¿Que es un pipeline de CPU?

La ejecución de una instrucción que existe en memoria puede ser **dividida en varias operaciones** más pequeñas que deben ser **realizadas en sucesión**.

Cada una de estas operaciones es realizada por partes diferentes del CPU. Por lo tanto, si se llevan a cabo todas las operaciones para una instrucción y luego se comienza con la siguiente, algunas partes del CPU quedan inactivas mientras no se usan.

Una arquitectura con pipeline resuelve este problema ejecutando algunas operaciones para una instrucción y otras operaciones para las instrucciones siguientes.

Un pipeline sencillo

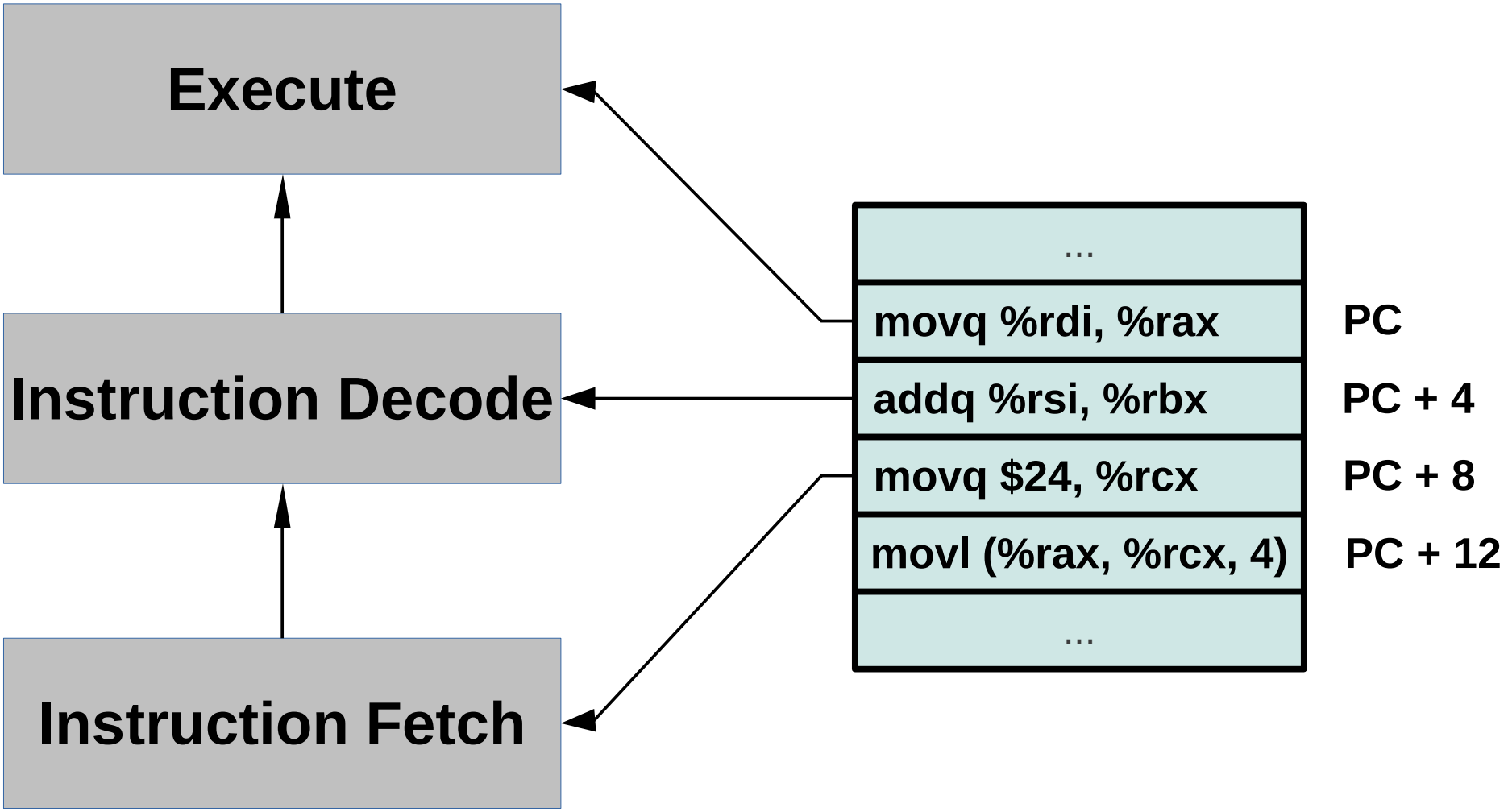
En el caso más sencillo puede separarse la ejecución de una instrucción en 3 partes:

Instruction Fetch: cargar de memoria la instrucción a ejecutar.

Instruction Decode: decodificar la instrucción para saber la operación a realizar y los operandos que se necesitan.

Execute: ejecutar la instrucción en la ALU.

