

Paralelismo a nivel de instrucciones: *pipelining*

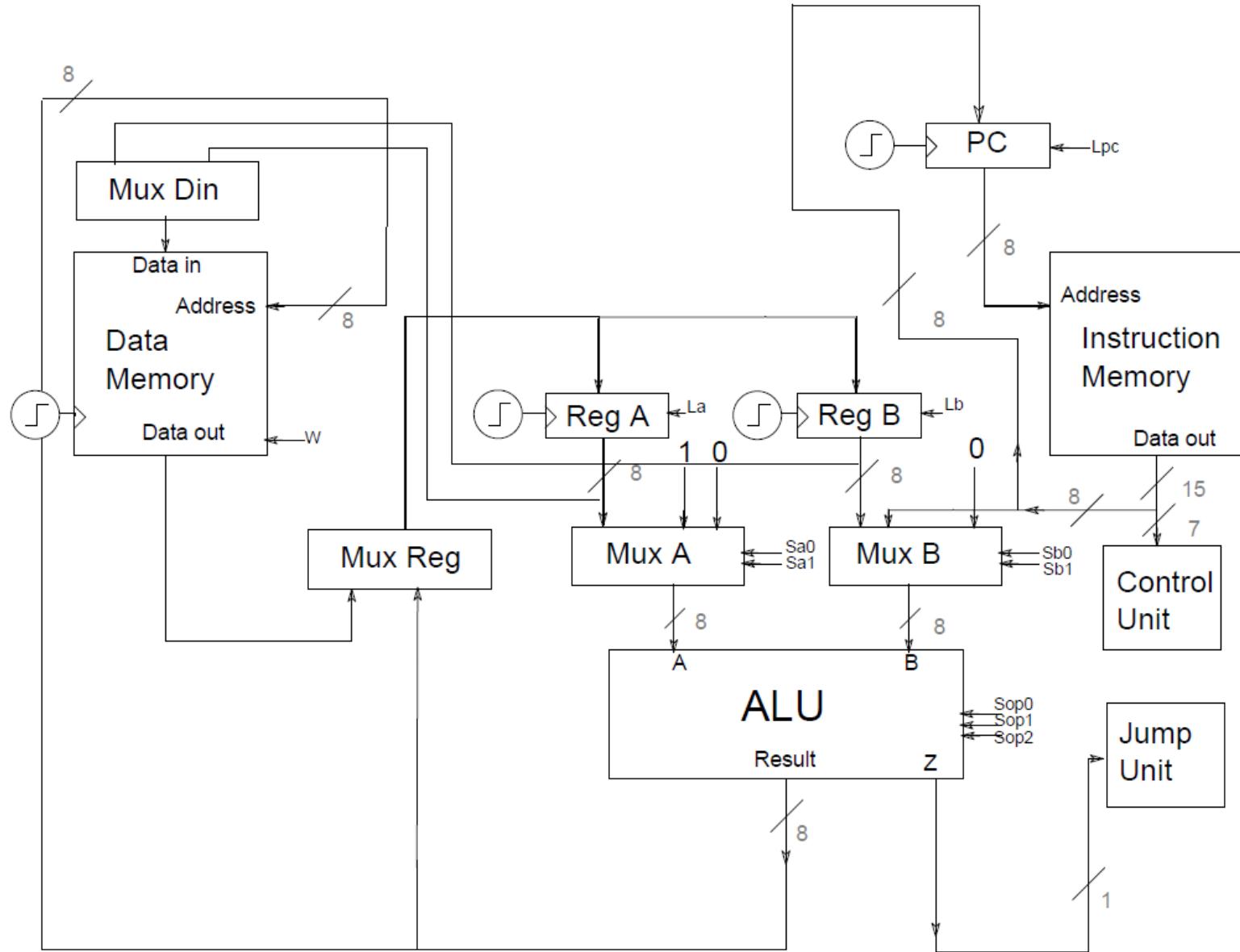
Arquitectura de Computadores – IIC2343

La CPU ejecuta cada instrucción en una secuencia de pasos, central a la operación de todos los computadores

... el ciclo ***fetch-decode-execute***:

1. Traer (*fetch*) la instrucción desde la memoria
2. Decodificar la instrucción y leer los registros involucrados
3. Ejecutar la operación o calcular una dirección de memoria
4. Tener acceso a un operando en la memoria
5. Escribir el resultado en un registro

El computador básico simplificado, en que separamos la *Jump Unit* de la *Control Unit* (las implicaciones las pueden leer en los apuntes de Alejandro y Hans)



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

Una posibilidad es 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— **que es lo que vamos a ver a continuación**
- paralelismo a nivel de procesadores —múltiples CPUs trabajan juntas en el mismo problema— que queda pendiente para otra oportunidad

P.ej., traer instrucciones desde la *Instruction Memory* (paso 1) es un 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 registros especiales
- esto divide la ejecución de la instrucción en dos partes: *fetching* y la ejecución misma

El cuello de botella también se produce cuando hay acceso a la *Data Memory* como consecuencia de la ejecución de la instrucción (paso 4):

- → podríamos dividir la ejecución de la instrucción en *fetching*, ejecución, y acceso a la *Data Memory*

Por otra parte, mientras se está decodificando la instrucción (paso 2)

... no se está ejecutando la instrucción (paso 3) al mismo tiempo:

- la ALU está desocupada

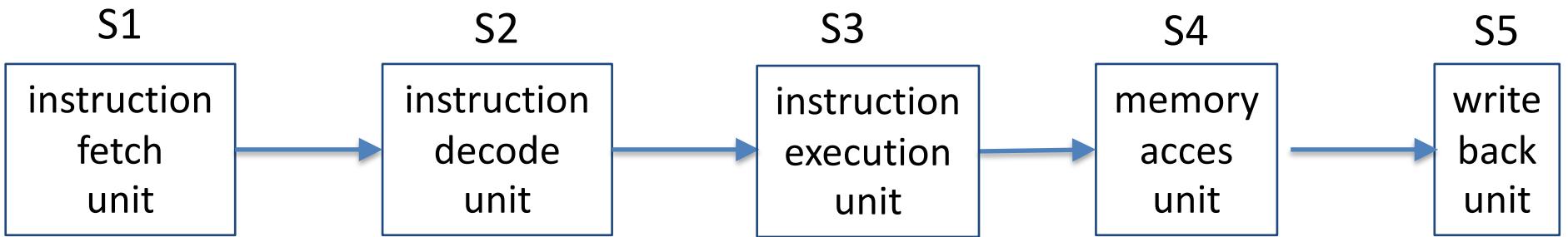
Un **pipeline*** permite implementar esta estrategia:

- 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

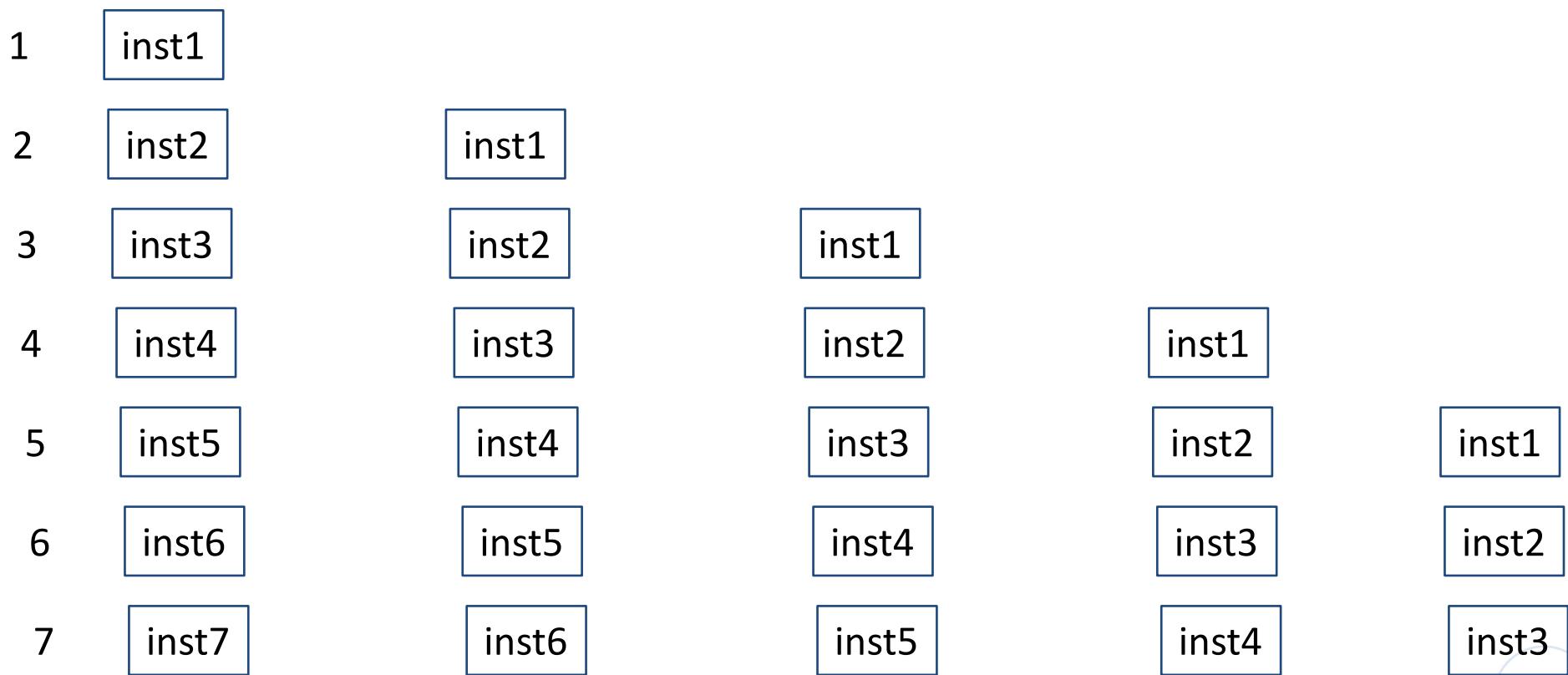
Pipelining: Técnica de implementación en la que las ejecuciones de múltiples instrucciones son traslapadas

... 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

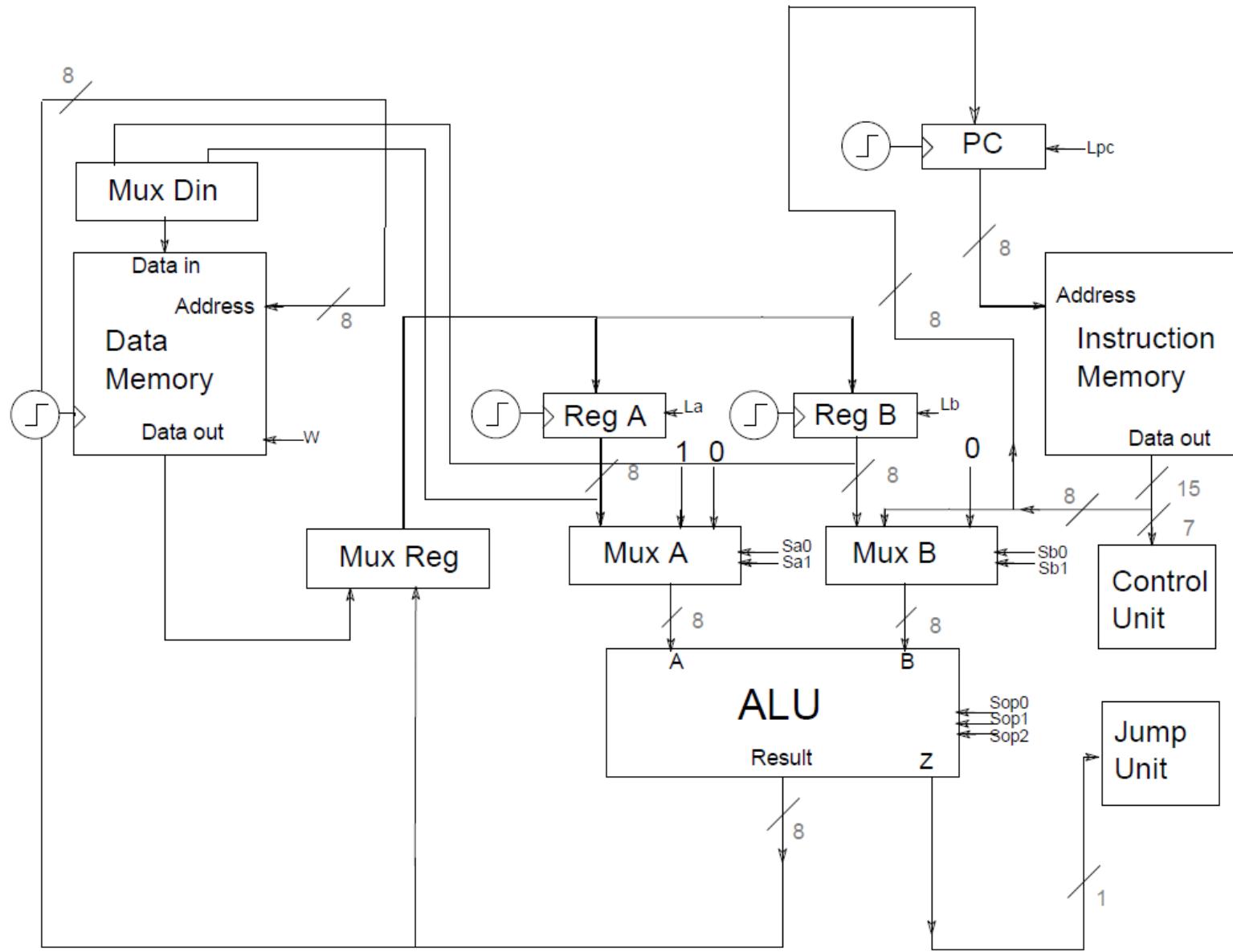
Consideremos las siguientes tres instrucciones y su división en etapas:

| 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 | |

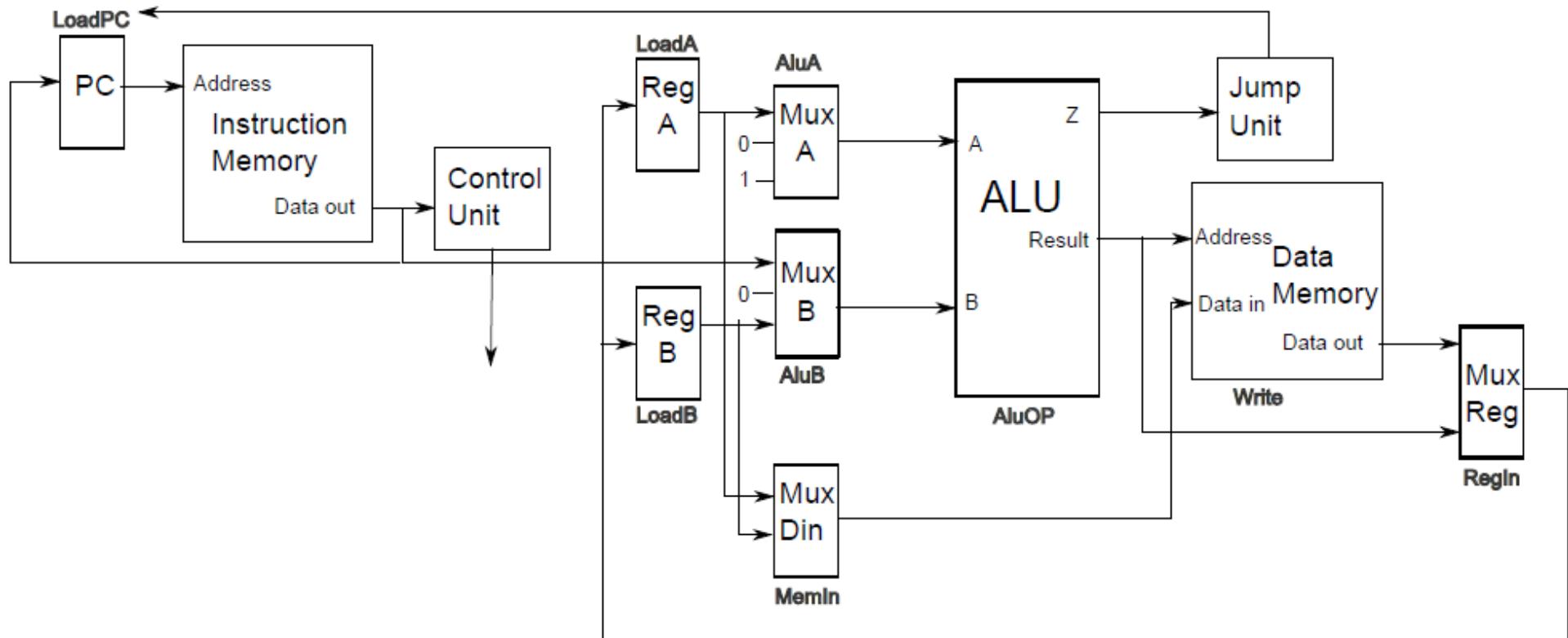
Supongamos entonces un pipeline de 5 etapas:

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)*

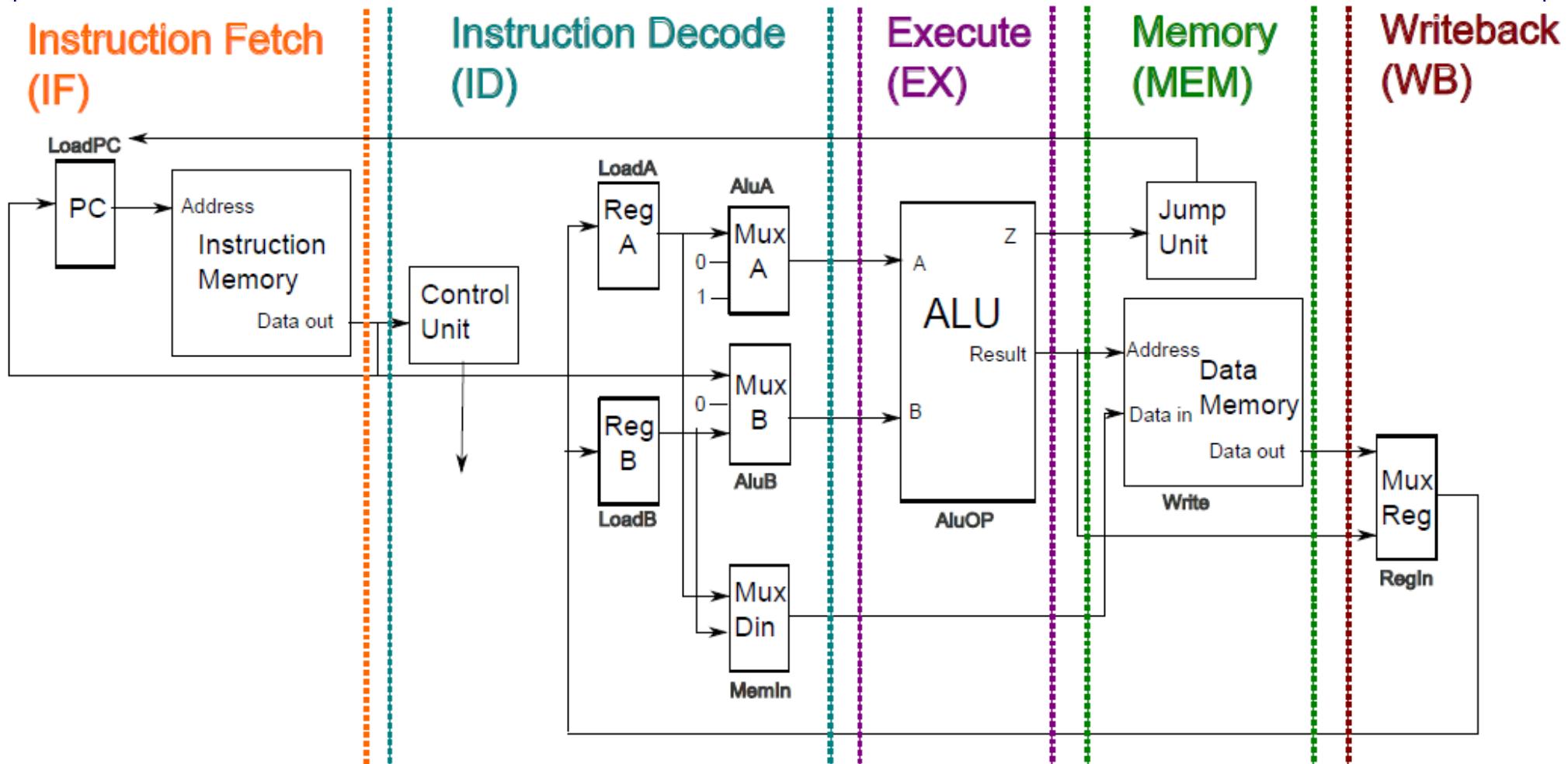
Tomemos la versión simplificada del computador básico: *Data Memory*, *Instruction Memory*, *PC*, *ALU*, registros *A* y *B*, *Control Unit* y *Jump Unit*



