Organización del procesador

Procesadores comerciales

Héctor Lacueva Sacristán

Índice

Riesgos

Estructurales

Causa

No disponibilidad de un módulo cuando se necesita.

Solución

En general se solucionan duplicando módulos y segmentando ALUs.

De datos

Causa

No disponibilidad de un dato cuando se necesita.

Solución

Ver algunas soluciones en *Dependencias*.

De control

Causa

Desconocimiento de cuál es la próxima instrucción que debe ejecutarse, culpa de los **saltos**.

Solución

Solución 1: Detener pipeline. Detener hasta que se conozca si el salto se produce. Demasiado lenta,

Solución 2: Asumir salto no tomado (NT). Si fallo de predicción, desechar todas las instrucciones en ejecución equivocadamente. Funciona bien si hay muchos saltos NT.

Solución 3: Predicción dinámica de saltos. Basada en eventos pasados. Predecimos que ocurrirá lo mismo que la vez anterior.

El predictor almacena: - La parte menos significativa de la dirección de la instrucción de salto. - Un bit indicando si T o NT la última vez.

Funciona bien en bucles.

Solución 4: Saltos retardados La siguiente instrucción a la de salto siempre se ejecuta. El compilador selecciona la instrucción adecuada.

Dependencias vs. Riesgos

Un riesgo es una limitación hardware.

Una dependencia es una propiedad del código.

```
subcc **r1**, r2, r3
addcc r5, **r1**, r4
```

Si hay dependencia puede existir o no riesgo.

Dependencias

Dependencia verdadera (True dependency)

Descripción

También conocida como dependencia **productor-consumidor**.

```
sub **$2**, $1, $3 ; productor
add $4, **$2**, $5 ; consumidor a distancia 1
or  $6, $7, **$2** ; consumidor a distancia 2
```

Causan riesgo de **lectura después de escritura** (LDE o RAW).

Soluciones

Solución 1: NOPs El **compilador** se encarga de garantizar que no se produzcan riesgos, para ello se **insertan** instrucciones **NOP** entre aquellas instrucciones que tengan dependencias de datos.

Desventaja: la ejecución se hace más lenta.

Solución 2: Anticipación de operandos Anticipar operandos mediante cortos. No siempre soluciona los riesgos, por ejemplo la instrucción **LOAD**. En ese caso **habría que detener el procesador**. De esto se encarga la **Unidad de detención del pipeline.** (introducir una NOP).

También se conoce como fordwarding.

Solución 3: Reordenamiento del código **Reordenamiento del código** para minimizar el número de detenciones. Lo hace el compilador. Algunas de las estrategias más comunes son el **loop unrolling** y el software pipelining.

Dependencia de salida (Output dependency)

Aparece cuando dos instrucciones escriben en el mismo registro.

```
sub **$2**, $1, $3
add **$2**, $4, $5
```

Causan riesgo de **escritura después de escritura** (WAW).

Antidependencia (Antidependency)

Aparece cuando una instrucción lee un registro y después otra lo modifica.

```
sub $2, **$1**, $3
add **$1**, $4, $5
```

Causan riesgo de **escritura después de lectura** (WAR).

Interrupciones

Multiciclo

Las etapas tienen latencias dependiendo de lo que se quiera realizar. Por ejemplo podríamos pensar en:

- Añadimos multiplicación float (*):
  - Latencia de operación 5 ciclos (*1,* 2, *3,* 4, *5).
  - Latencia de inicio de $\frac{1}{5}$ ciclos.
- Modelamos una cache más realista:
  - Latencia de operación 3 ciclos (M1, M2, M3).
  - Latencia de inicio de $\frac{1}{3}$ ciclos.

Aparecen nuevos casos de parada:

- Load seguido de consumidora a distancias 1, 2 y 3.
- Multiplicación seguida de consumidora a distancias 1, 2, 3, 4 y 5.
- Estructurales.

Un camino de ejecución

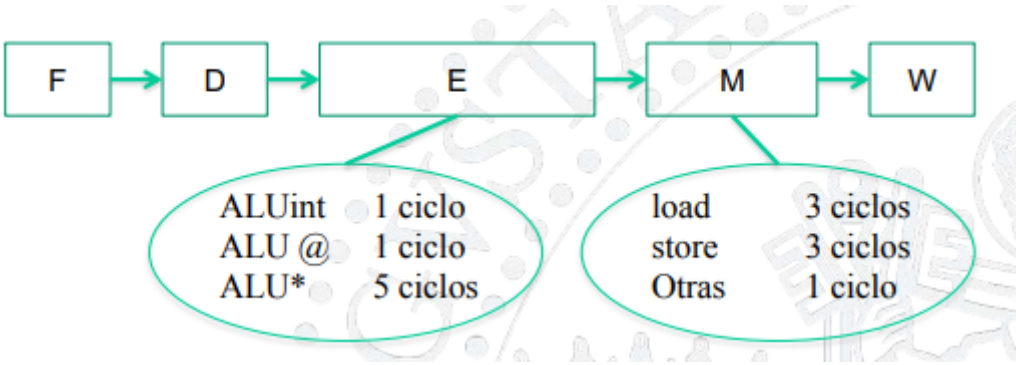


Figure 1: Un solo camino de ejecución

- **Terminación en orden**.
- **No hay riesgos WAW, WAR**.
- **Riesgos estructurales**:
  - Instrucción FLOAT seguido de cualquiera.
  - ld/st seguido de ld/st.
  - ld/st seguido de +/-.

Varios caminos de ejecución

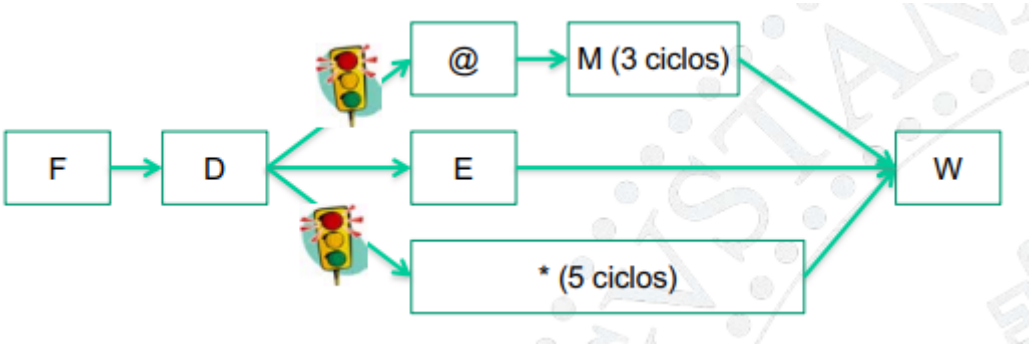


Figure 2: Varios caminos de ejecución

- **Terminación en desorden**.
  - Interrupciones imprecisas.
- **Riesgos WAW**.
- **Riesgos estructurales**:
  - FLOAT seguido de FLOAT.
  - FLOAT seguido de ld/st en W.
  - FLOAT sesguido de +/- en W.

– ...

Se crean cortocircuitos desde salida de cada UF a entradas de cada UF.

SCOREBOARD

Se usa para el correcto funcionamiento del procesador con varios caminos. Proporciona gestión de riesgos estructurales, de datos y de control.

Estructurales en UFs **Para cada UF multiciclo**:

- Contador que indica cuantos ciclos quedan en estado ocupado. Cuando se lanza una operación de latencia de inicio X, el contador se inicializa con el valor X-1.

Estructurales en BR **Vector de bits puerto escritura BR**:

Indica para los próximos ciclos, si el puerto de escritura está ocupado o libre. Por ejemplo:

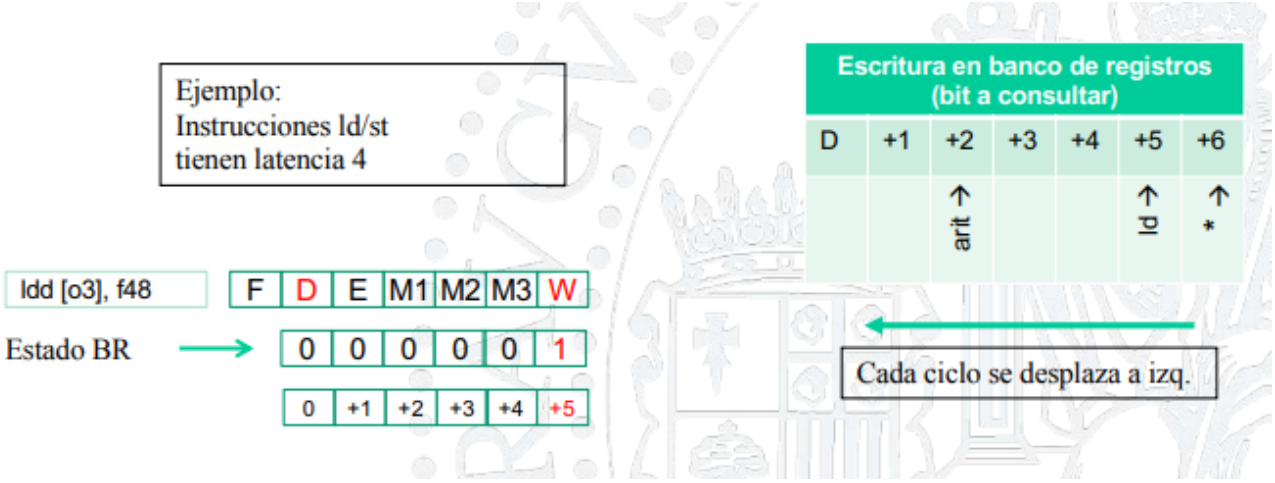


Figure 3: Vector del puerto de escritura

Cada instrucción lanzada desde etapa D, si escribe en BR:

- Comprueba que el ciclo que necesita el wBR está libre.
  - Si está **ocupado**, **espera**.
  - Si está **libre**, lo **ocupa** y continúa.

