

Paralelismo a nivel de instrucciones

Arquitectura de Computadores – IIC2343

La CPU ejecuta cada instrucción en una serie de pasos —el ciclo ***fetch-decode-execute***, central a la operación de todos los computadores:

1. Traer (*fetch*) la próxima instrucción desde la memoria al *instruction register*
2. Cambiar el *program counter* para que apunte a la instrucción subsiguiente
3. Determinar el tipo de la instrucción recién traída
4. Si la instrucción usa una palabra en memoria, determinar dónde está
5. Traer la palabra, si es necesario, a un registro de la CPU
6. Ejecutar la instrucción
7. Ir al paso 1 para comenzar la ejecución de la próxima instrucción

Principios de diseño *RISC* para computadores modernos

Todas las instrucciones son ejecutadas directamente por el hardware

Hay que maximizar la tasa a la cual se inicia la ejecución de instrucciones (pasos 1 y 2)

Las instrucciones deben ser fáciles de decodificar (paso 3)

Solo los *loads* y *stores* deberían hacer referencia a la memoria (pasos 4 y 5)

Hay que tener muchos registros

Los arquitectos computacionales están siempre tratando de mejorar el desempeño de los computadores

Una posibilidad: aumentar la velocidad del reloj —pero para cada diseño hay un límite a lo que se puede lograr por esta vía

Otra: **paralelismo** —hacer dos o más cosas al mismo tiempo:

- **paralelismo a nivel de instrucciones** —qué se puede hacer dentro de las instrucciones individuales para obtener más instrucciones ejecutadas por segundo
- paralelismo a nivel de procesadores —múltiples CPUs trabajan juntas en el mismo problema

Traer instrucciones desde la memoria (paso 1) es un importante cuello de botella para la velocidad de ejecución de las instrucciones:

- desde hace 60 años los computadores tienen la capacidad de traer instrucciones desde la memoria por adelantado y almacenarlas en un registro especial
- esto divide la ejecución de la instrucción en dos partes: *fetching* y la ejecución misma

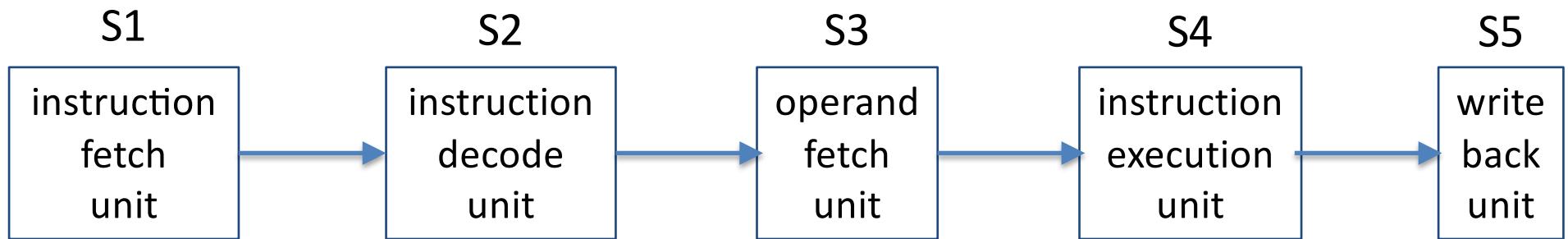
Un **pipeline*** lleva esta estrategia más allá:

- la ejecución de una instrucción es dividida en varias partes —o **etapas**
- ... cada una manejada por una pieza de hardware (una *unidad*) dedicada
- ... todas las cuales pueden correr en paralelo

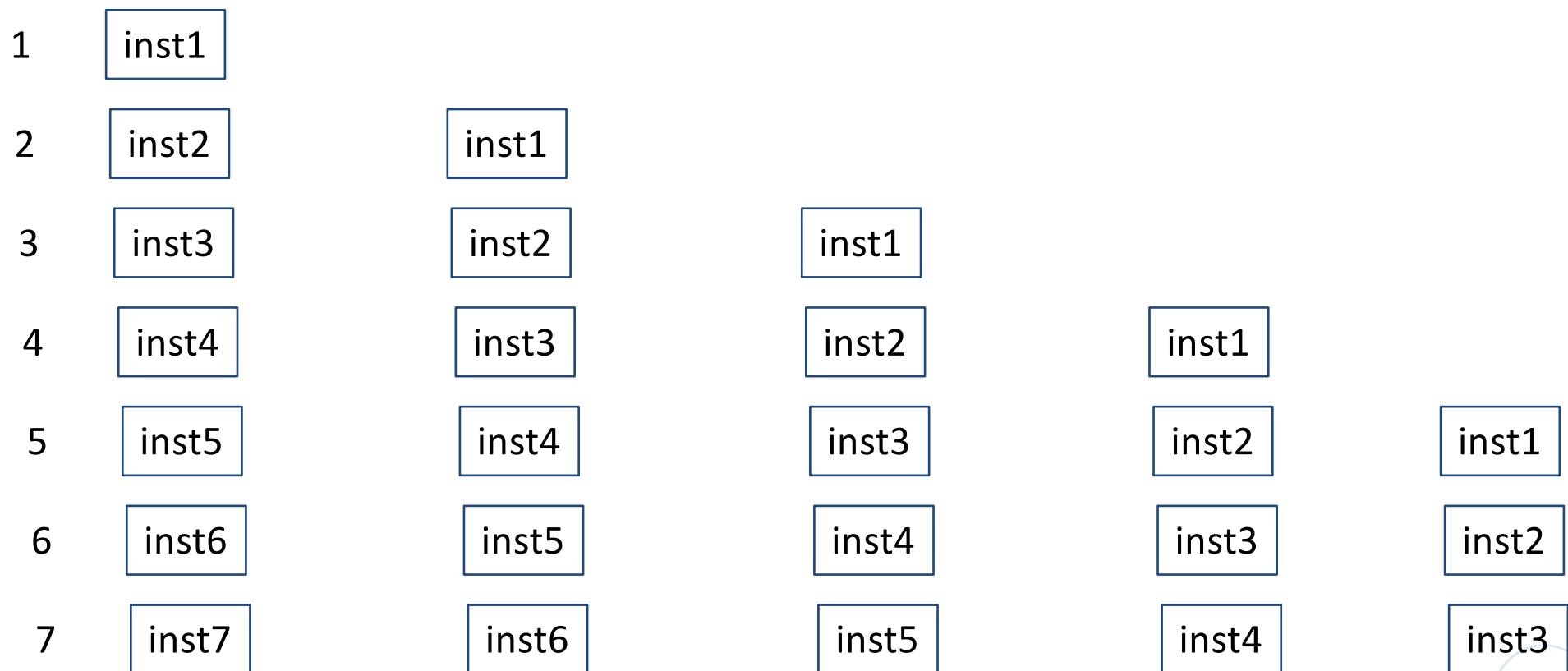
Pipelining: Técnica de implementación en la que las ejecuciones de múltiples instrucciones son traslapadas

... de modo que en un mismo ciclo del reloj las distintas instrucciones ocupan distintas unidades del pipeline

**Pipeline*: Proceso secuencial de varias etapas independientes



time



Para concretar un poco más las ideas, volvamos al computador básico

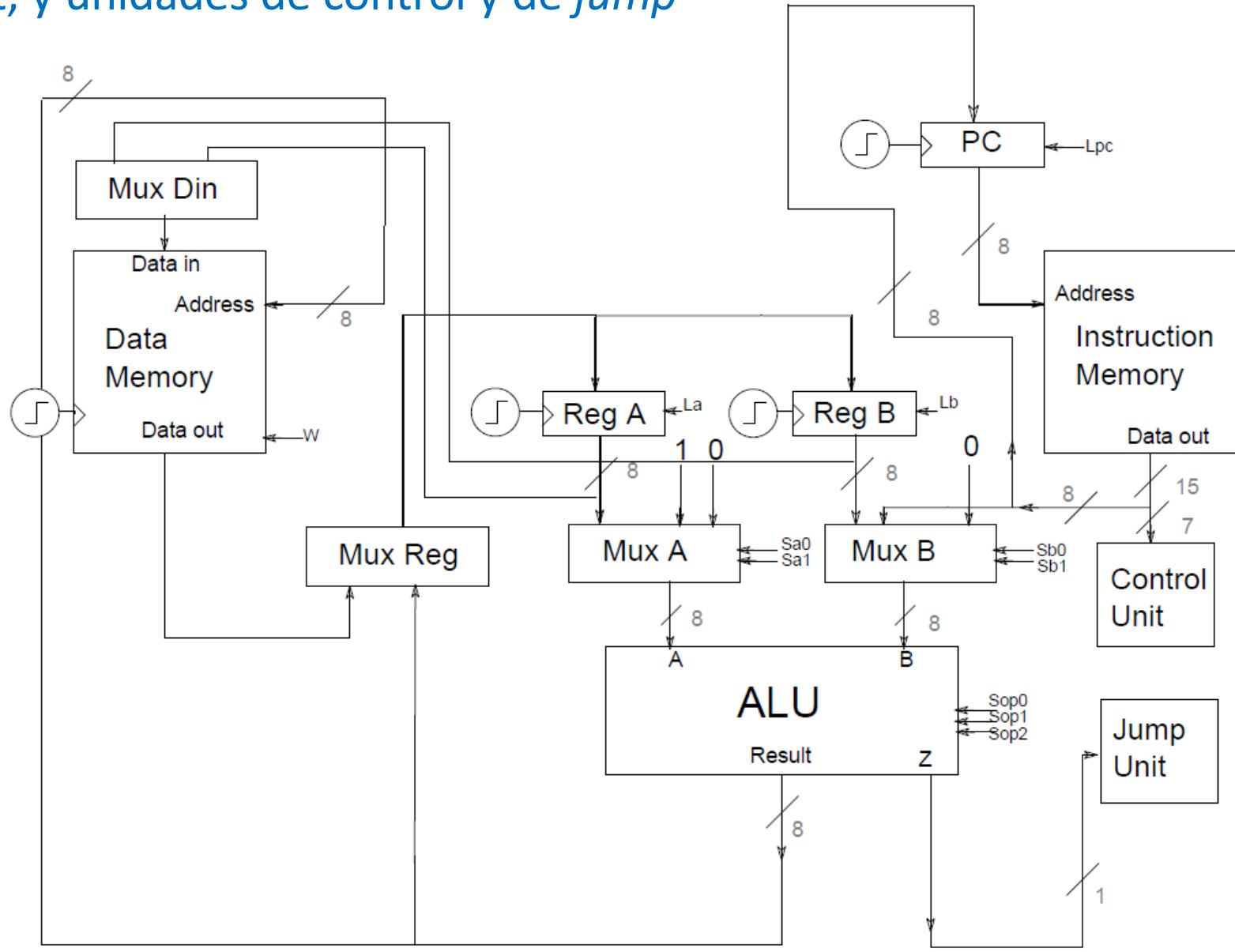
Consideremos las siguientes tres instrucciones y su división en etapas:

