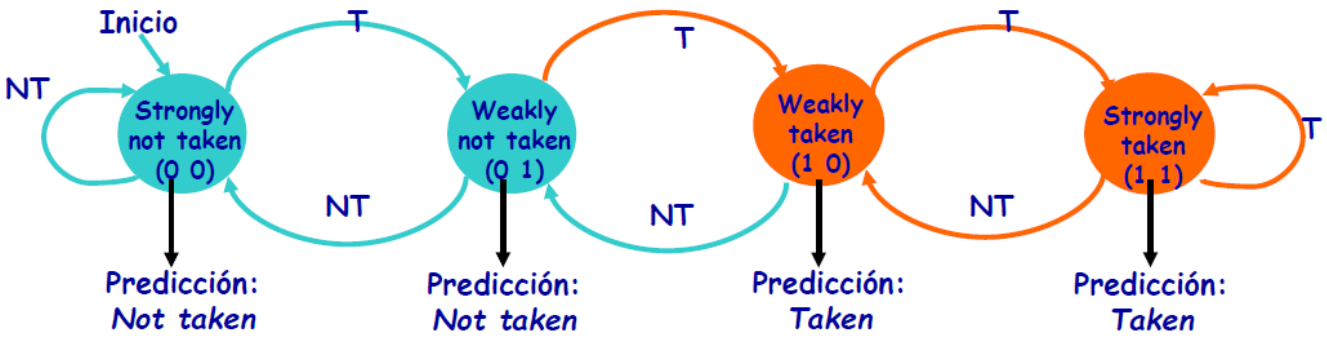
➢ 2.5 Predictor de 1 y 2 bits

Esencialmente son la misma idea, solo que el predictor de 2 bits es un poco más complejo ya que almacena el doble de estados.



El predictor al recibir una instrucción condicional toma la decisión de saltar en función del estado actual en el que se encuentra.

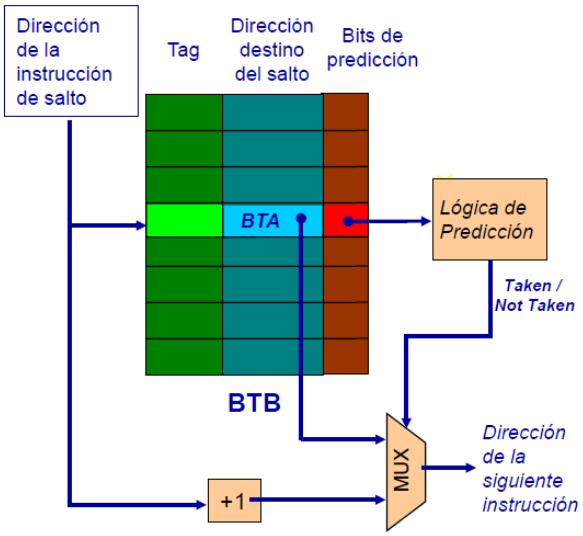
Después de la resolución de la condición transita dependiendo de si se tomó o no de verdad.



Estos predictores dinámicos que usan la historia local de los saltos del código se implementan de 2 formas posibles:

BTB	Se construye una BTAC ampliada a la que se le añaden los bits de predicción.
BTAC+BHT	Se dispone de dos tablas por separado, la BTAC y la BHT que contiene la historia.

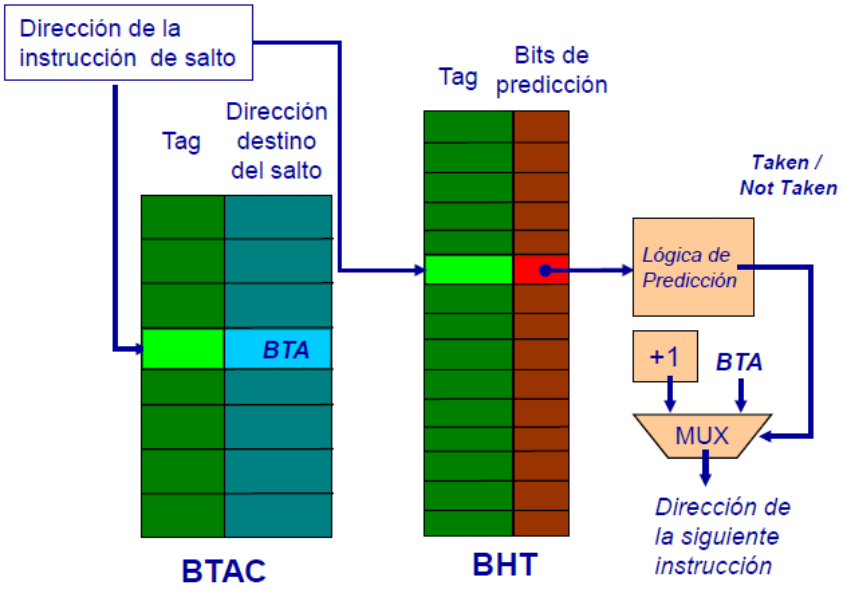
- **2.5.1 Implementación BTB (Branch Target Buffer)**