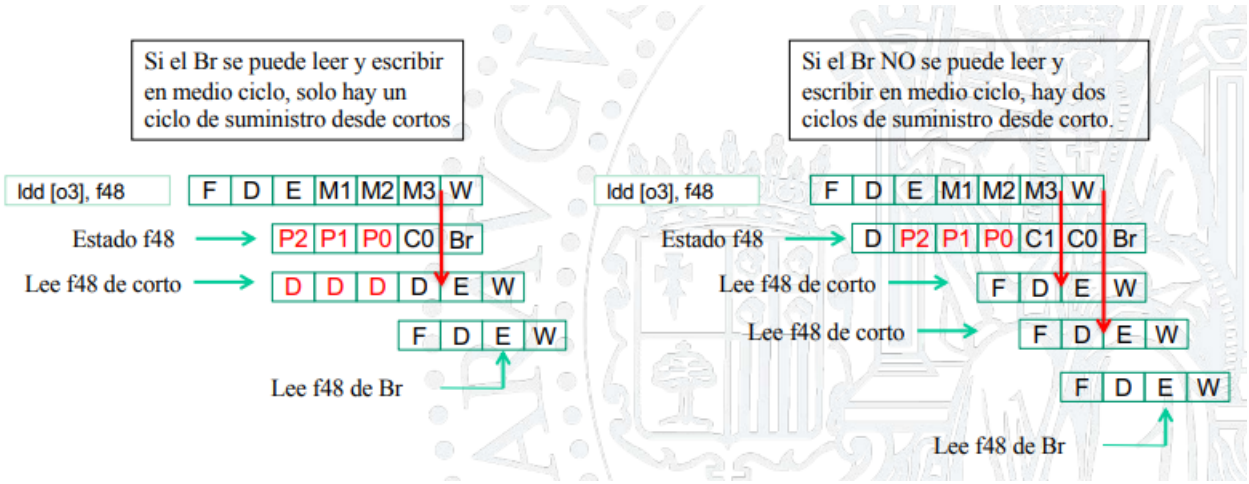


Figure 4: Muestra la ejecución con BR de medio ciclo y de ciclo entero

RAW y control de cortos Cuando una instrucción lee en su etapa D el estado de un registro fuente, éste puede ser:

- P: pendiente. La instrucción no puede avanzar.

- C: corto. Se podrá servir desde corto cuando la instrucción llegue a etapa E.
- Br: banco de registros. El valor leído en este mismo ciclo en etapa D es el válido. No se activa corto.

Dependiendo de si tenemos banco de registros capaz de leer y escribir en medio ciclo o no, tendremos un solo ciclo de serviccio desde cortos o dos.

WAW Instrucción "simple" (latencia 1) lee en su etapa D el estado de su registro destino.

- Estados P3, P2 y P1: la instrucción no puede avanzar.
- **Si avanza escribiría en desorden**.

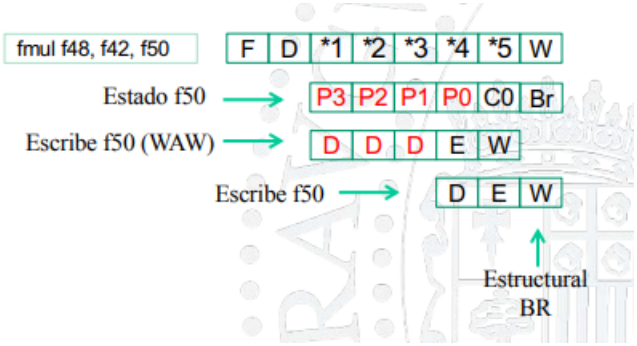


Figure 5: Muestra la ejecución de dos instrucciones consecutivas que escriben en el mismo registro

Problemas

Riesgos RAW y estructurales.

- Detectar y parar en decodificación

Terminación en desorden: interrupciones imprecisas.

- Asegurar terminación en orden
  - Añadir etapas artificiales para igualar caminos.
  - Reorder Buffer (ROB).

Riesgos WAW y WAR

- Detectar y parar en decodificación.
- Permitir varias versiones de cada registro (más adelante).

Terminación en orden: Añadir etapas

Añadir etapas para igualar latencias: etapas de Tránsito (T_i).

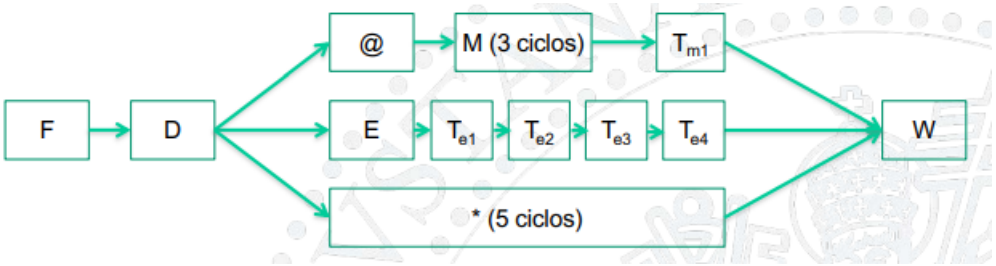


Figure 6: Ruta de datos con ejecución con misma latencia

Interrupciones precisas con punto de consolidación justo antes de etapa W.

NO riesgos WAW

- La escritura se realiza en orden de programa.

Riesgos estructurales

- FLOAT seguido de FLOAT.
- Ld/St seguido de Ld/St.
- Desaparecen los riesgos en W, todas las instrucciones llegan 5 ciclos después de D.

Riesgos RAW

- Todas las instrucciones tardan 6 ciclos hasta escribir en BR (de D a W).
- Mientras, las dependencias tienen que esperar o capturar valor mediante corto.
- **Aumenta mucho el número de cortos** o diminuyen prestaciones.
- También aumenta la complejidad del **scoreboard**.

Terminación en orden: ROB (Reorder Buffer)

Volvemos a los tres caminos con diferentes latencias.

- Moveremos el punto de consolidación tras la etapa de escritura (W).
- El banco de registros pasará a ser un **banco de registros especulativo (BRe)**, en el sentido de que algunos de estos datos aún no han sido consolidados y pueden ser erróneos. La escritura en este será en desorden en etapa W.
- Las instrucciones leen operandos de este BRe en etapa D, o de cortos.
- Solo un cortocircuito en cada camino, desde la UF.
- Añadiremos otro banco de registros, el **banco de registros consolidado (BRc)** en el que los valores están consolidados.
- Añadimos una nueva estructura: el **Reorder Buffer (ROB)**.

Banco de registros consolidado (BRc) Contiene el valor preciso de los registros tras la ejecución de la última instrucción consolidada y antes de ejecutar la instrucción más vieja que queda en ROB.

Reorder Buffer (ROB)

- Cuenta con una entrada por instrucción, en orden de programa
  - Se reservan en etapa D, si el ROB está lleno bloquea D.
  - Cuando una isntrucción produce resultado, lo escribe en ROB y en BRe.
  - Cada ciclo, si la instrucción más vieja del ROB ya ha terminado escribe resultado en BRc de valores consolidados y sale del ROB.

Interrupciones En caso de interrupción:

- Borrar el contenido del ROB (Todo lo que está a la izquierda del punto de consolidación se tira).
- Anular todas las etapas del segmentado.
- Guardar como PC de retorno el de la siguiente instrucción más vieja del ROB.
- Copiar BRc de consolidados a BRe de especulativos.
  - Por hardware, parte del tratamiento de la interrupción.
- Ejecutar rutina de interrupción.

Organizaciones alternativas

Organización con un banco de registros y ROB

- Los valores especulativos solo se guardan en ROB.
- ROB sirve operandos a las UFs en etapa D, junto a cortos y a BRc.
  - **Requiere búsquedas asociativas**.
- BRc: con valores consolidados. Sirve valores consolidados. También se usa para recuperar en casos de interrupción o salto mal predicho.

Organización con un banco de registros, ROB y tabla de mapeo (MT)

- Todo igual que en el caso anterior
- MT sirve para **evitar las búsquedas asociativas**.
- MT: tabla que asocia a cada registro el lugar donde está su resultado
  - Entrada del ROB o entrada del BRc.

MEJOR SOLUCIÓN HASTA EL MOMENTO.