Un pipeline sencillo

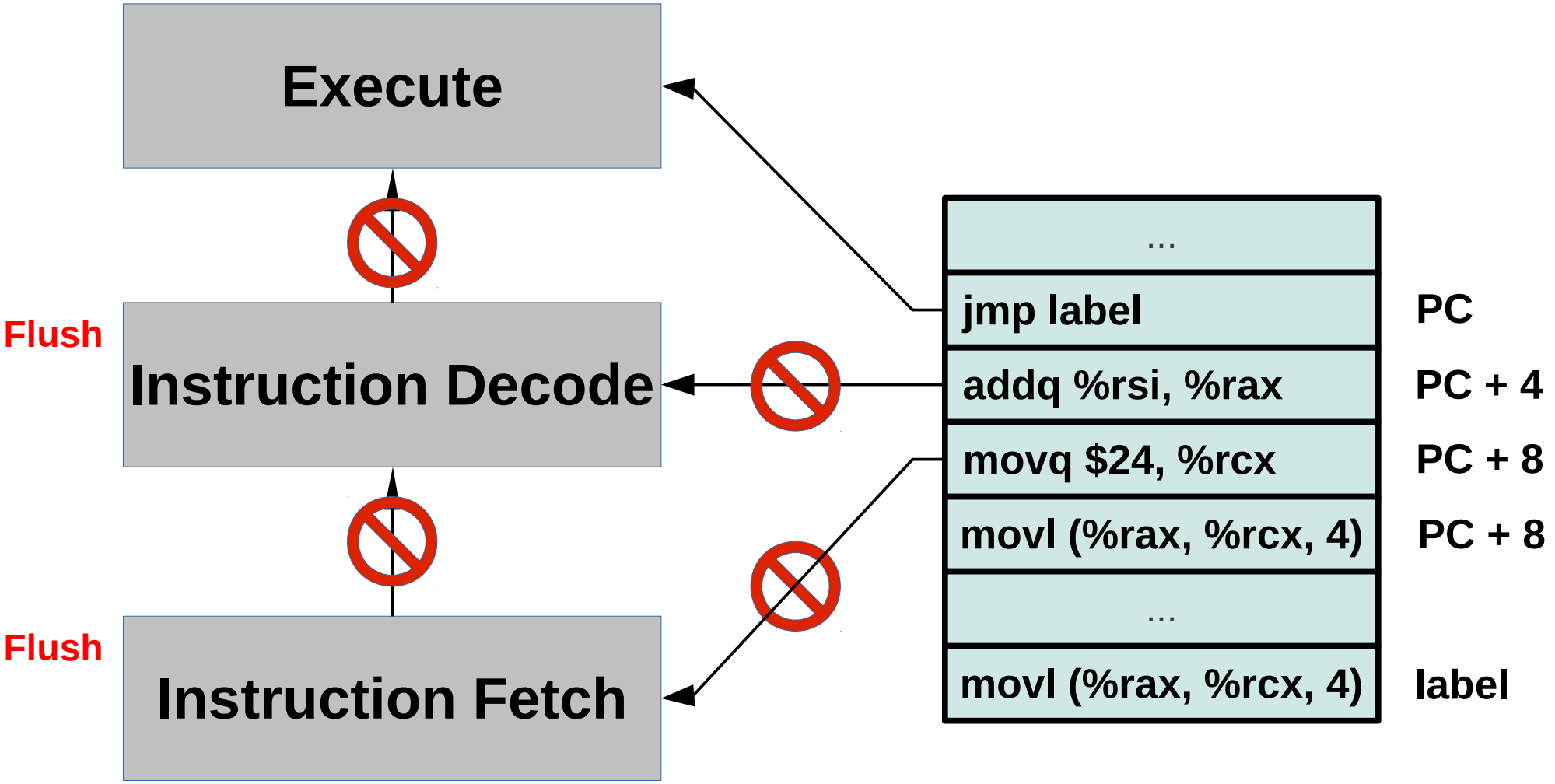


Problemas de este pipeline

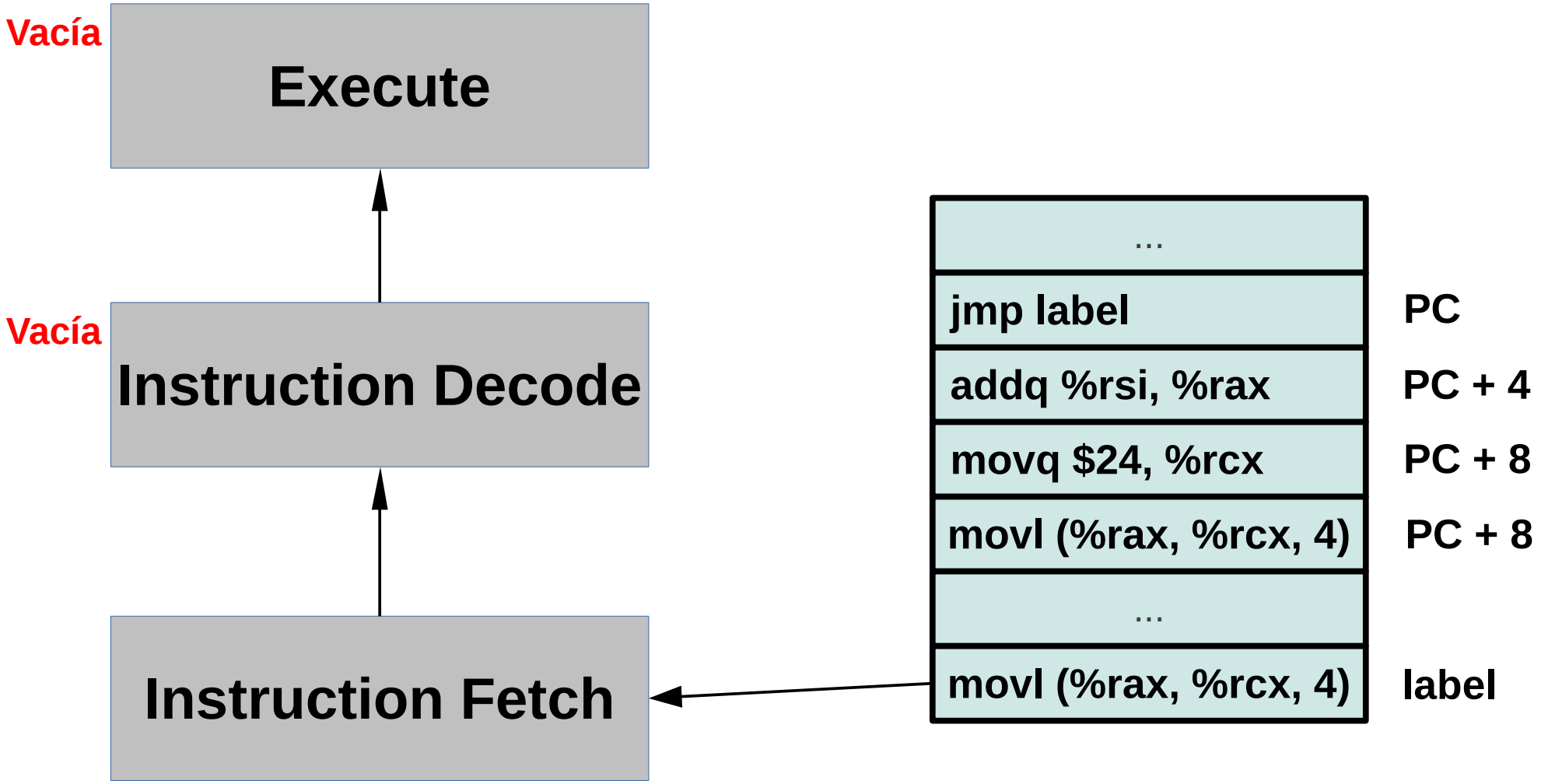
A pesar de que el pipeline tiene sólo 3 etapas, existen una serie de problemas que pueden darse:

- ¿ Qué sucede si hay un salto o interrupción ?
- ¿ Qué sucede si el resultado de la ejecución de una instrucción afecta un operando de la siguiente ?

Pipeline y saltos



Pipeline y saltos



Pipeline y saltos

Cuando se ejecuta un salto, se produce un **flush del pipeline**. Esto quiere decir que se vacían todas las etapas del pipeline dado que contienen información sobre instrucciones que no van a ejecutarse.

Luego del flush, se carga la próxima instrucción a ejecutar, y se empieza nuevamente a utilizar el pipeline.

El flush del pipeline naturalmente **introduce una demora** en el sistema, por lo cual es mejor tratar de evitar crear programas con muchos saltos.

Branch prediction

Una forma de eliminar los **flushes** es saber de antemano cuando un salto condicional va a ejecutarse y seguir la traza correcta.

Dado que eso es imposible (problema de la parada), lo mejor que se puede hacer es tratar de adivinar. Esto se llama '**branch prediction'**.

El procesador, a través de alguna heurística, toma la decisión de seguir alguna de las 2 ramas de un salto condicional. Si se equivoca, habrá un **flush.**

Branch prediction

Formas de predicción:

- Estática
- Basado en historia local de la instrucción
- Basado en historia global de saltos

- Muchas más ...

Un pipeline más complejo

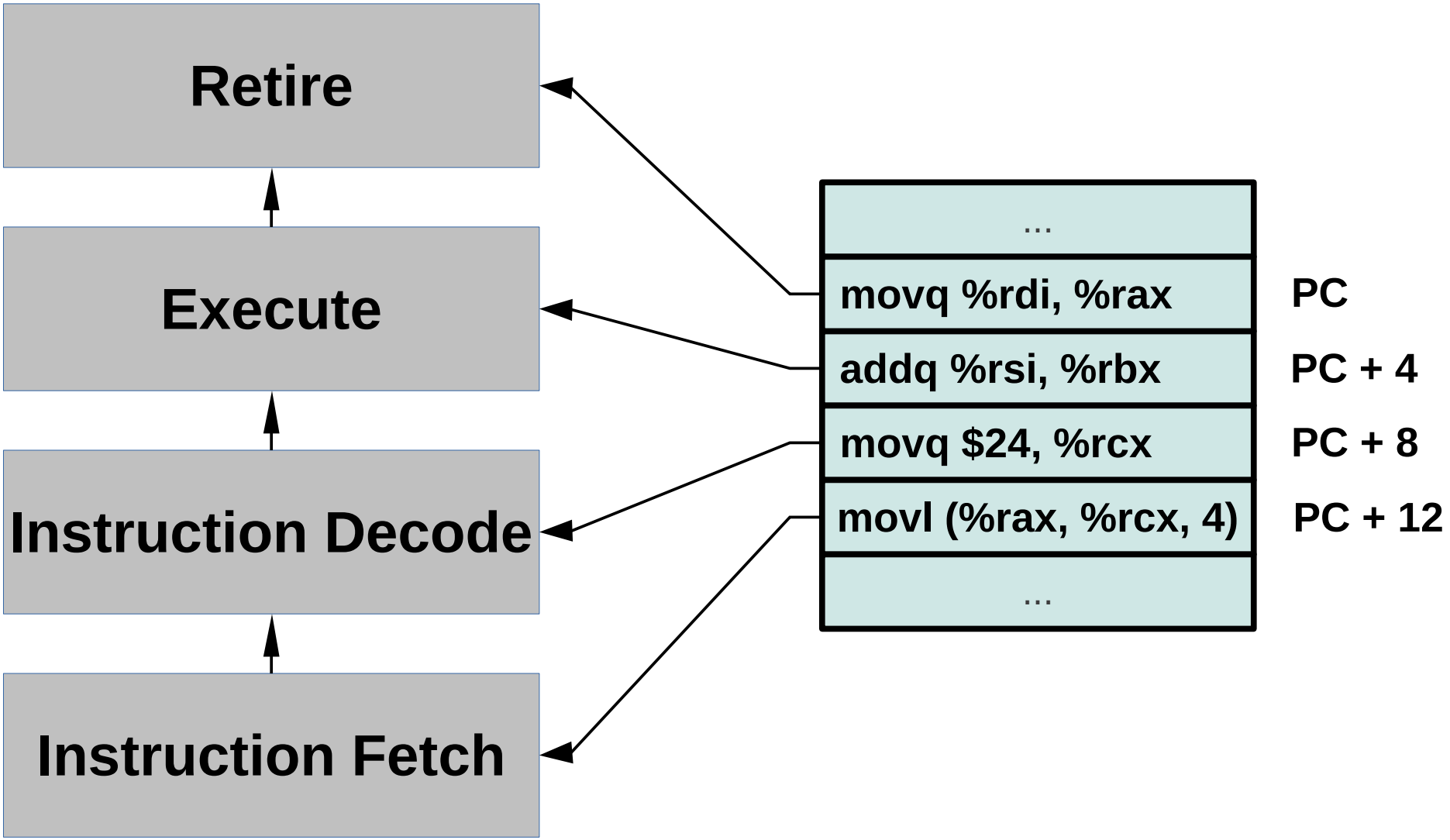
Para responder al resto de los problemas que planteamos, vamos a agregar una etapa más al pipeline básico que propusimos. Vamos a separar la etapa de execute en dos:

Execute: Ahora sólo ejecuta la instrucción en la ALU, pero no escribe el resultado en un registro o la memoria.

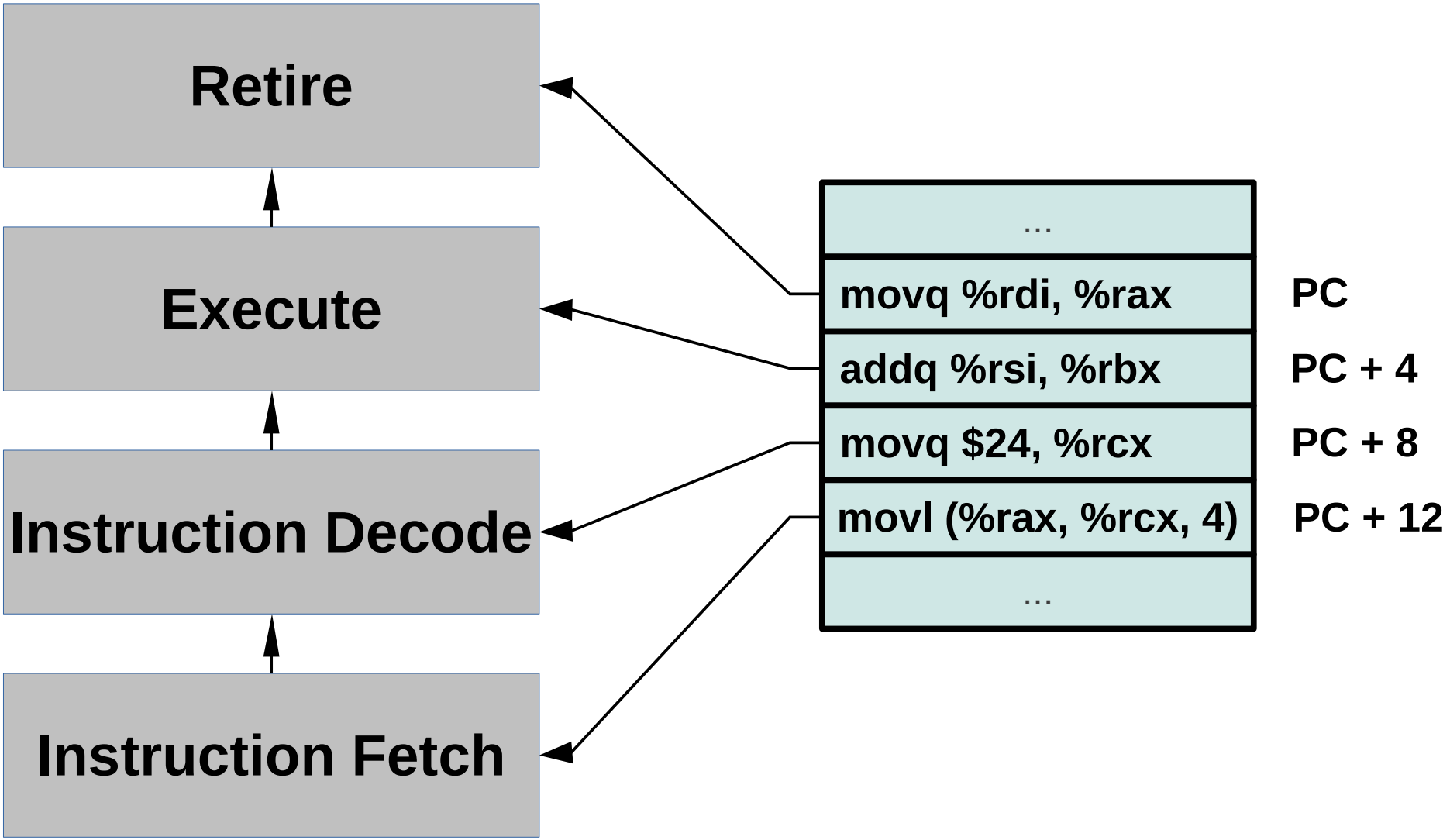
Retire: Esta etapa es la que copia el resultado en la ALU al registro o espacio de memoria correspondiente.

La velocidad de un CPU puede medirse por la cantidad de instrucciones que llegan a la etapa de retire por segundo.

