Arquitectura de Computadores

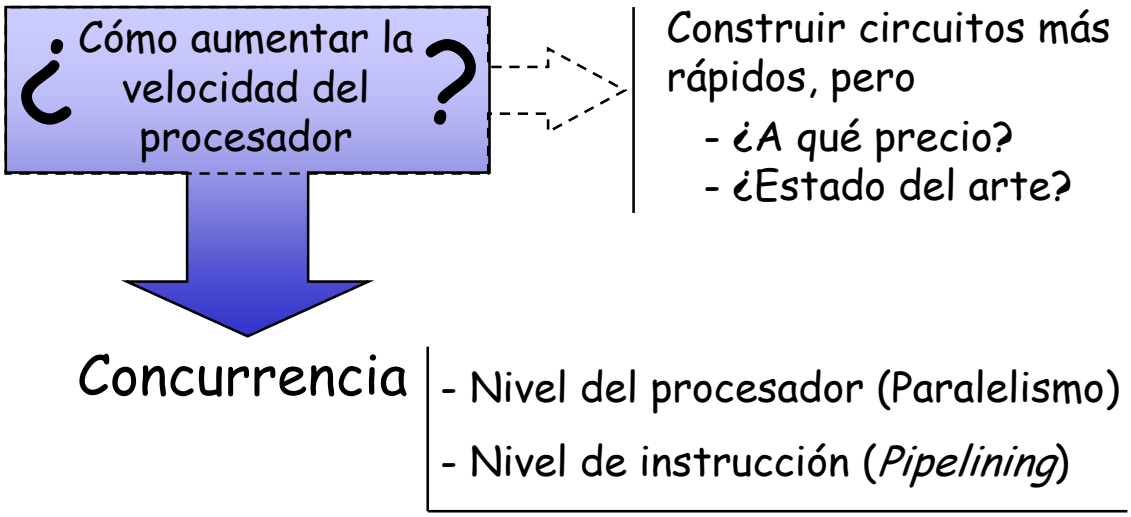
6. CPU Segmentada (*Pipeline*)

1. Conceptos Básicos
2. Causas de Ralentización

En los dos capítulos siguientes vamos a tratar dos formas de optimizar substancialmente la arquitectura convencional que hemos estudiado hasta ahora. Esta mejora se centra en la CPU y en la memoria.

En primer lugar empezaremos por mejorar las prestaciones de la CPU, que, en cuanto a velocidad, se pueden mejorar mediante los procesadores segmentados (o en *pipeline*) los cuales incorporan una técnica para acelerar el ritmo de ejecución de instrucciones.

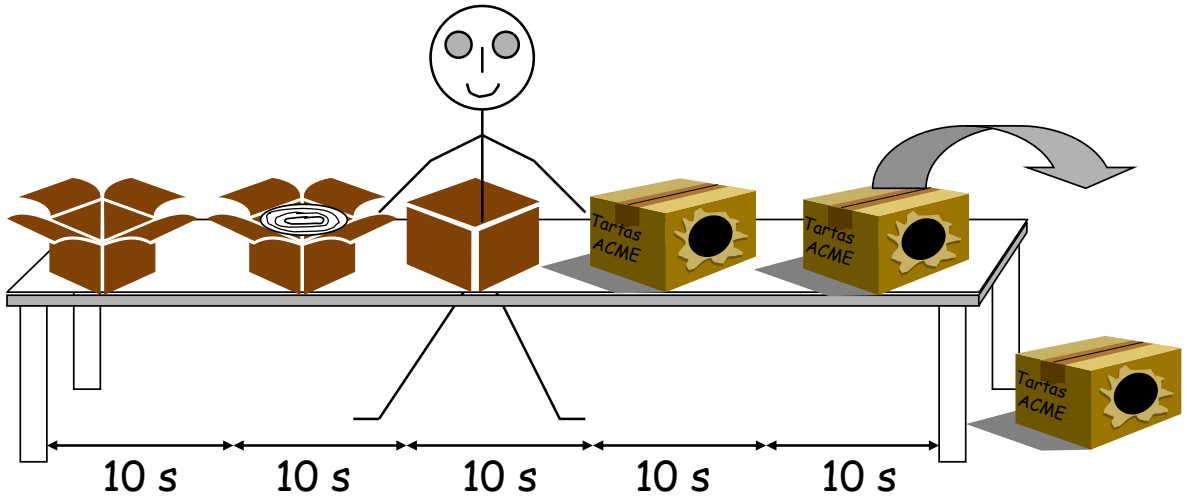
Como veremos, hay diversos factores que pueden impedir un aprovechamiento óptimo del concepto de *pipeline* en la CPU. Trataremos estos factores y las técnicas para evitarlos en la medida de lo posible.



La velocidad de ejecución de los programas depende de diversos factores. Una forma de aumentar esta velocidad es hacer más rápidos los circuitos con los que se construyen los procesadores y la memoria principal. No obstante, se debe considerar el coste que supone una mejora y que el límite a esta velocidad lo impone el estado del arte actual de la tecnología.

Otra posibilidad es organizar el hardware para poder ejecutar más de una instrucción simultáneamente: **concurrencia**. La concurrencia se puede obtener en dos niveles: al nivel del procesador y al nivel de la instrucción. La concurrencia al nivel de la CPU se obtiene disponiendo de múltiples procesadores ejecutando simultáneamente varias instrucciones. Obtener concurrencia a nivel de la instrucción significa poder ejecutar varias instrucciones simultáneamente con una única CPU. Este último tipo de paralelismo se denomina segmentación o encadenamiento, aunque suele ser más conocido por su denominación en inglés: ***pipelining***.

Las arquitecturas con múltiples procesadores suelen utilizarse en máquinas de muy altas prestaciones (y muy alto precio). Sin embargo, con arquitecturas segmentadas se consigue una muy buena mejora del rendimiento y a un coste asequible. Por esto, es normal que todos los microprocesadores actuales de propósito general incorporen el *pipelining*. Ya que es muy común su utilización en los actuales procesadores, vamos a abordar aquí esta técnica del *pipelining*, mientras que las arquitecturas multiprocesador las dejaremos para asignaturas o textos de arquitecturas paralelas o avanzadas.



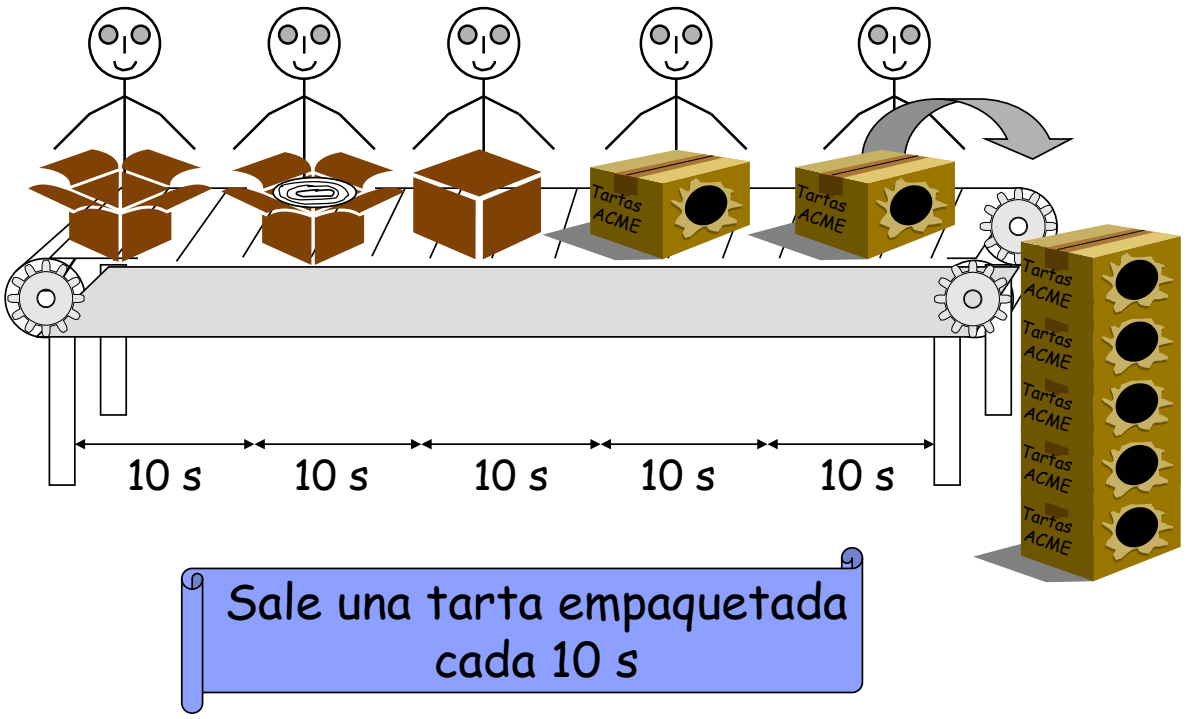
Sale una tarta empaquetada cada 50 s

El proceso en *pipeline* o encadenado es similar al utilizado en cualquier cadena de montaje, y el nombre *pipeline* (tubería) se debe al hecho de que como en una tubería, en la entrada se aceptan nuevos elementos (instrucciones) antes de que los previamente aceptados salgan por la salida.

Empecemos con el ejemplo de una cadena de montaje. Supongamos una gran pastelería en la que las tartas primero se hacen en el horno y después se empaquetan para la venta. El proceso de empaquetar una tarta consiste en:

1. Poner una caja vacía en la mesa.
2. Meter una tarta en la caja.
3. Cerrar y precintar la caja.
4. Poner una etiqueta en la caja.
5. Llevar la caja a un gran contenedor.

Si cada una de estas operaciones la realiza un operario en 10 segundos, parece claro que se tarda 50 s en empaquetar una tarta y, por lo tanto, en empaquetar 10 tartas se tardaría 500 s.