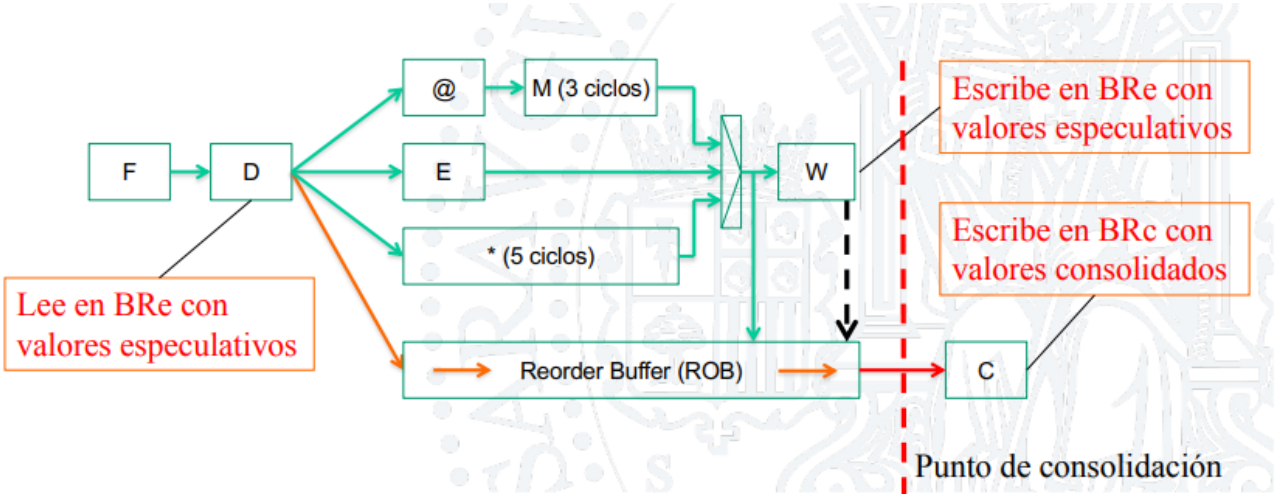


Figure 7: Ruta de datos con escritura en BRe en desorden, ROB y BRc

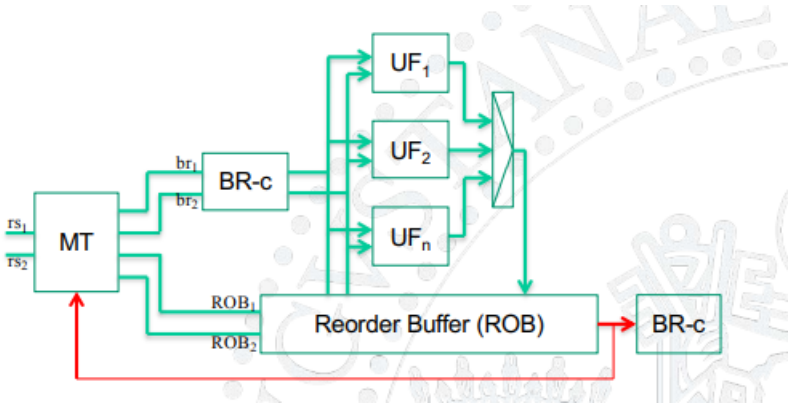


Figure 8: Ruta de datos con un BRc, ROB y MT

Ejecución fuera de orden

Procesadores que lanzan en orden

Resumen

- No existen puntos donde acumular instrucciones.
  - Los bloqueos se propagan hacia la etapa de búsqueda.
- Control de riesgos:
  - **WAR**: NO
    - Porque no se acumulan instrucciones y se lanzan con todos los operandos leídos.
  - **Resto de riesgos** se resuelven en **decodificación**:
    - RAW, WAW, estructurales.
- Para disminuir el número de detenciones, se debe **planificar en compilación**:
  - Alejar productoras de consumidoras respetando las dependencias.

Limitaciones

- Las instrucciones, tal como entran, deben lanzarse a una UF o bloquean la entrada.
- Una UF (alu, branch, +fp, *fp ...) puede bloquear la entrada de instrucciones aunque existan UF desocupadas.
- Una instrucción que debe esperar por riesgo RAW bloquea la entrada de instrucciones.
- Es posible que en el futuro existe trabajo que puede hacerse pero del que no hay noticia. Por ejemplo:

```
divd f0 , f2, f4
addd f10, f0, f8 ; bloqueo por riesgo RAW
subd f6 , f8, f4 ; PODRÍA EJECUTARSE
```

Procesadores que lanzan fuera de orden

Se realiza mediante planificación dinámica de instrucciones. El hardware cambia el orden secuencial de ejecución.

Si una instrucción no se puede ejecutar:

1. Acumulo la instrucción no ejecutable en un **buffer**, conocido como **Ventana de lanzamiento** (**IW**).
2. Observo si la siguiente se puede ejecutar.

Hay que tener cuidado con los riesgos de memoria. Se debe **desambiguar**: una instrucción de memoria (**i+k**) solo puede ejecutarse después de un store (**i**) si se comparan sus direcciones y son diferentes.

División del trabajo en etapas

Etapa D (Decode)

- Inserta las instrucciones en la ventana de lanzamiento y en ROB.
- Para si no hay hueco en alguna de las dos estructuras.

Etapa I (Issue)

- En cada ciclo, selecciona, entre todas las instrucciones de la ventana, la más antigua de las que tienen sus operandos preparados y su UF libre.
- Organización en dos sub-etapas:
  - **Despertar**: entre las instrucciones de la ventana, detectar las que tienen preparados sus operandos.
  - **Seleccionar**: entre las instrucciones despertadas, seleccionar las que pasan a ejecutar en el siguiente ciclo.
    - Las más antiguas entre las que tienen su UF libre.

Etapa R (Read)

- Lectura de operandos en el banco de registros.
- Se puede colocar después o antes de etapa de lanzamiento.

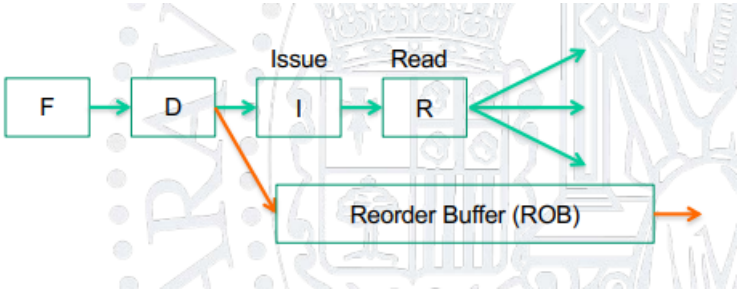


Figure 9: En esta figura se muestra la ruta de datos correspondiente al procesador fuera de orden.

Ventana de lanzamiento: Despertar

Implementación mediante CAM (Content-Addressable Memory)

- Instrucciones leen bits Ready para sus operandos en etapa D.
- Las instrucciones productoras difunden el id de su Rd cuando está disponible. ***(Esto no podría causar problemas)***.
- Se despiertan las instrucciones con operandos disponibles (desde BR o Cortos) y su UF disponible.

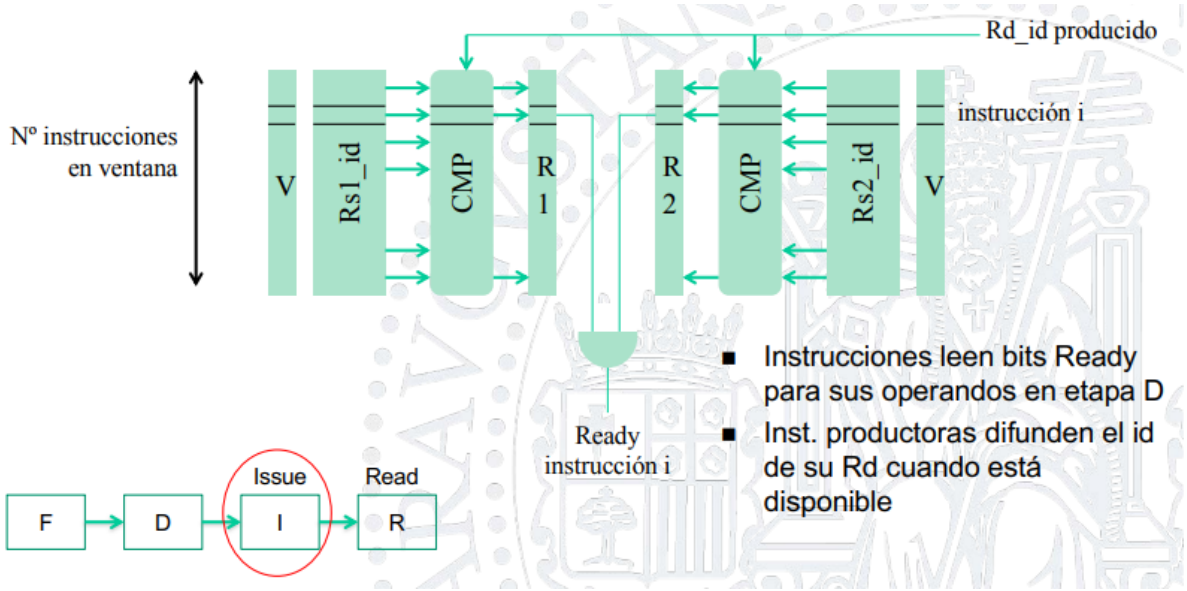


Figure 10: Estructura de la ventana de lanzamiento implementada con CAM.

Implementación mediante matriz

- A la larga no es viable por el coste hardware, hay que añadir muchas puertas.

Ventana de lanzamiento: Seleccionar

El tiempo es crítico: despertar y seleccionar en un ciclo.

- Implementación mediante varios schedulers (planificadores, árbitros, puertos).
- Cada scheduler es un codificador con prioridad.
- Ejemplo:

Temporización

Latencia entre instrucciones dependientes. Se soluciona teniendo en cuenta cuándo estará disponible el registro.