Un pipeline más complejo



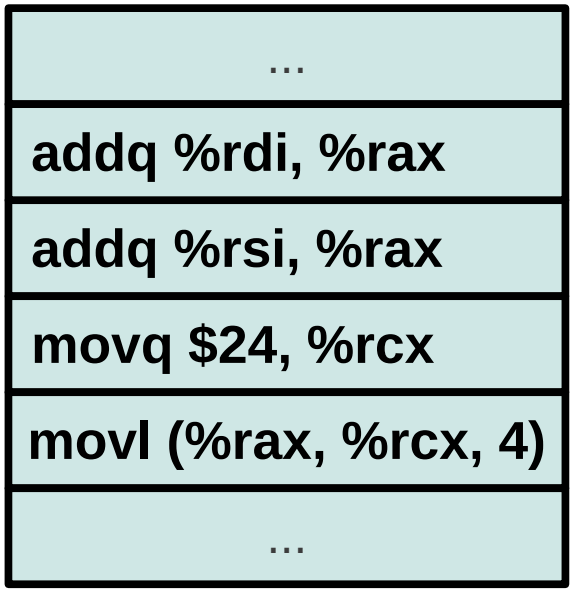
Un pipeline más complejo



Pipeline stalls

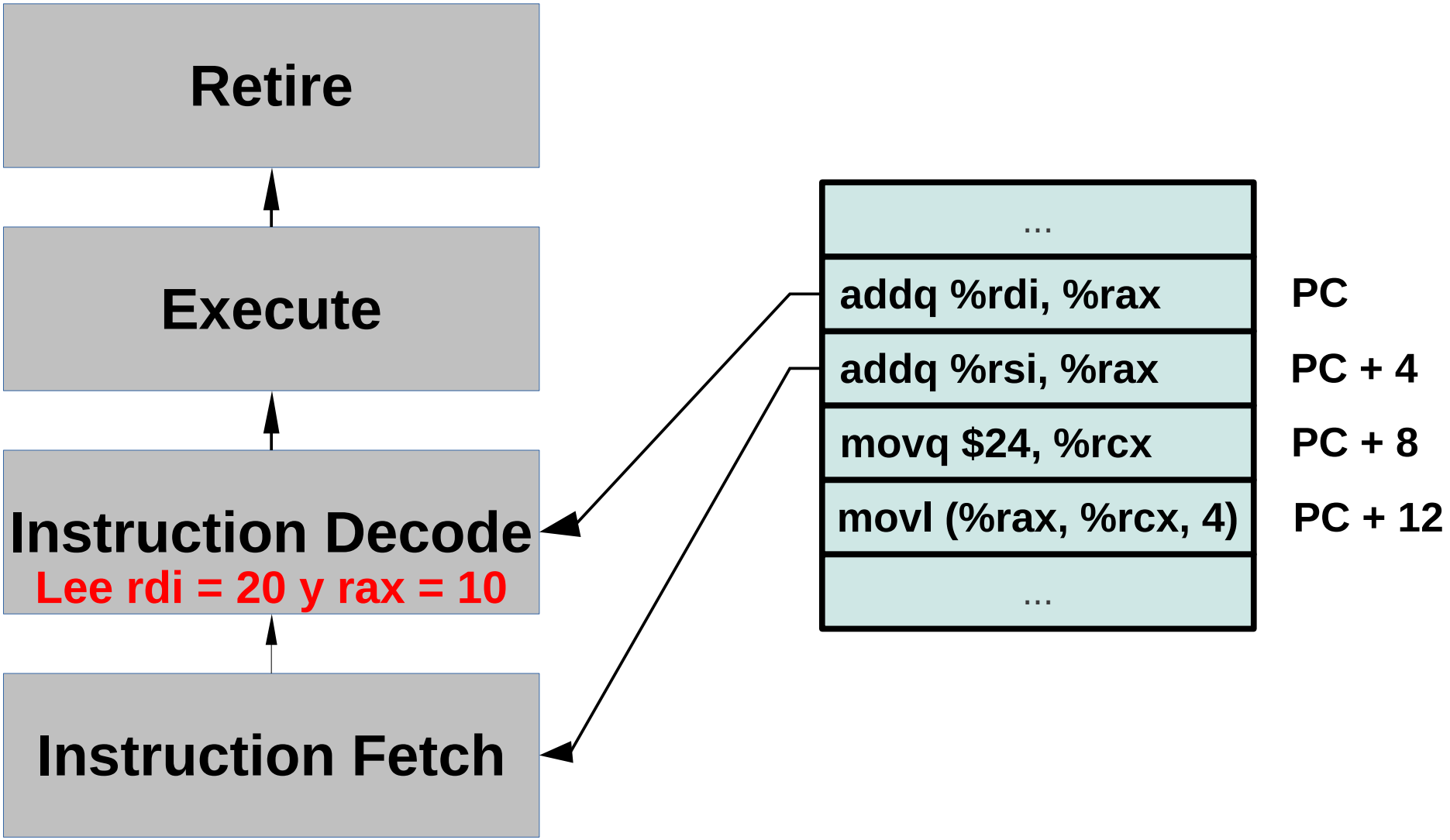
Ahora vamos a analizar que sucede si una instrucción escribe un registro o parte de memoria que necesita la próxima instrucción.

Pensemos en el pedazo de programa:

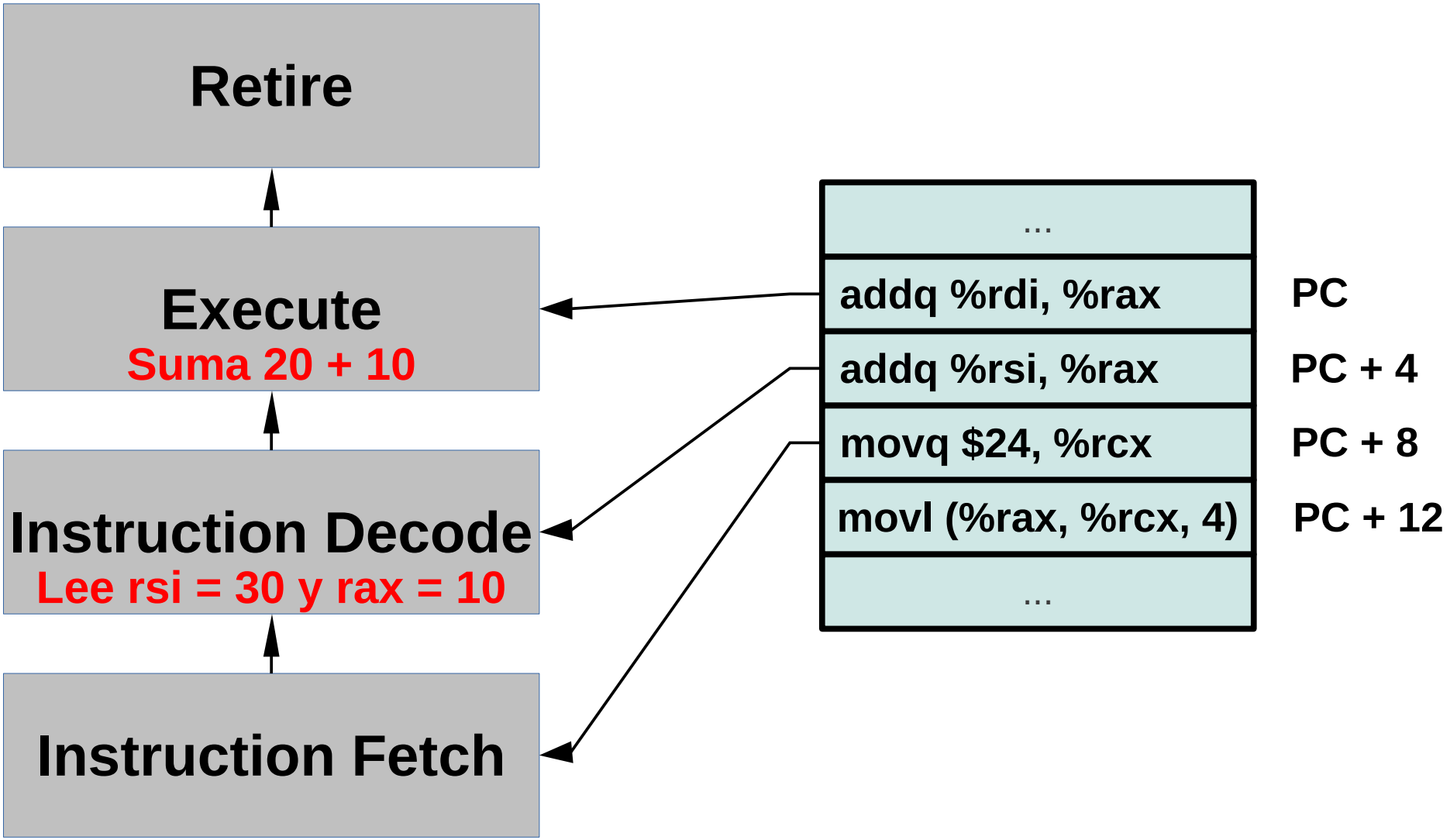


La primera instrucción escribe rax y la segunda instrucción la utiliza. Asumimos que rax, rdi y rsi tienen el valor 10, 20 y 30 respectivamente antes de empezar la ejecución.

