Arquitectura de Computadores

Parte 4

Carmen Azorín Martí

Introducción 1

Recordatorio del tema 1 2

Apartado 1: Microarquitectura de núcleos ILP superescalar 5

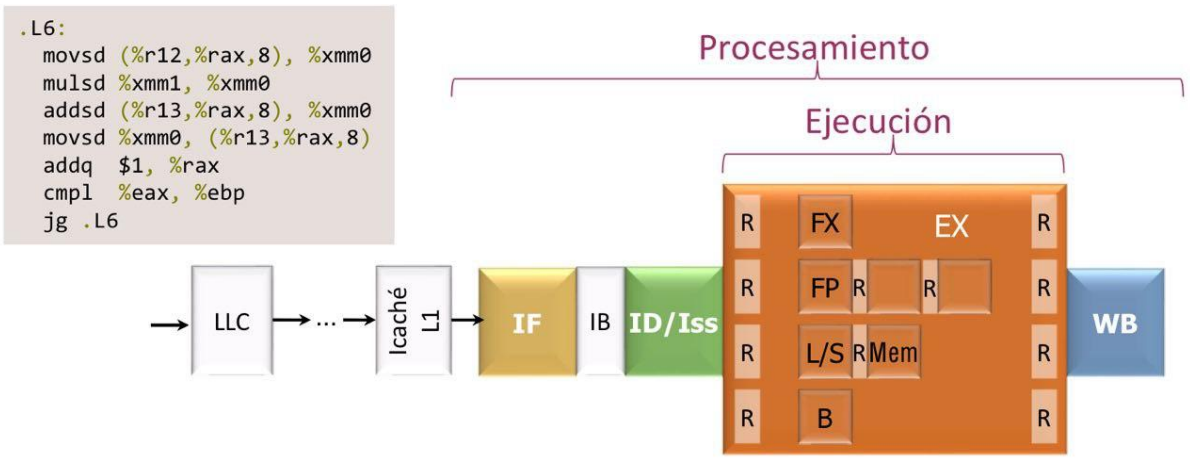
Apartado 2: Microarquitectura ILP VLIW 22

Introducción

Vamos a ver los procesadores que disponen de un cauce en el que cada etapa puede procesar más de una instrucción simultáneamente. Esto significa que se pueden arrancar o emitir varias instrucciones **en paralelo**.

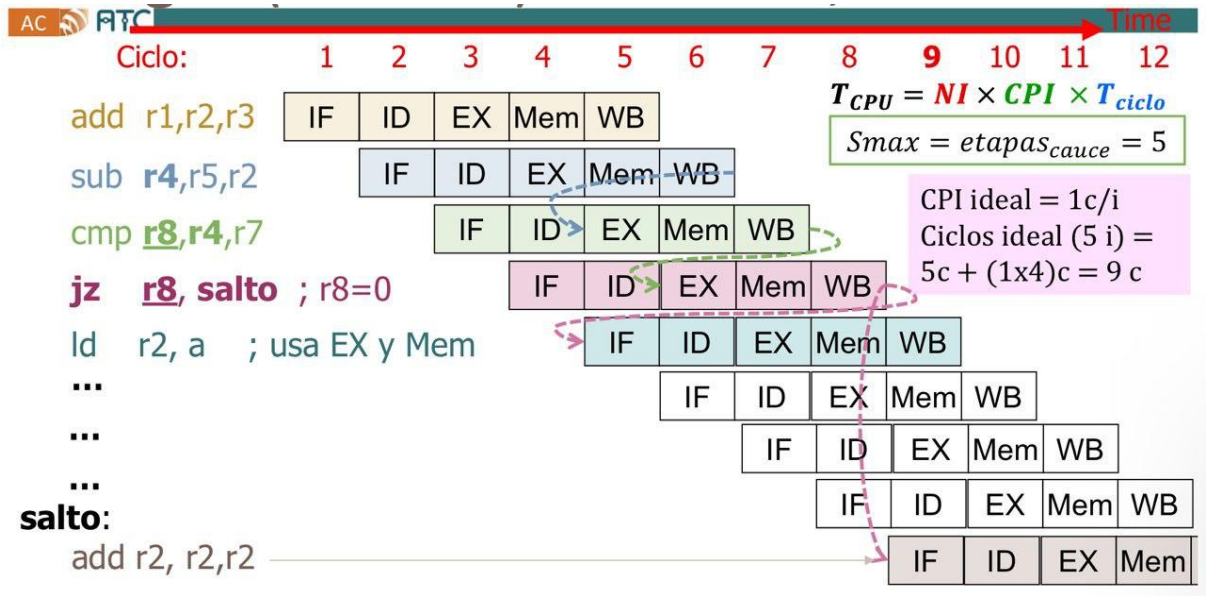
La planificación de qué instrucciones van a ejecutarse en paralelo puede hacerse estática o dinámicamente (la hace el compilador o la hace el procesador). El modelo de **planificación estática** que vamos a ver es el de los procesadores **VLIW** (Very Long Instrucction Word), y en la **planificación dinámica** vamos a abordar los procesadores **superescalares**.

Tiene sentido entonces que, en los procesadores VLIW, el **software** sea el que determina qué instrucciones se ejecutan en paralelo. Mientras que en los superescalares, es el **hardware** el que lo determina en tiempo de ejecución.



Vemos en la imagen un código y el diseño de una arquitectura segmentada.
A la hora de procesar este código, lo primero que se hace es captar las instrucciones de la caché 1 en la etapa de captación (**IF**), se captan **en el orden** en que aparecen en el programa.
Estas instrucciones captadas se colocan en el buffer de instrucciones (**IB**).
Gracias a este buffer, las instrucciones se van cogiendo **en orden** para decodificarlas en la etapa de decodificación (**ID/Iss**).
Posteriormente, pasan a la etapa de ejecución (**EX**). En esta etapa se encuentran las unidades funcionales, que se encargan de hacer las operaciones pedidas por la instrucción. Por ejemplo, en este caso hay una unidad funcional dedicada a las instrucciones para enteros (FX); otra para las instrucciones de punto flotante (FP); otra unidad dedicada a la carga y al almacenamiento (L/S); y, finalmente, una encargada de las instrucciones de salto (B). Cada unidad puede ejecutar las instrucciones en una o varias etapas. Por ejemplo, la unidad de FP se divide en 3 subetapas, mientras que la de FX se ejecuta en una única etapa.
Además, hay que destacar que las operaciones se pueden emitir a ejecución **desordenadas**.
Finalmente, la etapa de write-back (**WB**), donde los resultados se guardan en los registros **en orden**, de manera que parezca que se han procesado de forma secuencial.

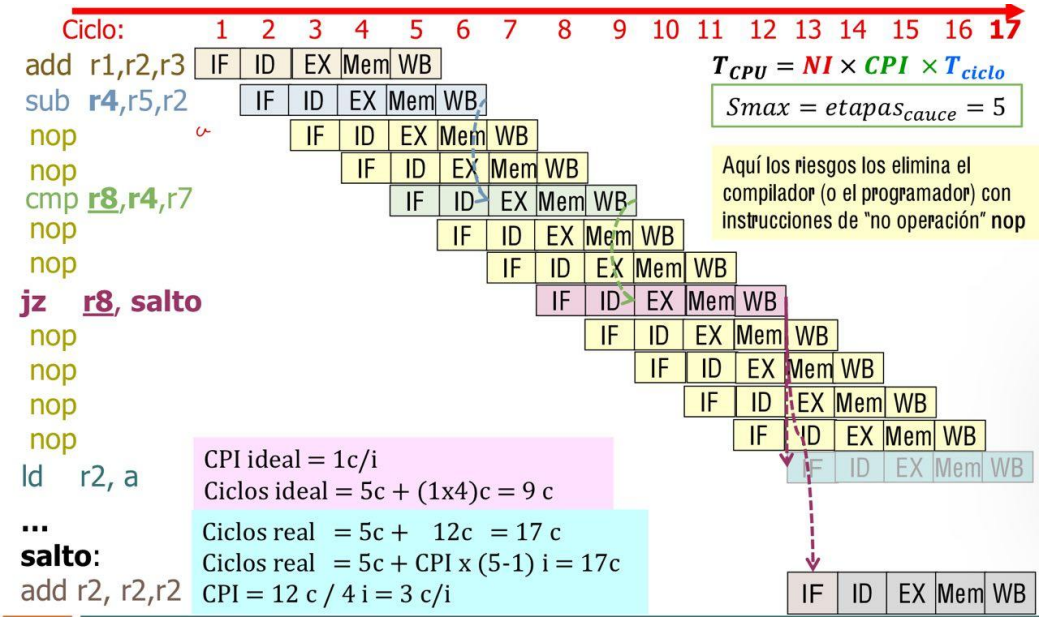
Recordatorio del tema 1



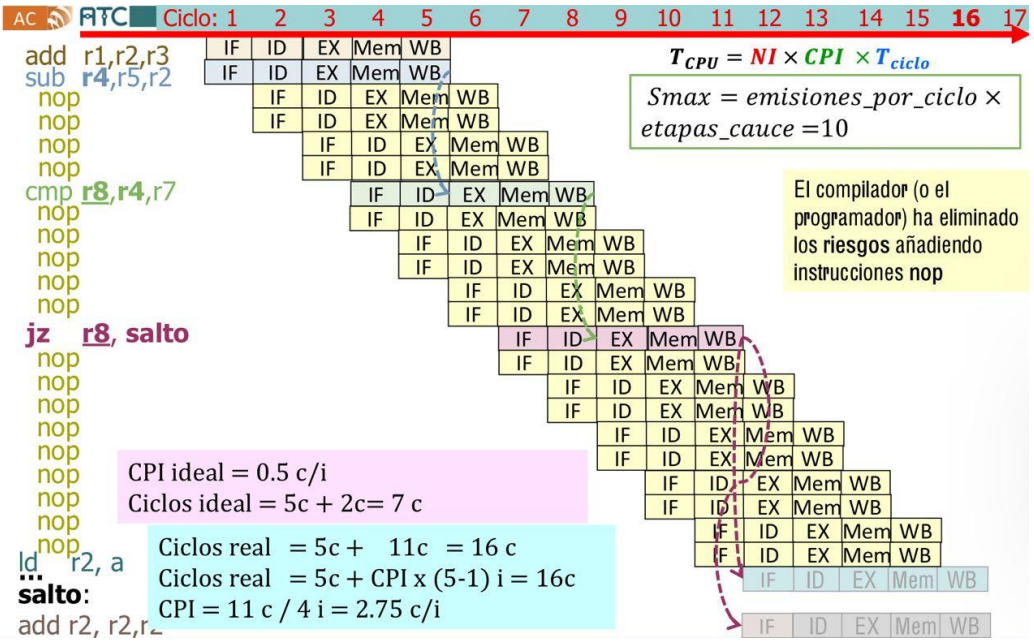
Estamos viendo una ejecución concurrente de instrucciones con riesgos de datos y de control. En particular, un cauce segmentado de 5 etapas.
Importante recordar que la segmentación pretende reducir el T_{ciclo} de la fórmula T_{CPU}.
Además, el incremento de prestaciones máximo o ideal es $S_{máx} = nºetapas = 5$, y el número de ciclos por instrucción (CPI) ideal es de 1c/i.