Otras alternativas: Read antes de Issue

- Etapa R: se leen los operandos disponibles.
- Etapa I: la ventana de instrucciones almacena los valores de los operandos disponibles y los identificadores de los no disponibles

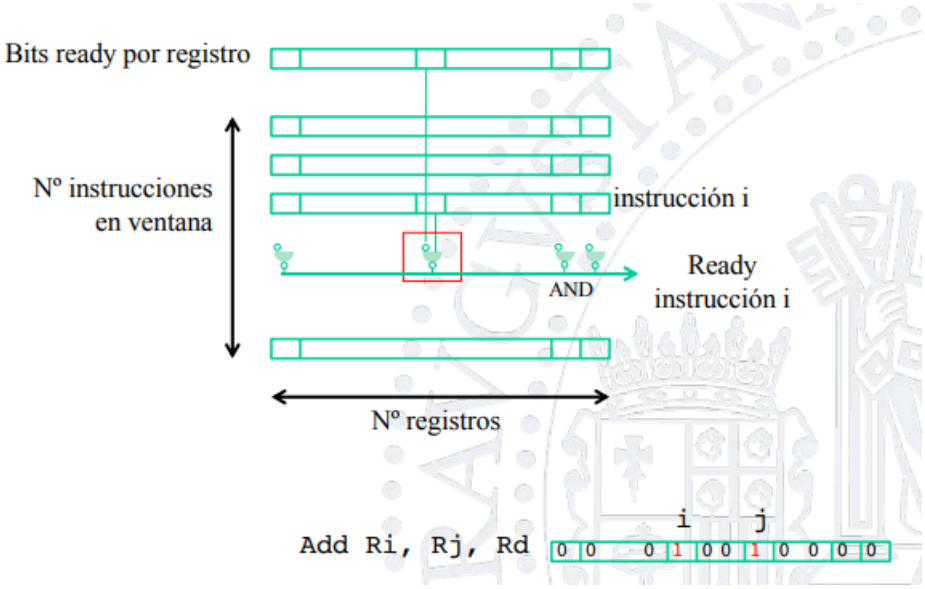


Figure 11: Estructura de la ventana de lanzamiento implementada mediante matriz.

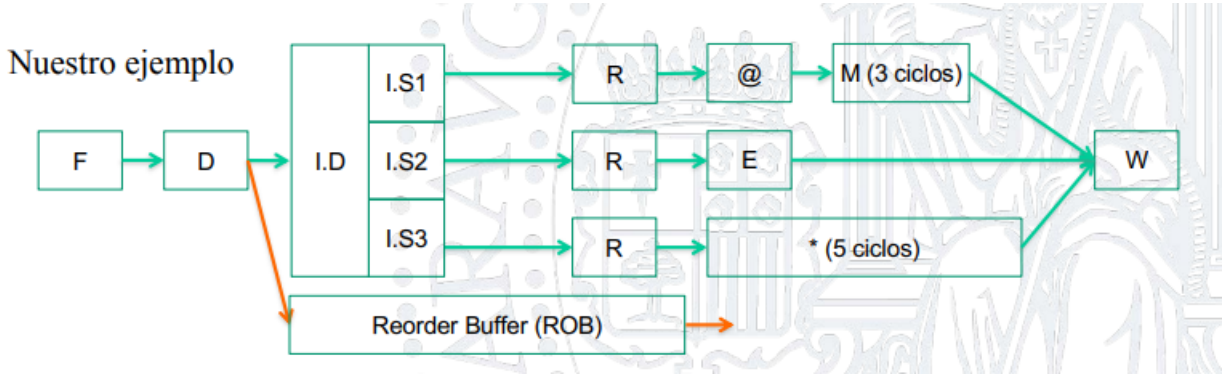


Figure 12: Estructura de ruta de datos con schedulers para asignación de caminos.

- Cuando una instrucción produce resultado, éste se difunde junto con el identificador del registro destino.
- Instrucciones en la ventana capturan el valore de los operandos que les faltan.
- Cuando tienen todos los operandos despiertan.

Otras alternativas: Estaciones de reserva

Propuesta por Tomasulo.

- Cada UF tiene un almacen de instrucciones que quieren ejecutarse en ella: estaciones de reserva (RS).
- Las RS tienen hueco para guardar el valor de los operandos: etapa R antes de etapa I.
- Los operandos se difunden desde las UF hacia las RS junto al identificador de registro (en realidad identificador de RS).

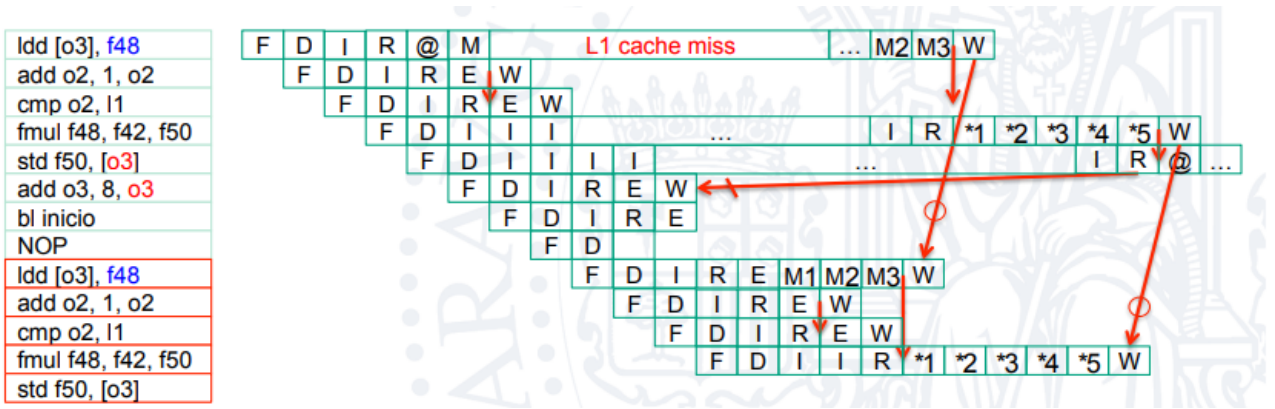
Renombre de registros

Partimos de un procesador con varios caminos de ejecución y lanzamiento fuera de orden.

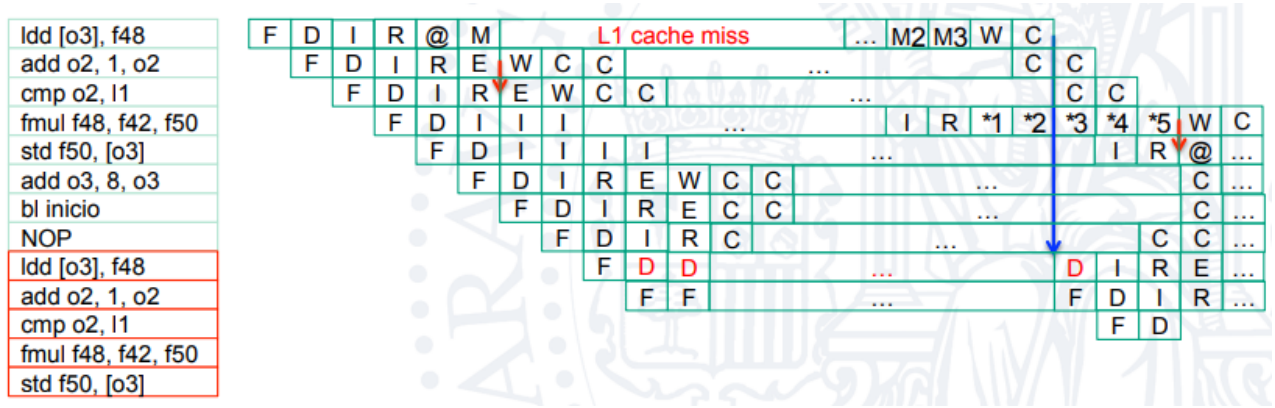
Como resolver las dependencias:

- Detención y parada en D (suponemos ROB):
  - Detener a una instrucción hasta que la instrucción previa que escribe el mismo Rd haya consolidado.
  - Tabla con una entrada por registro. Nos indica si su último valor ha consolidado o NO.
  - Se escribe un 0 en etapa D, y un uno en C (Consolidación) para el registro destino.