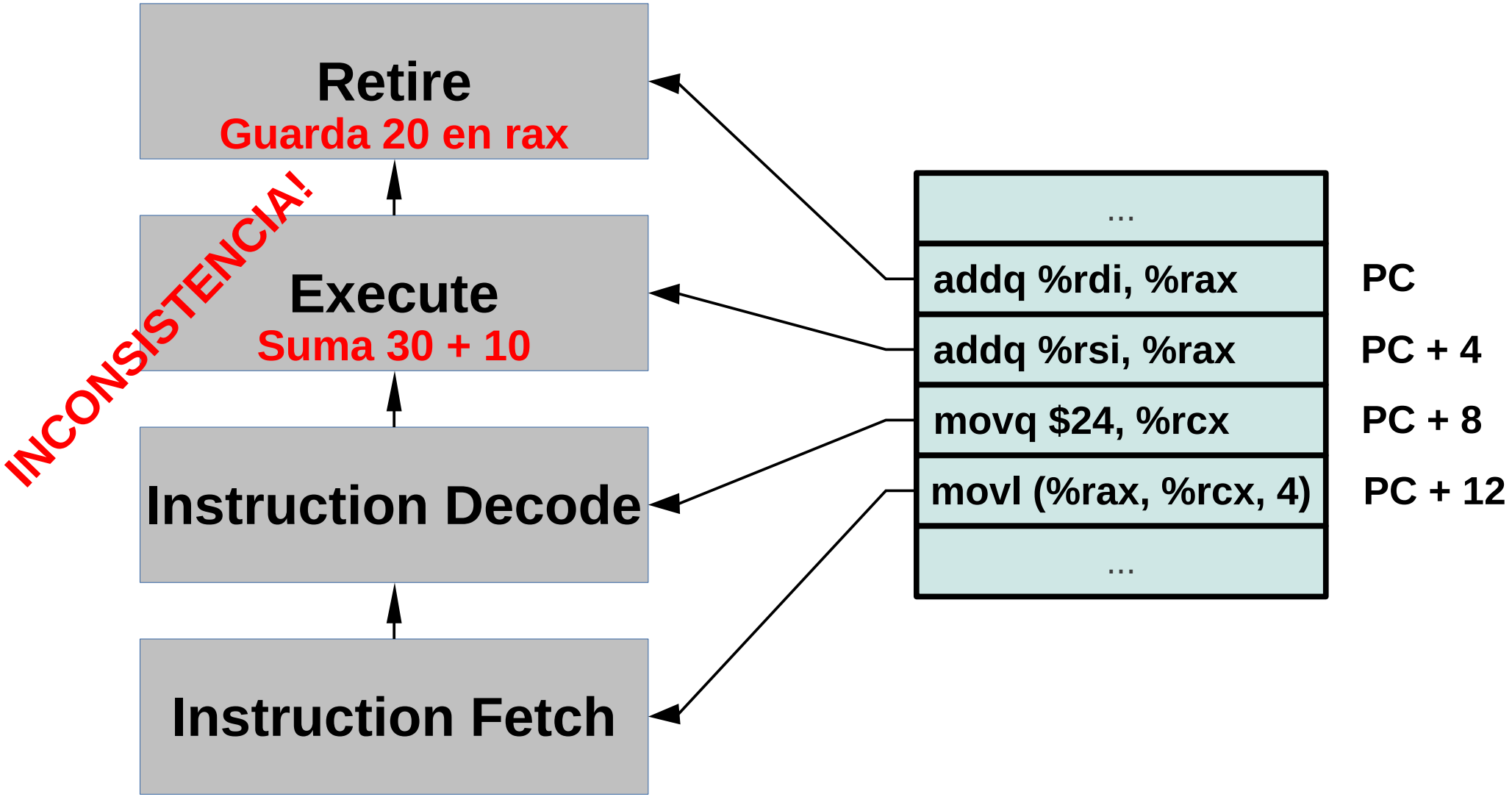
Pipeline stalls



Pipeline stalls



Pipeline stalls



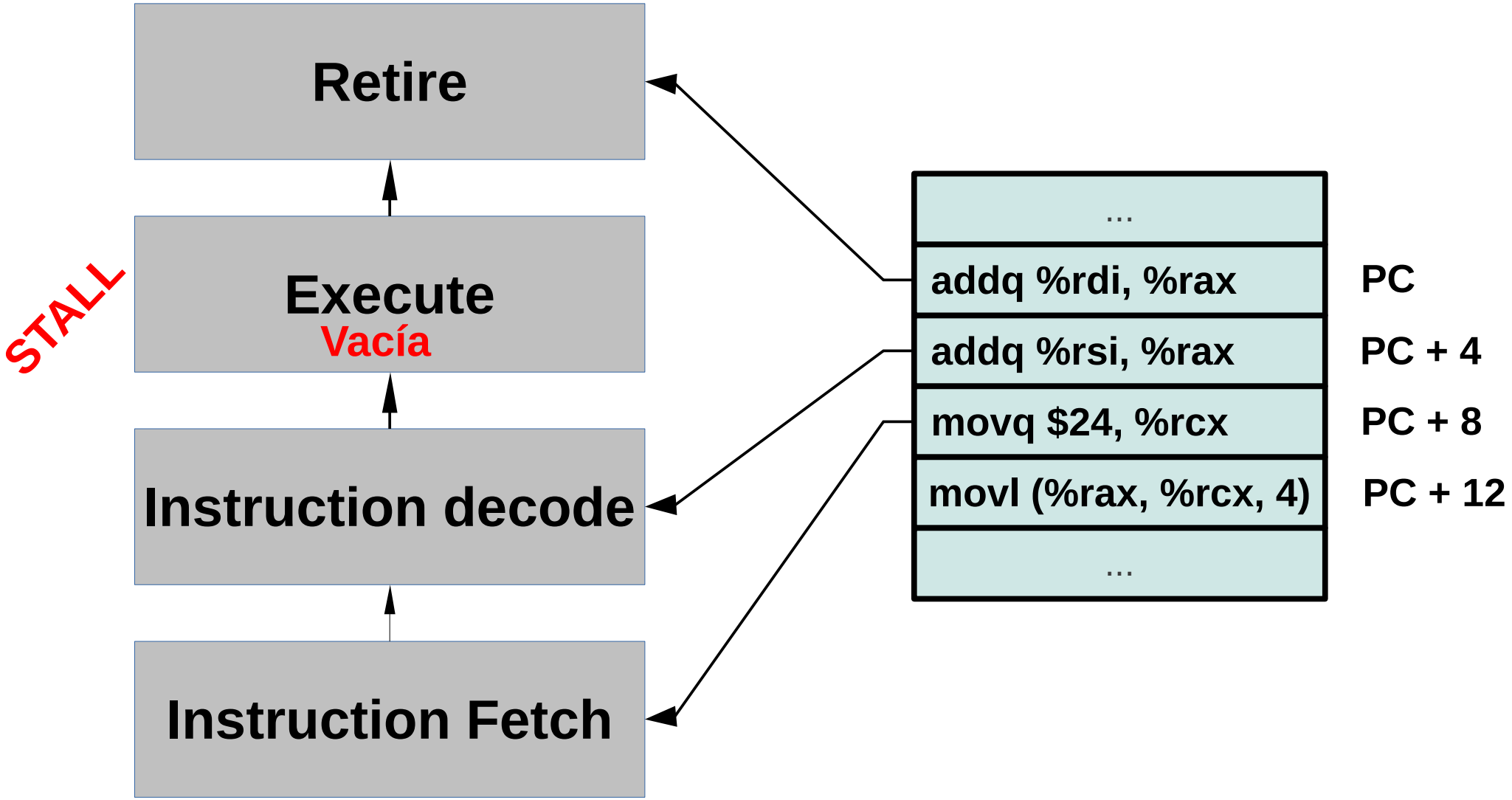
Pipeline stalls

¿ Cuando puede detectarse este caso ?

- Cuando se decodifica una instrucción se puede saber si depende del resultado de un instrucción anterior.

Si es así, se 'paraliza' (**stall**) el pipeline hasta que se retira la instrucción de la cual depende esta instrucción.

Pipeline stalls



Pipeline stalls

Al ocurrir un **stall** la instrucción que estaba siendo decodificada permanece en esa parte del pipeline esperando que sus argumentos estén listos luego que la instrucción precedente termine su paso por el pipeline.