Sin embargo, en este código se presentan dos dependencias RAW y una de control:

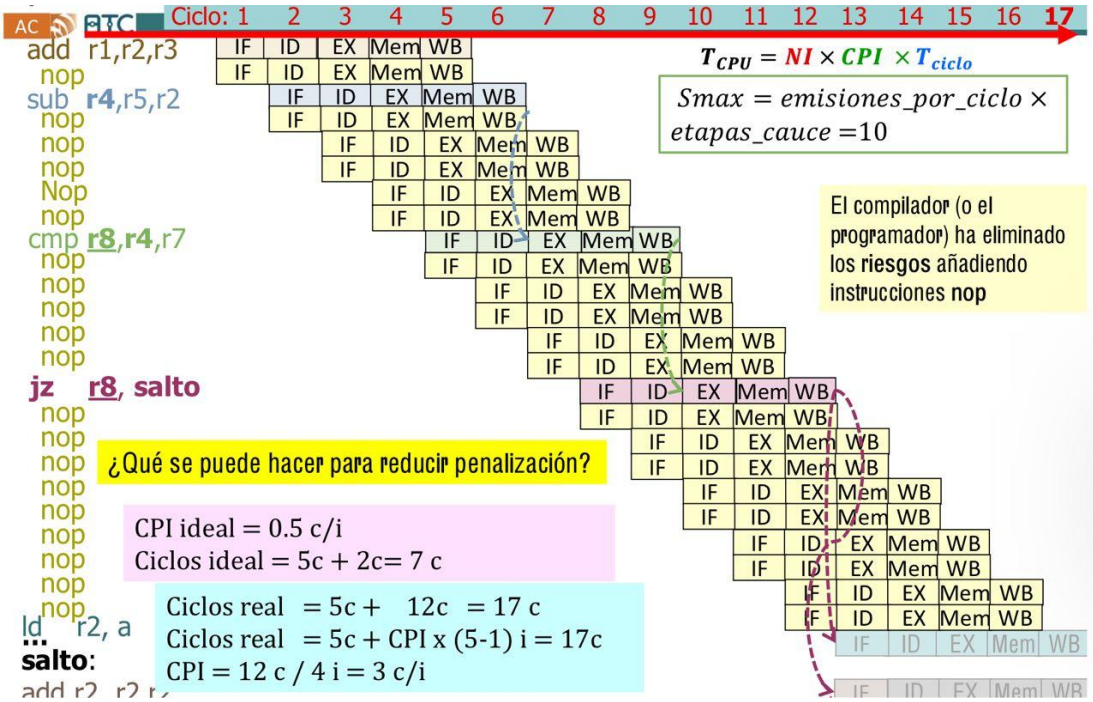
- RAW: se pretende leer r4 en la tercera instrucción. Sin embargo, todavía no se habría escrito el resultado de la instrucción anterior que cambia r4. Habría que conseguir que la segunda instrucción termine la etapa de WB antes de que la siguiente empezase a ejecutar la instrucción.
- RAW: se pretende leer r8 en la cuarta instrucción cuando la tercera instrucción no ha guardado el resultado de su operación en el registro.
- De control: suponiendo que r8==0, habría que saltar a la etiqueta salto.
  Sin embargo, esto no va a ocurrir hasta que la cuarta instrucción termine de procesarse y se haya actualizado el registro contador de programa (PC) en la etapa de WB.
  Es decir, se van a estar procesando instrucciones que NO se deberían estar procesando hasta que se actualice el PC. Además, podría ocurrir que alguna de estas cuatro instrucciones modificara un registro que NO se debería modificar.



Estos riesgos podrían solucionarse creando "no operaciones", es decir, instrucciones que no modifican ningún registro y se dedican a **perder tiempo**.
Se introducen 2 instrucciones entre la segunda y tercera instrucción, para que le dé tiempo a la segunda a **escribir** en r4 antes de que la segunda **lea** r4. Igual se ha hecho con la tercera y cuarta instrucción para solucionar la dependencia RAW. E igual se ha hecho con la cuarta instrucción que es de salto.
Problema: se impide llegar al CPI ideal (1 c/i), y pasa a ser a 3c/i. En vez de ejecutarse el código en 9 ciclos, lo hace en 17.



Así sería el ejemplo suponiendo que hay 2 cauces, es decir, en cada etapa se procesan dos instrucciones.
El CPI ideal debería es, en este caso, la mitad que el CPI ideal del caso anterior. Sin embargo, el CPI apenas decrece con respecto al CPI del caso anterior.

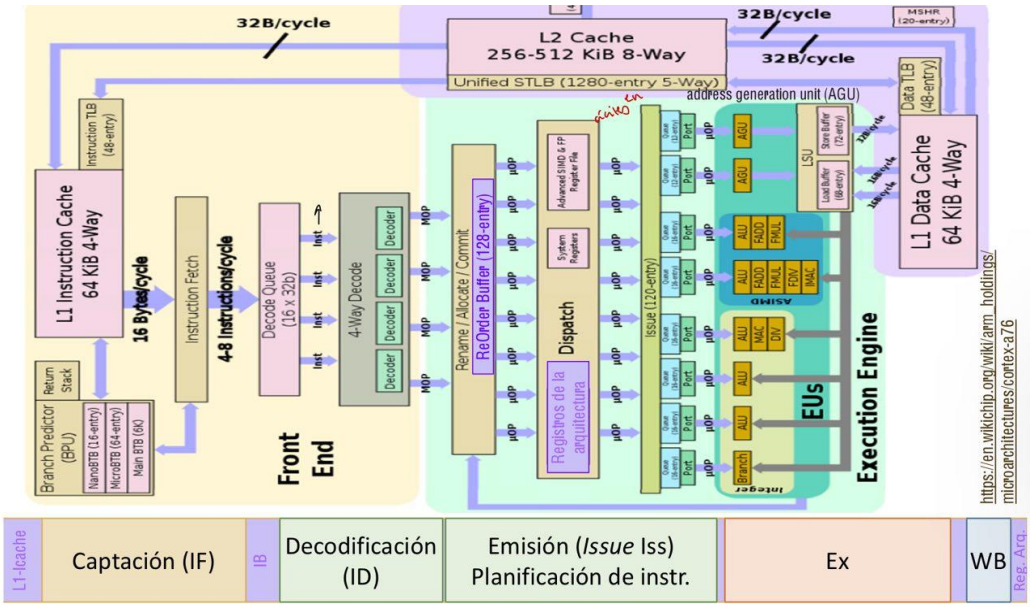


En este caso añadimos un riesgo estructural ya que suponemos que las operaciones de suma y resta utilizan la misma unidad funcional al ejecutarse. Es decir, que las operaciones primera y segunda no se pueden ejecutar en el mismo ciclo porque sólo se dispone de una unidad funcional de suma/resta. Por tanto, habría que añadir otra instrucción nop.
Esto no ocurriría si hubiese dos unidades funcionales de suma/resta.
Tenemos entonces que el CPI ideal es de 0.5 instrucciones por ciclo. Sin embargo, el CPI real es de 3 c/i.

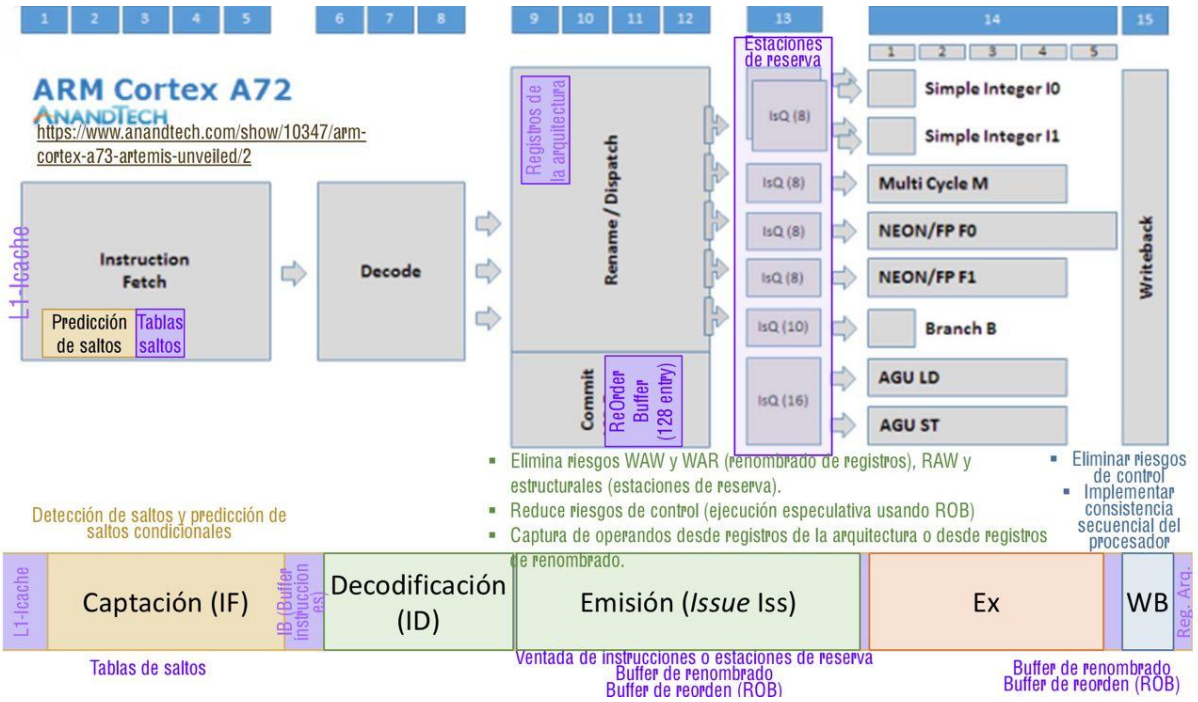
Si nos fijamos, no hay riesgos WAW porque se escriben en los registros en orden, ya que todas las instrucciones tardan lo mismo en procesarse. Tampoco hay riesgos WAR.

Apartado 1: Microarquitectura de núcleos ILP superescalar

Cauce superescalar



Ejemplo de Cauce ARM cortex A76.
Tenemos en la parte inferior de la imagen la barra que indica qué elementos se utilizan en cada etapa.
En la etapa de captación destacamos la tabla de saltos (tabla inferior izquierda) que se utiliza a la hora de detectar los saltos del código y hacer predicciones.
En la etapa de decodificación donde se decodifican las instrucciones en orden.
En la etapa de emisión es donde las instrucciones se dividen en microinstrucciones más fáciles de procesar.
En la etapa de ejecución se destacan dos unidades AGU utilizadas para calcular las direcciones a las que se van a acceder de memoria.
Si estamos en un multiprocesador, en el buffer de escritura se almacenan las escrituras que se van a llevar al sistema de memoria. El buffer de lectura puede leer un dato que esté en el buffer de escritura y que todavía no se ha actualizado en el sistema de memoria.