Es una mejora de BTAC. Incorpora a la dirección del salto el/los bits de predicción que serán gestionados por la lógica de predicción para decidir el sentido.

Al resolver la condición de salto se actualizan todos los campos.

- **2.5.2 Implementación BTAC + BHT (Branch History Table)**



La BHT es una tabla con más entradas ya que contiene muy poca información, solo los bits de predicción.

Se pueden usar los bits más bajos de la dirección de la instrucción para indexar la BHT.

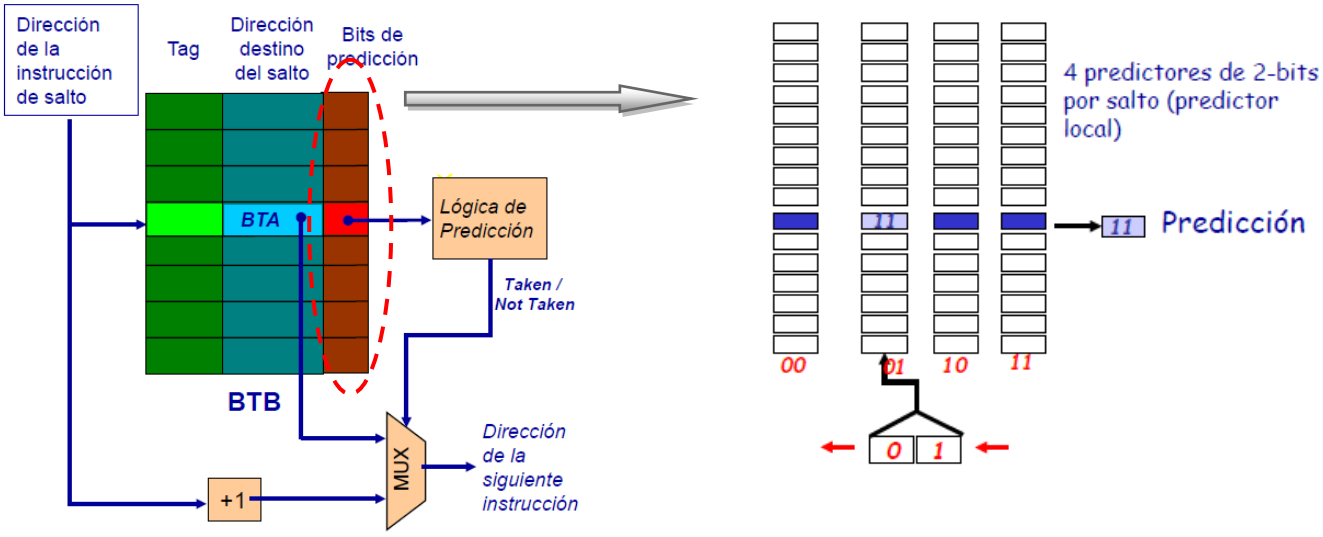
Tanto la BTB como esta configuración de BTAC + BHT son predictores de historia local (tienen bits de predicción propios de cada salto).

- **2.5.3 Predictor de dos niveles**

El predictor de dos niveles es un sistema que combina el concepto de predictor básico de 1 ó 2 bits local, con una tabla de predicción global.

Para nombrar un predictor de dos niveles se especifican con una dupla de números (m, n). El predictor (m, n) dispone de 2^m predictores de n bits para cada salto.