ADD A, B	MOV A, (B)	MOV (B), A
1. traer instrucción	1. traer instrucción	1. traer instrucción
2. decodificar instrucción	2. decodificar instrucción	2. decodificar instrucción
3. ejecutar en ALU	3. acceder a memoria	3. acceder a memoria
4. escribir en registro	4. escribir en registro	

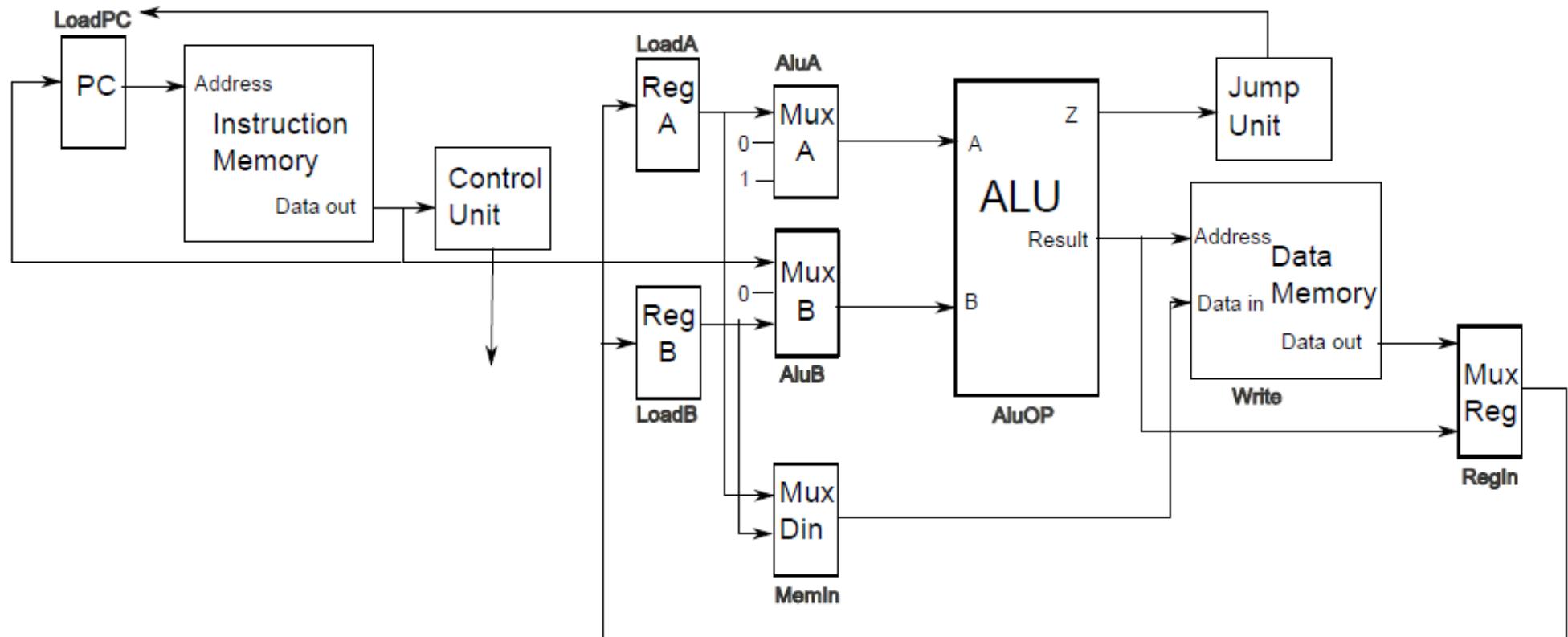
Supongamos entonces un pipeline de 5 etapas:

1. traer instrucción desde memoria —*instruction fetch (IF)*
2. decodificar instrucción en unidad de control —*instruction decode (ID)*
3. ejecutar instrucción en ALU —*execute (EX)*
4. tener acceso a memoria —*memory (MEM)*
5. escribir resultado en registro —*writeback (WB)*

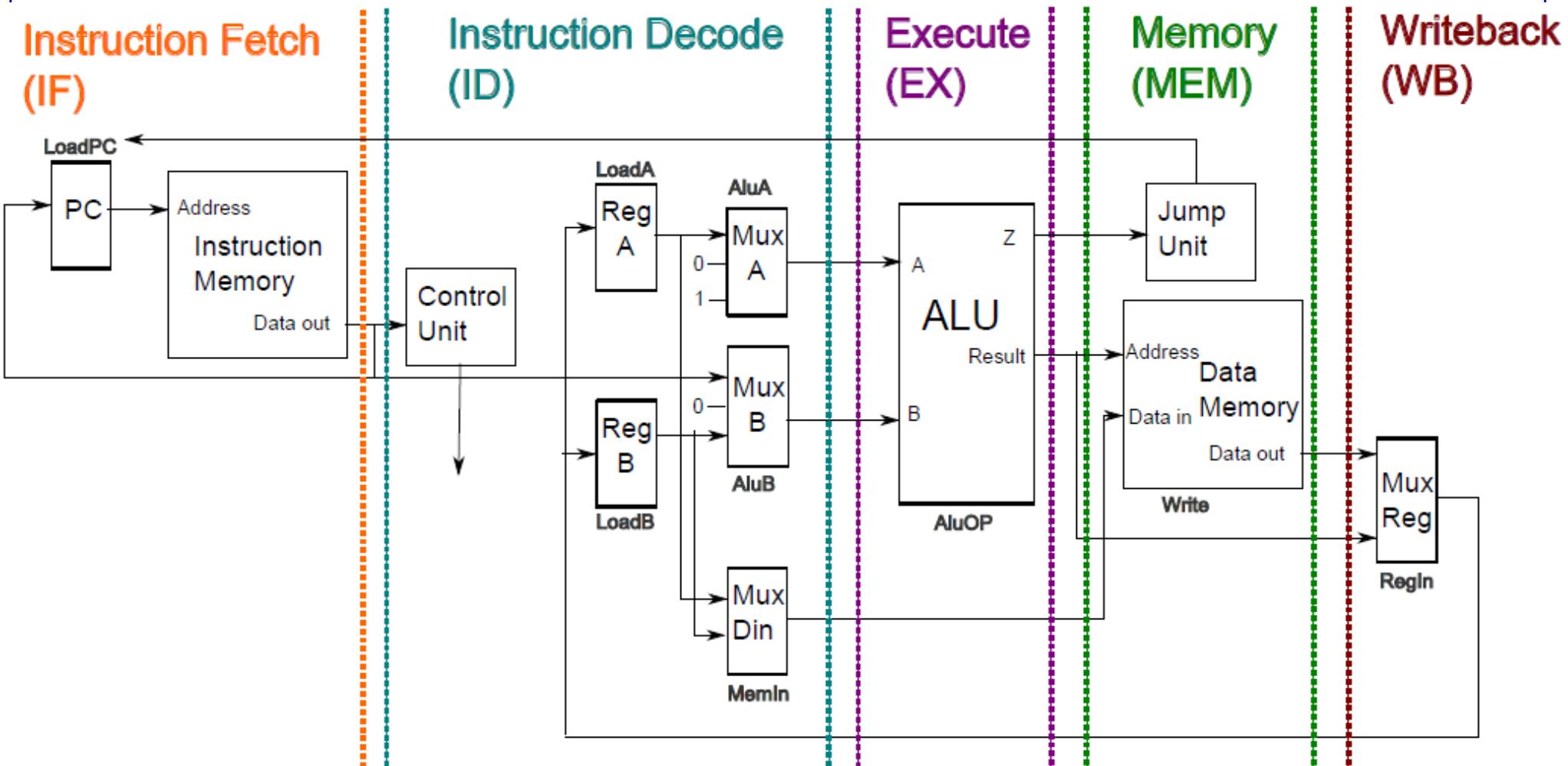
Tomemos una versión simplificada del computador básico: memoria de datos, memoria de instrucciones y *PC*, ALU y registros de input y output, y unidades de control y de *jump*