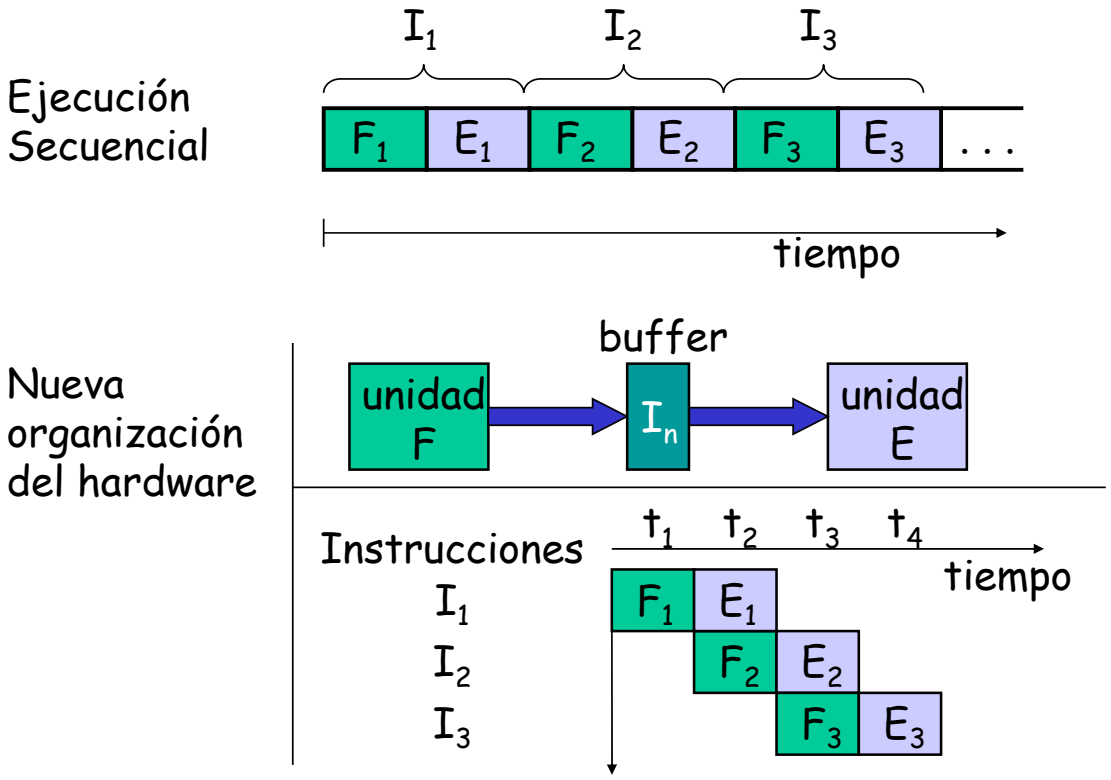


Sale una tarta empaquetada cada 10 s

Ahora supongamos que se dispone de una cadena de empaquetado de tartas con una cinta transportadora sobre la que trabajan cinco operarios especializados en tareas distintas. El primer operario pone la caja-1 en la cinta transportadora, y ésta avanza hasta que la caja-1 está donde el segundo operario, que introduce una tarta dentro de la caja-1, al mismo tiempo que el primer operario pone otra caja-2 en la cinta. La caja-1 sigue avanzando hasta el tercer operario, que la cierra y la precinta, al mismo tiempo que el segundo operario mete otra tarta en la caja-2 y el primer operario pone otra caja-3 en la cinta. La caja-1 sigue su camino en la cinta pasando por el cuarto operario, que pone una etiqueta, hasta llegar al quinto operario, que la retira de la cinta.

En el momento que el quinto operario retira la caja de la cinta, hay cuatro cajas más en la cinta. Si cada una de estas fases de empaquetado se realiza en 10 s, a partir de ahora, cada 10 s saldrá una nueva tarta empaquetada, en lugar de hacerlo cada 50 s que se tardaba cuando no había cadena de empaquetado. A partir de ahora, en tener 10 tartas empaquetadas se tardará solamente 100 segundos, mientras que en el caso de cuando se tenía un solo operario se tardaba 500 segundos.

Debe quedar claro que aunque ahora sale una nueva tarta empaquetada cada 10 s, la preparación completa de cada tarta sigue requiriendo 50 s (igual que cuando había una sola persona preparando las tartas).



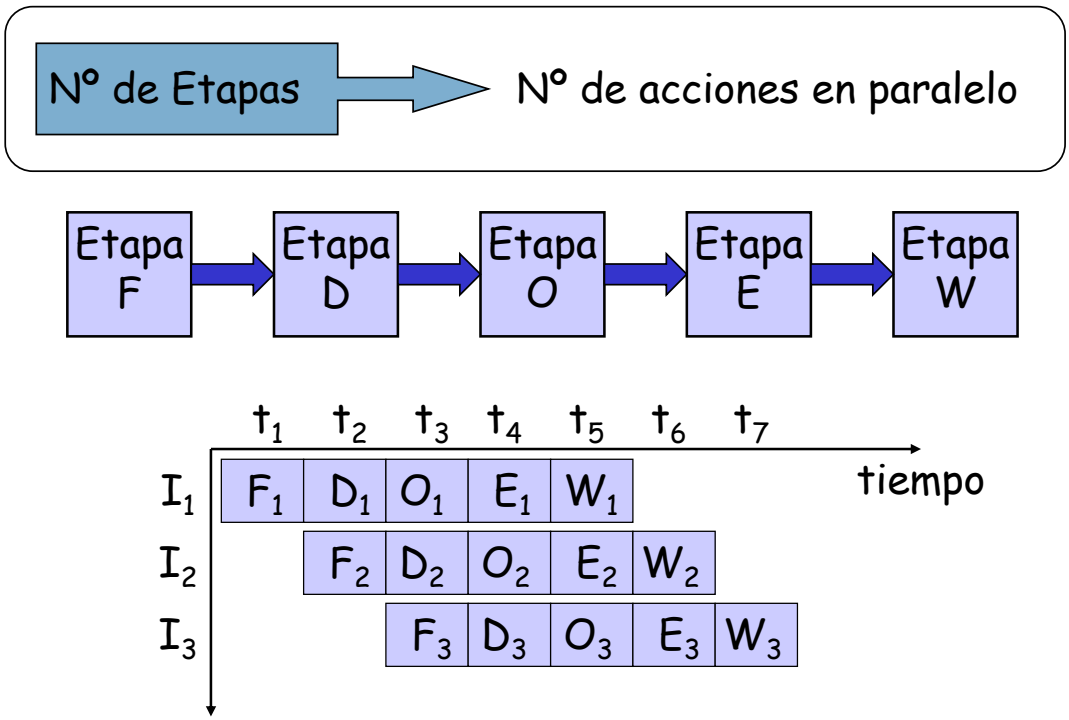
En una primera aproximación, se puede observar que para ejecutar una instrucción en la CPU se requieren 2 pasos:

1. Alimentación o extracción de la instrucción desde memoria (*fetching*).
2. Ejecución de la instrucción.

En 1959, el ordenador Stretch de IBM, teniendo en cuenta que durante la fase de ejecución hay momentos en los que no se accede a memoria principal, aprovechaba para alimentar instrucciones por adelantado y guardarlas en un *buffer de prealimentación*, todo ello en paralelo con la ejecución de la instrucción en curso, con lo que al terminar de ejecutar dicha instrucción podía cargar la siguiente instrucción directamente desde el buffer sin tener que esperar a traerla de memoria.

Esta técnica de **prealimentación** o *prefetching* puede verse como un *pipeline* de dos etapas. En la primera etapa se alimenta una instrucción de memoria y se guarda en un buffer. La segunda etapa toma una instrucción del buffer y la ejecuta. Mientras en la segunda etapa se está ejecutando una instrucción, la primera etapa aprovecha (los ciclos en los que la segunda etapa no accede a memoria) para leer la siguiente instrucción y guardarla en el buffer. Cuando la segunda etapa acabe la ejecución y vacíe el buffer de prealimentación, la primera etapa puede volver a leer una nueva instrucción de memoria.

Con estas dos etapas de alimentación y ejecución de instrucciones, parece que la velocidad de ejecución de instrucciones por segundo (rendimiento) se duplica. Y si nos fijamos en el ejemplo de la línea de empaquetado de tartas, tenemos que su velocidad de tartas empaquetadas por minuto se multiplica por cinco cuando se establece una cadena de empaquetado de cinco etapas. Esto es así, simplemente porque el número de etapas dice cuántas cosas se están haciendo simultáneamente, y claro, cuantas más mejor.

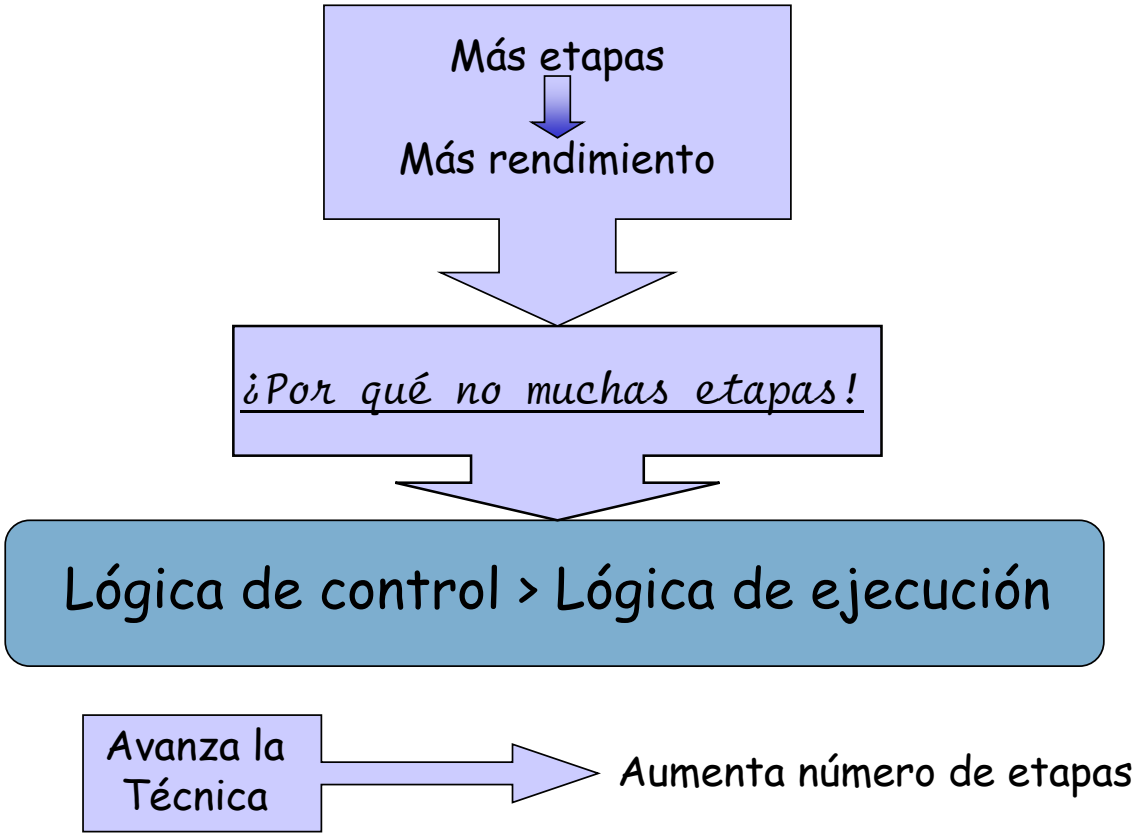


Según lo que acabamos de ver, parece que interesa dividir las fases de ejecución de las instrucciones en más etapas, para así obtener un mayor rendimiento en la ejecución. Poco después del Stretch, UNIVAC sacó el LARC, con 4 etapas. Actualmente, tenemos que el PowerPC 750 tiene 6 etapas; el Pentium de Intel consta de tres unidades de proceso en pipeline, cada una dividida a su vez en varias etapas. El Motorola 68040 tenía 6 etapas para el tratamiento de los enteros.