Ejemplo de cauce ARM cortex A72.
En violeta oscuro aparecen los almacenamientos que se van a usar en la planificación dinámica de las instrucciones, propia de los superescalares.
Podemos ver que la emisión se divide en dos subetapas: emisión a las estaciones de reserva (**dispatch**) y emisión a las unidades funcionales (**issue**).
En la subetapa de dispatch vemos el buffer de reorden, donde se encuentra el **buffer de renombrado**, que se encarga de e**liminar los riesgos WAW y WAR** que pueda haber.
Las **estaciones de reserva** son colas en las que se ponen las instrucciones que esperan a entrar a las unidades funcionales. Puede haber una estación por unidad funcional, una estación compartida por varias unidades o, incluso, una estación única para todas las unidades (en este caso se habla de la **ventana de instrucciones**).
Con las estaciones de reserva se consigue **eliminar los riesgos RAW y estructurales**.

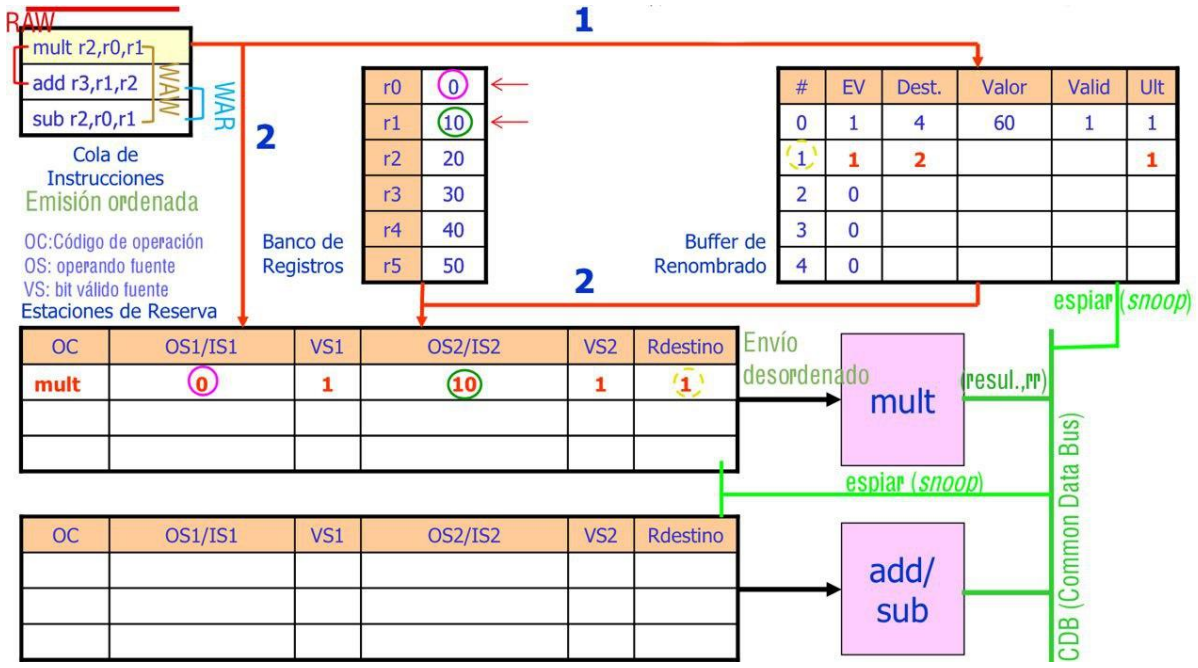
<u>Emisión (Algoritmo de Tomaculo)</u>

- Renombramiento de Registros
  Técnica para evitar el efecto de las dependencias WAR y WAW

Se pretende asignar a cada registro en el que se escribe el siguiente registro de renombramiento no utilizado. Por ejemplo, se escribe en la primera instrucción en r3

y los registros de renombramiento rr1 y rr2 ya están utilizados, así que se escribe r3 en rr3.
En la siguiente instrucción se escribe en r4, así que se le asigna el siguiente registro de renombramiento no utilizado (rr4).
En la siguiente instrucción se escribe en r3 otra vez, pero aun así, se asigna al siguiente registro de renombramiento rr5.

Veamos un ejemplo de la etapa de emisión completa, es decir, utilizando el buffer de renombramiento (para renombrar los registros) y los propios registros de la arquitectura.



Vamos a aplicar el algoritmo de Tomasulo para eliminar los riesgos de datos.
Entre la primera y la segunda instrucción hay una dependencia RAW. Entre la primera instrucción y la tercera hay una dependencia WAW. Ambas dependencias son causadas por el registro r2.
En la parte superior derecha tenemos la tabla de los registros de renombrado:

- #: número de la entrada (número de registro de renombramiento)
- EV: indica si el registro de renombramiento está ocupado (ya se ha asignado)
- Dest: indica qué registro se está renombrando
- Valor: guarda el resultado de la ejecución y se pone Valid a 1
- Valid: estará a 1 mientras no se termine el cálculo y mientras no se realice WB
- Ult: indica si el registro Dest se ha asignado por última vez a este registro de renombrado (si el registro tiene la última actualización de Dest)
    - Si un registro se ha renombrado varias veces es porque interviene una dependencia WAW

En la parte inferior vemos las tablas de estaciones de reserva. Suponemos que hay una estación por cada unidad funcional:

- OC: código de la operación

- OS1/IS1: valor del primer operando (se mira el valor en la tabla de registros de renombramiento y, si no está, se mira en el banco de registros de la arquitectura)
- VS1: indica si el dato está disponible
- OS2/IS2: valor del segundo operando
- VS2: indica si el dato está disponible
- Rdestino: indica el registro de renombrado donde se va a guardar el resultado

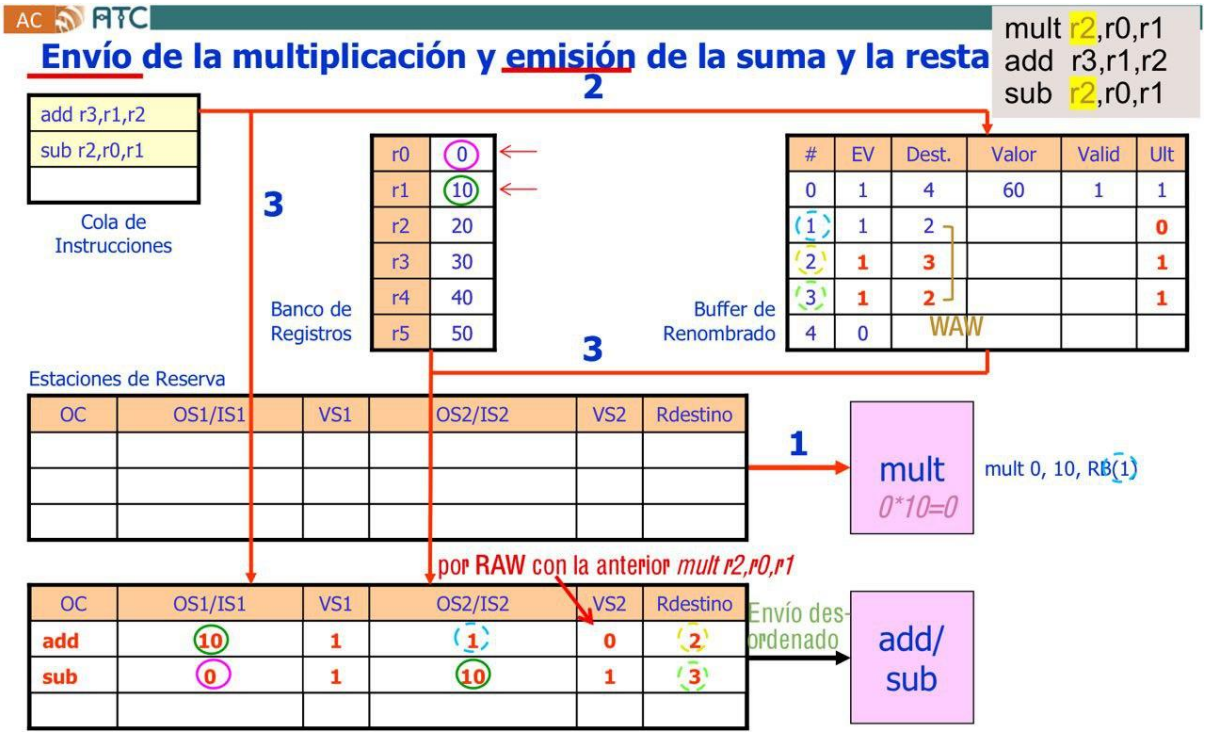
Bueno, empecemos con el ejemplo. Suponemos que en el primer ciclo sólo se emite la instrucción mult y, en el segundo ciclo, se emiten las otras dos instrucciones.
Empezamos con la primera instrucción mult. Como se escribe en el registro r2, se añade este registro al buffer de renombrado. Se inicializan los campos: como la entrada 0 ya está ocupada (EV=1), se inserta en la entrada 1 y se pone EV=1. Dest es r2 porque es el registro que se va a renombrar. Y se pone Ult=1 porque es la última asignación del registro r2.

Además, se añade esta instrucción a la estación de reserva de la unidad funcional de mult. Rdestino es 1 porque r2 se ha asignado al registro de renombrado rr1. Cojamos los operandos:

1. El primer operando (r0): miramos si en el buffer de renombrado aparece 0 en la columna de Dest. Como no es así, leemos del banco de registros el valor de r0. El valor de r0 es 0 (OS1=1) y es válido porque está sacado del banco de registros (VS1=1)
2. El segundo operando (r1): miramos si en el buffer de renombrado aparece en la columna Dest el número 1. Como no es así, se busca en el banco de registros el valor de r1 (OS2=10). Este valor es válido porque se saca del banco.

En el siguiente ciclo, esta instrucción, como tiene los operandos disponibles, se va a enviar a ejecución. Se puede enviar una instrucción de la estación de reserva a ejecución si tiene los operandos disponibles y si la unidad está también disponible. Así se eliminan problemas por dependencias RAW y dependencias estructurales.

En este nuevo ciclo, se van a emitir las dos instrucciones siguientes.