Ejemplo con riesgos WAW y WAR



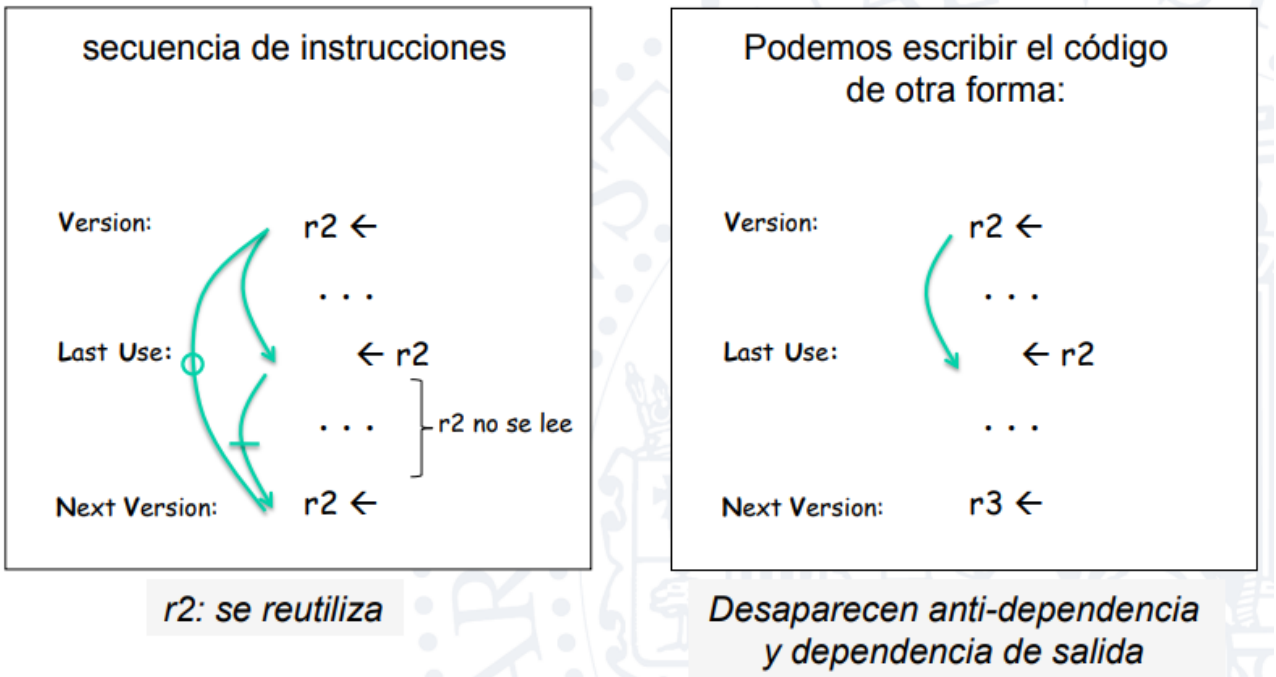
Solución



Concepto de versión

Las dependencias WAW y WAR aparecen por la reutilización de almacenes. Cada valor requiere un almacén desde que se produce hasta que lo consume la última instrucción consumidora. **Después el almacén puede reutilizarse**. También es posible que, según el orden de programa, tengamos que guardar un nuevo valor de un registro antes de que el anterior haya sido leído por todas sus consumidoras. **Necesitamos un almacén para varias versiones de cada registro**.

Concepto de reutilización de almacenes



Registros lógicos vs físicos

Los registros lógicos son los que usa el programador, compilador para crear el programa. Internamente el procesador asigna a cada registro lógico un registro físico dependiendo de las dependencias que pueda haber en el código para así reducirlas.

Dispondremos de más almacenes (registros lógicos) que registros:

- Guardaremos cada versión en un nuevo almacén (renombre).
- A las consumidoras les anotaremos el almacén donde deben leer.
- Liberaremos los almacenes cuando sepamos que ya no van a ser leídos por ninguna consumidora (consolida la siguiente versión).

Tabla de renombre en D

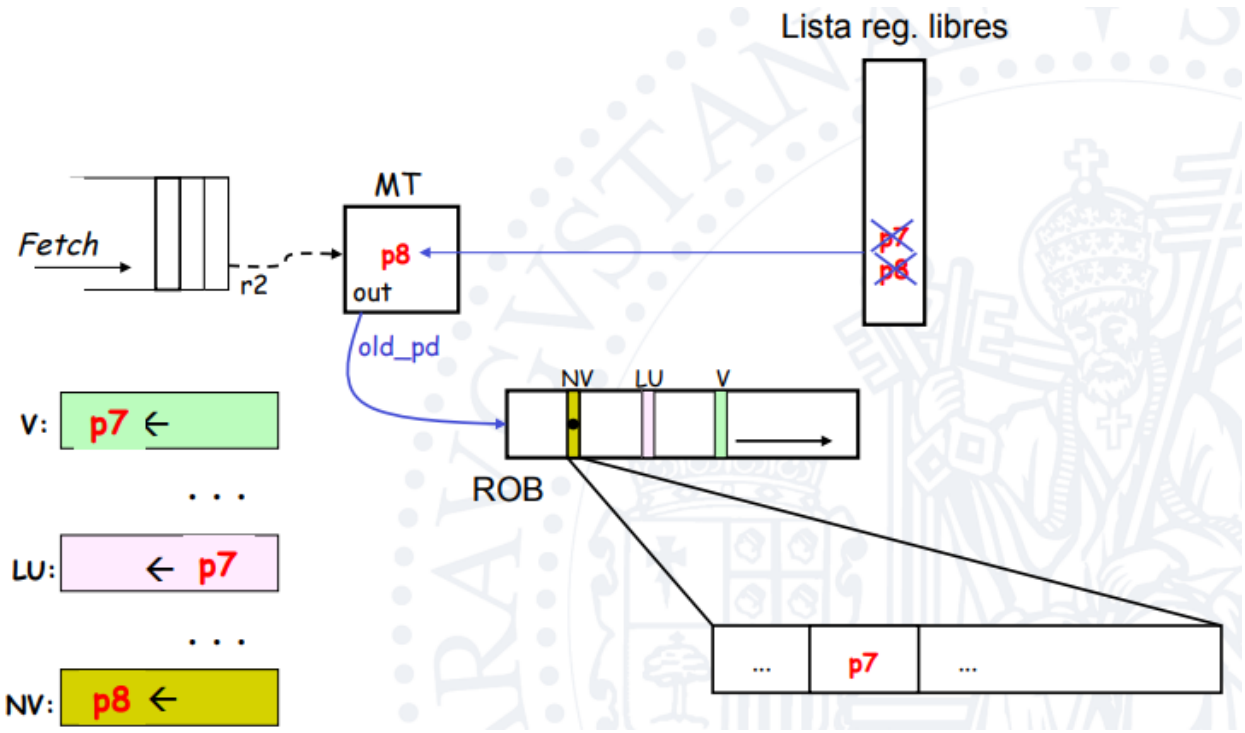
- Mantiene el nombre asignado a la última versión de cada registro.
- Se modifica cada vez que se crea una nueva versión.
- Se consulta para renombrar los fuentes de cada instrucción.

El renombre se puede llevar a cabo en el banco de registros, en el ROB o en las estaciones de reserva.

Renombre en banco de registros

- Partimos de la organización con ROB y dos bancos de registros.
- Aumentamos el tamaño del BR-e para que pueda guardar varias versiones de cada registro.
  - Por ejemplo: para 32 registros lógicos → 64, 100 entrada en el BR-e.
  - **R0..R31** se traducen en los registros físicos **P0..P63**
- Añadimos tabla de renombre o de mapeo **MT**.

Ejemplo de funcionamiento de renombre de registros



Implicaciones en etapa Issue

El despertar de la etapa Issue se guía por los registros físicos, no por los lógicos, esto implica que habrá más entradas y **alguna opción de implementación ya no es viable**.

Recuperación del estado preciso

En la **organización sin renombre se copiaba el BR-c sobre BR-e**. Al añadir renombre, **BR-e tiene más entradas que BR-c**.

Recuperación del estado preciso 1

- Copiar BR-c sobre algunas entradas de BR-e.
- Modificar la MT para reflejar nuevo estado.

Esto es muy complicado.

Recuperación del estado preciso 2 Solo se emplea un banco de registros y dos tablas de mapeo.

Renombre en el ROB

- El ROB puede almacenar varias instrucciones que producen distintas versiones del mismo registro.
- MT guarda, para cada registro, la entrada del ROB que almacena la última versión.
- A MT se accede en etapa D y nos traduce de un registro lógico a una entrada del ROB donde está la instrucción productora.
- En la etapa de Issue detecta riesgos RAW a través de este identificador.

Renombre en las estaciones de reserva

- MT guarda, para cada registro no disponible, la estación de reserva (RSid) que producirá la última versión.
  - Las consumidoras leen sus registros fuente disponibles en etapa R y renombran sus registros fuente no disponibles con el RSid que indica MT.
  - Las instrucciones productoras difunden RSid y resultado por el CDB.
- Si aparece una nueva instrucción con el mismo Rd (nueva versión).
  - Cambia la MT asociando al Rd la estación de reserva de la nueva instrucción.
  - Las consumidoras futuras leerán en MT la RSid de la nueva productora.

Orden en el acceso a memoria

- Stores modifican memoria al consolidar.
- Loads y stores ejecutan en orden de programa, por tanto, todos tras consolidar. Esto implica **mucho retardo para las consumidoras de load**.
- Podemos añadir especulación:
  - Las direcciones no se conocen hasta la etapa de cálculo @
    - No podemos detectar dependencias en etapa D.
  - Apostar **solo** que las instrucciones previas a un load no provocan excepción.
  - Apostar además que los stores previos a un load no escriben en la misma dirección de memoria.
  - Predecir situaciones de conflicto.

Ejecución conservadora de loads OOO

- Apostar que las **instrucciones previas a un load no provocan excepción**: consultar direcciones de stores previos.
- Store se ejecuta en dos fases: **cálculo de dirección y escritura**.
  - Lanzar cálculo de dirección en cuanto se pueda.
  - Almacenar las direcciones de todos los stores en vuelo.
- **Load espera en issue hasta que se conozcan las @ de todos los stores previos**
  - Después: load calcula @ y compara con la de stores previos.
  - Si hay coincidencia con @ de un store, dos posibilidades:
    - Store conoce dato → se suministra el dato al load.
    - Store no conoce dato → el load se aborta.

Vida de la instrucción store

- Etapa D de store:
  - Reserva entrada en Store Queue (STQ).
  - Inserta en ventana las dos subinstrucciones
- Cuando los registros @ están preparados:
  - Se lanza cálculo de @.
  - @ se escribe en STQ.
- Cuando @ calculada y registro dato preparado:
  - Se lanza la 2^{a} subinstrucción.
  - Escribe dato en STQ.
- Store en cabeza de ROB y @ y dato disponibles: **store consolida**:
  - Escribe dato en memoria (cache).
  - Libera entradas ROB y STQ.