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

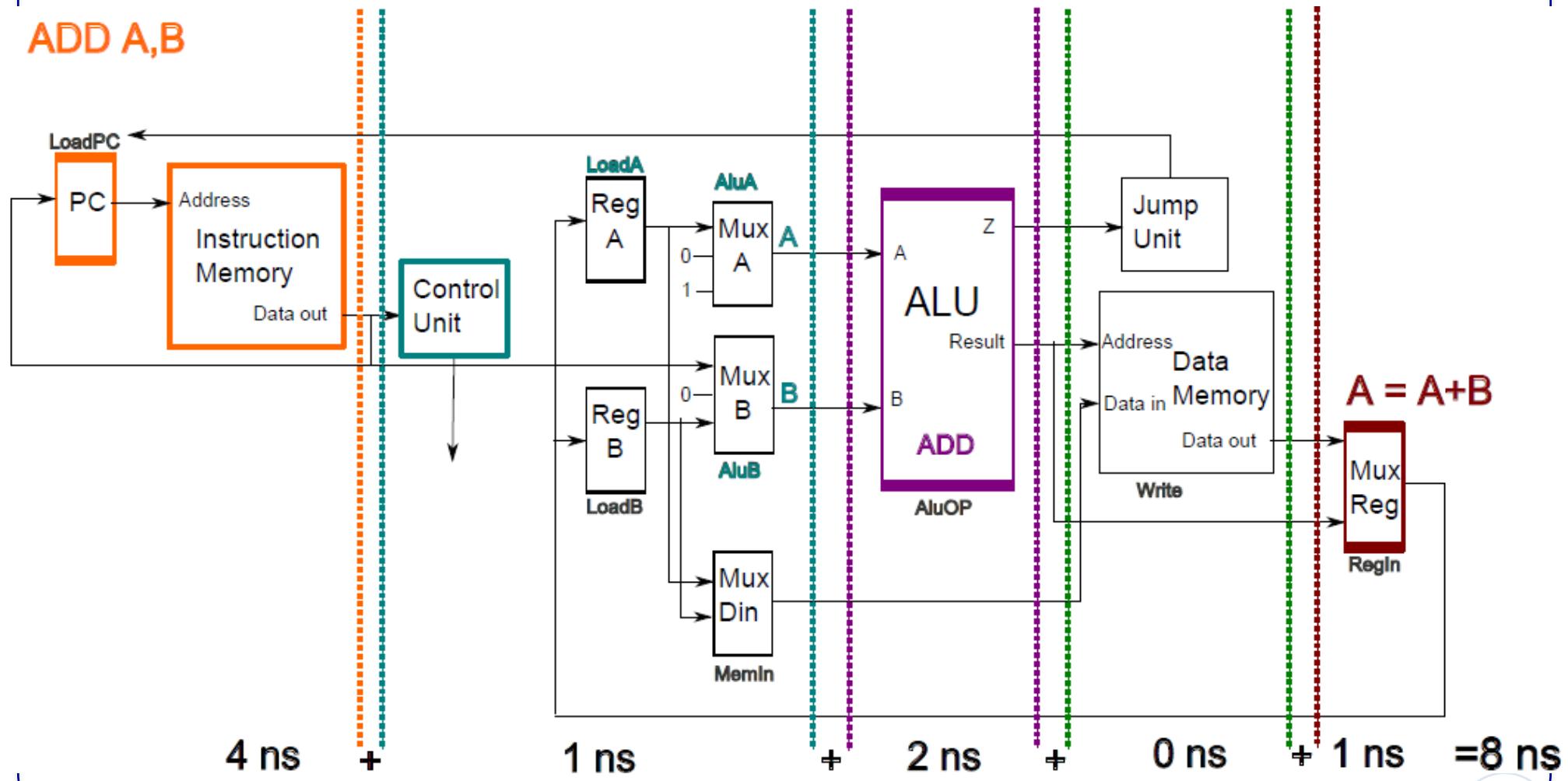


Agrupemos los componentes según cada una de las 5 etapas en que dividimos la ejecución de una instrucción