Dispongamos los componentes de izquierda a derecha en el orden en que participan en la ejecución de una instrucción

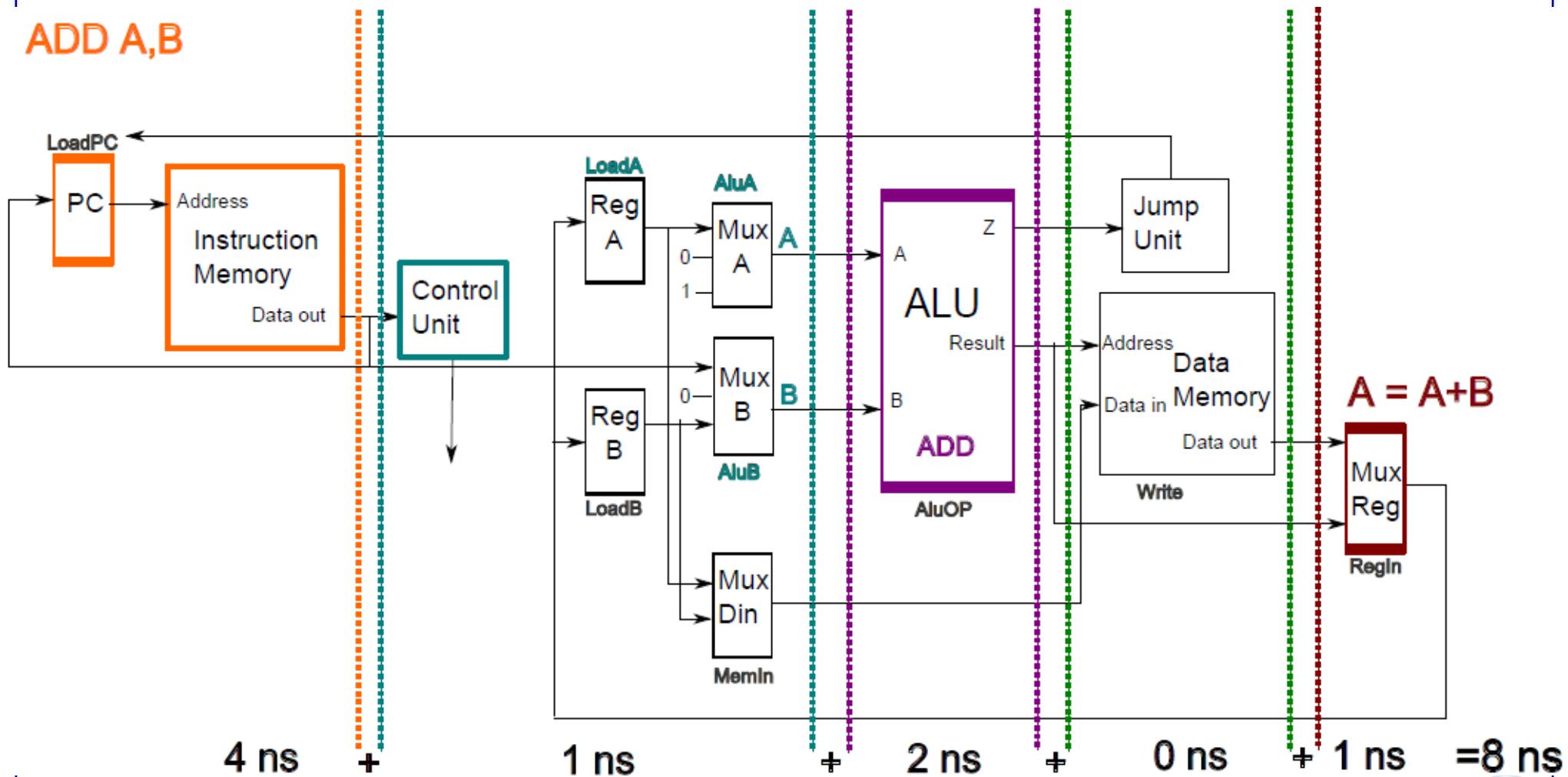


Agrupemos los componentes según cada una de las 5 etapas en que dividimos la ejecución de una instrucción



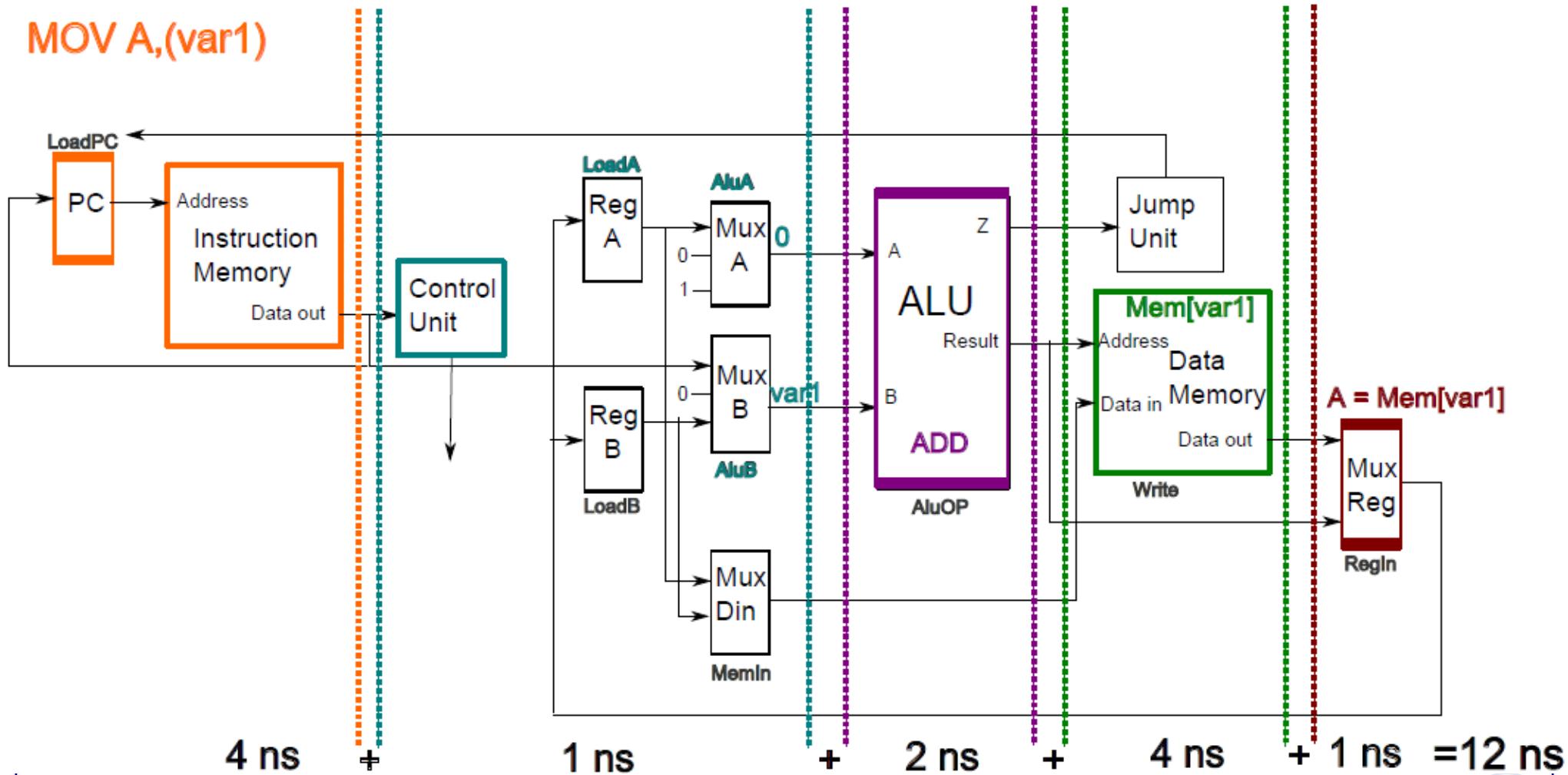
Etapas que participan en la ejecución de ADD A, B, y sus duraciones