Por ejemplo, pensemos en una BTB y convirtamos la columna final en una tabla. Predictor $(2, 2)$

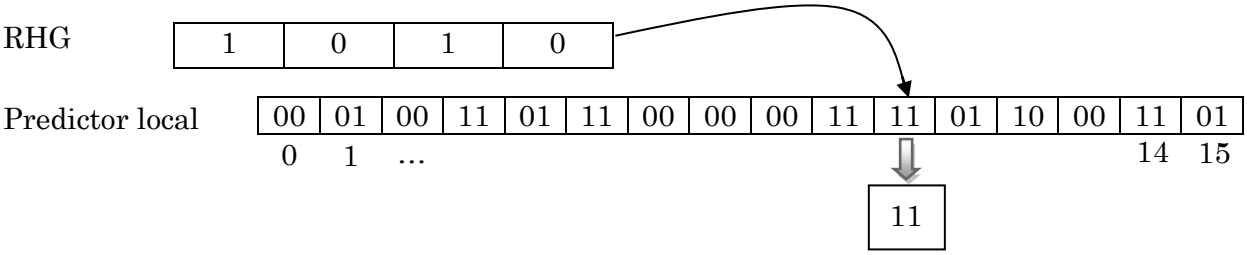


Globalmente estaba activado el predictor de la columna (0 1), estos dos números contienen un historial de los 2 últimos saltos globalmente, funciona como un registro de desplazamiento a izquierdas. Con esto se comprueba el predictor local de ese salto que indica (1 1) (Strongly Taken) y esa es la predicción que se realiza.

Una vez se obtiene el resultado de la condición de salto (por ejemplo TAKEN) se actualiza el predictor local según la lógica de dos bits (en este caso se mantendría en Strongly Taken) y después el global cambia echando al bit más a la izquierda (0 1) ← 1 ∴∴ 0 ← (1 1).

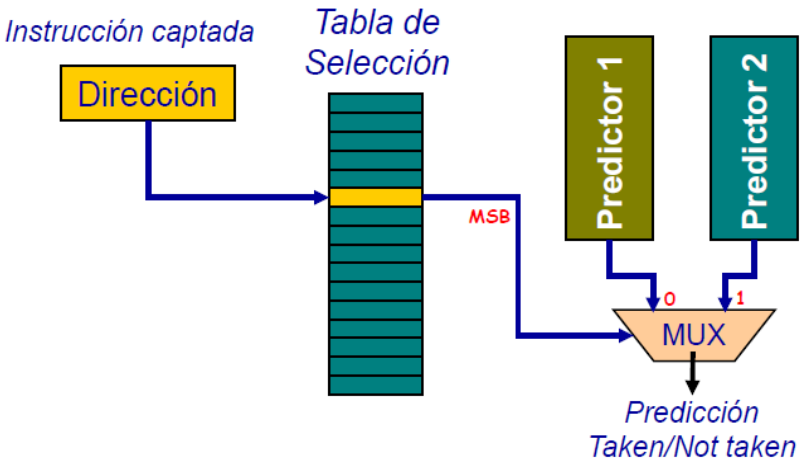
Ahora se consultaría el predictor local de la columna (1 1) con la dirección del próximo salto.

Un predictor que se pregunta con cierta frecuencia es el predictor $(4, 2)$. Este predictor tiene 16 predictores locales de 2 bits para cada salto y un mini registro de 4 bits con la historia global que selecciona el predictor (que es un registro de desplazamiento).



➢ 2.6 Predictores Híbridos

La idea consiste en aprovechar las bondades de varios predictores seleccionando al que lleve mejor tasa de aciertos en lo que va de ejecución de programa para cada salto concreto.