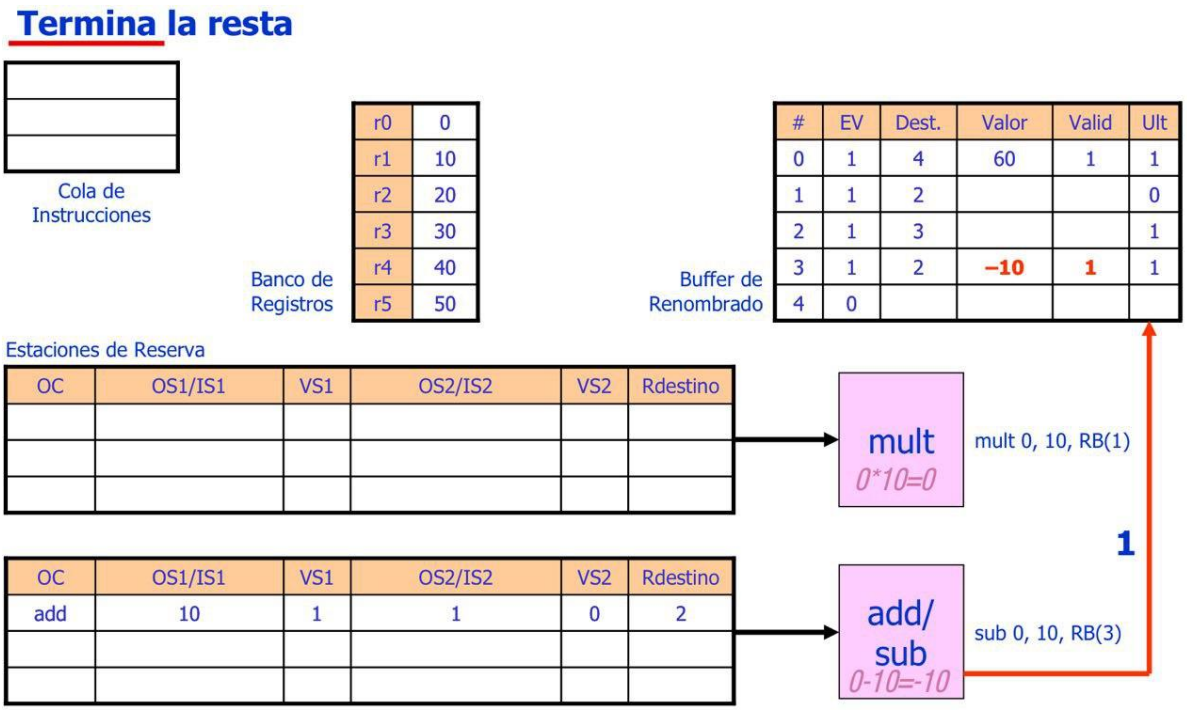
Se asignan los registros donde se escriben a un registro de renombrado. La primera entrada que hay libre es la de rr2, a la que se le asigna el registro r3 que es el que está antes en el código del programa (#2 se ocupa (EV=1) con el registro r3 (Dest=3) y éste es su último renombramiento (Ult=1)).
Al siguiente registro de renombrado rr3 se le asigna el registro r2 (#3 se ocupa (EV=1) con el registro r2 (Dest=2) y ésta es la última vez que se renombra dicho registro (Ult=1)).
El bit Ult de la entrada de #1 se pone a 0 porque ya no es el último renombrado de r2. Esto quiere decir que había una dependencia **WAW**.

También se utiliza la estación de reserva de la unidad funciona de add/sub, ya que ambas instrucciones son de este tipo. Se asignan en el orden del programa.
En la primera entrada de la estación de reserva:

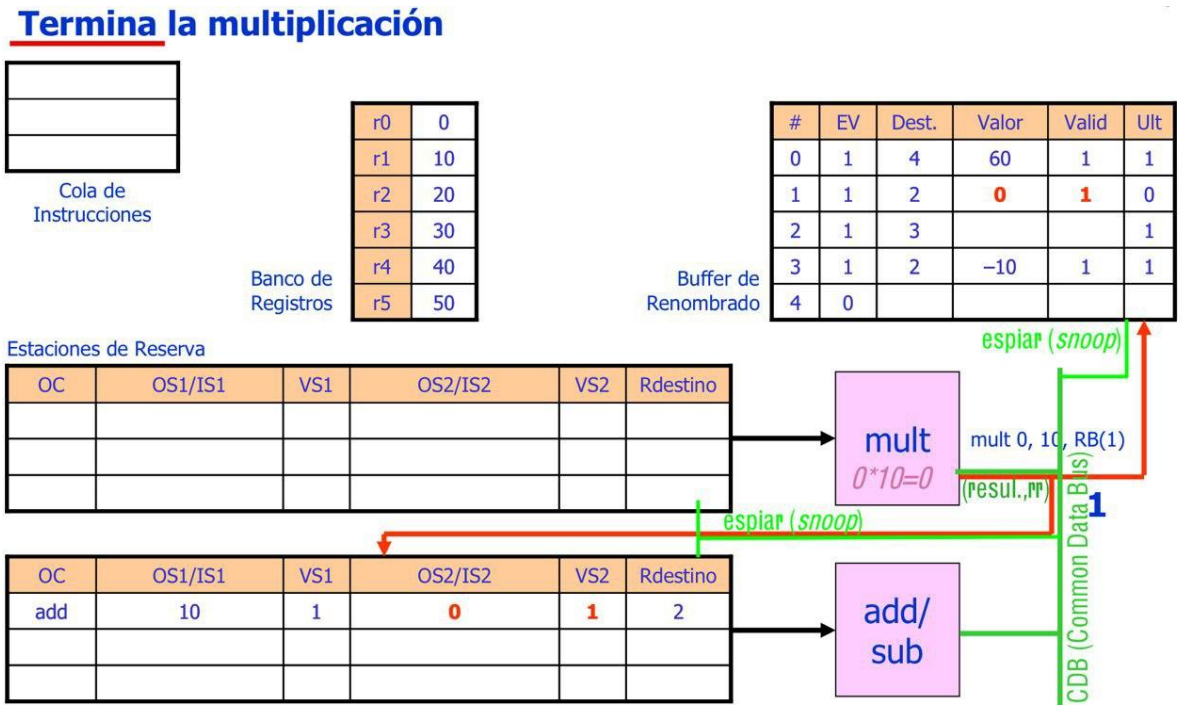
1. OC=add
2. Rdestino=2 porque dicha instrucción se ha asignado al registro de renombrado rr2
3. OS1=10: el valor de r1 se busca en la tabla de renombrados. Como no aparece r1 en ninguna casilla de la columna Dest, se pasa a buscar al banco de registros.
4. VS1=1 porque el valor se ha encontrado en el banco
5. OS2: el registro r2 está renombrado, así que se busca si está ya calculado el valor de esa operación (columna valor de #1). Como no está calculado, ponemos OS2=1 porque rr1 es el registro de renombrado donde se va a guardar el valor de la operación cuando haya terminado su ejecución.
6. VS2=0 porque OS2 no tiene un valor válido. Indica que hay un **RAW**

La siguiente instrucción no opera con el valor de ningún registro renombrado, así que se cogen ambos valores del banco de registros de la arquitectura, que son válidos.

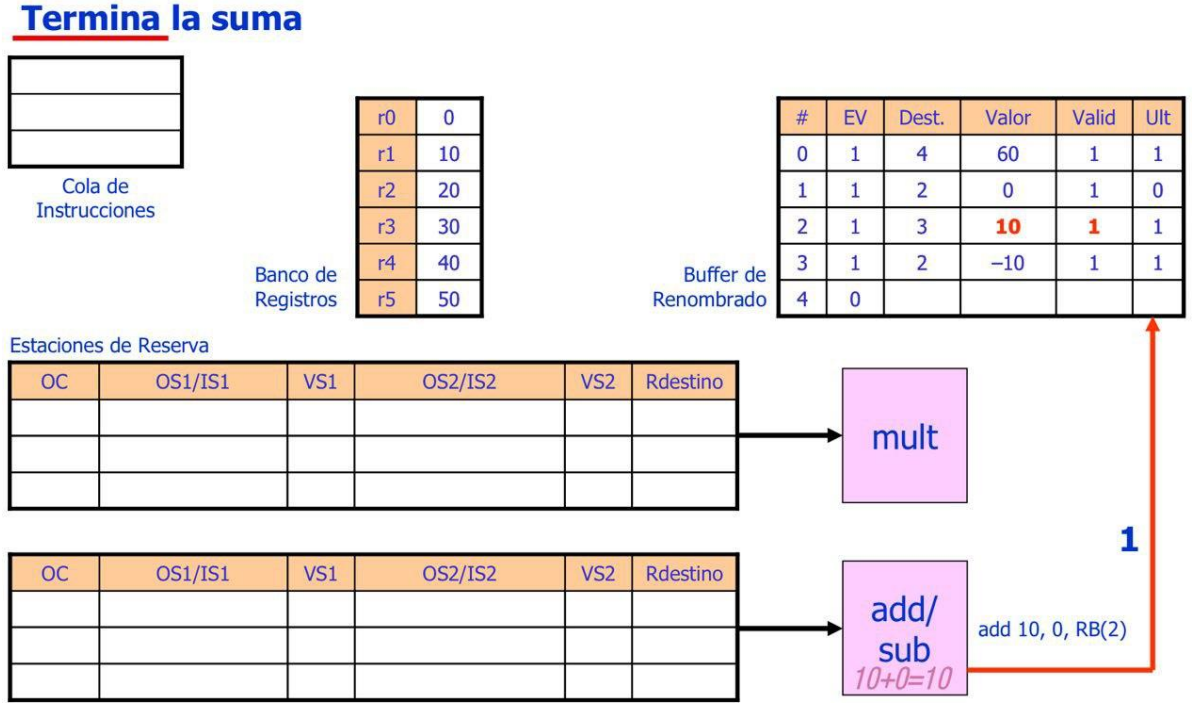
Como el envío a ejecución es desordenado, se puede mandar la segunda operación a ejecutar aunque la primera no se mande porque no tiene los operandos válidos.



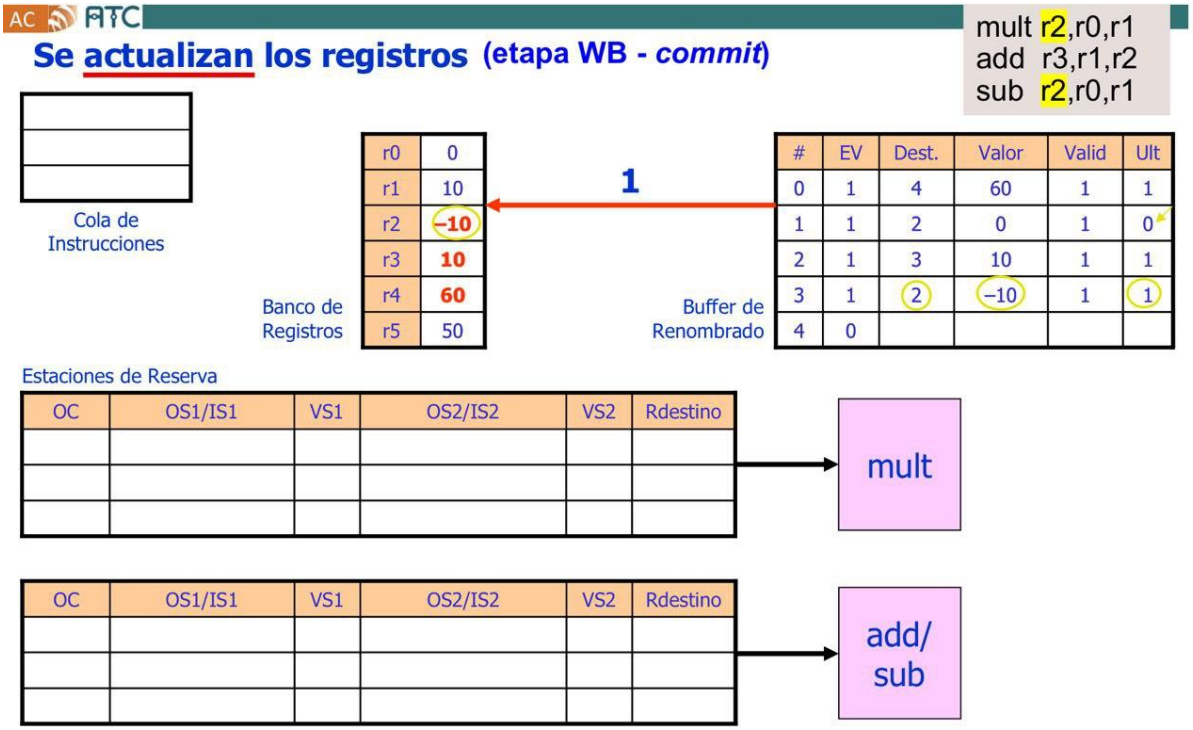
Antes de que acabe la instrucción de multiplicación va a terminar la instrucción de resta, ya que la resta supone menos tiempo (suele ocurrir en los núcleos actuales).
Cuando termine la ejecución de la resta, se colocará el resultado (-10) en la casilla correspondiente del buffer de renombrado. Pone el valor de válido para indicar que ya sí está el valor calculado.
Para pasar el valor y el la entrada del buffer de renombrado donde se debe colocar se utiliza un bus compartido por el buffer de renombrado, las estaciones y las unidades. Se pasa al bus la orden RB(3) para que se sepa que es en el registro de renombrado rr3.