ADD A,B

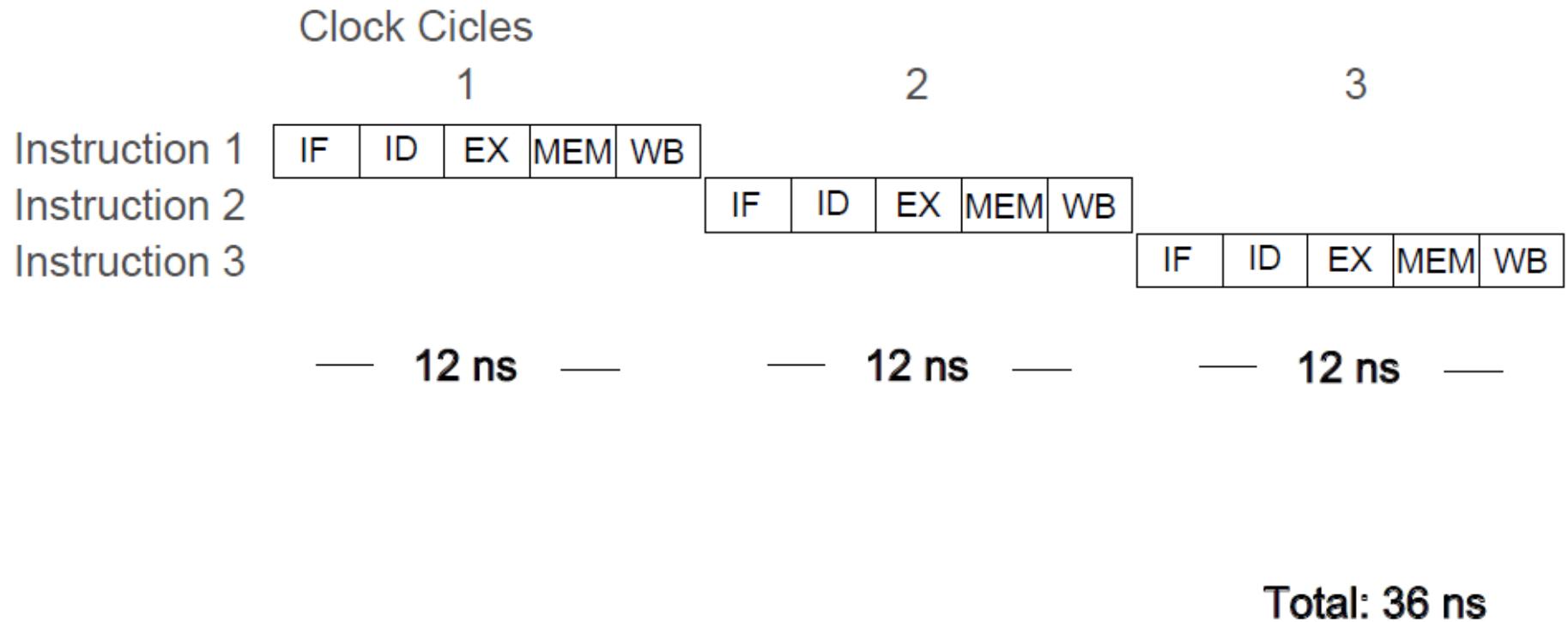


Etapas que participan en la ejecución de MOV A, (var1), y sus duraciones

MOV A,(var1)



Si simplemente ejecutamos una instrucción tras otra, entonces el ciclo del reloj no puede ser más rápido que la **instrucción más lenta**



En cambio, si aplicamos *pipelining* y traslapamos las ejecuciones de varias instrucciones, de modo que cada una ejecute una etapa distinta, el ciclo del reloj puede ser tan rápido como la **etapa más lenta**:

- ahora, todas las etapas deben durar lo mismo que la etapa más lenta
- ... alargando la duración de la ejecución de cada instrucción individual
- ... pero **disminuyendo el tiempo total para la ejecución del conjunto de instrucciones**

Cicles										
	1	2	3	4	5	6	7	8	9	10
Instruction 1	IF	ID	EX	MEM	WB					
Instruction 2		IF	ID	EX	MEM	WB				
Instruction 3			IF	ID	EX	MEM	WB			
	4ns									

Total: 28 ns

La disminución del tiempo total es más significativa mientras mayor sea el número total de instrucciones:

- un millón de instrucciones sin pipelining demora 12 millones ns
- **un millón de instrucciones con pipelining demora 4 millones ns**

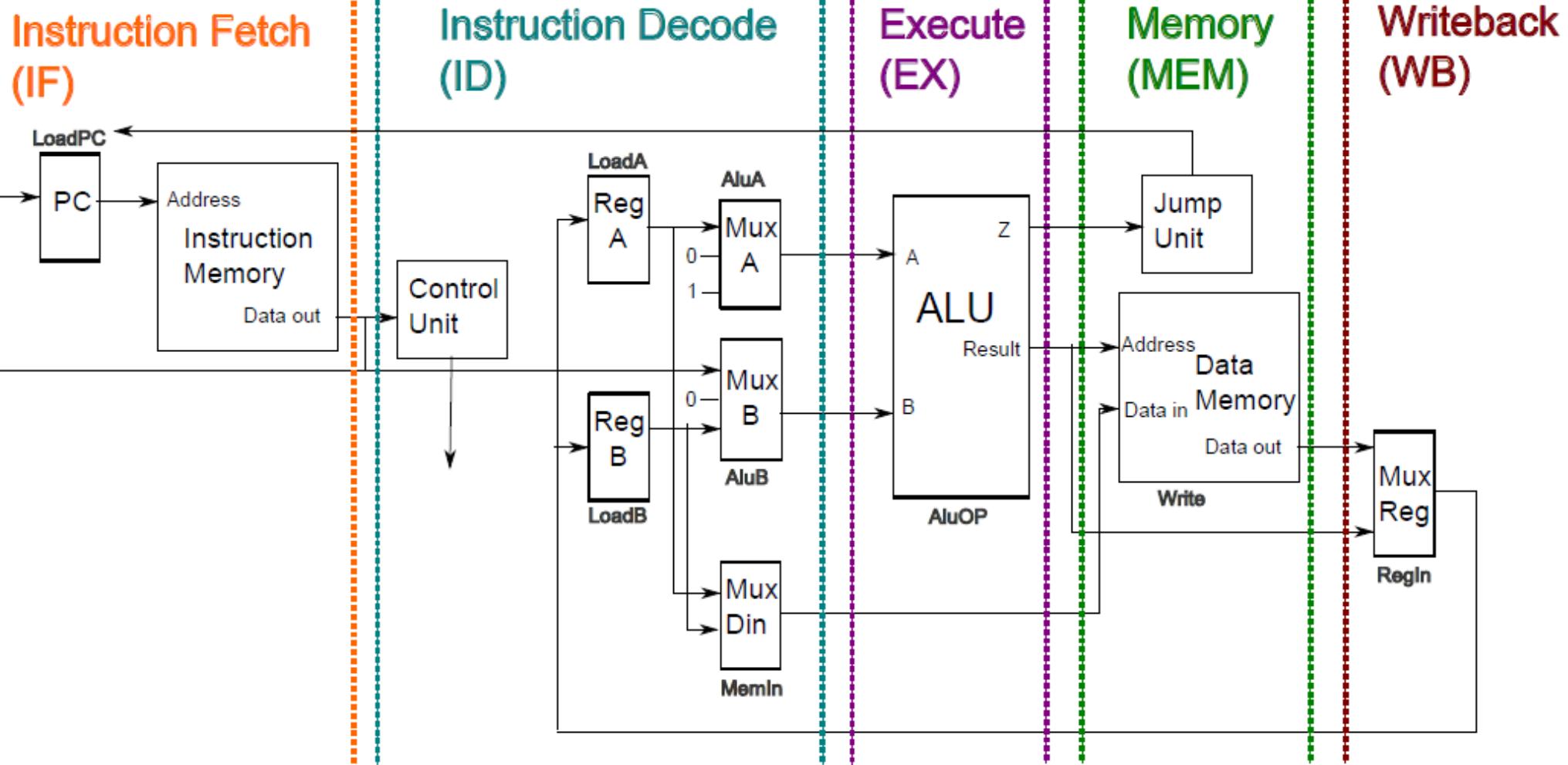
	Cicles				
	1	2	3	4	5
Instruction 1	IF	ID	EX	MEM	WB
Instruction 2		IF	ID	EX	MEM
Instruction 3			IF	ID	EX
Instruction 4				IF	ID
Instruction 5					IF
Instruction 6					

Pipelining permite un compromiso entre

... **latencia** (cuánto toma ejecutar una instrucción)

... y **ancho de banda del procesador** (cuántos MIPS tiene la CPU):

- en un pipeline de 5 etapas
 - ... si la duración del ciclo es 2 ns
 - ... entonces una instrucción toma 10 ns para pasar por todo el pipeline
 - parecería que el computador corre a 100 MIPS
 - ... pero en realidad como cada 2 ns se termina de ejecutar una nueva instrucción
 - ... la tasa de procesamiento es 500 MIPS



Como vemos, las instrucciones y los datos se mueven de izquierda a derecha a lo largo de las cinco etapas

... excepto (las dos flechas que apuntan de derecha a izquierda):

- la salida de la etapa *WB* escribe el resultado de vuelta en los registros
- la salida de la *Jump Unit* escribe la dirección de la próxima instrucción en el *PC*