La ejecución de una instrucción podría descomponerse en las siguientes 5 etapas:

1. **F**: Alimentación de la instrucción (*fetch*)
2. **D**: Decodificación de la instrucción
3. **O**: Extracción y cálculo de los operandos
4. **E**: Ejecución (en la ALU)
5. **W**: Escritura del resultado (*write*)

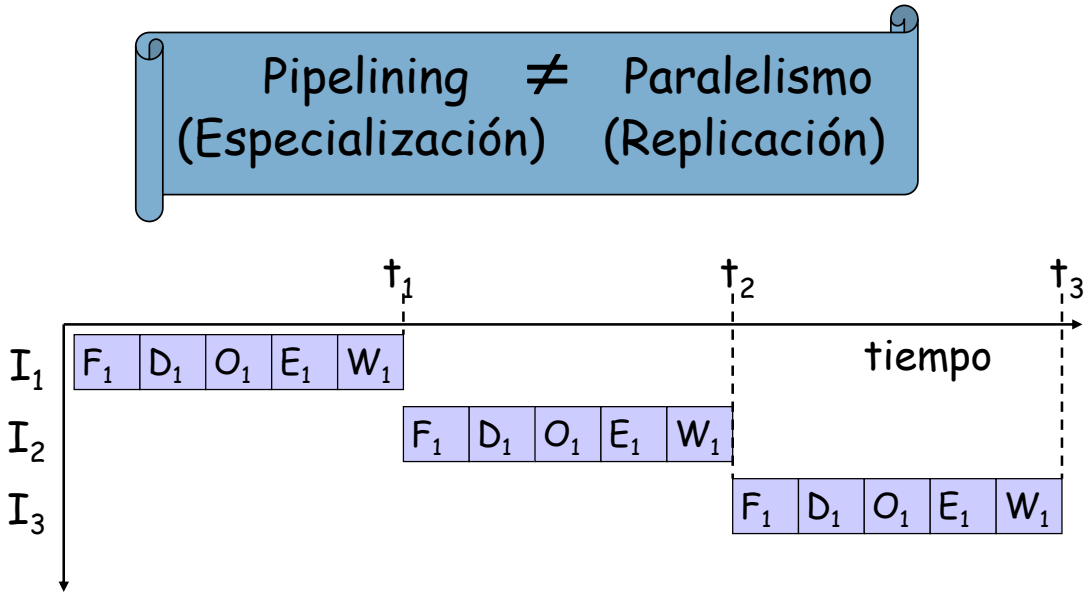
Si ahora la ejecución de una instrucción está descompuesta en 5 etapas, cada etapa puede durar aproximadamente 1/5 de la duración total de la ejecución de la instrucción. Si suponemos que la duración de un ciclo de reloj es igual a la duración de cada una de estas pequeñas etapas, podemos decir, en principio, que con la técnica del *pipelining* se consigue que a cada ciclo de reloj finalice una instrucción, o lo que es lo mismo, una velocidad de instrucción por ciclo.



Por lo que hemos dicho hasta ahora, esta técnica puede reducir el número de ciclos/instrucción en un factor igual a la profundidad del *pipeline* (número de etapas). Según esto, parece que cuanto mayor sea el número de etapas de un *pipeline*, mayor es la velocidad de ejecución.

Sin embargo, los diseñadores del S/360 de IBM (años 60) ya se dieron cuenta de que la cantidad de lógica de control necesaria para gestionar y optimizar los buffers intermedios y las dependencias entre las etapas del *pipeline* crece enormemente con el número de etapas, hasta el punto de que esta lógica de control entre etapas puede llegar a ser más compleja y costosa (en tiempo) que la lógica propia de cada etapa.

Dada la conveniencia de un alto número de etapas, a medida que se consiguen avances en la tecnología, los procesadores cada vez disfrutan de un mayor número de etapas, consiguiendo así, la correspondiente mejora en sus pretaciones.



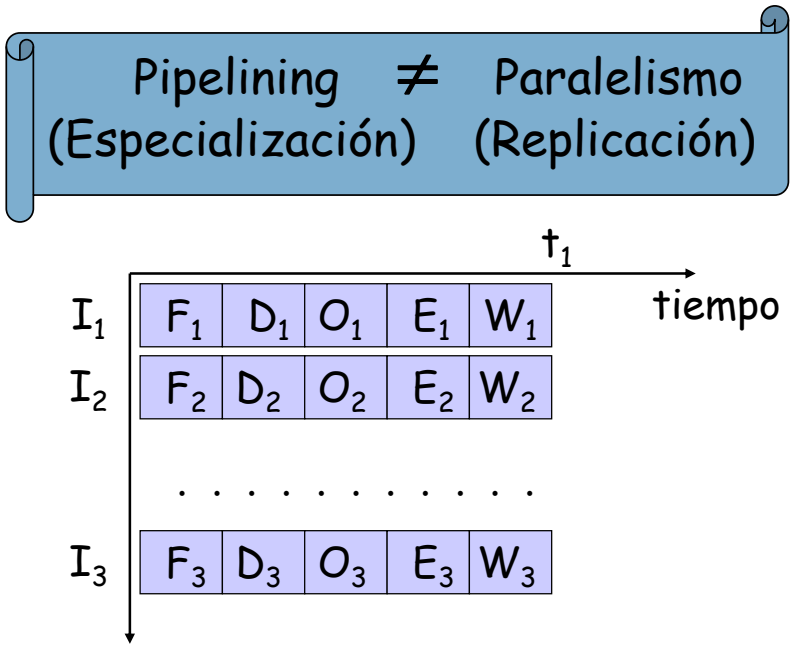
(a) Ejecución Secuencial Pura

Obsérvese que el *pipelining* no es lo mismo que el paralelismo (aunque, en cierto modo, en el *pipeline* también hay paralelismo). Ambas técnicas están dirigidas a mejorar el rendimiento (número de instrucciones por unidad de tiempo) incrementando el número de módulos hardware que operan simultáneamente, pero en el primer caso, el hardware para ejecutar una instrucción **no está replicado, simplemente está dividido en varias etapas distintas especializadas**, mientras que en las arquitecturas paralelas, el hardware (la CPU) sí está replicado (hay varios procesadores), por lo que varias operaciones pueden ejecutarse de manera completamente simultánea.

El incremento del rendimiento con el *pipelining* está limitado al máximo número de etapas del procesador, mientras que con el paralelismo, las prestaciones mejoran siempre que se añadan más procesadores (en situaciones ideales) y el trabajo se pueda descomponer en varias tareas para poder repartirlo.

Veamos, a continuación, los diagramas correspondientes a la ejecución serie, en paralelo y mediante *pipeline*, suponiendo que las instrucciones se ejecutan en cinco pasos o etapas de un ciclo de reloj cada una.

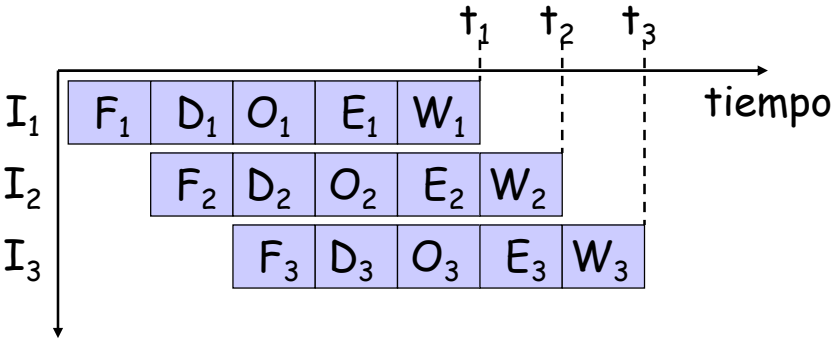
En la ejecución **en serie**, la primera instrucción debe ejecutarse completamente antes de comenzar la segunda, y ésta debe completarse a su vez antes de que comience la tercera. De esta manera, si las instrucciones son sumas, por ejemplo, **se obtiene un resultado cada cinco ciclos** (en t_1, en t_2 en t_3, ...).



(b) Ejecución Paralela Pura

Con un **paralelismo de *N* vías** (*N* procesadores), se pueden ejecutar simultáneamente *N* instrucciones, pero producirán resultados solamente cada 5 ciclos (***N* resultados cada 5 ciclos**). Ya que se producen *N* resultados en el mismo tiempo en el que la ejecución en serie obtiene un único resultado, el incremento o aceleración (en el caso ideal) es *N*.

Pipelining ≠ Paralelismo
(Especialización) (Replicación)



(c) Ejecución en *Pipeline*

En el caso del ***pipeline***, la segunda instrucción puede comenzar en cuanto la primera instrucción haya finalizado su primera etapa. A partir del momento en que se llena el *pipeline* (después de cinco ciclos) se tienen cinco instrucciones ejecutándose en distintas fases, y se puede empezar a obtener **un resultado por ciclo**, pues finalizará una instrucción después de cada ciclo.

Obsérvese que **el rendimiento de un *pipeline* no depende exactamente del número de etapas, sino de la duración de su etapa más larga**.

Aunque con una organización totalmente distinta, en cuanto al rendimiento, el paralelismo y el *pipeline* se pueden considerar equivalentes.

No olvidar que la técnica de la segmentación o *pipelining* **mejora el rendimiento no el tiempo de ejecución de cada instrucción.**

Ralentización

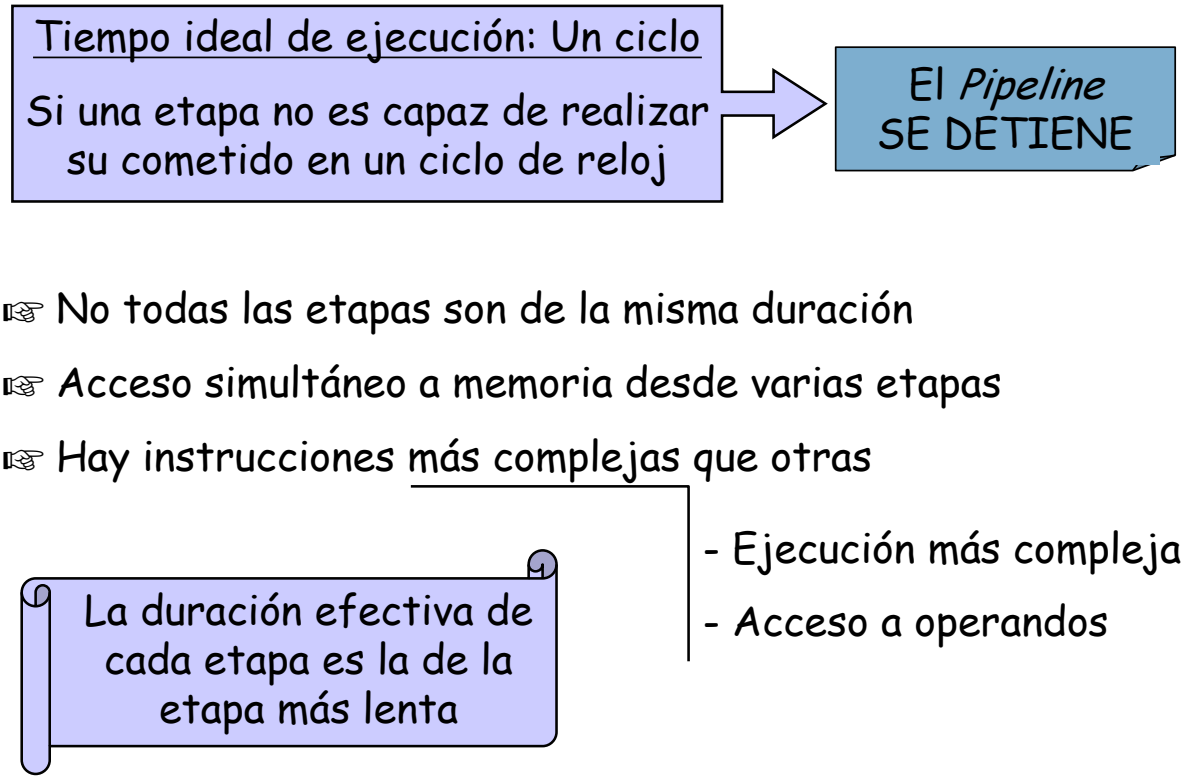
Una vez elegido el número óptimo de etapas, para que el factor de aceleración sea igual al número de etapas se requiere que todas las etapas del *pipeline* siempre estén llenas de instrucciones útiles, y que nada retrase el avance de las instrucciones a través del *pipeline*.