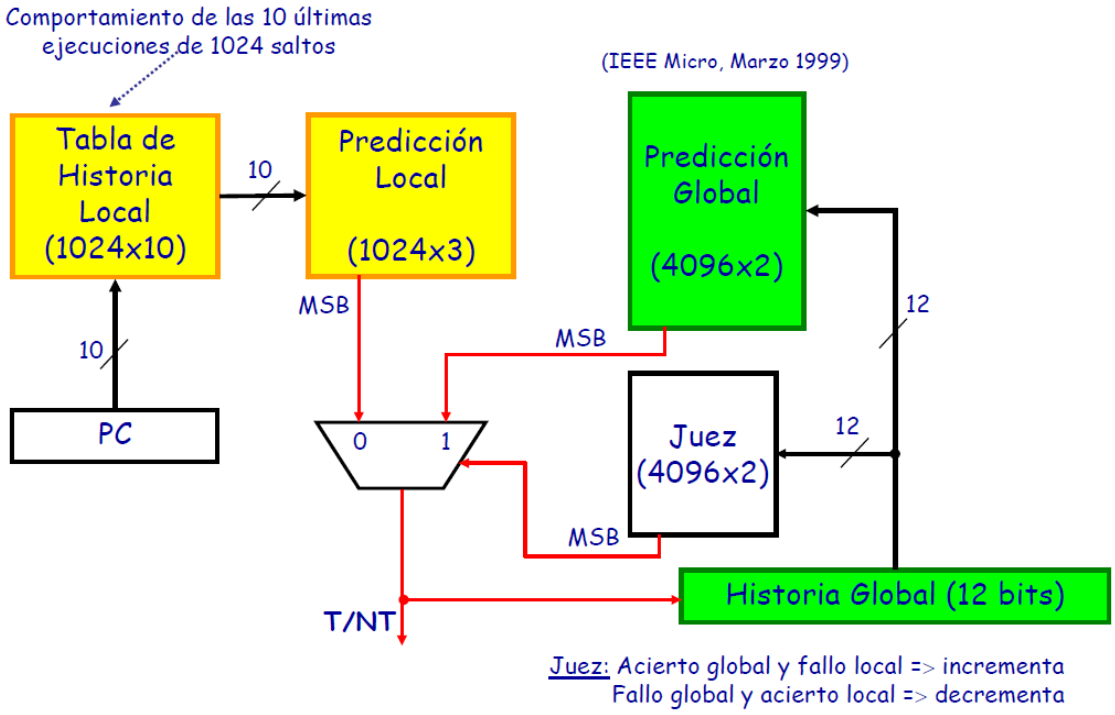


P1	P2	Actualiz. del contador
Fallo	Fallo	Cont no varía
Fallo	Acierto	Cont = Cont +1
Acierto	Fallo	Cont = Cont -1
Acierto	Acierto	Cont no varía

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

En este caso general la tabla de selección se suele implementar con contadores saturados. Se usa la predicción mirando el bit más significativo de este contador.

- **2.6.1 Tournament Predictor**



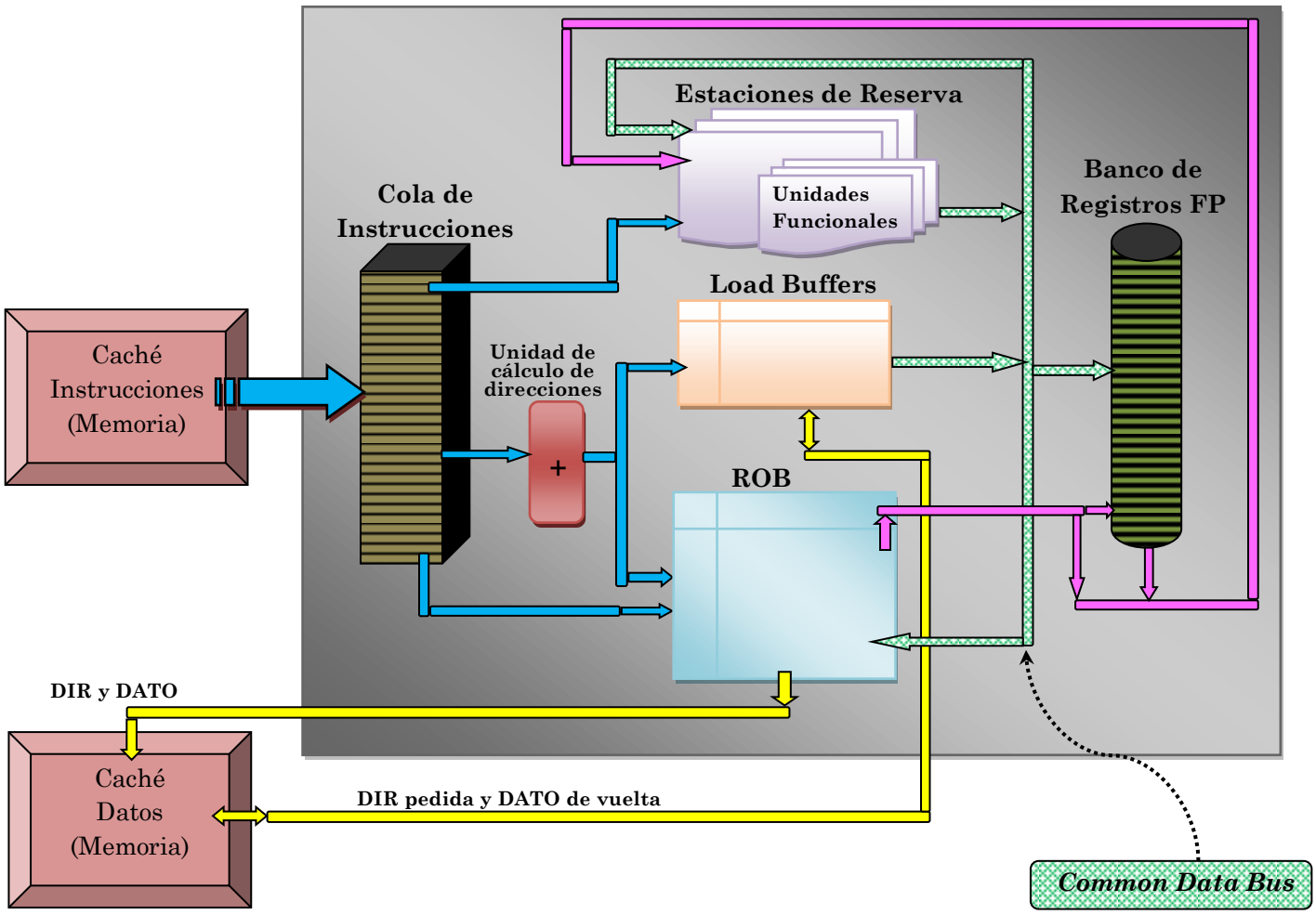
Es un caso particular de predictor híbrido, compiten la predicción local y la global. Veremos su funcionamiento con la resolución de los ejercicios de teoría.