Estos dos casos no afectan la ejecución de la instrucción vigente

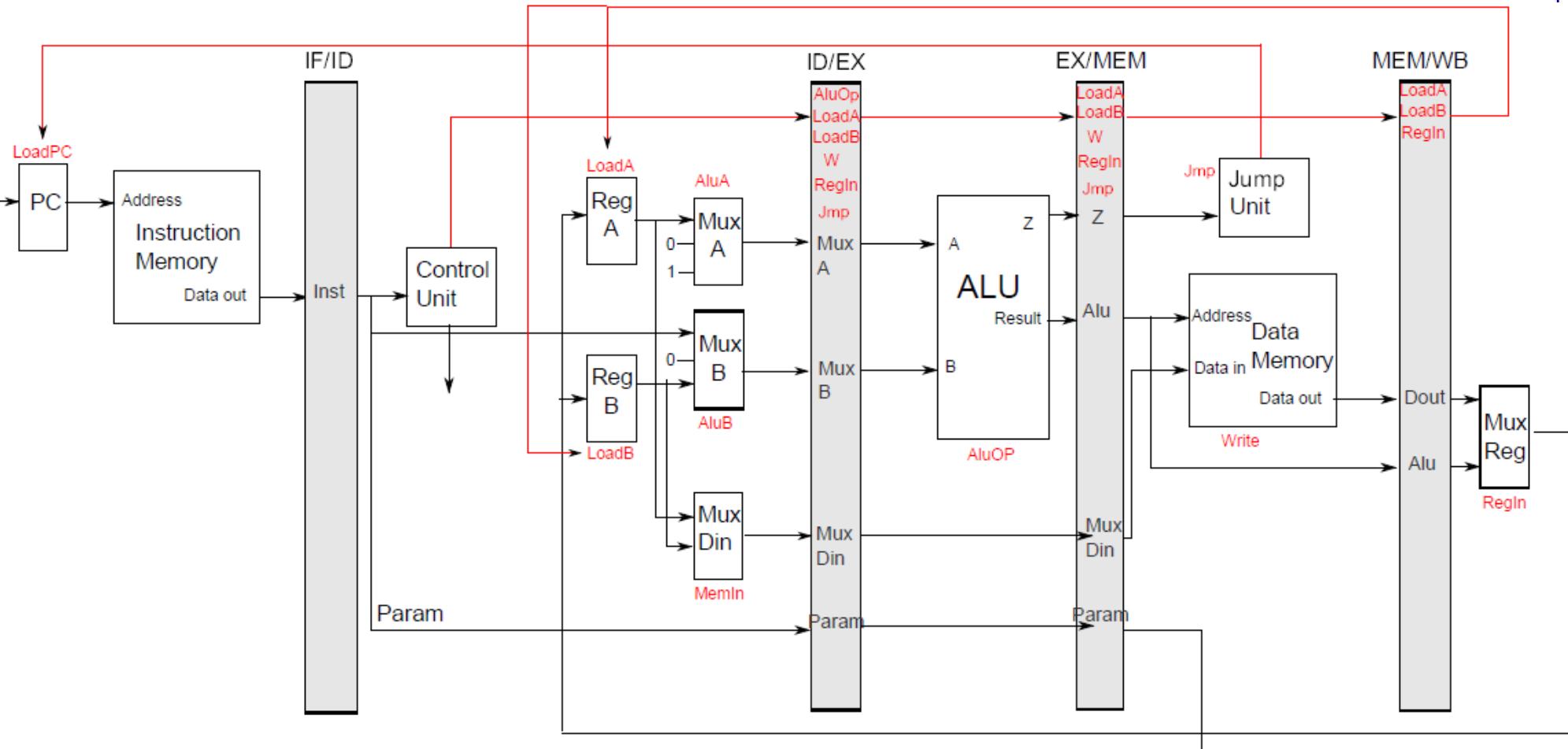
... pero sí pueden afectar a instrucciones posteriores en el pipeline
(como veremos luego) —respectivamente:

- *hazards* de datos
- *hazards* de control

Para que una misma unidad pueda ser usada (compartida) por varias instrucciones (a medida que van avanzando por el pipeline)

... agregamos registros especializados entre etapas consecutivas:

- **IF/ID**: almacena la instrucción vigente (leída desde la memoria) y la dirección de la próxima instrucción (calculada como $PC+4$, ya escrita también en el PC)
- **ID/EX**: almacena los valores de los dos registros, y un posible *offset* constante, especificados en los operandos de la instrucción, y también la dirección $PC+4$
- **EX/MEM**: almacena el resultado de la operación ejecutada en la ALU, el que puede ser una dirección de memoria, y posiblemente el valor de uno de los registros
- **MEM/WB**: almacena el dato leído desde la memoria (y que posiblemente va a ser escrito en alguno de los registros)



También es necesario actualizar las señales de control (líneas rojas)

Extendemos los registros especializados para que puedan almacenar información de control:

- durante la lectura y decodificación de la instrucción (*IF* e *ID*) no es necesario controlar nada en particular: estas etapas ejecutan siempre lo mismo, ya que aún no se sabe cuál es la instrucción
- creamos la información de control durante la decodificación de la instrucción (*ID*) y la almacenamos en *ID/EX*
- algunas de estas líneas son usadas durante la ejecución de la instrucción (*EX*), mientras que las otras son almacenadas en *EX/MEM*
- similarmente, algunas de estas últimas líneas son usadas en el acceso a memoria (*MEM*), mientras que las restantes son almacenadas en *MEM/WB*, para ser usadas en el write back (*WB*)

En *pipelining* hay situaciones en que la próxima instrucción no se puede ejecutar en el siguiente ciclo de reloj —***hazards***:

Hazards estructurales:

- el hardware no permite la combinación de instrucciones que queremos ejecutar → dos instrucciones en diferentes etapas de su ejecución necesitan usar la misma unidad del pipeline en el mismo ciclo de reloj

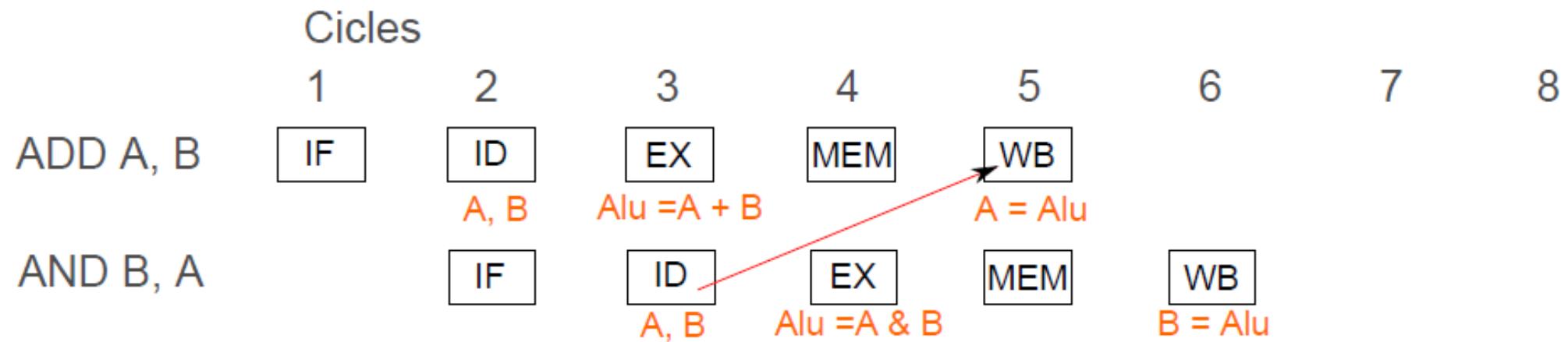
Hazards de datos:

- una instrucción no puede ejecutarse en el ciclo de reloj que le corresponde porque el dato que necesita para su ejecución aún no está disponible

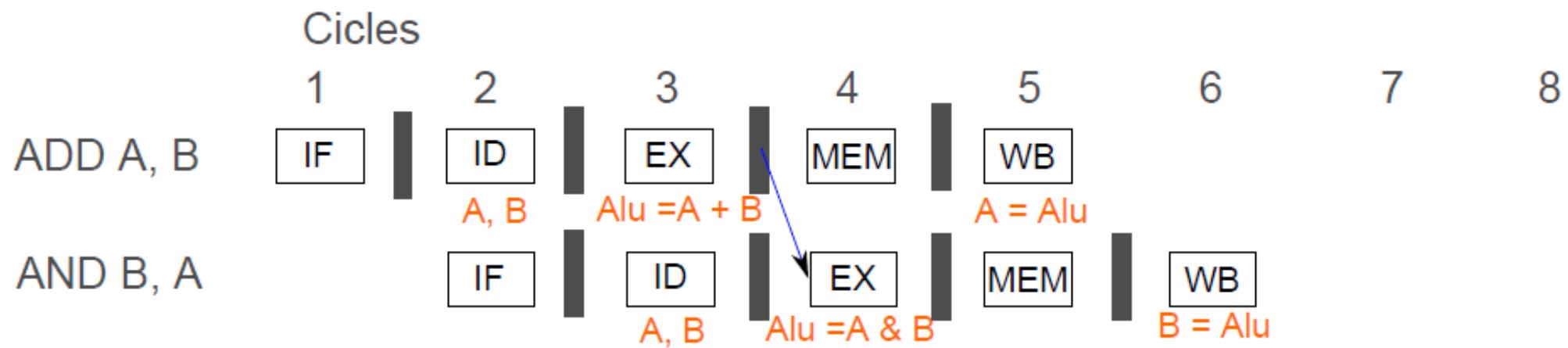
Hazards de control:

- una instrucción no puede ejecutarse en el ciclo de reloj que le corresponde porque no es la instrucción que se necesita → el flujo de direcciones (de las instrucciones) no es el que el pipeline esperaba

Ejemplo de *hazard de datos*



La mayoría de los *hazards* de datos tiene solución simple:
forwarding



Consideremos el pipeline anterior de 5 etapas (*IF*, *ID*, *EX*, *MEM*, *WB*)

... y la siguiente secuencia de instrucciones:

```
sub    $2, $1, $3  
and    $12, $2, $5  
or     $13, $6, $2  
add    $14, $2, $2  
sw     $15, 100($2)
```

Las últimas cuatro dependen del valor en el registro **\$2** de la primera
... valor que cambia durante el ciclo de reloj 5, p.ej., de 10 a –20:

- el valor leído desde el registro **\$2** no va a ser el resultado de la resta a menos que la lectura ocurra durante el ciclo 5 o después
- suponemos que en el ciclo 5 el *write* ocurre en la primera mitad del ciclo y el *read* en la segunda
- las instrucciones que obtendrían el valor correcto son **add** y **sw**