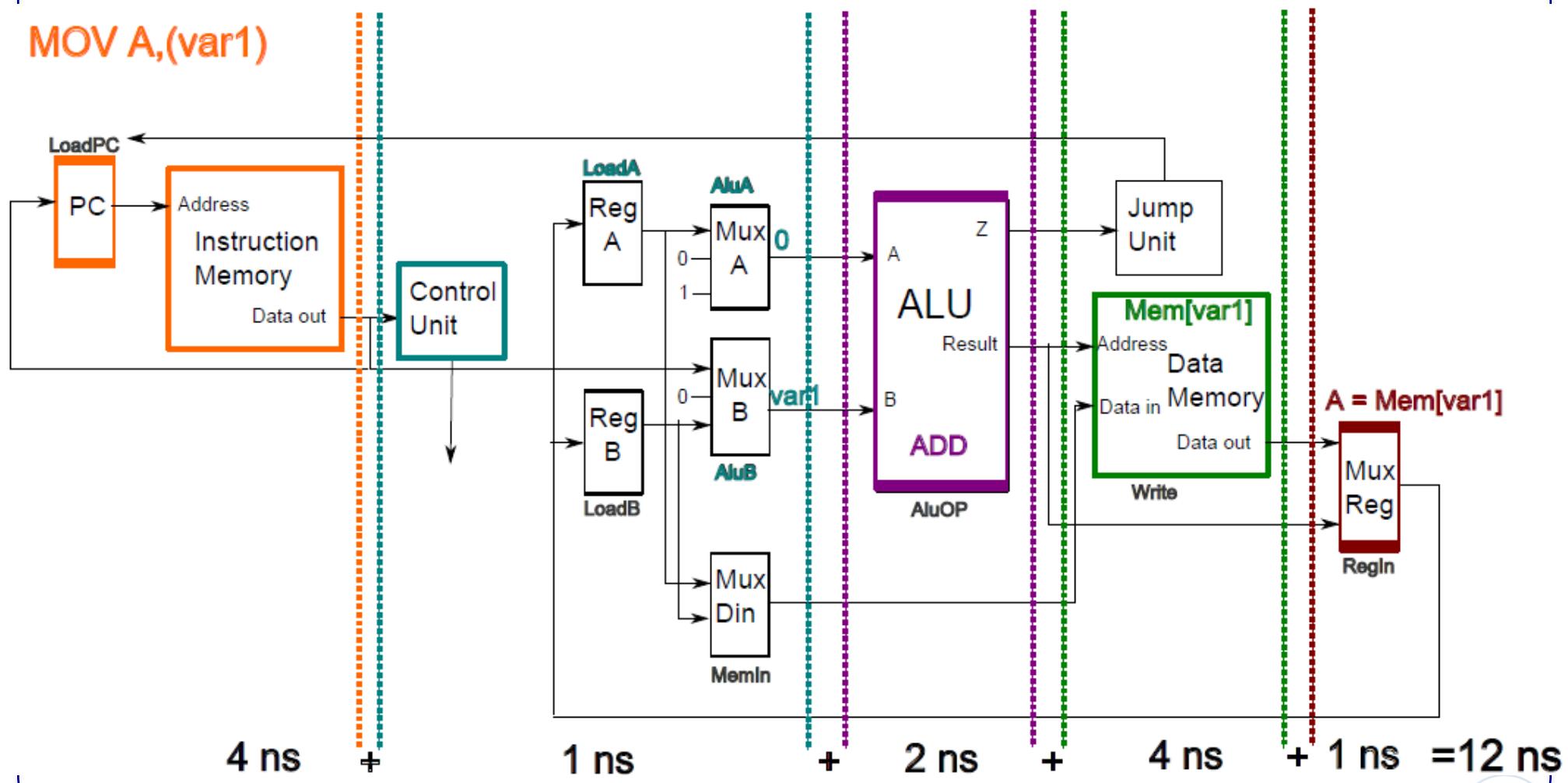
Etapas que participan en la ejecución de ADD A, B y sus duraciones

ADD A,B



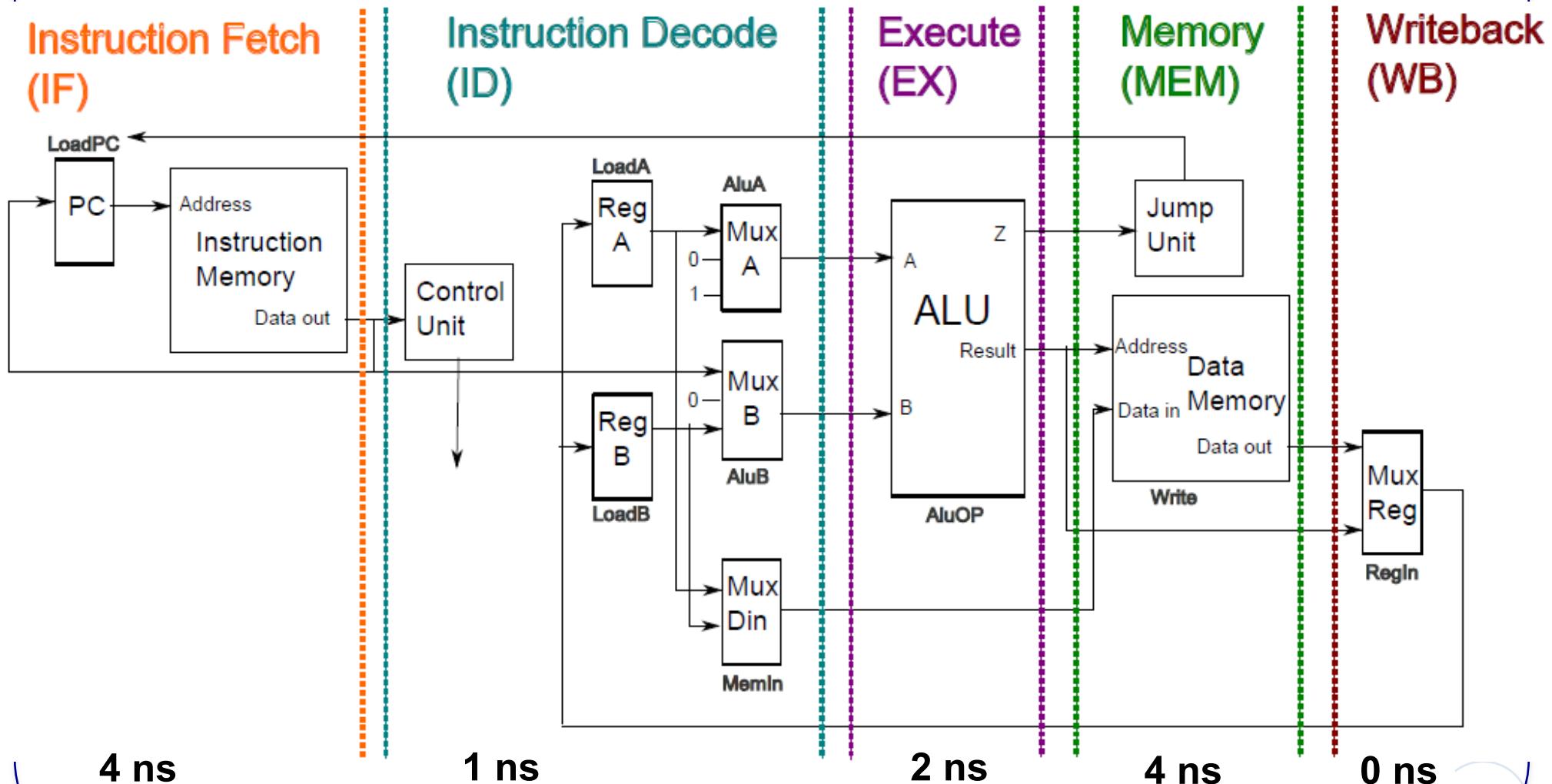
Etapas que participan en la ejecución de MOV A, (var1) y sus duraciones