Por desgracia, no es fácil mantener siempre ocupadas todas las etapas del *pipeline*. Hay tres causas que lo impiden:

- Motivos estructurales.
- Dependencias de operandos.
- Instrucciones de bifurcación.

En las siguientes transparencias las comentaremos con cierto detalle.

Motivos Estructurales



Como ya veremos, se tiende a que la ejecución de cada etapa se realice en un ciclo de reloj. Pues bien, **cuando una etapa no es capaz de realizar su cometido en un ciclo de reloj, el pipeline se detiene** hasta que dicha etapa finaliza su trabajo. Hay varias causas estructurales (arquitectura del *pipeline*) que pueden hacer que el *pipeline* se detenga.

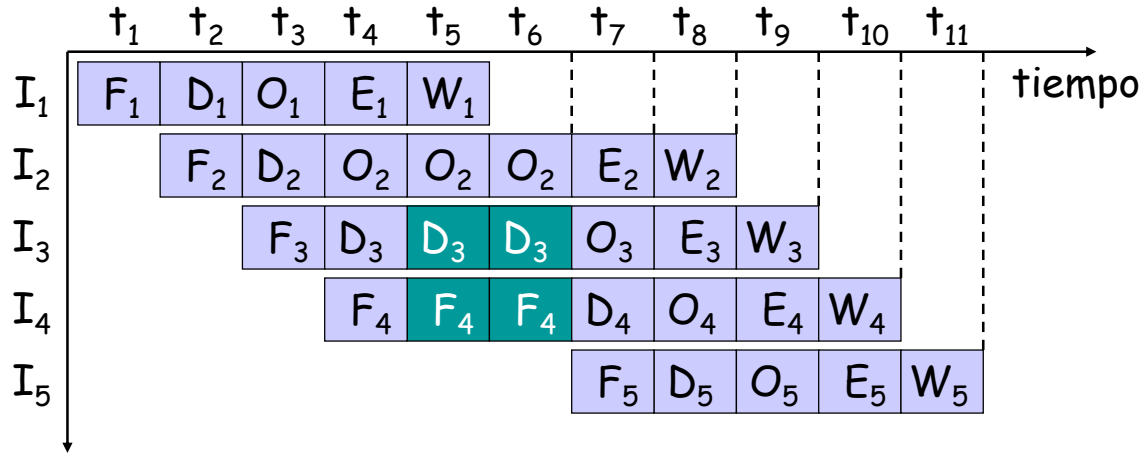
Por ejemplo, puede ocurrir que **no todas las etapas sean de la misma duración**, con lo que alguna etapa de corta duración debería esperar a que acabe la siguiente que es más larga. Esto hará que la duración efectiva de cada etapa sea igual a la duración de la etapa más larga. Normalmente los procesadores actuales tienden a un alto número de etapas, con lo que automáticamente tienden a igualarse los tiempos.

Otra cosa que también puede ocurrir es que **desde varias etapas se quiera acceder a memoria simultáneamente** (por ejemplo en la etapa de alimentación de instrucción y en la escritura del resultado). Y, claro, si una etapa se detiene para esperar a poder realizar el acceso a memoria, el pipeline se para.

También tenemos que considerar que **no todas las instrucciones hacen las mismas cosas**, por lo que requieren tiempos distintos de CPU. Pasemos a la siguiente página para tratar este caso con más detalle.

...Motivos Estructurales

☞ Hay instrucciones más complejas que otras



¿Y si alguna instrucción no utiliza todas las etapas?

No todas las instrucciones hacen las mismas cosas y requieren el mismo tiempo de CPU. Unas pueden necesitar más tiempo en la etapa de ejecución (por ejemplo, la carga o escritura de un registro requiere menos trabajo de ALU que una división en coma flotante), mientras que otras pueden necesitar más tiempo para obtener los operandos o escribir el resultado (si están en memoria principal se tarda más que si están en registros).

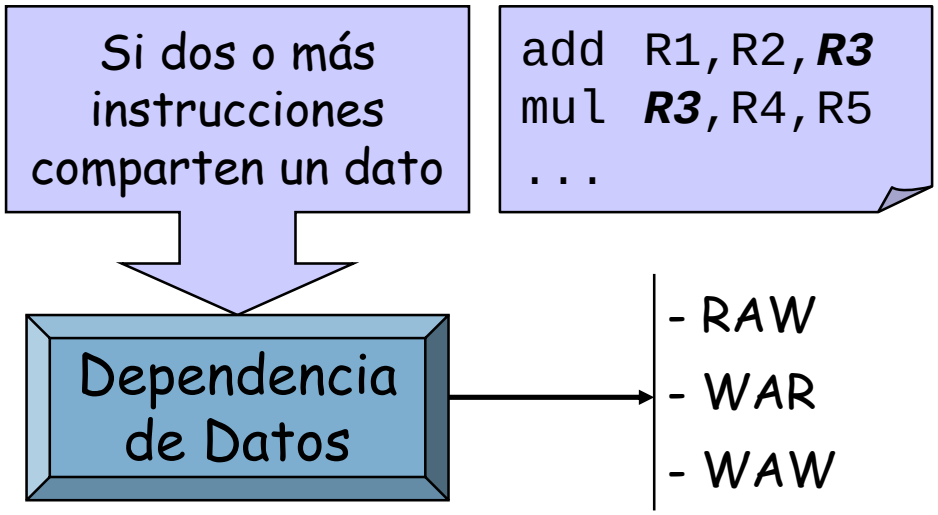
En el ejemplo de la transparencia vemos que la instrucción I_2 no puede completar la fase de alimentación de operandos en el ciclo 4, necesitando para ello también los ciclos 5 y 6. Esto hace que en el 5º ciclo no pueda alimentarse la instrucción I_5 por estar ocupada la etapa de extracción de instrucción, debiendo esperar ésta al ciclo 7 para poder continuar extrayendo instrucciones. Obsérvese que como consecuencia del sobretiempo de O_2, al término de los ciclos 6 y 7 no finaliza ninguna instrucción (lo cual va en perjuicio del rendimiento).

Puede suceder incluso que **alguna de las etapas ni siquiera necesite ejecutarse**. Por ejemplo, en un procesador cuya última etapa se dedique a escribir en memoria principal, la carga de un registro no requerirá dicha última etapa. En este caso, simplemente sucederá que cuando una instrucción corta va después de una normal, ambas finalicen su ejecución simultáneamente, y en el siguiente ciclo de reloj no termine ninguna instrucción; por lo tanto, el rendimiento no varía.

Aquí, para simplificar el problema, supondremos que todas las instrucciones pasan por todas las etapas y que todas son de la misma duración.

Nos permitimos realizar estas simplificaciones para poder centrarnos en las principales causas de la ralentización del *pipeline*: las dependencias de datos y las bifurcaciones. Veámoslas a continuación.

Dependencia de Datos



Las dependencias de datos se producen cuando dos instrucciones comparten un dato (operando o resultado). La situación es la siguiente: Una instrucción I_j actualiza el valor de una variable, pero una instrucción posterior, I_k, accede a esa variable antes de que I_j haya terminado la operación.

Hay tres tipos de dependencias de datos, pero aquí vamos a comentar solamente el más evidente, el que se denomina “lectura después de escritura” (*Read After Write*, o RAW).

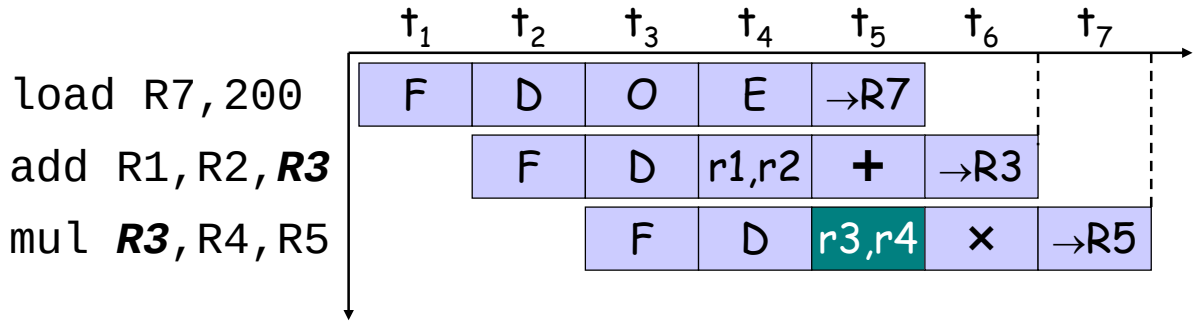
...Dependencia de Datos

Dependencia RAW

```
    ...
I1  load  R7,200
I2  add   R1,R2,R3
I3  mul   R3,R4,R5
I4  load  R1,200
I5  load  R2,300
    ...
```

2 SOLUCIONES

- ☞ Prevención
- ☞ Detección y Resolución



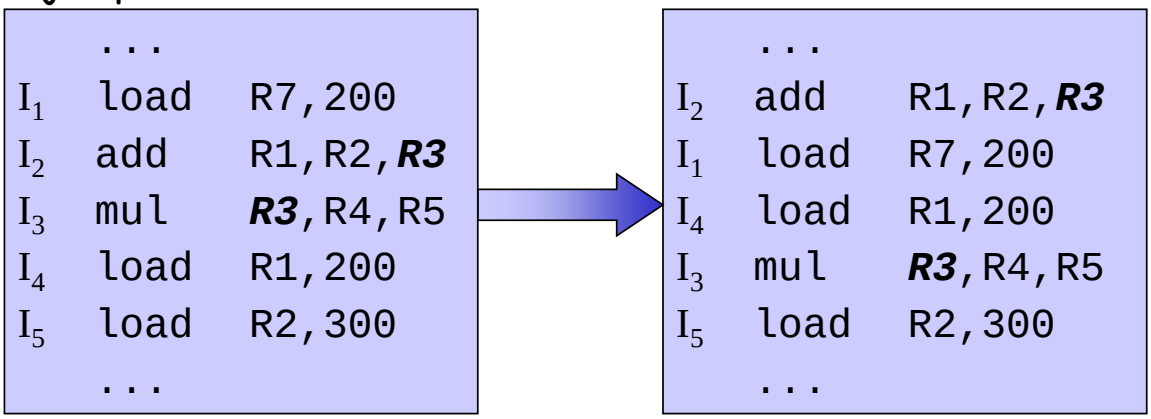
En el programa del ejemplo, la dependencia que se denomina “lectura después de escritura” (*Read After Write*, o RAW) puede producirse entre las instrucciones I_2 e I_3 si la instrucción `MUL` lee el contenido de R3 (en el ciclo 5) antes de que el resultado de la suma anterior (al final del ciclo 6) se cargue en él. Obviamente, la operación `MUL` no se ejecutará con los operandos esperados por el programador, por lo que el resultado del programa será incorrecto.