➢ 2.7 Algoritmo de Tomasulo CON Especulación

Añadimos al algoritmo de Tomasulo la ESPECULACIÓN, esto significa la aparición de saltos condicionales y por tanto un potencial riesgo de ejecutar instrucciones incorrectas. Riesgo de control.

- En la etapa de ISSUE las instrucciones se inscriben en el ROB (ReOrder Buffer) para finalizar su ejecución en orden.
- En Write las instrucciones no han terminado, escriben su resultado en un buffer auxiliar. El ROB. Después va el Commit, una etapa de finalización.
- No existe buffer de Store, los Store esperan en el ROB al resultado y de ahí se escriben en memoria.
- Las instrucciones realizan sus escrituras en memoria o en registro en la fase de Commit. Una instrucción que depende de un salto sin resolver no puede hacer el Commit.

Tomasulo CON Especulación



(*) El CDB solo puede tener volcado un dato en cada ciclo, por lo tanto se debe arbitrar su uso por parte de todos los componentes de la ruta.

- **2.7.1 Estructura del ROB**

	Destino	Valor	Tipo de Instrucción	Listo	#ROB
					5
					4
ISSUE →					3
	F10		ADDD F10, F4, F0	N	2
COMMIT →	F0		LD F0, 10(R2)	N	1

El ROB es una cola FIFO de instrucciones. Entran por el puntero ISSUE y salen por el puntero COMMIT. Los punteros van siempre en orden creciente, cuando el puntero llega al tope vuelve abajo y así se van ciclando las posiciones del ROB.

Se le llama CABECERA del ROB al puntero de Commit. Cuando la instrucción que está en la cabecera tiene la etiqueta de Listo activa, se efectúa su fase de Commit y se sube el puntero.

Si una instrucción de salto llega a la cabecera y está bien predicha se elimina sin más en el Commit porque las instrucciones cargadas posteriormente a ella son todas correctas.

Si una instrucción de salto **MAL predicha** llega a la cabecera se borra todo el ROB, las Estaciones de Reserva y los Buffer de Load. Se borran todos los etiquetados del banco de registros relativos a entradas de ROB. Vamos que la ruta se inicializa por completo.

-) El destino de los operandos con Especulación. Uso del CDB.

Ahora las estaciones de reserva, los Load Buffers y el ROB tienen un campo de destino. Antes la gestión del CDB consistía en que un hardware que produce un resultado ponía el (resultado + su nombre) y lo recogía cualquiera que lo quisiese. Ahora también tiene una indicación de a qué #ROB debe ir.