A continuación, cuando termine la multiplicación, colocará el resultado de la multiplicación (0) en el bus y se almacena en la entrada #1 como indica RB(1) y pondrá Valid a 1. Además, la estación de reserva con una entrada que tiene bit de operando inválido, al ver en el bus compartido que se va a escribir en rr1, recoge del bus el resultado. Ya tiene OS2=0 y el bit VS2=1.
Entonces, en el siguiente ciclo se manda a ejecución la instrucción de suma.



El resultado de la suma es 10, lo manda al bus junto a RB(2). El buffer de renombrado actualizará sus datos.



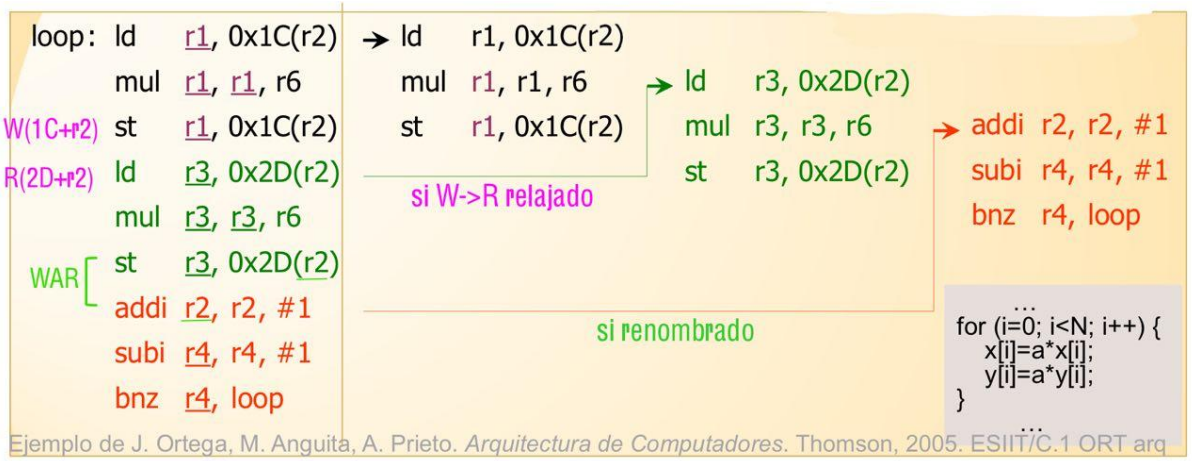
Cuando las instrucciones pasen por la última etapa de WB, el banco de registros de la arquitectura se actualizará.
Hay cuatro renombrados pero 3 registros Dest, del registro que está repetido, se cogerá el valor del segundo renombrado que tiene Ult=1.

Por tanto, se ha evitado el WAW. A pesar que la primera instrucción que escribe en r2 tarda más en ejecutarse que la segunda que lo hace, las siguientes instrucciones van a poder encontrar en r2 el valor de la resta.

Consistencia del procesador y buffer de orden

Como hemos visto, las instrucciones pueden terminar la ejecución en un orden distinto del orden del programa. A pesar de ello, terminan de procesarse en la última etapa en el orden del programa, es decir, modifican los registros en el orden en el que se han captado las instrucciones del código. Se garantiza así, como resultado final, el resultado que se obtendría si las instrucciones se hubiesen procesado en orden secuencial.
Para hacerlo, se utiliza un **buffer de reordenamiento (ROB) FIFO**. Las instrucciones cuando se emiten, se introducen en el buffer en el orden del programa. Y, siguiendo ese orden, se van retirando del buffer de reorden en la etapa de WB cuando finalizan su ejecución.
Si una instrucción finaliza su ejecución, pero hay otras antes en el orden del programa que han terminado de ejecutarse, no se podrá retirar del buffer de reorden. Tendrá que esperar a que se retiren las instrucciones anteriores.

En el procesador se implementa **consistencia secuencial** utilizando el ROB. Mientras que **la consistencia de memoria** es **relajada**, por tanto, los accesos de memoria pueden realizarse de forma desordenada.



La eliminación de riesgos por parte de los procesados y el hecho de que los accesos sean relajados, permite **reordenar el código** y **extraer paralelismo**. Veamos el ejemplo:

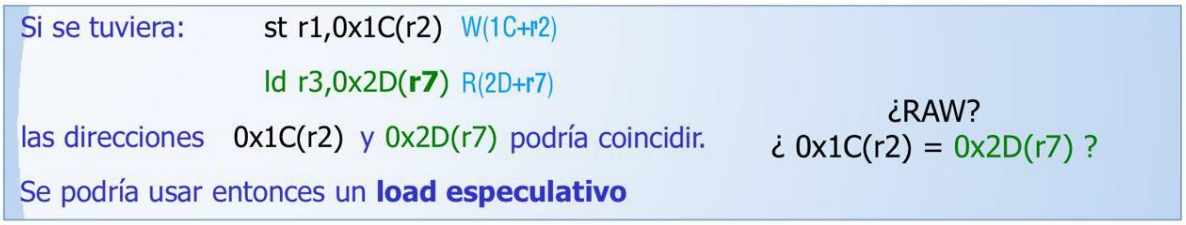
En este código, si no se relajan los accesos ni se eliminan riesgos WAR, no se puede ejecutar ninguna instrucción en paralelo con otra.

El código pretende recorrer dos vectores y multiplicar cada componente por un valor. Suponiendo que el vector x comienza en 0x1C, las primeras tres instrucciones realizan los siguiente:

1. ld carga el siguiente componente de x en r1
2. mul multiplica dicha componente por el valor a (que se encuentra en el registro r6)
3. st almacena el resultado en la misma posición de memoria

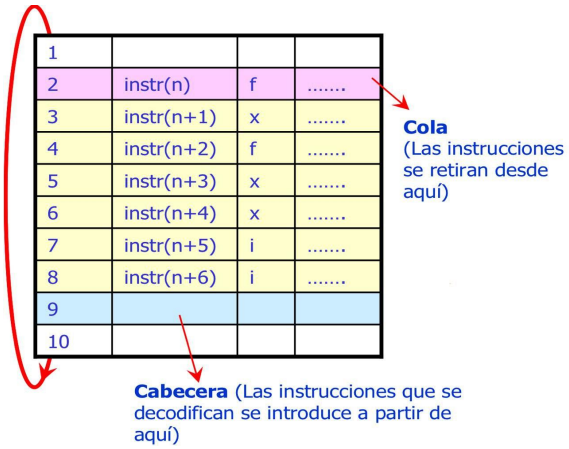
El vector y comienza en 0x2D, y se realiza exactamente lo mismo.
En cada iteración se incrementa i (r2) para pasar a la siguiente componente. Y el registro 4 sirve para comprobar que ya se han recorrido los dos vectores.

En cuanto a las dependencias, vemos que entre las instrucciones i1-i2-i3 y las instrucciones i4-i5-i6 hay dependencias RAW. Además, si dos instrucciones usasen la misma unidad funcional, tampoco podrían ejecutarse en paralelo por dependencia estructural.
Tenemos entonces que ninguna instrucción de este código podría ejecutarse en paralelo con otra si no se permite que las lecturas puedan adelantar a escrituras anteriores (W->R relajado) y si no se permite eliminar los riesgos WAR con renombrado.



Si se tuviera lo indicado, podría haber una dependencia RAW entre esas dos instrucciones. Si hay un RAW, la carga no podrá adelantar al almacenamiento. Se sabrá si hay o no un RAW cuando ambas instrucciones hayan ejecutado la primera etapa del cauce de carga y almacenamiento, etapa donde se calcula la primera dirección a la que se va a acceder. Como la mayor parte de los casos no se sabe si hay un RAW, ocurre que en realidad no lo

hay, se usa la lectura especulativa. Se permite que una lectura pueda adelantar a una escritura que hay antes en el código aunque no se tenga claro si hay un raw entre ellas. Hay que tener en cuenta, que en este adelantamiento, no solo es esa instrucción la que adelanta su ejecución, también las que hay detrás en el código, que puede usar el resultado que se ha cargado. Si cuando esta instrucción termina esta etapa se descubre que escribe en la misma dirección que la lectura especulativa lee, se marcará la instrucción de load y las que se han ejecutado detrás, como instrucciones a retirar del ROB (buffer de reorden) sin modificar los registros de la arquitectura. Es decir, que en la práctica sería como si no se hubieran ejecutado. Además se modifica el PC para volver a ejecutar estas instrucciones.