Hay dos opciones básicas para resolver este problema de dependencia de datos; uno es mediante la **prevención**: evitando que pueda llegarse a esta situación de dependencia; el otro es mediante la **detección y resolución**, es decir, no preocupándose de evitarlo, pero sí de detectarlo en caso de que se produzca y solucionarlo de alguna manera. Veámoslas en detalle.

Prevención de la Dependencia de Datos

El compilador debe retrasar la etapa de ejecución de la instrucción dependiente reordenando las instrucciones

Ejemplo 1



La dependencia de datos: **Prevención**.

El problema de la dependencia de datos entre una instrucción I_1 y otra instrucción I_2 que le sigue puede prevenirse retrasando la ejecución de I_2 un número *K* de etapas hasta que desaparezca el problema de que I_2 lea un operando que I_1 no ha escrito todavía. Este retraso puede conseguirse insertando un número *K* de instrucciones entre I_1 e I_2. Esto significa que el compilador tiene que reordenar el programa para encontrar *K* instrucciones que puedan ejecutarse después de I_1 y antes de I_2 sin que por ello varíe la estructura lógica del programa.

Ejemplo 1: En la transparencia tenemos el ejemplo de un programa en el que hay una dependencia entre las instrucciones I_2 e I_3 a causa del registro R3. Como vemos, en este programa el compilador puede detectar la dependencia de datos y reorganizar las instrucciones para retardar el acceso al registro R3 hasta que esté actualizado. Debe quedar claro que esta reorganización solamente puede hacerse si se mantiene la semántica original del programa.

Por lo que hemos visto en el ejemplo de la página anterior, para evitar la dependencia de I_3 respecto a I_2, se requiere que I_3 comience su ejecución tres ciclos después de que lo haga I_2. Como se puede apreciar, esto se ha conseguido con la reorganización que ha realizado el compilador, intercambiando I_2 por I_1 e I_3 por I_4.

...Prevención de la Dependencia de Datos

¡...Y si no se pueden reordenar las instrucciones sin alterar la lógica del programa!

Ejemplo 2

```
    ...
I1  load   R1,200
I2  add    R1,R2,R3
I3  mul    R3,R4,R5
I4  add    #1,R3,R3
    ...
```

Insertar **NOP**

```
    ...
I1  load   R1,200
    NOP
    NOP
I2  add    R1,R2,R3
    NOP
    NOP
I3  mul    R3,R4,R5
I4  add    #1,R3,R3
    ...
```

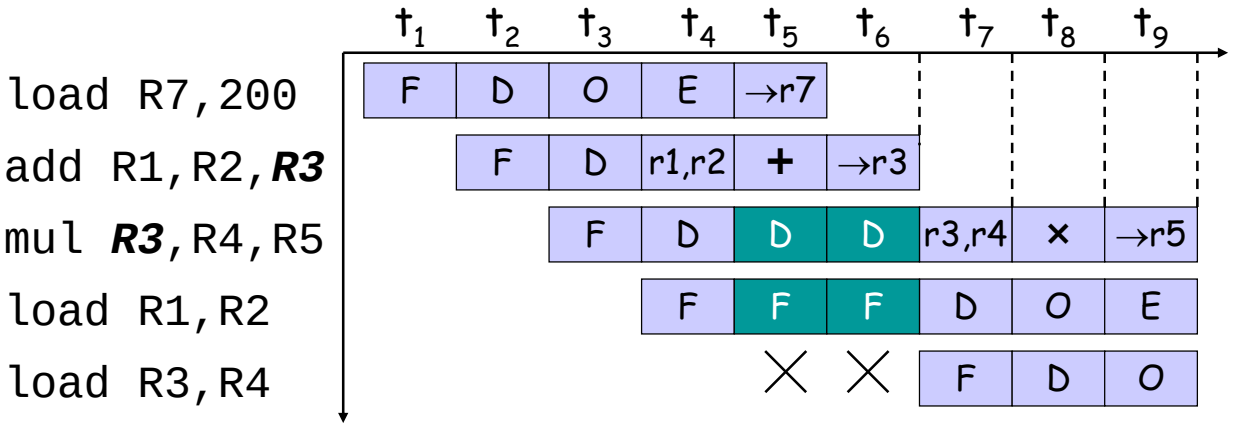

Si el compilador no puede reorganizar el código para encontrar estas *K* instrucciones que decíamos arriba, sin modificar la lógica del programa, **debe insertar operaciones `NOP`** (No Operación) entre I_1 e I_2.

Ejemplo 2: En el ejemplo inferior tenemos el fragmento de un programa en el que también hay dependencias entre las instrucciones I_1, I_2 e I_3. En este caso vemos que las instrucciones de este fragmento no se pueden reordenar sin alterar la lógica del programa, por lo que el compilador inserta las instrucciones `NOP` necesarias para evitar la ejecución errónea de instrucciones por las dependencias de datos.

La ventaja de la solución basada en la prevención es que no se requiere hardware adicional, pero a expensas de un compilador más complejo y una pérdida de tiempo si es necesario insertar instrucciones `NOP` (cuando no se puede reordenar el programa para insertar instrucciones útiles).

Detección y Resolución

☞ Detener el *Pipeline*
- Anticipación



La dependencia de datos: **Detección y resolución**.

Este método requiere un hardware adicional en la CPU, pues se deben detectar las dependencias de datos durante la ejecución y resolver estas dependencias. Detectarlas significa que debe darse cuenta de que en un momento dado hay dos instrucciones arrancadas I_1 e I_2, tal que I_2 depende de un resultado establecido por I_1. El dispositivo hardware que detecta estas dependencias se denomina *interlock*. Resolver las dependencias significa hacer algo para retrasar la ejecución de I_2 o para acelerar, en la medida de lo posible, la entrega a I_2 del resultado que produce I_1.

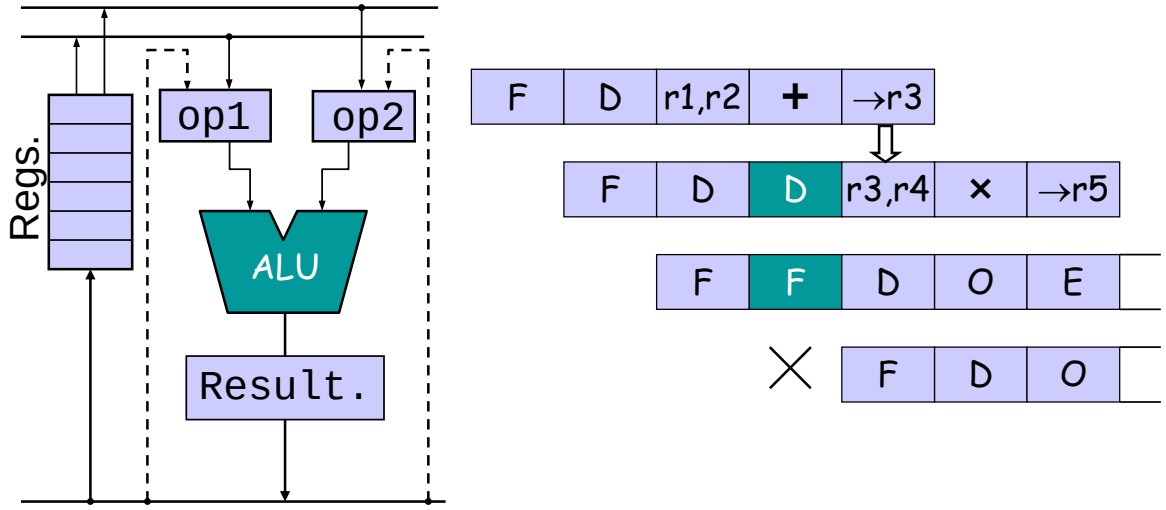
Veamos dos posibilidades de resolución (no excluyentes) para cuando se detecta una dependencia de datos entre dos etapas del *pipeline*.

- **Detener el *pipeline***. La aproximación más sencilla para evitar los problemas de dependencias de datos con ayuda del hardware es detener la actividad en las etapas necesarias del *pipeline* hasta que desaparezca la dependencia, es decir, hasta que se pueda ejecutar correctamente la instrucción dependiente. En el ejemplo de la transparencia, esto significa detener las instrucciones que siguen a la instrucción `add` desde el ciclo 5 hasta que el registro R3 pueda leerse debidamente actualizado en el ciclo 7.

  Con esta estrategia, primero se detecta la dependencia y después se detiene el *pipeline* a partir de la instrucción dependiente hasta que desaparece la dependencia.