Ejecución agresiva de loads OOO

- Apostar que las instr. previas no provocan excepción.
- Apostar que los stores previos a un load no escriben en la misma dirección de memoria.
- Store ejecuta cuando tiene operandos fuente preparados y no divide la instrucción en dos (o si ...).
- Load ejecuta sin conocer direcciones de stores previos
  - Continuamente se verifica la corrección.
  - Loads guardan su dirección en LDQ y stores en STQ en orden.
  - load calcula @: compara con stores previos ya calculados
    - En coincidencia: load toma valor del store.
  - store calcula @: compara con loads posteriores ya calculados.
    - En coincidencia: abortar load (y dependientes : ¿ también se debería puesto que no tendrían un valor correcto ?)
  - Puede provocar muchos loads abortados.

Predicción de dependencias

- Apostar que las instr. previas no provocan excepción.
- Predecir si los stores previos a un load escriben en la misma dirección de memoria o no.
- En principio todo funciona igual que en el caso anterior.
  - Load ejecuta sin conocer direcciones de stores previos.
- Cada vez que hay que abortar un load, se recuerda:

  - Próximas ejecuciones de este load tendrán que esperar hasta que todos los stores previos hayan ejecutado.
  - Disminuye el número de loads abortados.
  - Se requiere de hardware para recordar

¿Cómo abortar una instrucción load?

- Ha podido despertar a sus instrucciones dependientes.
- Tratarlo como a una excepción:
  - Marcar la instrucción y tratarla cuando llegue a consolidación.
  - Volver el load a etapa de Fetch.
  - **Penalización muy alta** (caso más frecuente que la excepción).
- **Repetir trabajo de load sólo desde Issue**:
  - No es necesario volver a buscar y decodificar la instrucción, ese trabajo se hizo correctamente.
  - Volver a lanzar el load desde la ventana
    - El load no se elimina de la ventana nada más lanzarlo, se espera hasta que calcule @ y confirme que no tiene problemas.
  - Dependientes: re-ejecución selectiva (solo dependientes) o no selectiva (todas las instrucciones más jóvenes).
    - Mantenerlas en la ventana hasta que sea segura su ejecución.

Saltos

El salto se suele resolver en D si son saltos relativos a PC o tras la ejecución si se calcula en base a registros. El retardo en la resolución de un salto aumenta con el número de etapas del frontend (de Fetch a Decode). Cuantas más instrucciones ejecutándose en el procesador (p.e. en un procesador superescalar), más penalización. En el caso de haber dependencias puede aumentar hasta cientos de ciclos (p.e. en un fuera de orden):

```
ldd [o3], o5
cmp o5, 0
beq destino
```

Si el load falla en cache, la penalización puede ser de cientos de ciclos.

Soluciones

Salto retardado

- Solución sencilla para segmentados de pocas etapas.
- Se complica con supersegmentados y superescalares.
- No sirve para fuera de orden.

Predicción de dirección destino de salto

Predicción del sentido de salto (T/NT)

- Predicción en la propia etapa de Búsqueda de instrucción.
- Estática (hardware/software).
- Dinámica (local o temporal/ global o espacial / híbrida).

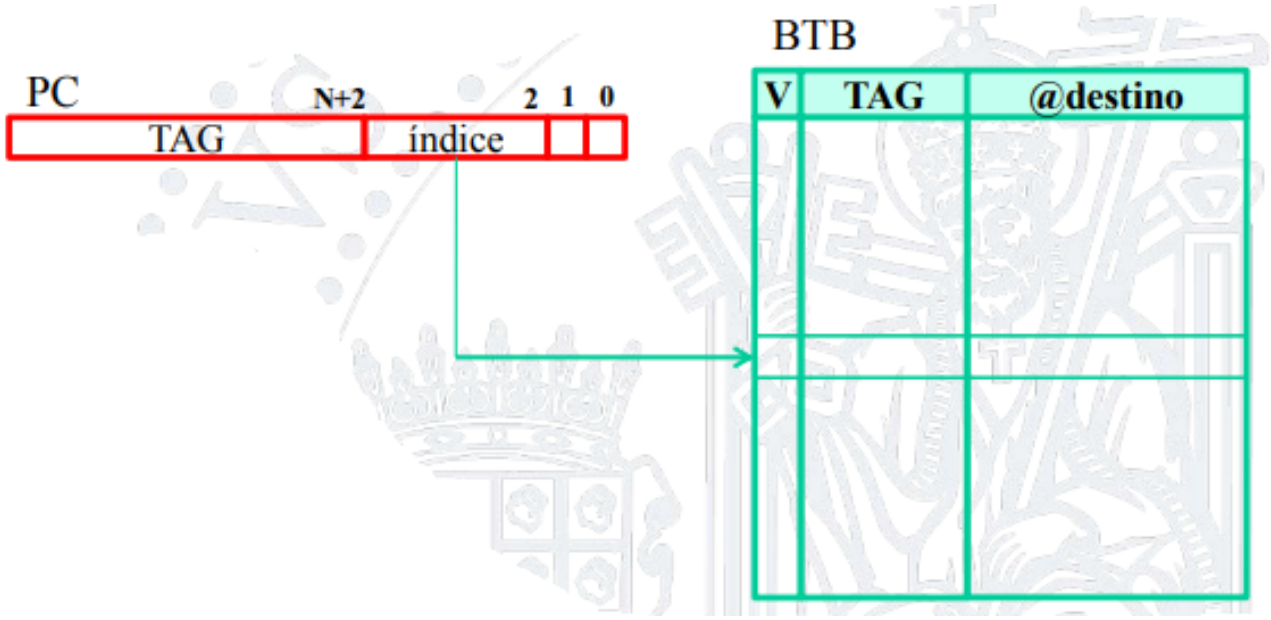
Predicción de dirección de destino de salto

Siempre se cuenta con un BTB y una pila de direcciones de retorno.

Branch Target Buffer (BTB)

- Un solo salto por entrada: guarda la dirección destino de su última ejecución.
- Se consulta en etapa de FETCH indexando con el PC de la instrucción.
- **Cuando se actualiza ??**

BTB



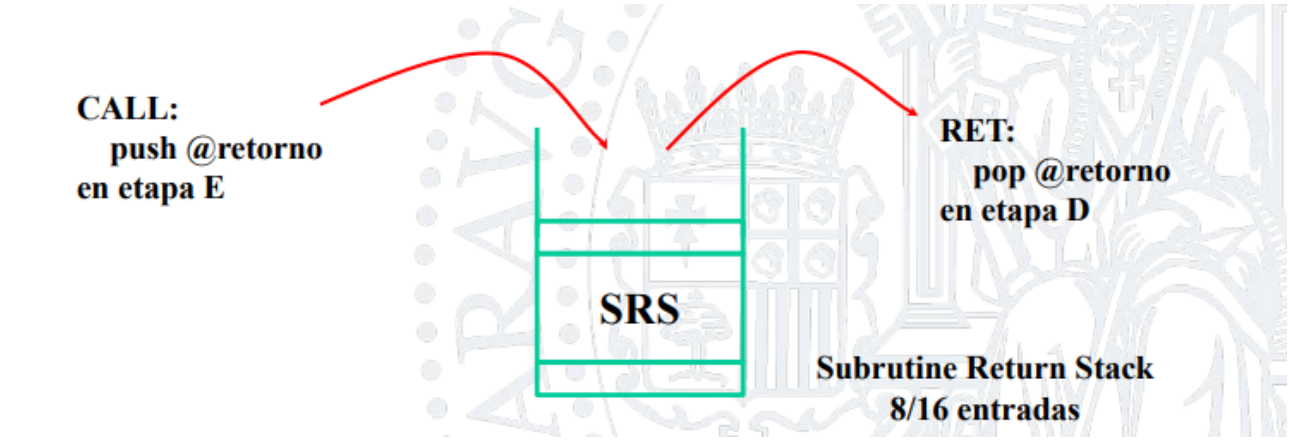
No siempre funciona bien, pero:

- Bucles → **funciona bien**.
- Condicionales → **funciona bien**.
- Switch → **funciona bien** si se traduce como if-then-else.
- Llamada a subrutina → **funciona bien**.
- Llamada a subrutina a través de puntero → **raro**.
- Retorno de subrutina → **desastre si la subrutina es llamada desde distintos sitios**.

Pila de direcciones de retorno

Se trata de una estructura sencilla con precisión mucho mayor que BTB para los saltos de retorno de subrutina. Es un hardware que actua como predictor.

Pila de direcciones de retorno



- Cada vez que se ejecuta una instrucción call, guarda su dirección.
- Estructura de pila.
- Cada vez que se ejecuta retorno, se saca de la pila la dirección guardada en el top y se salta a ella.