MOV A,(var1)

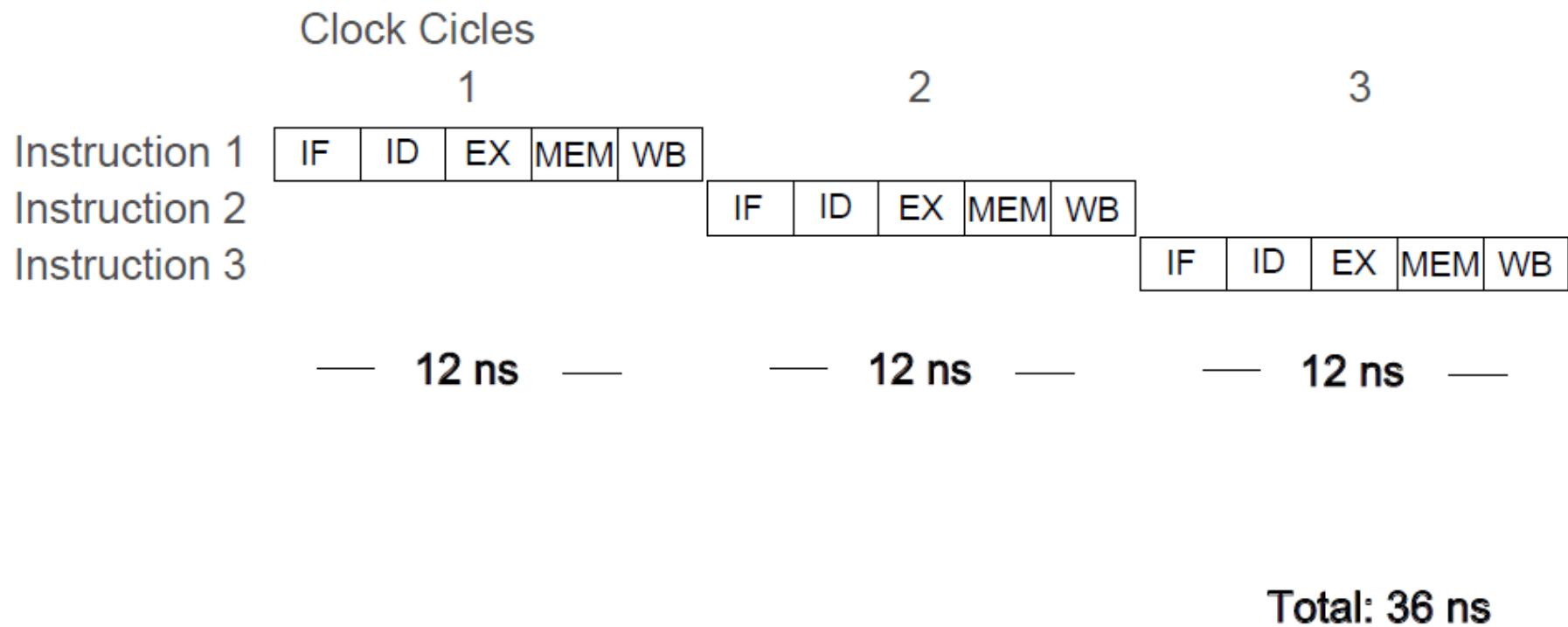


Etapas que participan en la ejecución de MOV (var1),A y sus duraciones

MOV (var1),A



Si simplemente ejecutamos una instrucción tras otra —esperamos a que termine la primera para empezar a ejecutar la segunda— 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 |
| Instruction 1 | IF | ID | EX | MEM | WB |
| Instruction 2 | | IF | ID | EX | MEM |
| Instruction 3 | | | IF | ID | EX |
| | | | | MEM | WB |
| | | | | | |

4ns 4ns 4ns 4ns 4ns 4ns

Total: 28 ns

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

- 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 | | | | | |

Nomenclatura relativa a *pipelines*:

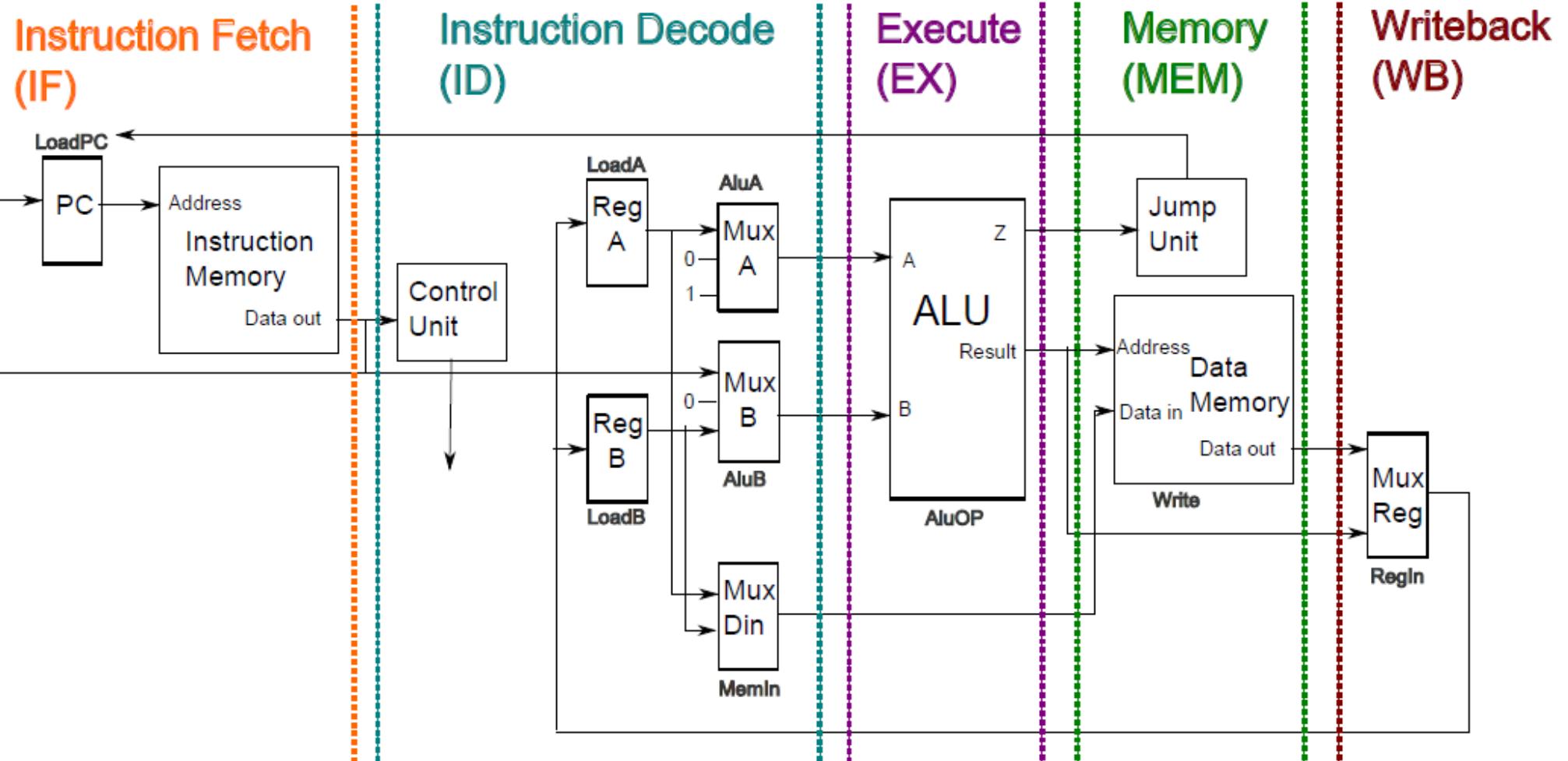
- **profundidad (depth)** es el número de etapas del pipeline; en el ej., 5
- **llenado (filling)** es el tiempo que toma al inicio llegar a que todas las etapas sean ocupadas simultáneamente; en el ej., 4 ciclos