Pero el resultado correcto está disponible al finalizar la etapa *EX*, en el ciclo 3, de la instrucción **sub**:

- las instrucciones **and** y **or** necesitan el dato al inicio de la etapa *EX*, en los ciclos 4 y 5, respectivamente
- podemos ejecutarlas sin necesidad de detenciones (*stalls*) si enviamos (*forward*) el dato a las unidades que lo necesitan tan pronto como esté disponible (y antes de que esté escrito en el registro **\$2**)

Forwarding a una instrucción en la etapa *EX* —cuando trata de usar un registro que otra instrucción anterior va a escribir en su etapa *WB*:

- una operación en la ALU
- el cálculo de una dirección

Hay un *hazard* entre las instrucciones **sub** y **and**:

- se detecta cuando **and** está en su etapa *EX*, tratando de usar el registro **\$2** como primer operando,
... y **sub** está en su etapa *MEM*

... y otro entre **sub** y **or**:

- se detecta cuando **or** está en su etapa *EX*, tratando de usar el registro **\$2** como segundo operando,
... y **sub** está en su etapa *WB*

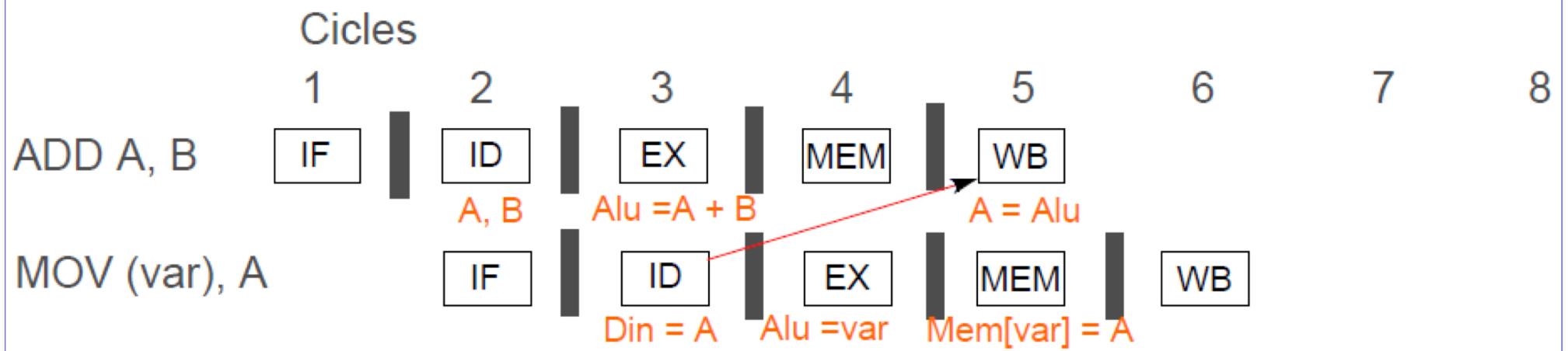
Pero si miramos los registros del pipeline notamos que

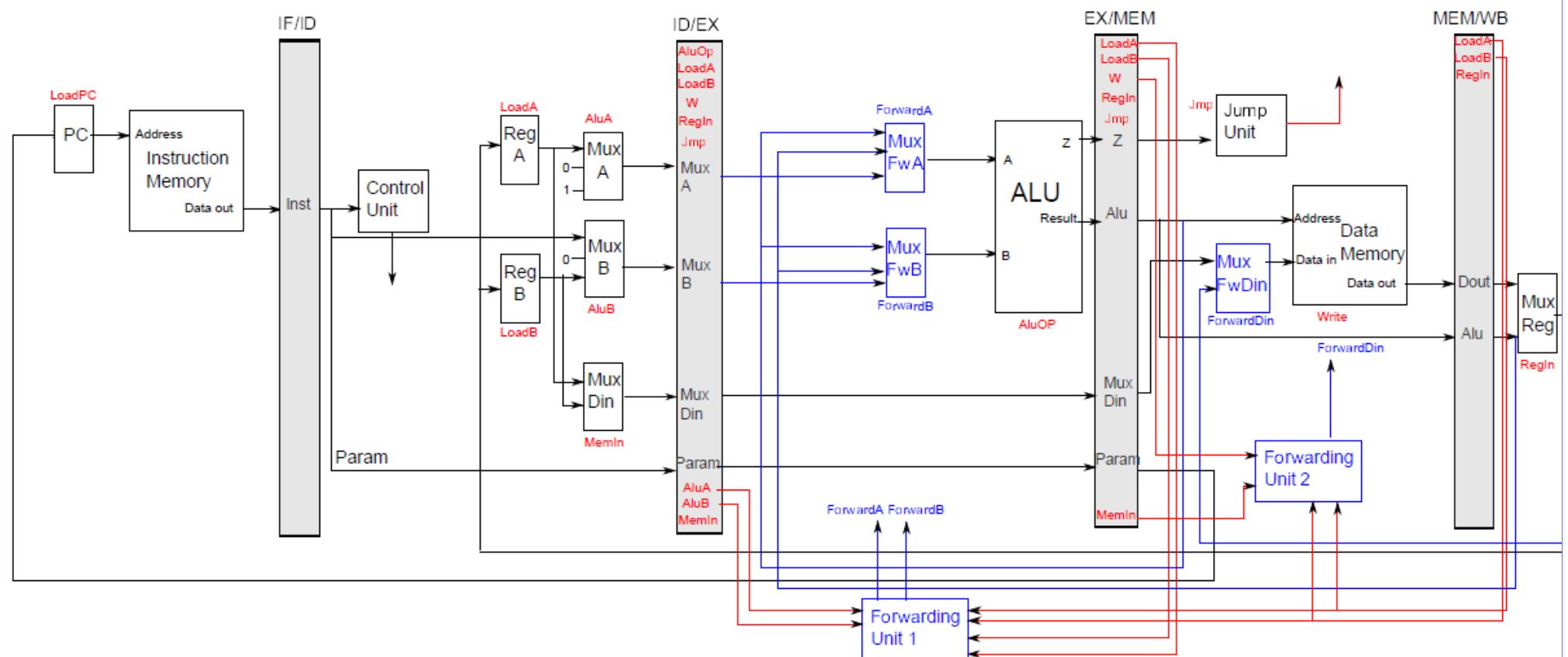
- el valor actualizado del primer operando de **and** está disponible en el registro *EX/MEM* cuando **and** lo necesita
- el valor actualizado del segundo operando de **or** está disponible en el registro *MEM/WB* cuando **or** lo necesita

Si podemos tomar los inputs para la ALU desde cualquier registro del pipeline y no solo de *ID/EX*

... entonces podemos enviar (*forward*) el dato apropiado:

- agregamos multiplexores a los inputs de la ALU
 - ... con los controles correspondientes
 - ... que seleccionan ya sea los registros A y B (o las constantes 0 o 1)
 - ... o bien los datos enviados (*forwarded*) desde los registros del pipeline
- ... y dos ***forwarding units*** que controlan a los multiplexores
 - ... en que la unidad en la etapa *MEM* ayuda en casos en que una instrucción *store* almacena en la memoria un valor calculado en la instrucción anterior (próx. diapo.)