Predicción estática hardware T/NT

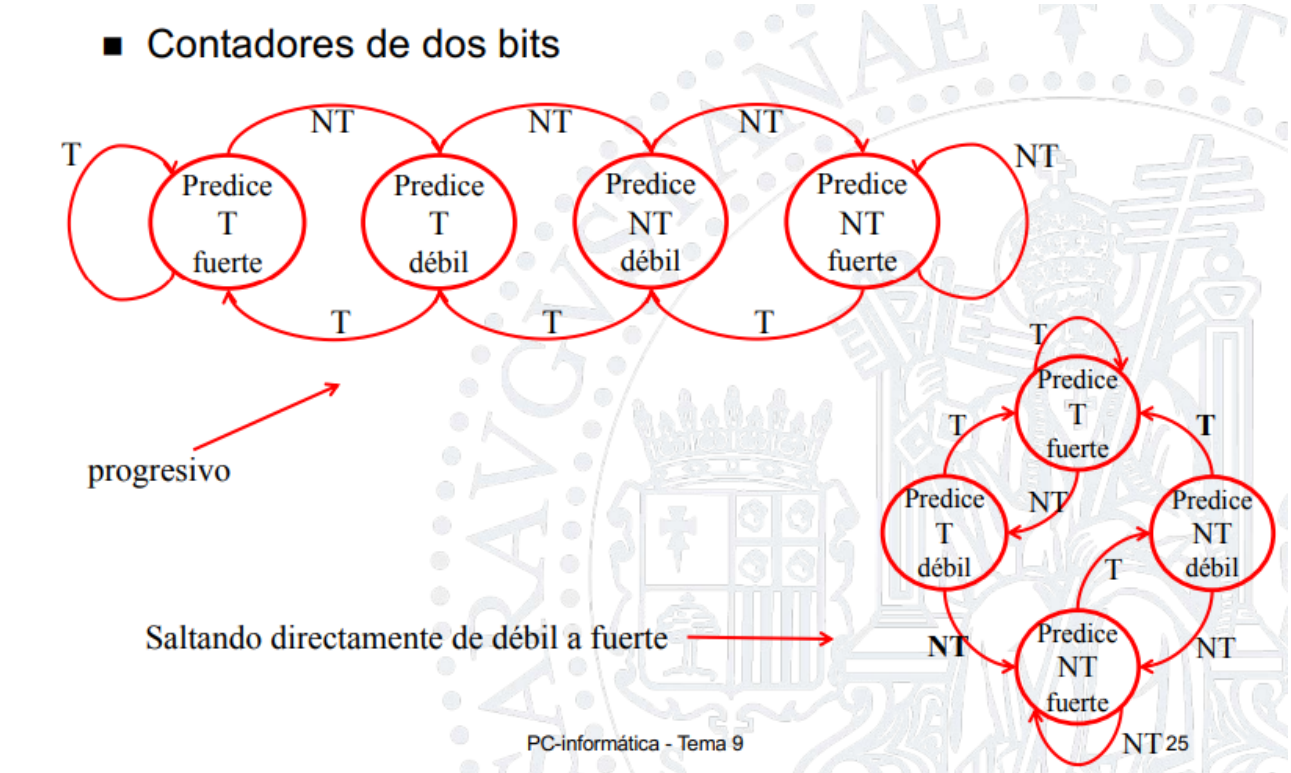
- **Predecir siempre NT** → simple, falla mucho en bucles.
- **Predecir siempre T** → necesita predecir dirección destino, más precisión que NT.
- **Hacia atrás T, hacia delante NT** → mejor que anteriores pero más compleja.

- **Compilador marca cada salto con predicción**:
  - Lo tiene que soportar el ISA.
    - Bit en instrucciones de salto.
  - Se conoce en etapa D
    - Al menos un ciclo de retardo.
  - Se acierta entorno al 80% de los saltos.

Predicción dinámica local

Cada salto usa información previa del propio salto. Explota correlación temporal:

Ejemplos de predictores dinámicos locales



- La forma en que se ha comportado un salto en el pasado puede ser un buen predictor de cómo se comportará en el futuro (ej. bucle).
- Un contador con saturación para cada salto.

Branch History Table (BHT)

Tabla con 2^N entradas, indexada con N bits del PC de la instrucción, con un contador por entrada (de 1 ó 2 bits).

- Se lee en etapa Fetch.
- Si BTB indica que la instrucción es un salto y BHT predice salto tomado
  - Saltar a @ del BTB.
  - No se pierde ningún ciclo.
- Se actualiza en etapa C.
- 4k entradas → 80/90% aciertos

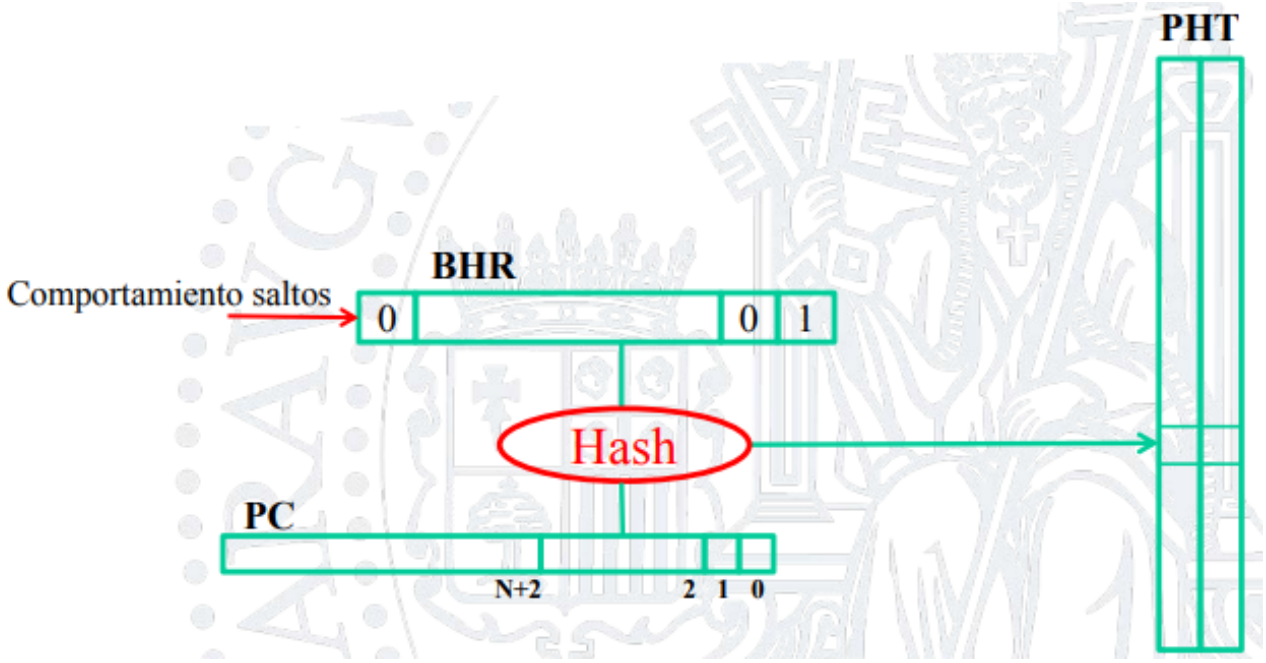
Predicción dinámica global

Cada salto usa información previa de los últimos saltos ejecutados por el procesador. Explota correlación espacial:

- La forma en que se comporta un salto puede dependeer de cómo se comportan los saltos cercanos en el código (ej. bucle).
- **Branch History Register (BHR)**: registro de desplazamiento con K bits que recuerda el comportamiento (T/NT) de los últimos K saltos ejecutados.
- **Pattern History Table**: guarda el comportamiento asociado a cada patrón.

Predictor híbrido: gshare

Gshare



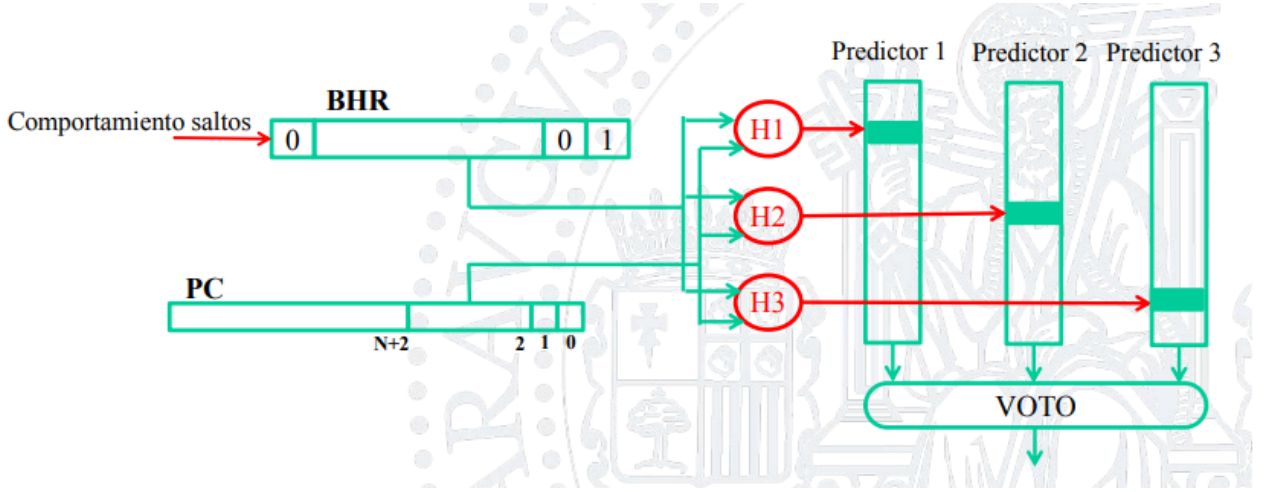
Guardar la historia global específica para cada salto:

- Tabla indexada con una mezcla de PC e historia global.
- Dos saltos pueden asociar a la misma historia global comportamientos distintos (entradas distintas en PHT).

Predictor híbrido: gskew

Objetivo: reducir conflictos en el predictor.