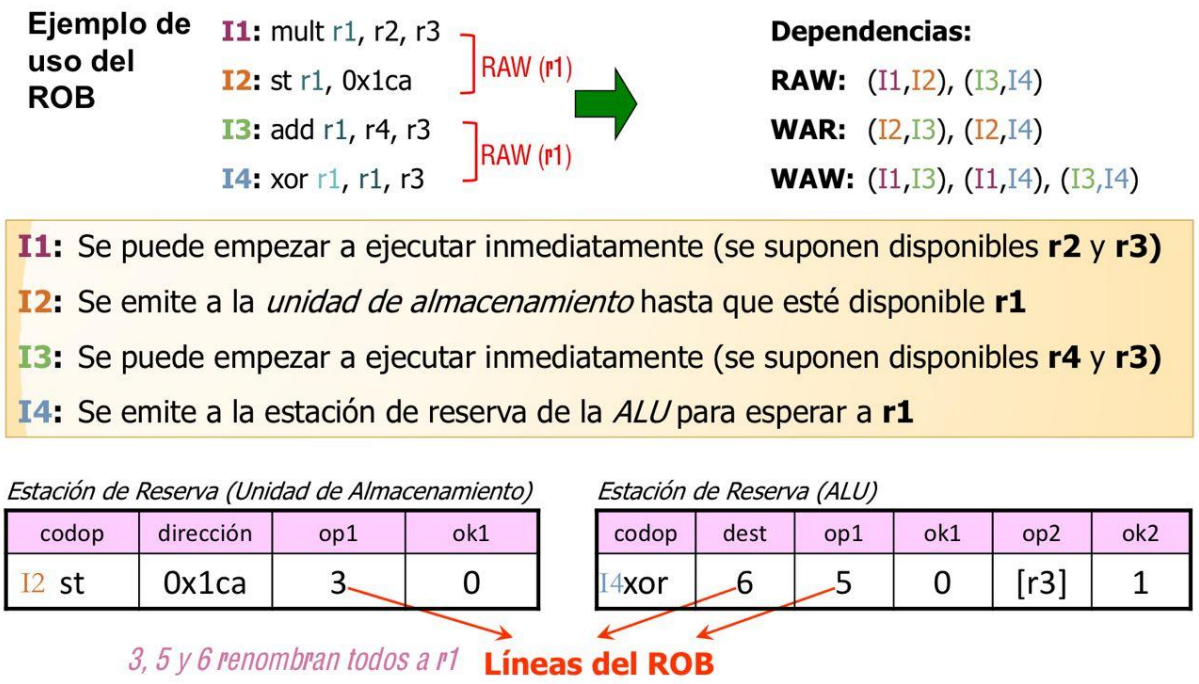


El **buffer de reorden (ROB)** es una **cola circular**.
Cuando se emiten instrucciones a ejecución, se le asignan **entradas** en el ROB. Ahí se van introduciendo en el orden del programa una después de otra.
Además, las instrucciones tienen **estados**:

- **x**: ejecutándose
- **i**: emitidas pero no ejecutándose, ya que están esperando en las estaciones de reserva a que quede libre la unidad funcional o esperando a tener todos los operandos disponibles
- **f**: estado finalizado

Una instrucción **se retira del ROB** cuando va a escribirse el resultado en los registros de la arquitectura correspondientes. En este momento, se modifica el estado del procesador, **terminando el procesamiento** de la instrucción.
Una instrucción o secuencia de instrucciones contiguas en estado finalizado en el ROB, se pueden retirar a la vez. Siempre y cuando, la primera de ellas esté la primera en la cola. Si la primera en la cola no está en estado finalizado, el resto de instrucciones detrás no pueden retirarse de la cola aunque sí que estén finalizadas.
En este ejemplo, la instrucción 4 **no** puede retirarse aunque esté en estado finalizado, porque no todas las anteriores a ella están finalizadas. Si las instrucciones i2, i3, i4 estuviesen finalizadas, podrían retirarse todas a la vez del ROB en el mismo ciclo de reloj, siempre que la arquitectura lo permita.

Además, se podría poner otra columna más para indicar cuando una instrucción ejecutada de forma **especulativa** se debe retirar **sin modificar los registros** de la arquitectura por el motivo de que se haya encontrado que la **predicción** fue **errónea** (predicción de adelantar una carga a un almacenamiento anterior).



Veamos el uso del ROB para una secuencia de instrucciones:

Entre mult y store hay una dependencia RAW, es decir, lo que almacena la instrucción segunda es el resultado de la multiplicación.También entre las dos últimas instrucciones hay una **dependencia RAW**.
Estas dos dependencias van a generar un riesgo que se va a eliminar con las **estaciones de reserva**.
La instrucción i2 estará **esperando** en su correspondiente estación de reserva mientras se ejecuta la multiplicación, esperará a tener disponible el operando que le falta.
La instrucción i4 estará **esperando** en la estación de reserva de la ALU, que es donde se va a realizar el xor, a que termine la ejecución de la suma.

Además entre las instrucciones i1-i3-i4 hay dependencias WAW y una dependencia WAR. Suponiendo que no hay renombramiento, podría ocurrir que la instrucción i4 lea r1 con el resultado de la multiplicación de i1. Esto se debe a que la **multiplicación tarda más** en ejecutarse **que la suma**.

Entre i2-i3 hay una dependencia WAR, que podría dar lugar a un riesgo si no se usa renombrado. Si la suma se emite a ejecución antes de que se ejecute el almacenamiento, podría leer en el almacenamiento de r1 el resultado de la suma, en vez del resultado de la multiplicación.
Estas dependencias **WAW** y **WAR** se eliminan con el **renombrado**.

Supongamos que ya se han emitido las tres primeras instrucciones:

Ciclo 7 Situación en este ciclo

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	I1 mult	7	r1	int_mult	-	0	x
4	I2 st	8	-	store	-	0	i
5	I3 add	9	r1	int_add	-	0	x
6	I4 xor	10	r1	int_alu	-	0	i

RAW

RAW

Ciclo 9 Terminó **add**, pero todavía no se puede retirar

#	codop	Nº Inst.	Reg.Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	**17**	**1**	**f**
6	xor	10	r1	int_alu	-	0	**x**

Ciclo 10 Terminó **xor**, pero todavía no se puede retirar

#	codop	Nº Inst.	Reg.Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	**21**	**1**	**f**

Por tanto tienen todas ellas una **entrada asignada** en el ROB en el orden del programa.
Vamos a suponer que se encuentran en el ciclo 7 de la ejecución.
La mult tiene asignado el registro rr3 y el xor tiene asignado el registro rr6.
en el buffer de reorden tiene:

- #: indica qué registro de renombrado se le ha asignado a la instrucción
- codop: código de la operación
- Nº inst: número de la instrucción que se esta ejecutando en el codigo
- RegDest: indica qué registro se ha renombrado
- Unidad: unidad funcional de la que va a hacer uso la instrucción en la etapa de ejecución
- Resultado: resultado de la operación de la instrucción
- ok: si el resultado es correcto
- marca: indica el estado en que están las instrucciones (x ejecución, i issue, f finalizado)

Como vemos en la tabla, la multiplicacion y la suma tienen los operandos validos. son el almacenamiento el xor los que tenian un raw y necesitaban el resultado de la anterior. asi que no se han mandado a ejecucion y se encuentran esperando en la estacion de reserva(estado i).

en el ciclo 9 ya ha tterminado la suma de ejecutarse, su resultado se habra almacenado desde la unidad funcional en la entrada 5 del buffer de reorden (resultado = 17), y tambien se habra almacenado en la correespondiente entrada de la estacionde reserva donde esta esperando el xor. como ésta ya tiene los operandos disponibles (op1=17 y ok1=1), pasa a ejecutarse (x). la instruccion de suma, aunque ya ha terminado, no se podra eliminar del rob, ya que no se encuentra la primera en la cola, es decir, hay instrucciones por delante que no se han eliminado del rob (mult y store).

en le siguiente ciclo tampoco ha terminado la multiplicacion. pero si ha terminado la xor, entonces se coloca en la entrada cuarta del rob el resultado=21 que es válido. ha finalizado, pero tampoco se podra retirar del rob, porque tiene instrucciones por delante que no se han retirado.

Ciclo 12 Terminaron las instr. de la cola, **mult** y **st**, y se retiran (completan el procesamiento)

#		codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	Ok	marca
3	I1	mult	7	r1	int_mult	**33**	**1**	**f**
4	I2	st	8	-	store	-	**1**	**f**
5	I3	add	9	r1	int_add	17	1	f
6	I4	xor	10	r1	int_alu	21	1	f

Ciclo 13 add y **xor** se pueden retirar ya al encontrarse en la cola (completan el procesamiento)

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

cuando termina la multiplicacion, ya tendra disponible el store el dato que le hacia flata. el resultado (71) de la multiplicacion se guarda en la entrada correspondiente del buffer de reorden (rob) y se pone ok=1; y tambien se pone el resultado en la estacion de reserva correspondiente al store, que ya tendria su operando valido. el almacenamiento ya tiene validos sus operandos y ya puede pasar al buffer de escritura del nucleo de procesamiento y se considera terminada su ejecucion, no se tiene que modificar ningun registro de la arquitectura.

ya estan terminadas entonces las instrucciones de mult y store en el ciclo 12. la multiplicacion esta la primera en la cola, asique se podra retirar. Ademas, si suponemos que la arquitectura deja retirar hasta dos instrucciones po ciclo: junto a la retirada dela multiplicacion, se retura el almacenamiento. en el ciclo 13, se podran retirar las otras dos instrucciones de suma y xor.

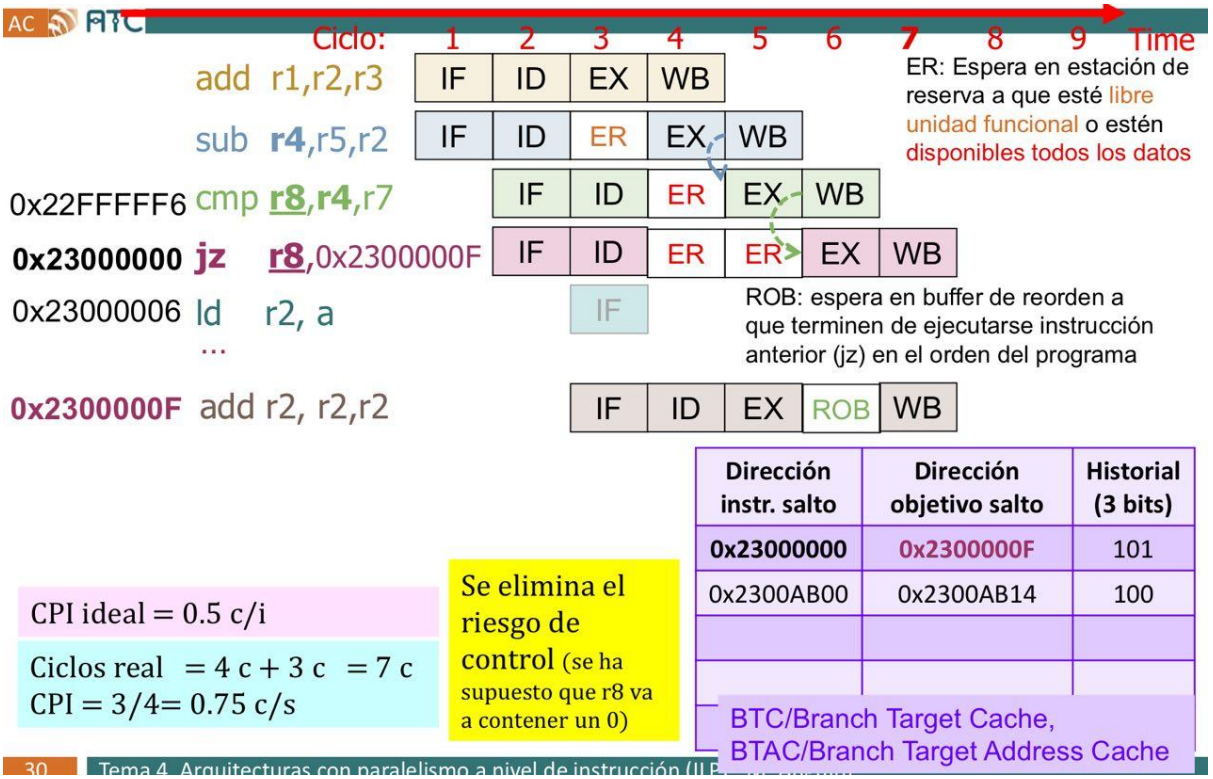
Procesamiento de saltos

para eliminar los riesgos que suponen los saltos, los de control, se usa en la etapa de captacion una tabla de saltos. Esta tabla tendrá una entrada por cada salto que se produce en el codigo. se indicara en un campo la direccion a la que saltara la instruccion. POr un lado se indica la direccion de la instruccion de salto y por otro lado, un campo que indica la direccion a la que se va a saltar.

por otro lado tambien se va a usar el buffer de reorden (rob) para la ejecucion especulativa. en la etapa de captacion se detectan ls instrucciones de salto, se ven consultando la tabla de saltos si, una instruccion que se va a captar es de salto o no, no hay que esperar a la etapa de decodificacion. si se trata de una instruccion de salto incondicional, se extrae la direccion del salto de la tabla y, sin ninguna penalizacion, se comenzara a captar las instrucciones a partri de esta direccion.

si es una instruccion de salto condicional, en la etapa de captacion se predice si se va a saltar o no , es decir, si se van a ejeuctar las instrucciones a partir del salo o a partir de la direccion de salto. teniendo en cuenta la prediccion, sin penalizacion, en ciclos, se comenzara a captar las instrucciones de una rama u otra teniendo en cuenta la prediccion.

cuando se ejcute la instruccion de salto, es decir, cuando ytermine la etapa de ejecucion, se sabra si la prediccion es correcta o no. si la prediccion no es correcta, las instrucciones que se han ejecutado teniendo en cuenta la prediccion, se mascaran en el buffer de reorden como instrucciones a retirar del buffer sin modificar los registros de la arquitectura. y para eliminar el riesgo, se actualiza el pc para que ejecute el camino correcto, eliminando entoces en la ultima etapa el riesgo que no haya podido ser eliminado con la prediccion.
si la prediccion es erronea, se han perdido ciclos de reloj.