Detección y Resolución

- Detener el *Pipeline*
- ☞ Anticipación



El otro modo de resolver las dependencias de datos mediante Detección y Resolución es acelerando, en la medida de lo posible, la entrega a una instrucción I_2 del resultado que produce una instrucción previa I_1. Veámoslo en detalle:

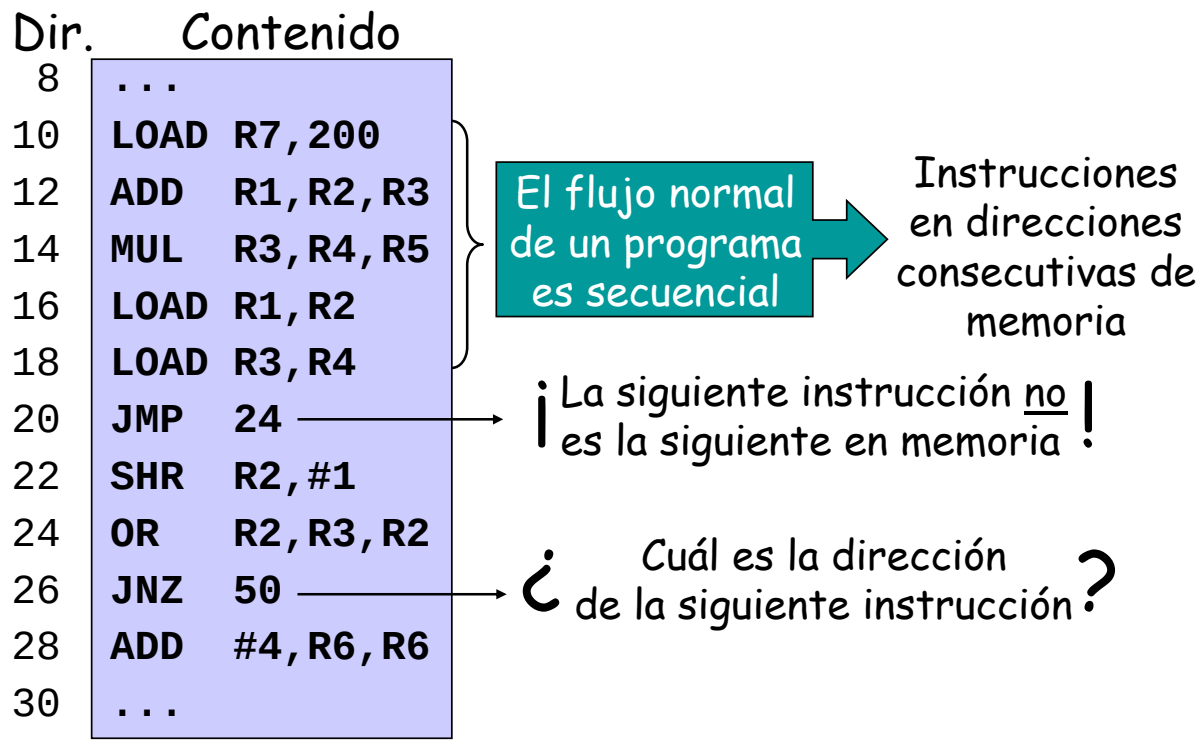
- **Anticipación (*data forwarding*)**. En una dependencia de datos RAW, puede suceder que una instrucción I_2 necesite un operando en la etapa O (obtención de operandos) que debe producirlo en el mismo ciclo la instrucción I_1 en su etapa E (ejecución). Esto obligaría a detener la instrucción I_2 hasta que I_1 escriba el resultado y entonces pueda continuar I_2 en su etapa O y leer el resultado que produjo I_1.

  Este retraso puede evitarse redirigiendo (*forwarding*) el resultado de la etapa E de la instrucción I_1 directamente a la entrada de la etapa E o a la etapa O de la instrucción I_2, obteniendo el mismo efecto que se obtendría en la ejecución de la etapa O de I_2.

  En la diapositiva se muestra el esquema de una CPU con hardware de anticipación. Una vez que en la etapa de ejecución se tiene el resultado para escribir en R3, se dirige dicho resultado directamente a la etapa de obtención de operandos de la siguiente instrucción, sin esperar a que finalice la etapa de escritura del resultado de la instrucción de suma.

  En este caso no se consigue evitar totalmente la detención del *pipeline*, pero sí se evita un ciclo de espera sobre la situación en la que no hay “anticipación”. La medida en que se consigue evitar que se detenga el pipeline depende del número de etapas y de sus funciones.

Bifurcaciones



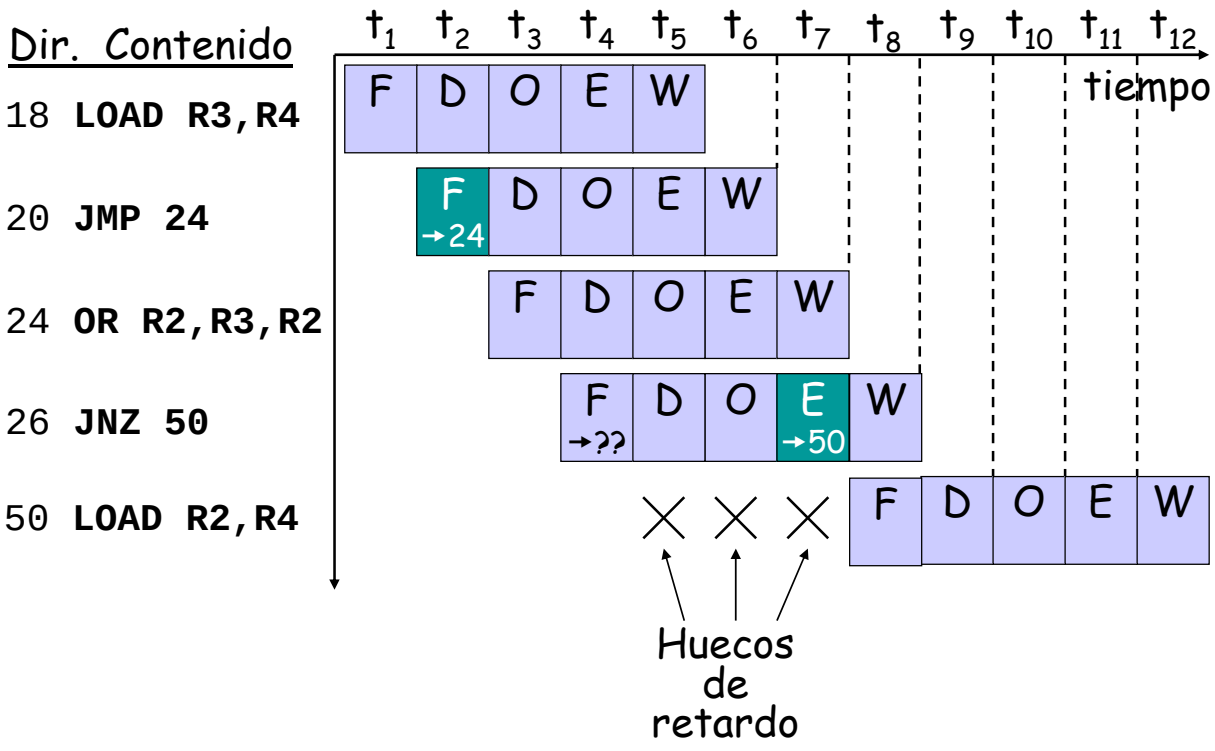
Ya hemos comentado que uno de los principales problemas en el diseño de un *pipeline* consiste en asegurar el mantenimiento de un flujo constante de instrucciones alimentando sus diversas etapas para así poder mantener también constante el ritmo de ejecución de instrucciones (idealmente, una por ciclo).

El flujo normal de ejecución de un programa es secuencial, por lo que las instrucciones que se van alimentando y ejecutando están en direcciones consecutivas de memoria. Por desgracia, las instrucciones de bifurcación (que suelen representar alrededor del 20% de las instrucciones ejecutadas) pueden romper el flujo constante de instrucciones alimentadas.

Cuando se alimenta una instrucción en la CPU, lo primero que se hace es incrementar el registro Contador de Programa para conocer la dirección de la siguiente instrucción a ejecutar y extraerla.

Pero si se trata de una instrucción de salto, hasta que no llega a la etapa de ejecución no se establece en el Contador de Programa la dirección de la siguiente instrucción a ejecutar, por lo que la etapa de alimentación de instrucción no sabe por dónde seguir alimentando instrucciones.

¡Tenemos un problema con los saltos!

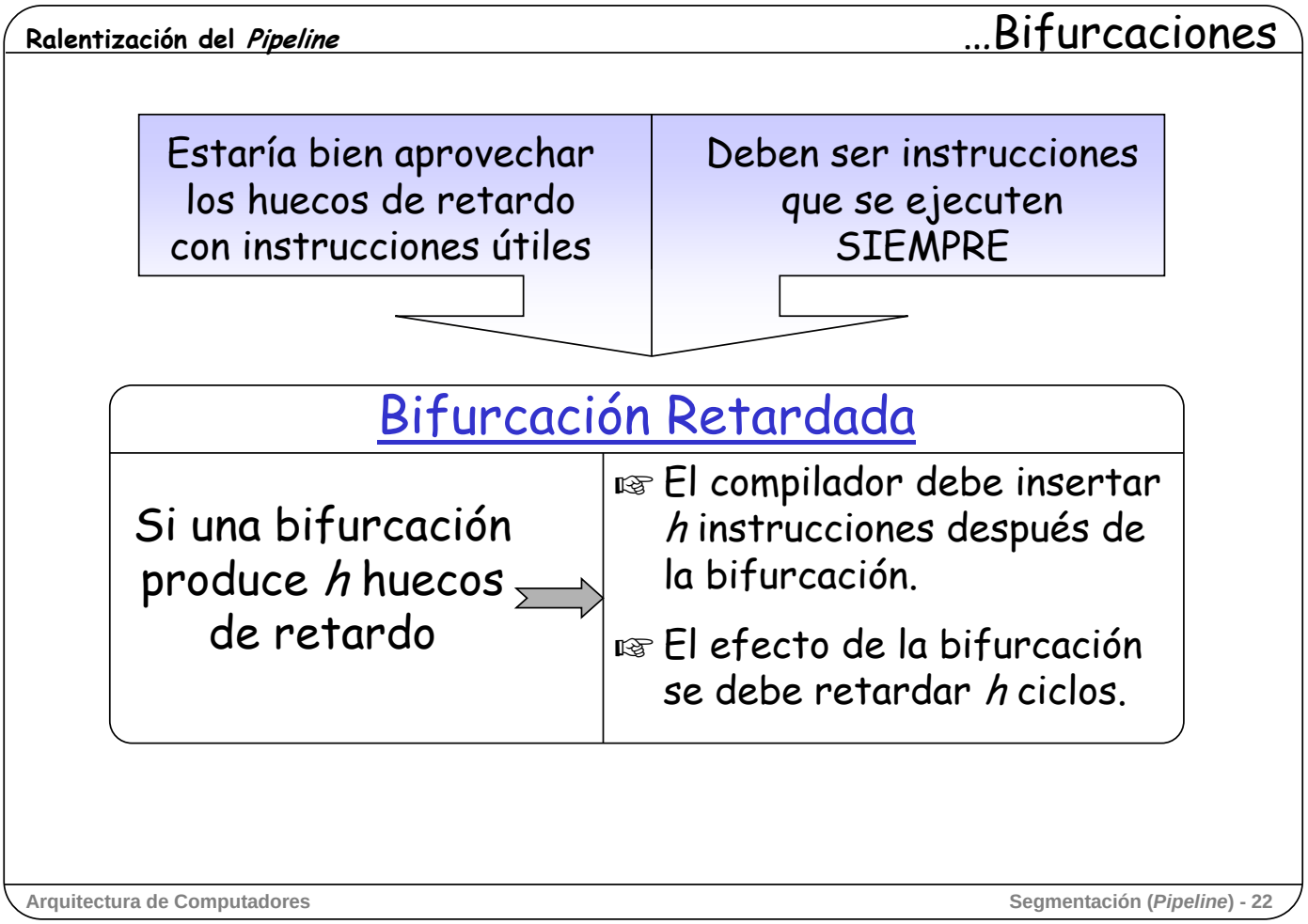


Una instrucción de bifurcación hace que la dirección de la siguiente instrucción a ejecutar no se conozca hasta el momento justo de la ejecución de la instrucción de bifurcación (en la etapa de ejecución), por lo que la etapa de alimentación no puede extraer la siguiente instrucción a una bifurcación hasta que esta última no finalice su etapa de ejecución, y por lo tanto al terminar la etapa de ejecución, ésta tendría que esperar hasta que se alimente la siguiente instrucción y vaya avanzando por todas las etapas anteriores, que se habrán quedado vacías. A estas etapas vacías que aparecen se las denomina **huecos de retardo** (*delay slots*).