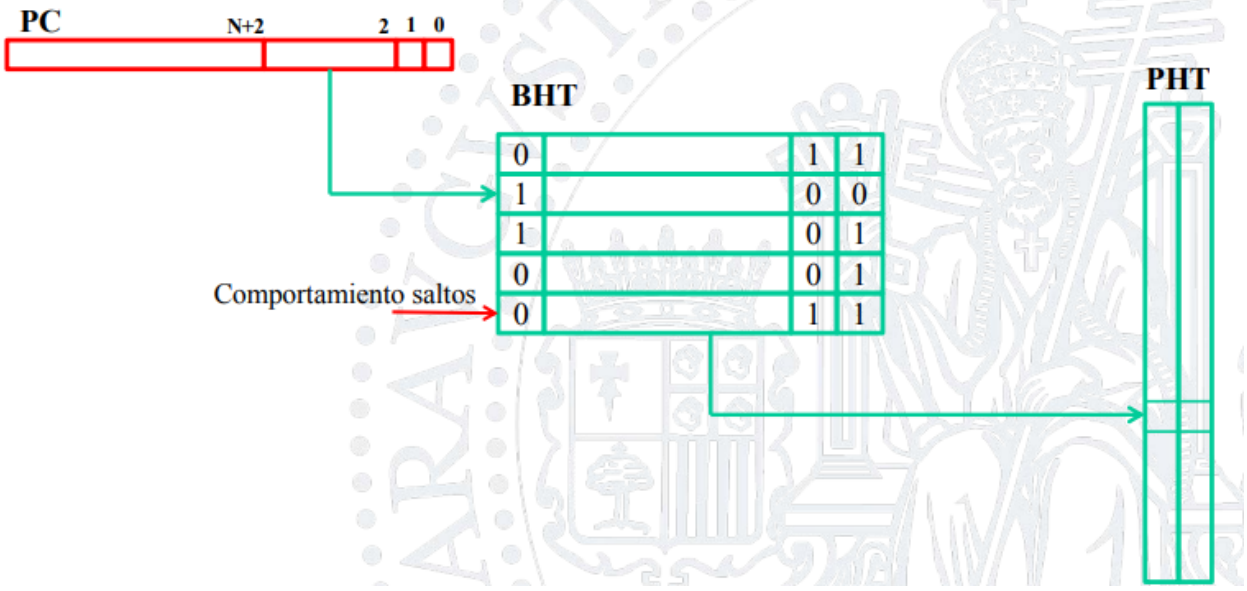
Gskew



Predictor local de dos niveles

Se emplea BHT (Branch History Table), guarda la historia local de cada salto.

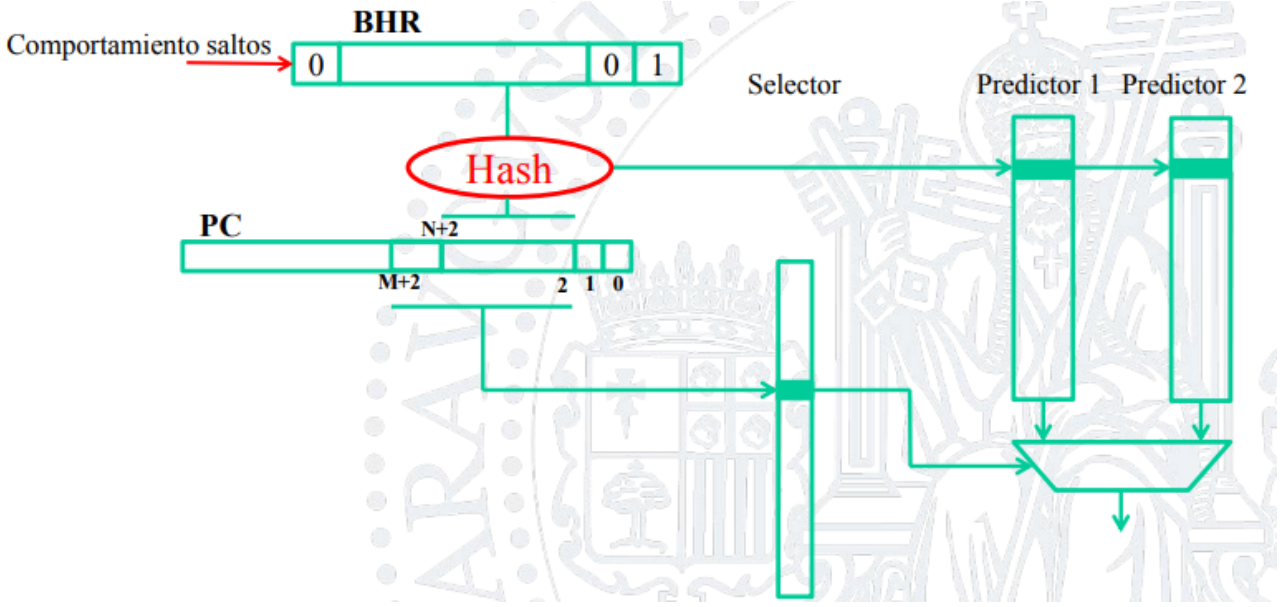
Predictor local de dos niveles



Predictor híbrido: bi-modo

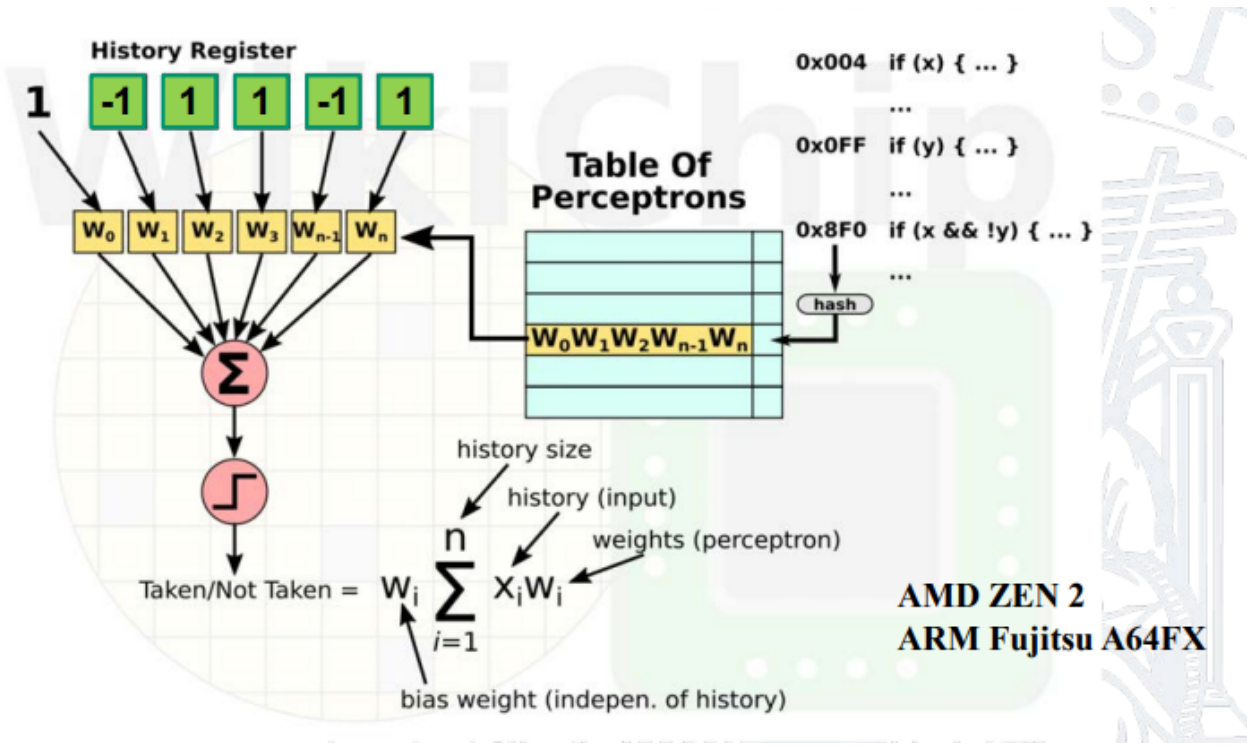
Objetivo: reducir conflictos en el predictor. Es complicado. Libro de *Shen y Lipasti pp473*

Predictor bi-modo



Perceptrón

Perceptrón



Recuperación en caso de error de predicción

- Las instrucciones posteriores al salto se deben anular y la etapa de búsqueda tiene que empezar en el camino correcto.
- El estado del procesador debe quedar como si las instrucciones del camino incorrecto no se hubiesen ejecutado.
  - **Problema**: han modificado las estructuras de la etapa decode:
    - ROB, colas de load y store, IQ.
    - tabla de renombre y lista de registros libres.
- Opción 1: recuperar en jubilación (como en excepción):
  - Frecuencia de error en predicción mucho mayor que de excepción.
  - **Pérdida de prestaciones muy grande**.
- Opción 2: recuperar en ejecución
  - Minimiza el tiempo perdido, recupera en cuanto se detecta error.
  - ROB, colas de load y store, IQ es fácil.
  - Tabla de renombre y lista de registros libres más complicado. Dos opciones:
    - **Rollback**: Partiendo de las tablas actuales, deshacer los cambios con la información almacenada en ROB de cada instrucción anulada: rd, pd y old_pd.
      Ejemplo: rd=r5, pd=p7, old_pd=p9 Cuando esta instrucción pasó por decode se solicitó un nuevo registro físico (p7) y se cambió el mapeo de r5 que estaba asignado a p9 que pasó a estar asignado a p7. Recuperación: p7 se devuelve a la lista de libres y r5 vuelve a estar asociado a p9.
    - **Snapshot**: En cada salto guardar copia de las tablas.