Al igual que el caso de jumps, ejecutar instrucciones que causan stalls ralentizan la ejecución de un programa.

Out of Order Execution

¿ Cómo mejorar un poco el problema de **stalls** ?

En el momento que se detecta una dependencia de argumentos que va a causar un **stall**, puede que una instrucción siguiente no tenga problemas de dependecias. En este caso se ejecutará la instrucción independiente mientras se espera que se resuelva la dependencia de argumentos.

Esta técnica se denomina 'Out of order execution' (**OOO**), y es común en procesadores modernos.

A pesar de que las instrucciones se ejecutan fuera de orden, la etapa de '**retire**' se ejecuta en el orden establecido por el código.

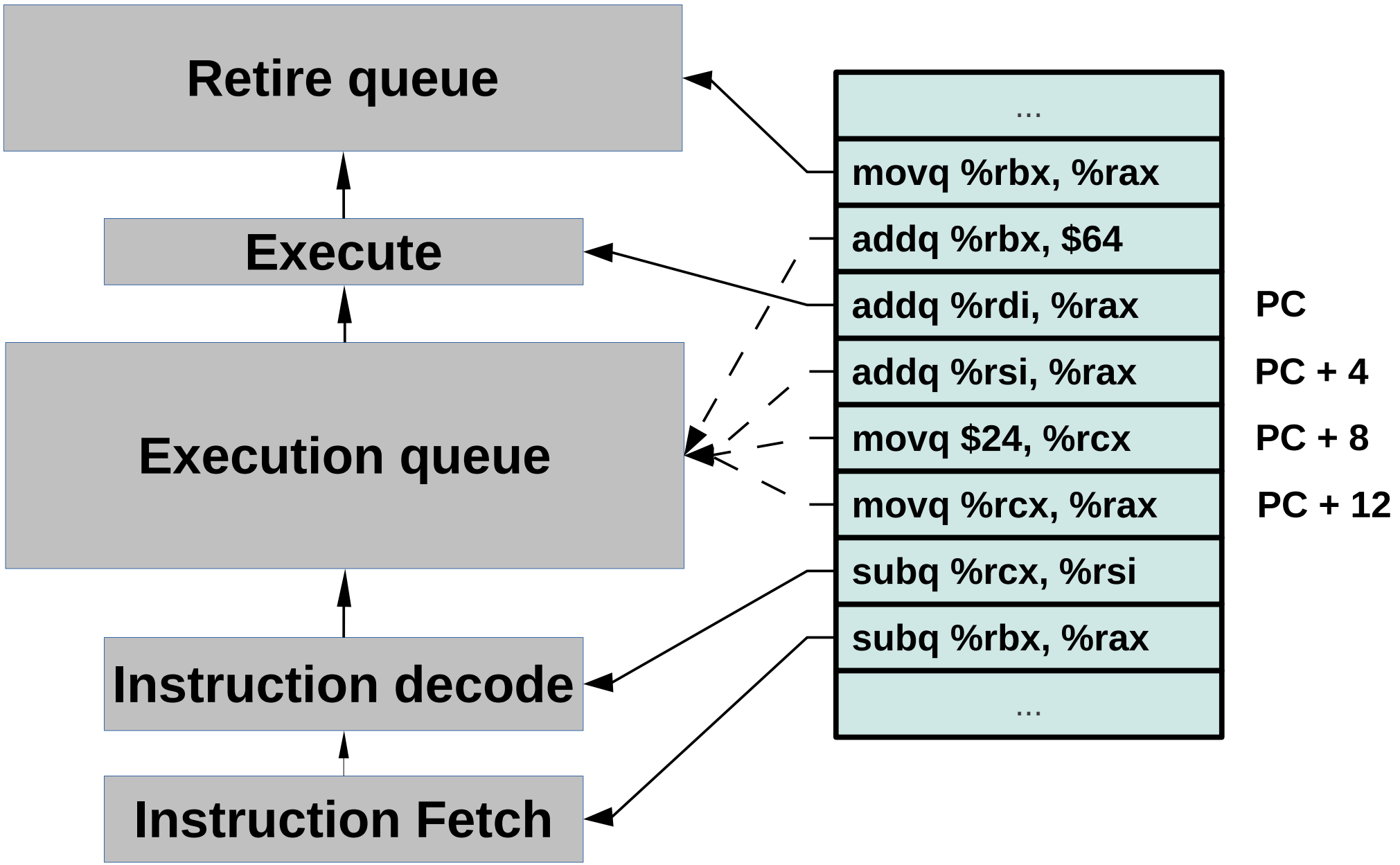
Out of Order Execution

¿ Cómo se implementa OOO ?

Para implementar OOO el procesador usa colas de espera para las instrucciones, ejecutándolas a medida que sus operandos son 'seguros' para usar.

Luego de la etapa de decodificación, cuando ya se sabe que argumentos necesita cada instrucción, se van pasando a la etapa de ejecución.

Out of Order Execution



Algunas características de pipelines modernos

En un CPU moderno, los pipelines tienen muchas más etapas que nuestro pipeline de ejemplo (hasta 30+, aunque comunmente entre 8 y 15).

Otra característica son los pipelines superescalares, que pueden ejecutar una misma etapa para 2 o más instrucciones al mismo tiempo.

Algunas características de pipelines modernos

Los CPU CISC de Intel tienen normalmente entre 13 y 15 etapas en sus pipelines. En CPUs de años atras han tenido hasta 30 etapas, pero la complejidad y el uso de energía los hace ineficientes.

Los CPU de ARM más viejos (hasta ARM-v3) tenían pipelines simples de 3 etapas. Los procesadores modernos (ARM-v7) tienen entre 7 y 24 etapas.