En algunos sistemas, las **bifurcaciones incondicionales** pueden detectarse en la fase de alimentación o extracción (*fetch*) si se le añade un poco hardware a esta primera etapa. Este hardware de ayuda también extrae la dirección de salto, con lo que se puede proseguir la extracción de instrucciones de la dirección del salto.

Sin embargo, en el caso de las **bifurcaciones condicionales** no se puede hacer esto, pues puede ocurrir que la condición del salto se establezca precisamente en la etapa de ejecución de la instrucción anterior, con lo que no hay más remedio que esperar a que la bifurcación llegue a la etapa de ejecución para conocer la dirección de la siguiente instrucción. Esto quiere decir que se debe detener la alimentación de instrucciones al *pipeline* hasta que en la etapa de ejecución se averigüe la dirección de la siguiente instrucción

Afortunadamente, hay diversas técnicas que pueden evitar o minimizar el impacto de las instrucciones de bifurcación, tales como la “bifurcación retardada”, la “predicción del salto” y algunas más. Veamos estas dos primeras técnicas.



El problema de los saltos: **Bifurcación retardada**.

Ya hemos visto que cuando entra una instrucción de salto en el *pipeline*, se producen *h* **huecos de retardo** por lo que hay que esperar *h* ciclos hasta que llega la siguiente instrucción de la secuencia a ejecutar. Estaría bien que estos huecos se pudieran rellenar con instrucciones que siempre se deban ejecutar, independientemente de si se toma la bifurcación o no. Pero claro, si estas instrucciones están en memoria inmediatamente después de la instrucción de bifurcación, según lo que sabemos hasta ahora, no se ejecutarían si la instrucción de bifurcación decide saltar. Para conseguir que se ejecuten siempre las *h* instrucciones que siguen a una bifurcación, en los procesadores que tienen bifurcaciones retardadas, las instrucciones de salto no tienen efecto hasta *h* instrucciones después de su ejecución, por lo que independientemente del resultado de la ejecución de la bifurcación, siempre se ejecutan las *h* instrucciones siguientes. De esta manera no se producen los huecos de retardo.

CPU SIN bifurcaciones retardadas

- Cuando NO se toma la bifurcación:

$I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow I_4 \rightarrow I_5 \rightarrow \boldsymbol{I_6} \rightarrow I_7 \rightarrow I_8 \rightarrow I_9 \rightarrow I_{10} \rightarrow I_{11} \rightarrow I_{12} \rightarrow ...$

- Cuando SÍ se toma la bifurcación:

$I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow I_4 \rightarrow I_5 \rightarrow \boldsymbol{I_6} \rightarrow I_{11} \rightarrow I_{12} \rightarrow ...$

CPU CON bifurcaciones retardadas (h = 3)

- Cuando NO se toma la bifurcación:

$I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow I_4 \rightarrow I_5 \rightarrow \boldsymbol{I_6} \rightarrow I_7 \rightarrow I_8 \rightarrow I_9 \rightarrow I_{10} \rightarrow I_{11} \rightarrow I_{12} \rightarrow ...$

- Cuando SÍ se toma la bifurcación:

$I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow I_4 \rightarrow I_5 \rightarrow \boldsymbol{I_6} \rightarrow \boldsymbol{I_7} \rightarrow \boldsymbol{I_8} \rightarrow \boldsymbol{I_9} \rightarrow I_{11} \rightarrow I_{12} \rightarrow ...$

Supongamos la serie de instrucciones de la transparencia, en la que I_6 es la instrucción de salto condicional a la instrucción I_{11}. En una CPU sin saltos retardados, la secuencia de instrucciones ejecutadas cuando la bifurcación no tiene lugar es: I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, ... Mientras que cuando sí se produce la bifurcación, la secuencia es: I_1, I_2, I_3, I_4, I_5, I_6, I_{11}, I_{12}, ...

Si a este mismo procesador se le ponen bifurcaciones retardadas (con tres huecos de retardo), la secuencia de instrucciones ejecutadas cuando se produce la bifurcación es: I_1, I_2, I_3, I_4, I_5, I_6, $\boldsymbol{I_7}$, $\boldsymbol{I_8}$, $\boldsymbol{I_9}$, I_{11}, I_{12}, ... Es decir I_7, I_8 e I_9 se ejecutan siempre, haya o no bifurcación.

Esto es así porque el efecto de la bifurcación se retarda 3 ciclos de reloj, es decir, después de alimentar una instrucción de salto, se siguen extrayendo y ejecutando las instrucciones siguientes, según su orden en la memoria, durante 3 ciclos de reloj. Después, se establece en el contador de programa la dirección indicada en la instrucción de salto, con lo que la 4ª instrucción después de la de salto, será ya la correspondiente a la de la dirección indicada en dicha instrucción de salto.

Ralentización del *Pipeline* …Bifurcaciones

EJEMPLO

Ejecución en procesador con 3 huecos de retardo en los saltos

Programa

```
     ...
     ...
I1:  MOV  #0,R2
I2:  MOV  #0,R7
I3:  LD   R8,VALOR
I4:  ADD  #1,R8
I5:  CMP  R2,R3
I6:  BEQ  I11
I7:  ADD  #1,R3
I8:  ADD  #2,R4
I9:  MUL  R3,R4
I10: ST   R4,TOTAL
I11: ADD  #1,R5
I12: ST   R5,TOTAL
     ...
     ...
     ...
```

Secuencia de Ejecución Cuando NO se toma la Bifurcación

```
     ...
     ...
I1:  MOV  #0,R2
I2:  MOV  #0,R7
I3:  LD   R8,VALOR
I4:  ADD  #1,R8
I5:  CMP  R2,R3
I6:  BEQ  I11
I7:  ADD  #1,R3
I8:  ADD  #2,R4
I9:  MUL  R3,R4
I10: ST   R4,TOTAL
I11: ADD  #1,R5
I12: ST   R5,TOTAL
     ...
     ...
     ...
```

Secuencia de Ejecución Cuando SÍ se toma la Bifurcación

```
     ...
     ...
I1:  MOV  #0,R2
I2:  MOV  #0,R7
I3:  LD   R8,VALOR
I4:  ADD  #1,R8
I5:  CMP  R2,R3
I6:  BEQ  I11
I7:  ADD  #1,R3
I8:  ADD  #2,R4
I9:  MUL  R3,R4
I11: ADD  #1,R5
I12: ST   R5,TOTAL
     ...
     ...
     ...
```


Veamos con un ejemplo el efecto de las bifurcaciones retardadas en un procesador con 3 huecos de retardo en los saltos.

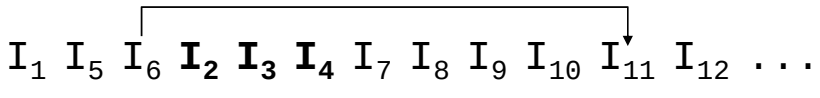
Supongamos que tenemos el fragmento de un programa como el de arriba a la izquierda. Ahora lo ejecutaremos y veremos su comportamiento.

En el cuadro del medio tenemos la secuencia de ejecución del programa en el caso en que, de acuerdo a la instrucción de comparación I_5, en el salto condicional I_6 se decide continuar la ejecución sin tomar el salto, con lo que se continúan ejecutando las instrucciones I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, …

Ahora veamos en el cuadro de la derecha el comportamiento del programa cuando, en ejecución, sí se debe tomar la bifurcación de la instrucción I_6. Como se puede observar, después de la ejecución de la instrucción de salto I_6, a pesar de que se determina que se debe saltar, se continúan ejecutando las 3 instrucciones siguientes: I_7, I_8 e I_9, y luego ya, sí se salta a la instrucción I_{11}, como establece la sentencia de salto I_6.

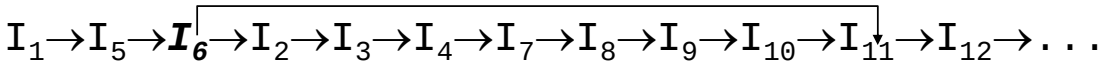
CPU con bifurcaciones retardadas (h = 3)

Conocido el retardo en las bifurcaciones, si es preciso, reordenamos el código para mantener la semántica del programa:

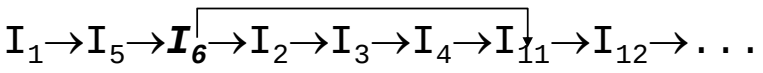


Ahora, la ejecución será así:

- Cuando NO se toma la bifurcación:



- Cuando SÍ se toma la bifurcación:



A partir de un fragmento de código escrito para un procesador sin saltos retardados, si se va a ejecutar en una CPU con saltos con 3 huecos de retardo, sabiendo lo que sucede en los procesadores con bifurcaciones retardadas (tarda 3 ciclos en bifurcar realmente), vamos a intentar reordenar nuestro fragmento de código para que su ejecución se produzca con la misma semántica que la esperada en un procesador sin saltos retardados.

Así, después de la instrucción de salto vamos a poner 3 de las instrucciones que hay antes de la de salto, es decir, 3 instrucciones que queremos que se ejecuten siempre (se produzca el salto o no).

Así, suponiendo que no se altera la semántica del programa, podríamos mover las instrucciones I_2, I_3 e I_4 y ponerlas justo a continuación de I_6 (la bifurcación condicional), quedando entonces la secuencia de instrucciones en memoria así:

$$I_1, I_5, I_6, \mathbf{I_2}, \mathbf{I_3}, \mathbf{I_4}, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, \ldots$$

Ahora vamos a ejecutar este programa en nuestro procesador con bifurcaciones retardadas de 3 ciclos.