Aqui podrian verse los ciclos que se pierden al ejecutarse las instrucciones del ejemplo visto anteriormente usando un diseño de nucleo de procesamiento con buffer de reorden que elimina las depen waw y war y con estaciones de reserva, que elimnan las dependencias estructurales si necesitan usar la misma unidad y se eliminan los riesgos raw que hay entre las instrucciones i2-i3 y las instrucciones i3-i4. las esperas en la estacion de reserva (ER) vienen determinadas por las esperas que hay que realizar para eliminar los problemas de las dependencias que acabamos de comentar. tambien con el diseño que hemos hecho, no es necesario la etapa de acceso a memoria que hay entre ejecucuion y wb, esa estapa la necesita las instrucciones de acceso a memoria como load y store. y el diseño, con las estaciones de reserva que se han ehcho, pues permte que las univas instrucciones que pasen por el acceso a memoria sean lead y store.
la primera espera en amarillo de en la estacion de reseva es pra que quede libre la unidad funcional que necesita la resta que es la misma que usa la suma. la espera en rojo en la estacion de reserva es porque esta eseradno a que la segunda instruccion termine de ejecutarse y, por tanto, pueda tener el dato que necesita.
los dos ultimos ciclos de espera de la cuarta instruccion de salto es porue esta esperando a que termine de calcularse la comparacion, terminara a terminarse la ejecucion y ya tendra disponible r8 para ser usado en la etapa de ejcucion.

para eliminar el riesgo de control se usa deteccion en la etapa dde capatcion de las instrucciones de salto y prediccion en la misma etapa. lo que se hace ne la captacion es que con la direccion que hay en el pc, se consulta la tabla de saltos, en esa tabla se idetfica los saltos por la direccion que ocupa en el programa. si esa direccion se encuentra en la tabla, se ha detectado un salto y si la instruccion es de salto, se consultaran los bits qu hay en el historial, que informan sobre lo que ocurrio en anteriores ocasiones en los que se ejeucto el programa con el mismo salto. en este caso, 101, el primer bit indica que en la ultima ejecucion de la instruccion se saltó, el segundo bit a 0 indica que en la penultima ejecucion no se salto y el tercer bit a 1 indica que en la ejecucion antepenultima se salto. como hay más unos, la prediccion es saltar. ya se tiene la prediccion y la direccion a la que se debe saltar, asi que sin ningun ciclo de penalizacion, se captara la instruccion ue hay en la direccion de salto y se seguira con las instruccionrd postriroes a ella.
de forma que la ejecucon de 5 instrucciones, las primeras cuatro y la ultima supondran 7 ciclos de reloj y un CPI de 0.75 c/i que se acerca bastante al CPI ideal de 0.5 c/i.
en esta implementacion se puede captar decodificar, ejecutar y escribir dos instrucciones al mismo tiempo . en la ultima etapa en el ciclo 7, podran retirarse del rob 2 instrucciones. la instruccion de salto que acaba de yterminar y la ulrima instruccion que estaba esperando en el roba que la instruccion que hay antes en el buffer de reorden finalicen su ejecucion.

cuando termina la ejecucion de la instruccion de salto, se va a saber si realmente r8 era 0 y habia que saltar o si, por el contrario, no habia que saltar y habia que ejecutar las instrucciones a continuacion del salto. suponemos que, en este caso, la prediccion era correcta y no habia que hcaer nada mas. se habria eliminado el riesgo de control con la predicicon. si por el contrario la preddiccion ha sido incorrecta, se marcaran la instruccion de suma y las posteriores que se hayan ejecutado como instrucciones qeue deben reitrarse del rob sin modificar los registros de la arquitectura. entonces, la instruccion de suma no va a modificar el registro r2 de la arquitectura.
se modifica tambien el pc para apuntar a la instruccipn de load (0x23000006), de forma que ya apartir del ciclo 7, se captarian las instrucciones del camino correcto. en este caso el resultado de la ejecucipn sera correcta.

en este caso hemos considerado una prediccion basada en un historial, es decir, se ha considerado una prediccion dinamica en tiempo de ejecucion.
tambien se pudede usar una prediccion fija, donde se decide que se van a realizar todos los saltos o que no se vana realizar todos los saltos.
tambien se puede usar una prediccion estatica, se usa el desplazamineto del salto. en los saltos con desplazamiento negativo (saltos hacia arriba), la prediccion dice saltar (porque probablemnete sea un bucle do-while). si el desplazamiento es positivo (saltos hacia abajo), se decide no saltar.

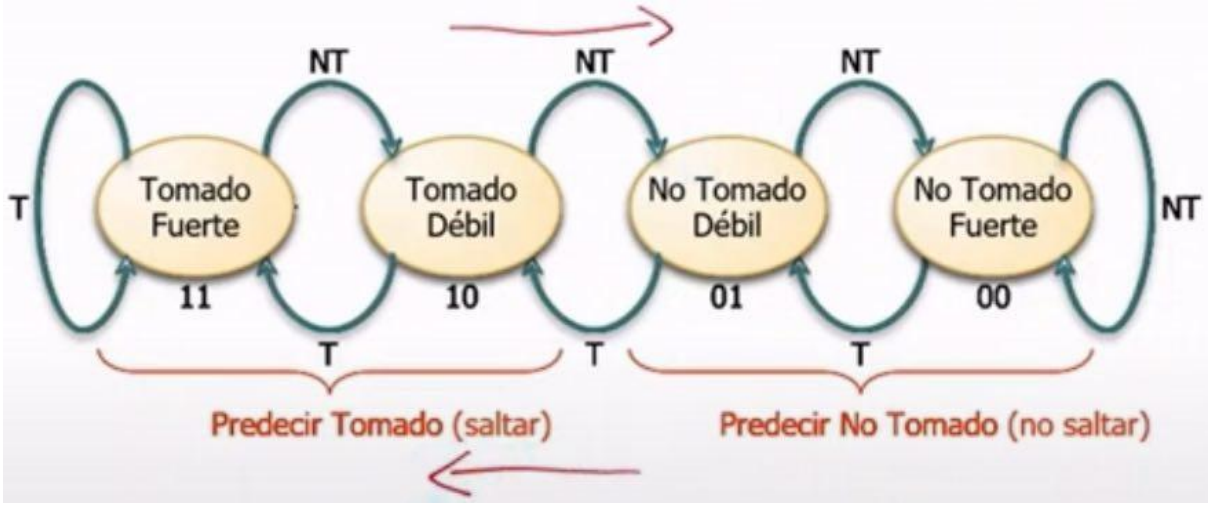
```
.L6:
  addq  $1, %rax
  cmpl  %eax, %ebp
  jle .L7
  movsd (%r12,%rax,8), %xmm0
  mulsd %xmm1, %xmm0
  addsd (%r13,%rax,8), %xmm0
  movsd %xmm0, (%r13,%rax,8)
  jmp .L6
.L7:
```

tenemos aqui ele ejmplo de un bucle donde se llegara a jmp l6 y se ejecutara otra vez el codigo y, en la ultima iteracion, sera en la instruccion jle l7 donde se saldra del bucle.
entonces, tiene sentido que, cuando llegue a jmp l6 la prediccion sea saltar porque el desplzamaniento es negativo y, cuando llegue en todas las iteraciones a jle l7 no salte porque el

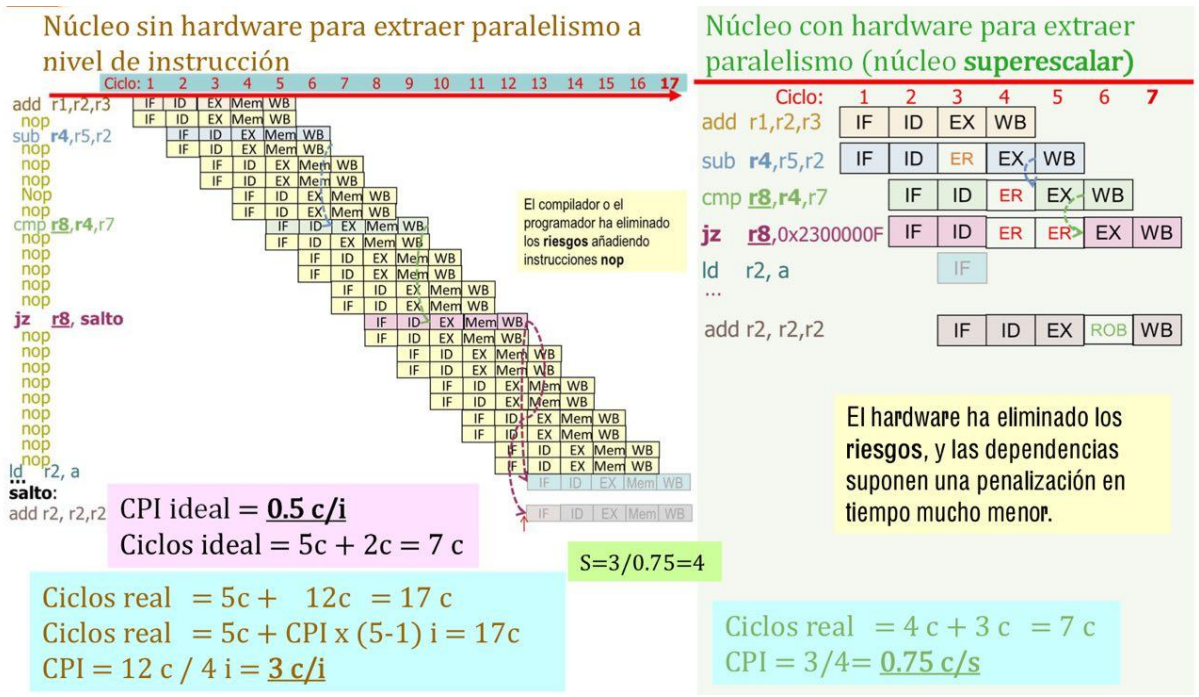
desplazamiento es positivo. asi solo se fallara una vez.