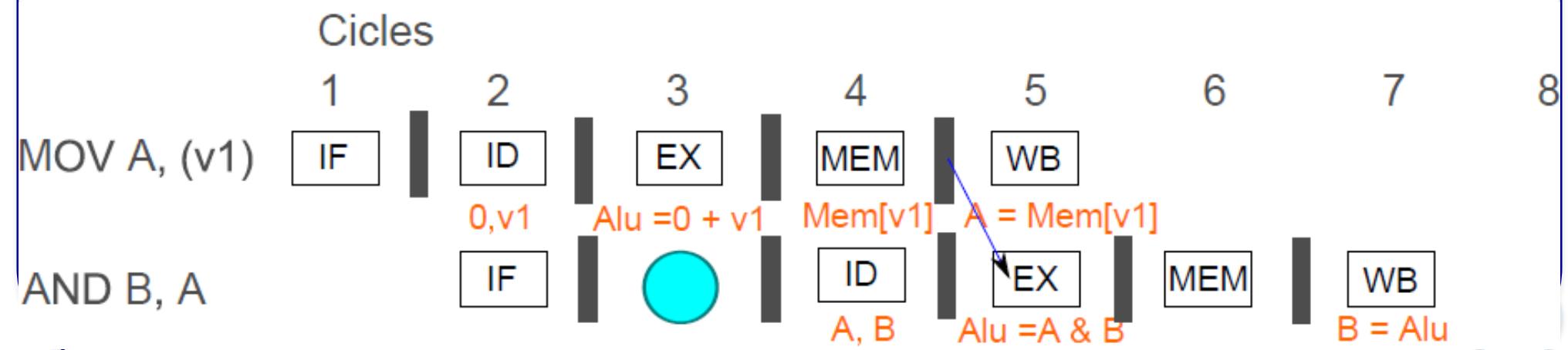
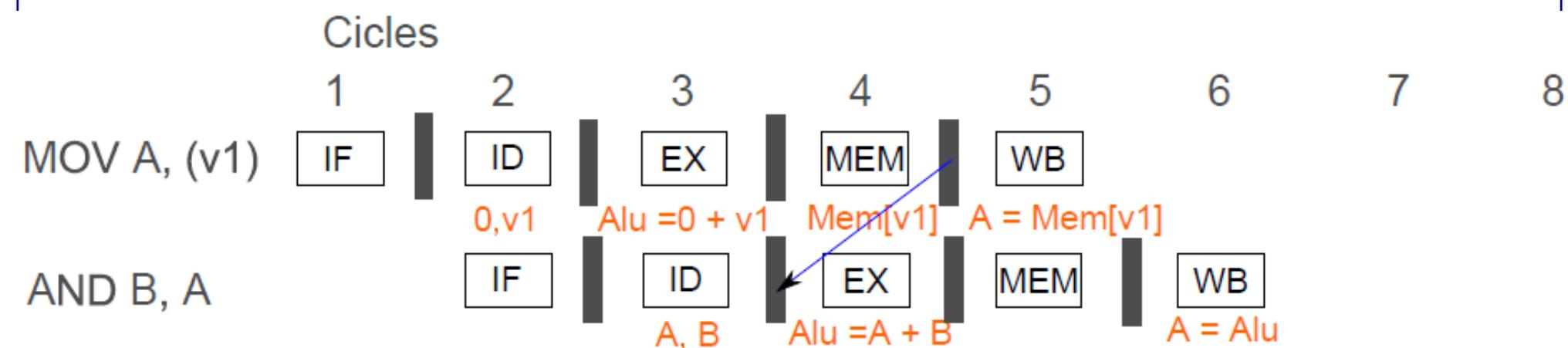
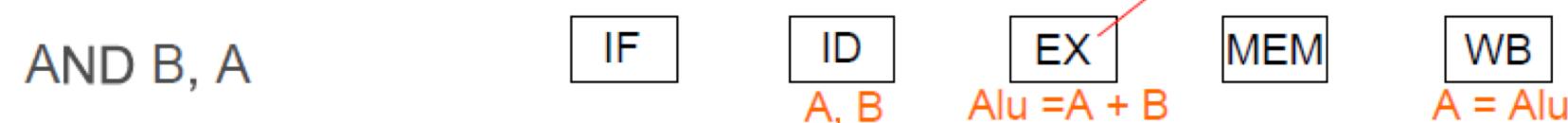
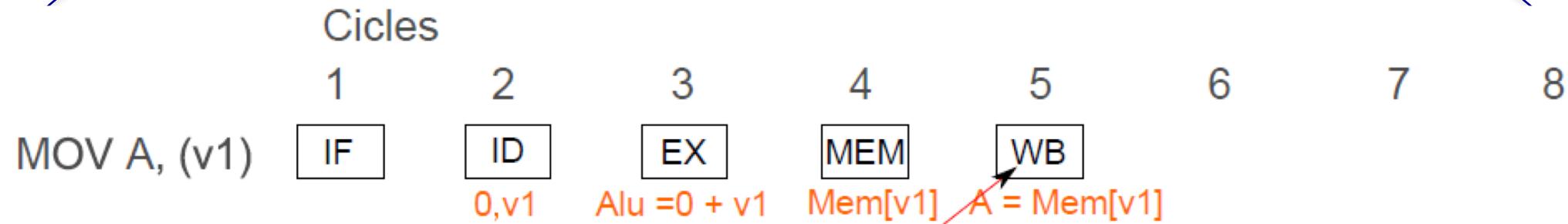




Pero *forwarding* no resuelve todos los *hazards*, p.ej.:

lw \$2, 20(\$1)
and \$4, \$2, \$5

- una instrucción trata de leer un registro justo después de una instrucción *load (lw)* que escribe el mismo registro
- el dato está aún siendo leído desde la memoria en el ciclo de reloj 4 (*MEM*) mientras la ALU está realizando al operación de la siguiente instrucción

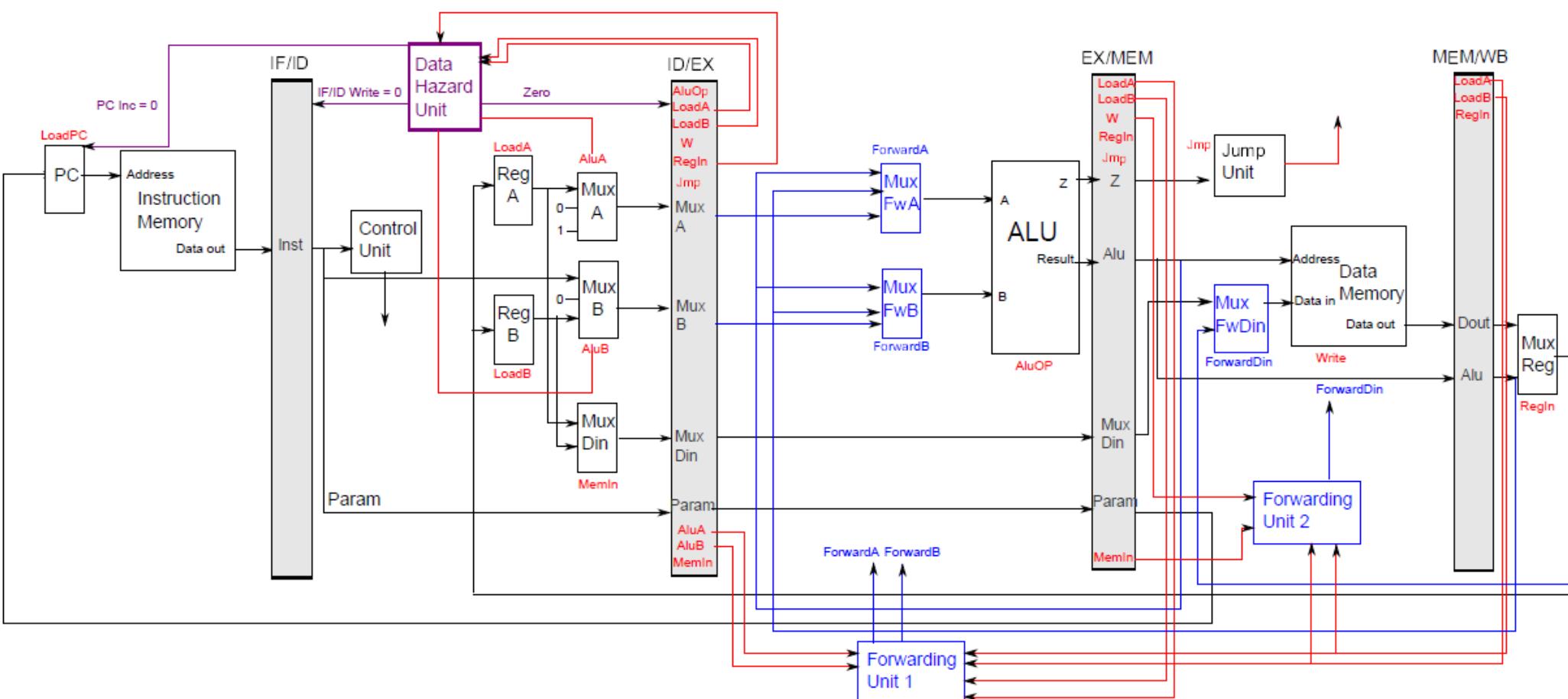


Es necesario detener (*stall*) el pipeline cuando una instrucción *load* es seguida por una instrucción que lee su resultado:

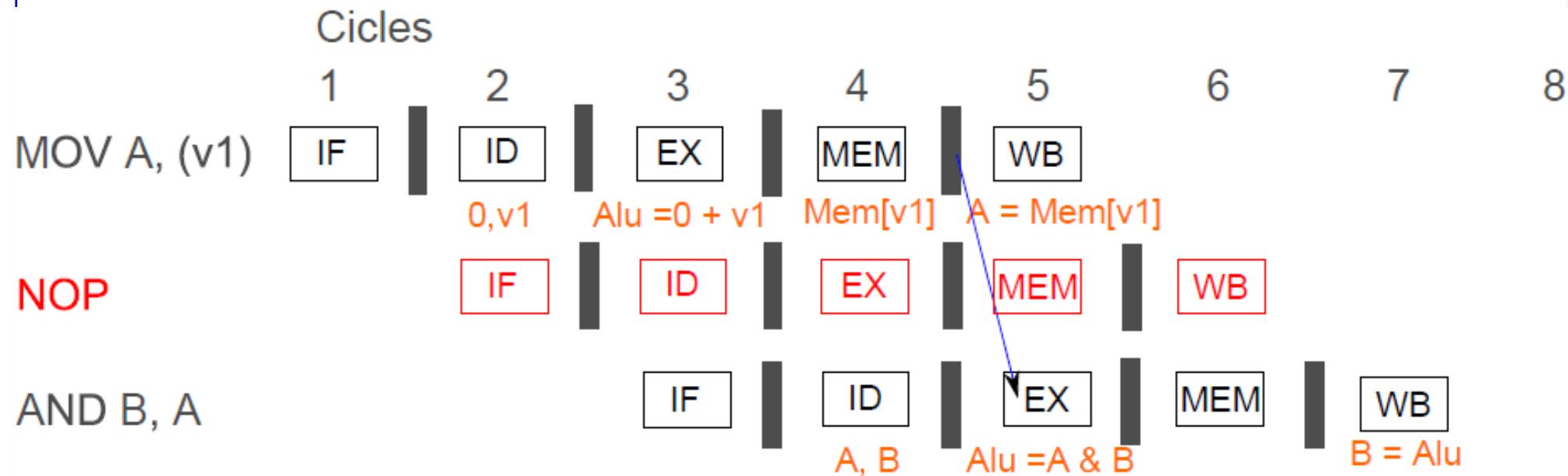
- agregamos una ***data hazard unit*** en la etapa *ID*
 - ... que produce un *stall* de un ciclo en la ejecución de la segunda instrucción
- luego del *stall*, la lógica de *forwarding* maneja la dependencia y la ejecución sigue