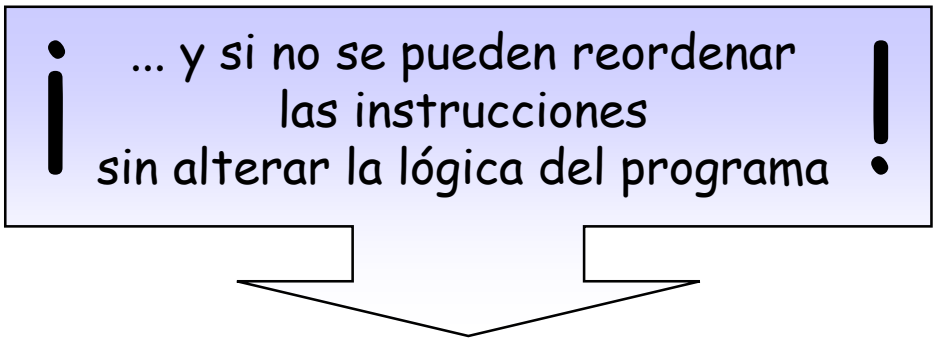
Cuando se no se produce la bifurcación, la secuencia de instrucciones que se ejecuta es:

$$I_1, I_5, I_6, \mathbf{I_2}, \mathbf{I_3}, \mathbf{I_4}, I_7, I_8, I_9, I_{10}, I_{11}, I_{12}, \ldots$$

Si la bifurcación tiene lugar, las instrucciones se ejecutarán en este orden:

$$I_1, I_5, I_6, \mathbf{I_2}, \mathbf{I_3}, \mathbf{I_4}, I_{11}, I_{12}, \ldots$$

Si no hubiese bifurcación retardada, al alimentar una instrucción de salto condicional habría que detener la alimentación de instrucciones hasta que la instrucción de salto pasase por la etapa de ejecución (hasta saber cuál es la siguiente instrucción a ejecutar). Con bifurcación retardada se aprovechan esos ciclos en alimentar y ejecutar instrucciones que se desea ejecutar incondicionalmente antes de que la instrucción de bifurcación tenga lugar (se salte o no).



Insertar Instrucciones **NOP**

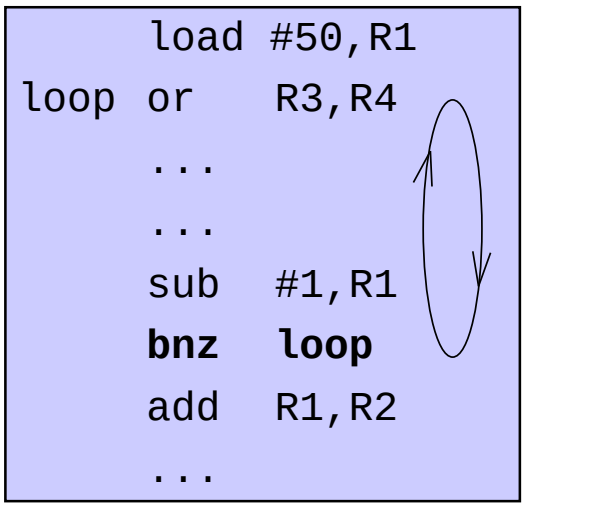
$I_1 \rightarrow I_2 \rightarrow \boldsymbol{I_3} \rightarrow NOP \rightarrow NOP \rightarrow NOP \rightarrow I_{11} \rightarrow I_{12} \rightarrow ...$

Esta técnica de los saltos retardados requiere la colaboración del compilador, que debe saber cómo reorganizar el código para rellenar los huecos de retardo con instrucciones útiles (de la misma manera que se hacía con las dependencias de datos).

Si el compilador no encuentra una manera de reordenar el código sin afectar a su semántica, debe insertar operaciones `NOP` en los huecos de retardo.

Predicción del Salto

Ante un salto ...



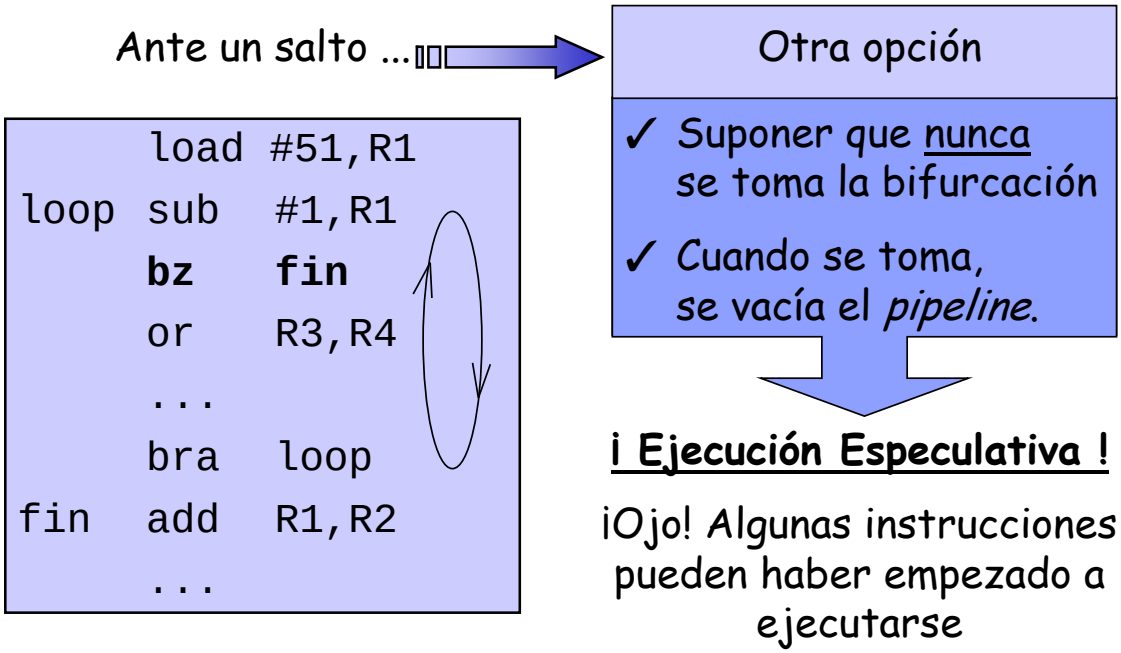
Una opción

- ✓ Suponer que siempre se toma la bifurcación
- ✓ Cuando no se toma, se vacía el *pipeline*.

El problema de los saltos: **Predicción del salto**.

Otra técnica para reducir el problema de las bifurcaciones consiste en intentar predecir si una instrucción de bifurcación saltará o no. Por ejemplo, una bifurcación al final de un bucle salta al comienzo de éste todas las veces excepto la última. Según esto, sería ventajoso que cuando el procesador se encuentra una instrucción de salto suponga que el salto sí se va a efectuar realmente, y cuando la etapa de alimentación detecte una bifurcación, empiece a extraer instrucciones de la dirección de destino del salto. Si una vez que se ejecuta la instrucción de bifurcación, resulta que efectivamente se salta, la ejecución continúa normalmente, pues las instrucciones de la dirección del salto son las que ya se están alimentando. Si por el contrario resulta que no se realiza el salto, se debe vaciar el *pipeline* (desechar todas las instrucciones alimentadas) y empezar a alimentar las instrucciones que siguen en secuencia.

... Predicción del Salto

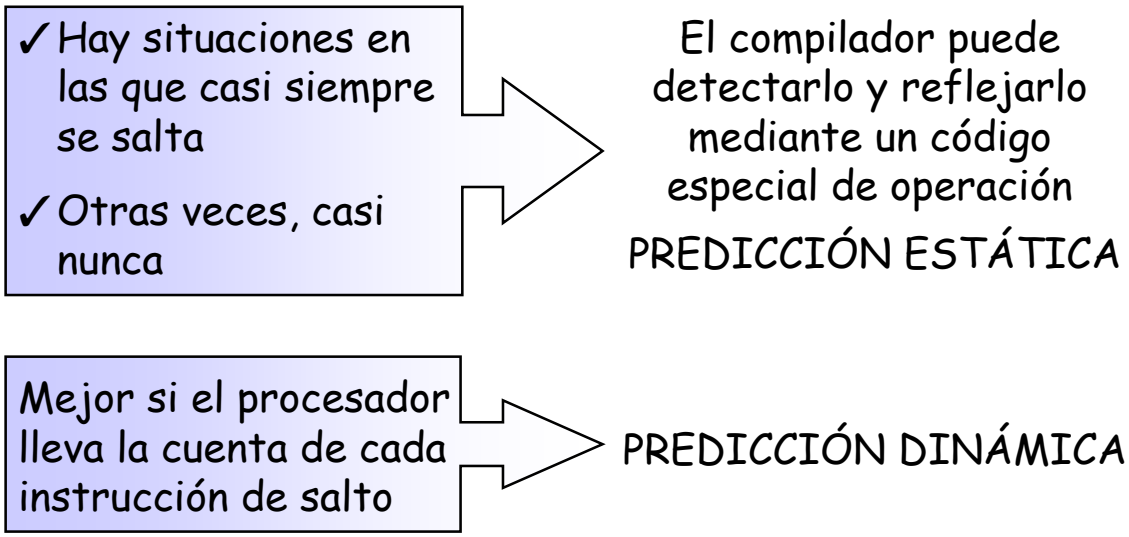


Si ahora suponemos que el control del bucle se realiza mediante una instrucción al comienzo del mismo, ahora lo normal será suponer que la bifurcación no se tomará hasta la última pasada del bucle. Es decir, hay veces que conviene suponer una cosa y otras veces otra.

Esto que hemos visto se denomina **ejecución especulativa**, pues las instrucciones pueden empezar a ejecutarse antes de que el procesador sepa que las instrucciones alimentadas son las realmente correctas. Supongamos que se predice que el salto tendrá lugar, por lo que se empiezan a alimentar instrucciones y a pasarlas a las siguientes etapas del *pipeline* antes de que la instrucción de bifurcación finalice su etapa de ejecución. ¡Y si al ejecutar la bifurcación no se realiza el salto! ¡Nos encontramos que algunas instrucciones ya se han empezado a ejecutar en las etapas anteriores!

Con ejecución especulativa se debe tener cuidado de que en las etapas anteriores a la de ejecución no se modifiquen registros o posiciones de memoria hasta que no se confirme que la predicción realizada ha sido la acertada.

... Predicción del Salto



La repetición de la bifurcación que se toma en el bucle la puede detectar el compilador y establecer la predicción mediante un cierto código de operación en la bifurcación, lo que quiere decir que, en ejecución, siempre que se alimente esa instrucción de bifurcación se hará la misma predicción. Por esto, se la conoce como **predicción estática**. El PowerPC, por ejemplo, dispone de códigos de operación para predicción estática.

La predicción se puede mejorar si se realiza **dinámicamente**, para lo cual el hardware del procesador debe establecer la posibilidad de que haya o no salto cada vez que se encuentre una cierta instrucción de bifurcación. Para ello, en la CPU se debe llevar la cuenta de los resultados de las últimas ejecuciones de cada bifurcación. Ciertos estudios estadísticos dicen que conservando solamente el resultado de la última ejecución (con un bit) ya se obtiene una gran probabilidad de acierto (del orden del 90%) y con la historia de las cuatro últimas ejecuciones (dos bits) se mejora ligeramente; manteniendo una historia mayor, la mejora es despreciable.