Para implementar el historial, se podria usar para cada entrada de la tabal de saltos un registro de desplazamiento con un numero de bit estables igual al numero de bits del historial (en nuestro caso 3 bits). entoces tendriamos un regstro de desplazamiento para cada una de las entradas de la tabala.
si resulta que no se salta, se meteria un 0 en el regstro de desplazaiento, se tendria 001.
si se prodeuce el salto, se tendria 111.
una mayoria de unos produce una prediccion de salto, ya que se ha saltado mas veces de las que no se ha saltado.

El esquema seria el siguiente:

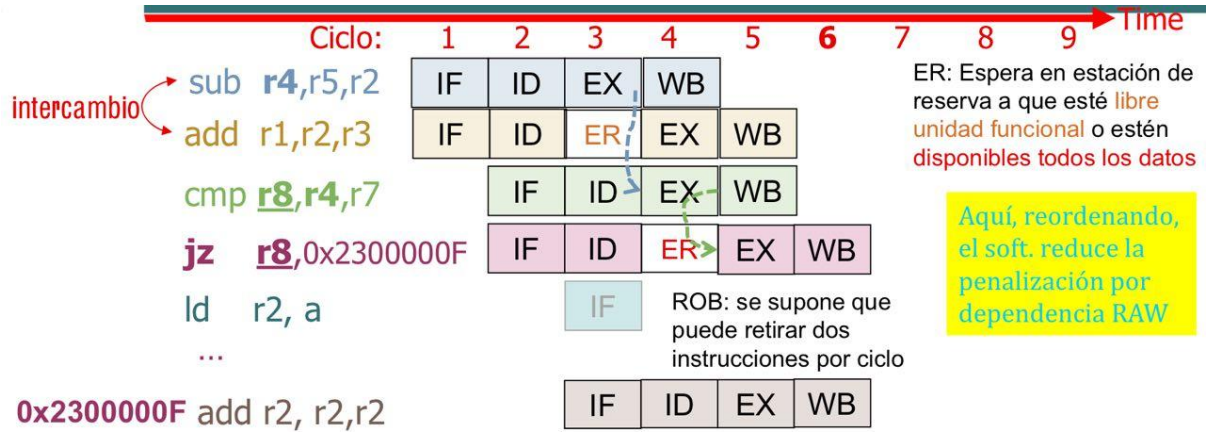


Veamos ahora un ejemplo de ganancia con la extraccion de paralelismo del hardware.



el nucleo original en el que no habia hardware para eliminar los riesgos y se utilizaba el compilador o el propio programador utilizando instrucciones de no operacion, donde el tiempo de ejecucion del trozo de codigo suponia 17 ciclos (3c/i), se ha pasado a un nucleo superescalar en el que hardware (el buffer de reorden, las estaciones de reserva, la tabal de prediccion de saltos) elimina los riegso y ademas hace que las dependencias suongan un tiempo de ejeucion mucho menor. llehando asi un tiempo de ejecucion de 7 ciclos (0.75c/i).

cabria preguntarse si se puede hacer algo mas, si el software puede ayudar al superescalar a reducir el tiempo de ejecucion y, realmente, si que puede. el software puede ayudara extraer paralelismo eliminando dependencias o reduciendo penalizacion que suponen esas dependencias.
en nuestro ehempoki de parydia, las dos primeras instruccionesno pueden ejecutarse a la vez, en el mismo ciclo, porque usan la misma unidad funcional. el restraso de esta instruccion por el riesgo estructural hace que se retrase la ejecución de la tercera instruccion, que necesita el resultado de la segunda instruccion y tenga que esperarse ne la estacion de reserva hasta que termine la ejecucion al segunda instruccion.



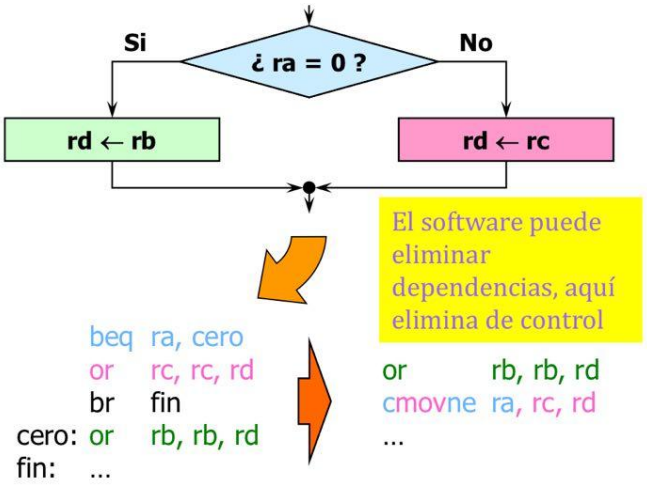
Intercambiando el orden de las dos primeras instrucciones, el software elimina la espera de la tercera instruccion en la estacion de reserva. se consiue asi la ejecucuion del programa en 6 ciclos, llegando al CPI de 0.5c/i que es el ideal.
tambien se pueden eliminar las propias dependencias en el codigo, dependencias de control, estructurales (usando alternatvas a la hora de generar el codigo ensamblador) y de datos (con renombrado).

Ejemplo: `cmovxx` de Alpha

`cmovxx ra.rq, rb.rq, rc.wq`

- **`xx`** es una condición
- **`ra.rq, rb.rq`** enteros de 64 bits en registros ra y rb
- **`rc.wq`** entero de 64 bits en rc para escritura
- El registro `ra` se comprueba en relación a la condición `xx` y si se verifica la condición `rb` se copia en `rc`.

Sparc V9, HP PA, y Pentium ofrecen también estas instrucciones.

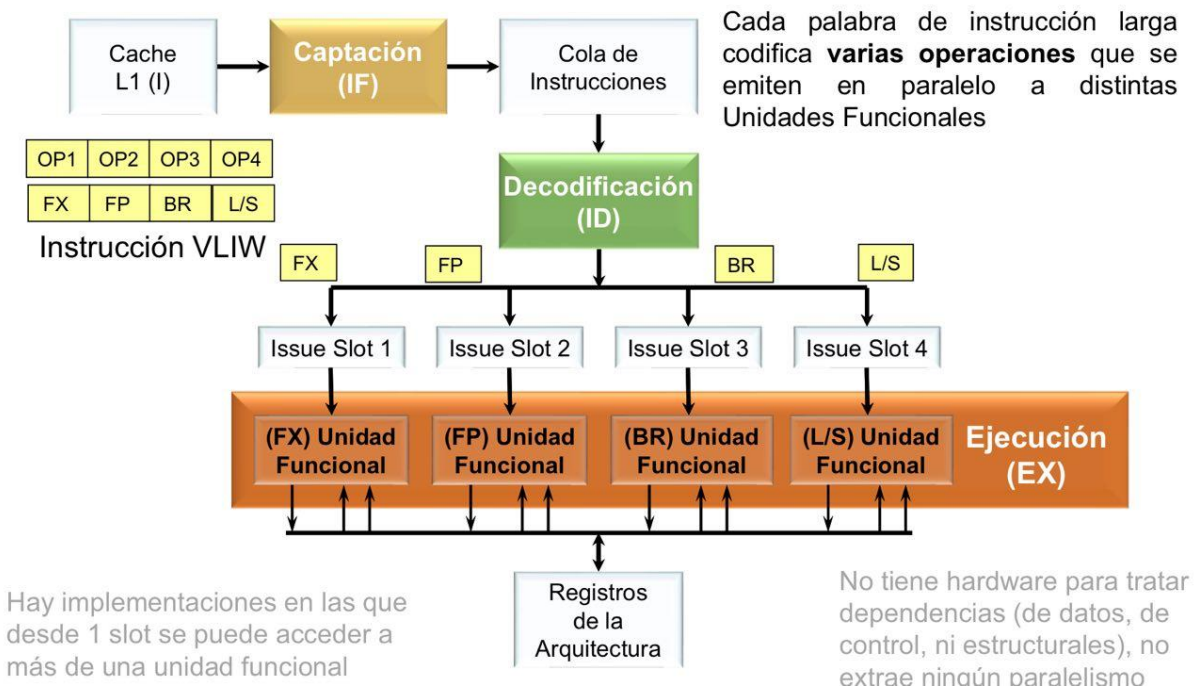


En este ejemplo, tenemos dos instrucciones de salto una condicional y otra no (beq y br). Para impkementar un if-else.
se han sustituido las primeras 4 intrucciones por las otras 2 de la derecha en las que no hay salto. se ha usado una instruccion de movimiento condicional, que mueve entre registros en funcion de si se cumple una condicion. en el bloque practicao 4 hemos visto que hay un ejemplo con un if-else en el que se genera por parte del compilador, una instruccion de este tipo. estas instrucciones son buenas para instrucciones de salto condicional no predecicble (como if-else en muchos casos).

Para reducir aun mas la penalizacionque supone por los riesgos y aprovechar en mayor medida el hardware de los superescalares, se ha recurrido a introducir hardware que permita la ejecución de instrucciones de distintos flujos (threads) en un mismo nucleo. bien se pueden ejeuctar las instruccion concurrentemente o en paralelo. este tipo de nucleos aparecieron en los 2000.

Apartado 2: Microarquitectura ILP VLIW

En este caso, no hay hardware para eliminar riesgos provocadospor dependencias. debe ser el propio software el que se encargue de eliminar los riesgos y de reducir la penalizacion que supoenn las dependencias. en este caso, el vliw la planificacion de las instrucciones se vana emitir en paralelo a las distintas unidades funcionales en un ciclo, la hace el softwre.



Para hacerlo, el compilador tiene que poner junto en memoria en una palabra de instrucciones larga, aquellas instrucciones que se van emitir en paralelo a las unidades funcionales. si hay 4 unidades funcionales, como en este ejemplo, una para enteros, una para punto flotante, otra para salto y otra para carga y almacenamiento. podrá poner junta en una palabar de instr larga una instruccion de cad auno de esos tipos que deben de ser independendientes (no deben tener dependencias raw).
las palabras de instrucciones largas las ha creado el compilador o el programador en ensmblador. se captan estas instrucciones y se decodifican y a continuacion se emiten a ejecucion ls intrsucciones que hay dentro de una palabra.

los problemas por riesgos los tiene que eliminar el compilador introduciendo instrucciones de no operacion. cuando no pueda incluir isntrucciones independientes de los 4 tipos, en alguno de los slot (cuadrado amarillo) debera poner una instruccion de no operacion. y tambien que poner instrucciones de no operacion en palabras de isntrucciones en todos los slot de la palabra de instrucciones con el fin de eliminar dependencias entre palabras de instrucciones largas distitas.
en el caso de los vliw, necesariamente lso compiladores deben de aplicar tecnicas a nivel software para extraer paralelismo disminuyendo los riesgos y disminuyendo la penalizacion. tecnicas como el desenrrollado de bucles