Si la instrucción en la etapa *ID* es detenida, entonces también es necesario detener la instrucción en la etapa *IF*:

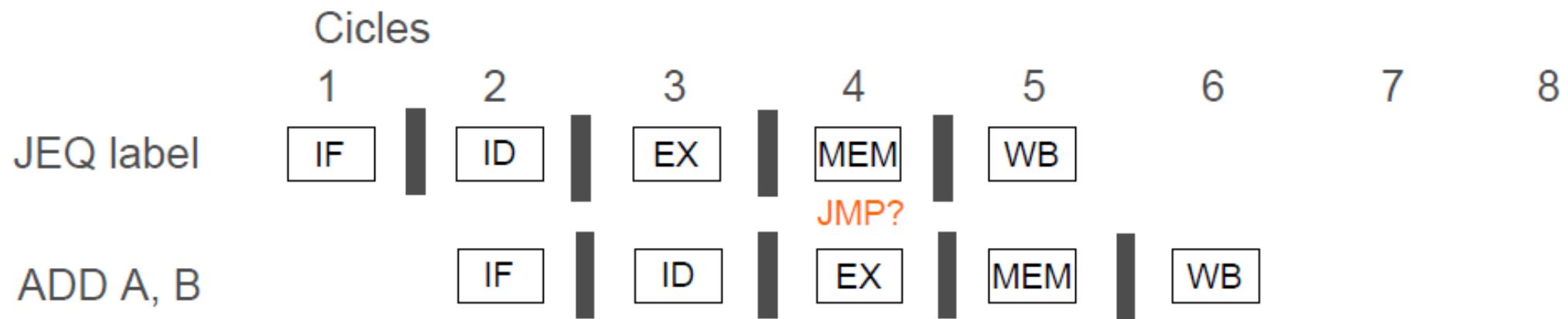
- hay que evitar que el registro *PC* y el registro *IF/ID* del pipeline cambien



Otra opción es insertar una instrucción **NOP**, que no hace nada, entre instrucciones

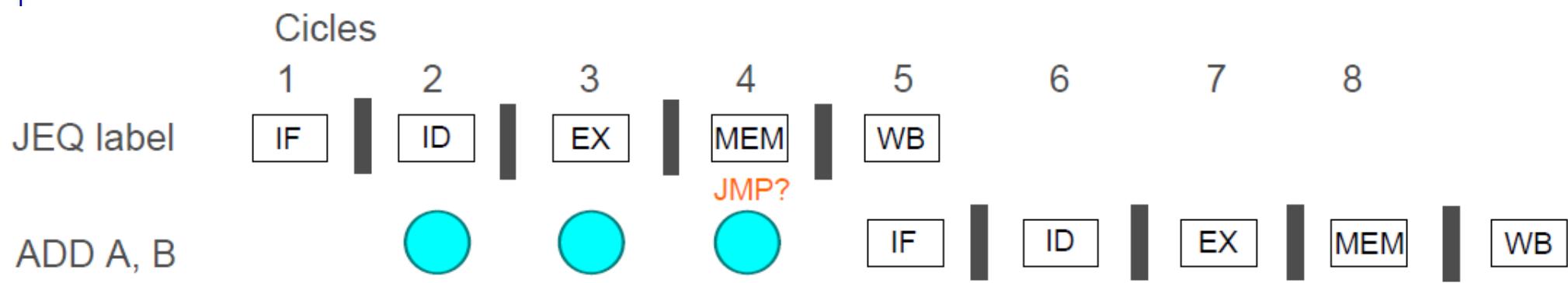


Ejemplo de *hazard de control* (o de *branch*)

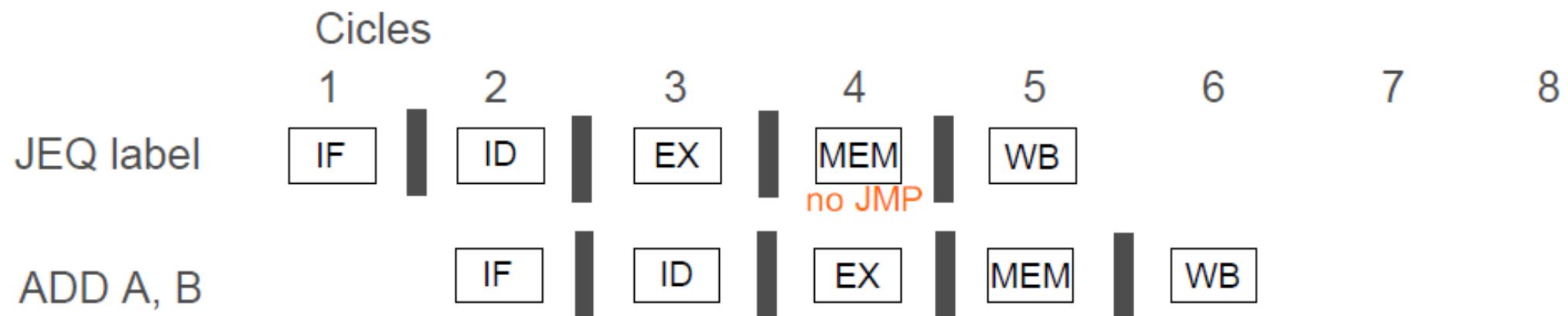


Para los *hazards* de control no hay nada tan eficaz como *forwarding* es eficaz para los *hazards* de datos

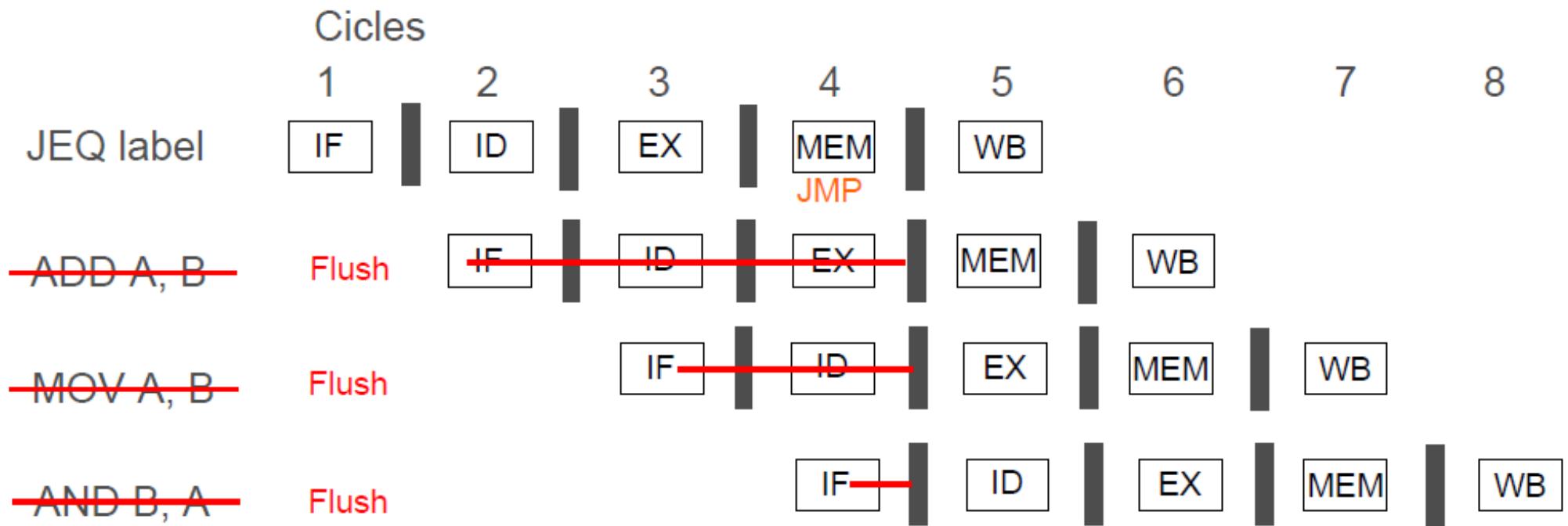
Una solución simple es hacer *stalling* del pipeline durante los ciclos de reloj que sean necesarios



Una solución más sofisticada es emplear predicción de saltos



... en que el único problema es qué hacer cuando nos equivocamos



La instrucción que sigue inmediatamente a un *branch* (o *jump*) incondicional es ejecutada siempre:

- *delay slot* — ¿qué podemos poner aquí?

Predicción **estática** de saltos:

- todos los saltos hacia atrás se toman
- todos los saltos hacia adelante no se toman

¿Cómo deshacemos (*flush*) instrucciones?

- permitimos que se ejecuten solo hasta que tratan de escribir en un registro
—el nuevo valor es almacenado temporalmente en un registro borrador
(*scratch register*)
- guardamos el valor actual del registro que va a ser escrito en un registro
borrador, para poder recuperar el estado que había al tomar el salto
equivocadamente

Predicción **dinámica** de saltos:

- un bit
- dos bits