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 4 ns
 - ... entonces una instrucción toma 20 ns para pasar por todo el pipeline
 - parecería que el computador corre a 50 MIPS
 - ... pero en realidad como cada 4 ns se termina de ejecutar una nueva instrucción
 - ... la tasa de procesamiento es 250 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 registro *PC*

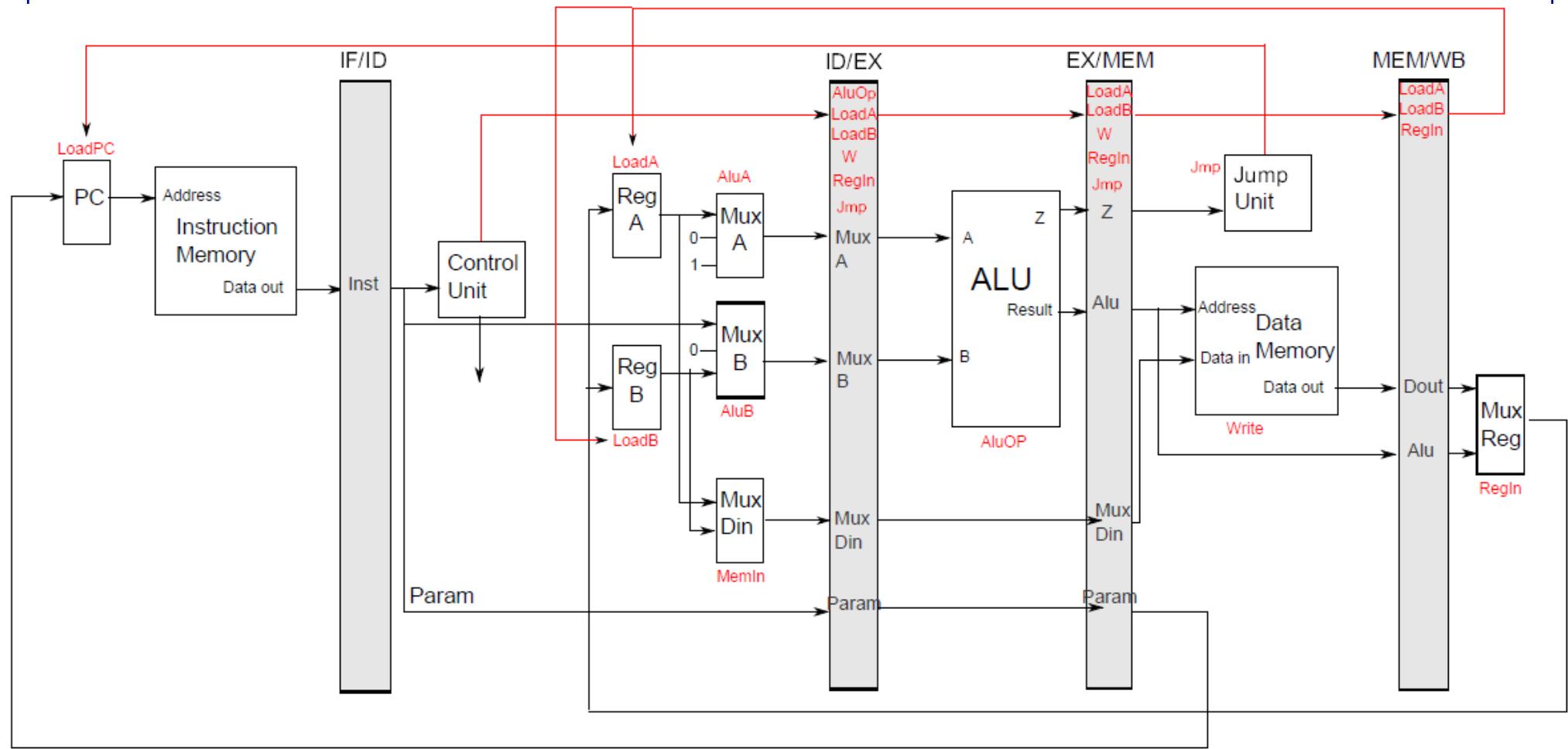
Estas dos excepciones no afectan la ejecución de la instrucción vigente
... pero pueden afectar a instrucciones posteriores en el *pipeline* (como veremos luego) —respectivamente:

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

Para que una misma unidad pueda ser usada por varias instrucciones (no simultáneamente, pero a medida que avanzan 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*); las otras son almacenadas en *EX/MEM*
- similarmente, algunas de estas últimas líneas son usadas en el acceso a memoria (*MEM*); las restantes son almacenadas en *MEM/WB*, para ser usadas en el *write back* (*WB*)

En *pipelining* aparecen 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:

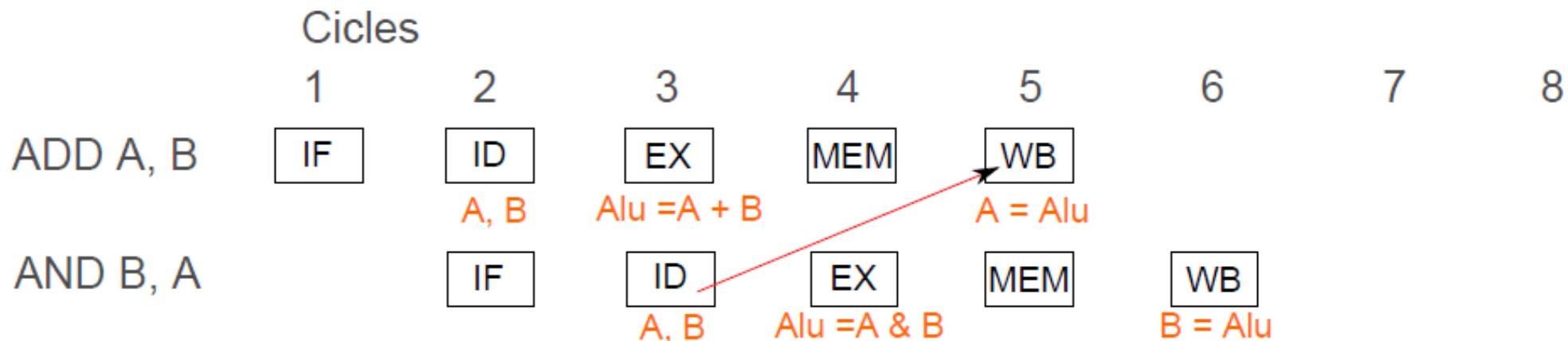
- una instrucción no puede ejecutarse en el ciclo de reloj que le corresponde porque el dato que necesita para su ejecución aún no está disponible

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

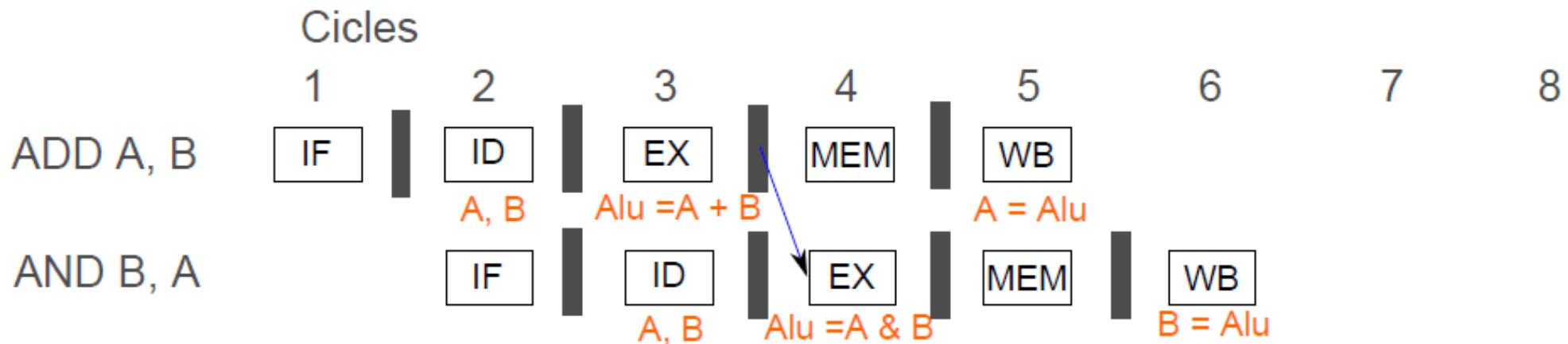
Ej. de *hazard* de datos:

- ejecución de la instrucción ADD A, B seguida por la ejecución de la instrucción AND B , A
- naturalmente, la instrucción AND B , A espera que el valor del registro A que va a usar sea el valor actualizado, después de la ejecución de ADD A, B
- sin embargo, como se ve en el diagrama, AND B , A necesita los valores de los registros en su paso por la etapa *ID*, durante el ciclo 3,
... pero el valor actualizado en el registro A sólo aparece una vez que ADD A, B pasa por la etapa *WB*, durante el ciclo 5 (la flecha roja indica dependencia de datos)



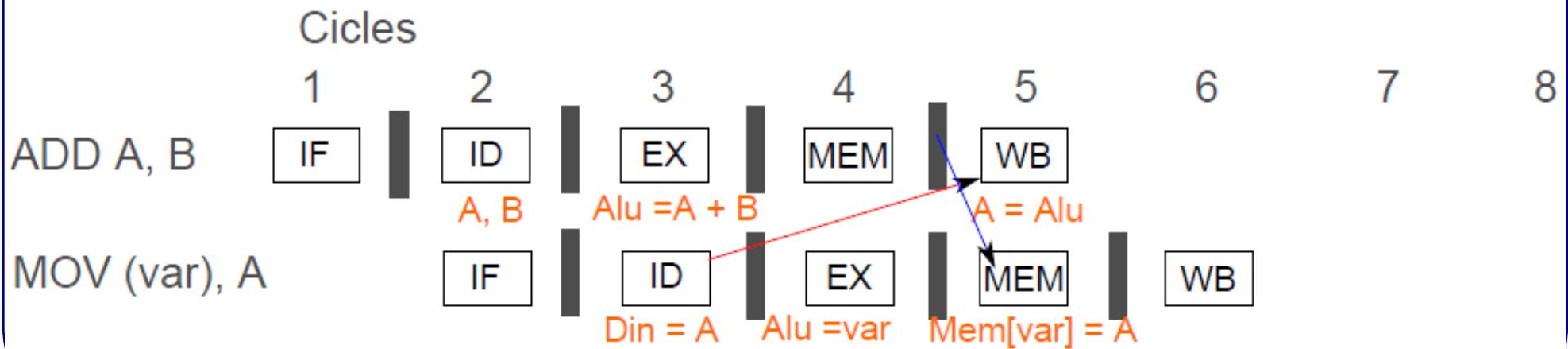
La mayoría de los *hazard* de datos tiene una solución simple:

- incluyamos en el diagrama los registros interetapas y tengamos presente el rol del registro *EX/MEM* (diap. #23)
- notamos que cuando ADD A, B pasa por su etapa *EX*, en el ciclo 3, almacena en el registro *EX/MEM* el valor del resultado de la suma
- así, cuando AND B, A pasa por su etapa *EX*, en el ciclo 4, el valor (del registro A) que necesita usar ya está disponible en el registro *EX/MEM* (aunque no en A aún) y puede usarlo (la flecha azul) → **forwarding**
- lo que se necesita es hardware adicional —una **forwarding unit**— que detecte la dependencia de datos y le pase a AND B, A, cuando ésta llega a su etapa *EX*, el valor almacenado en *EX/MEM* en vez del valor almacenado en A



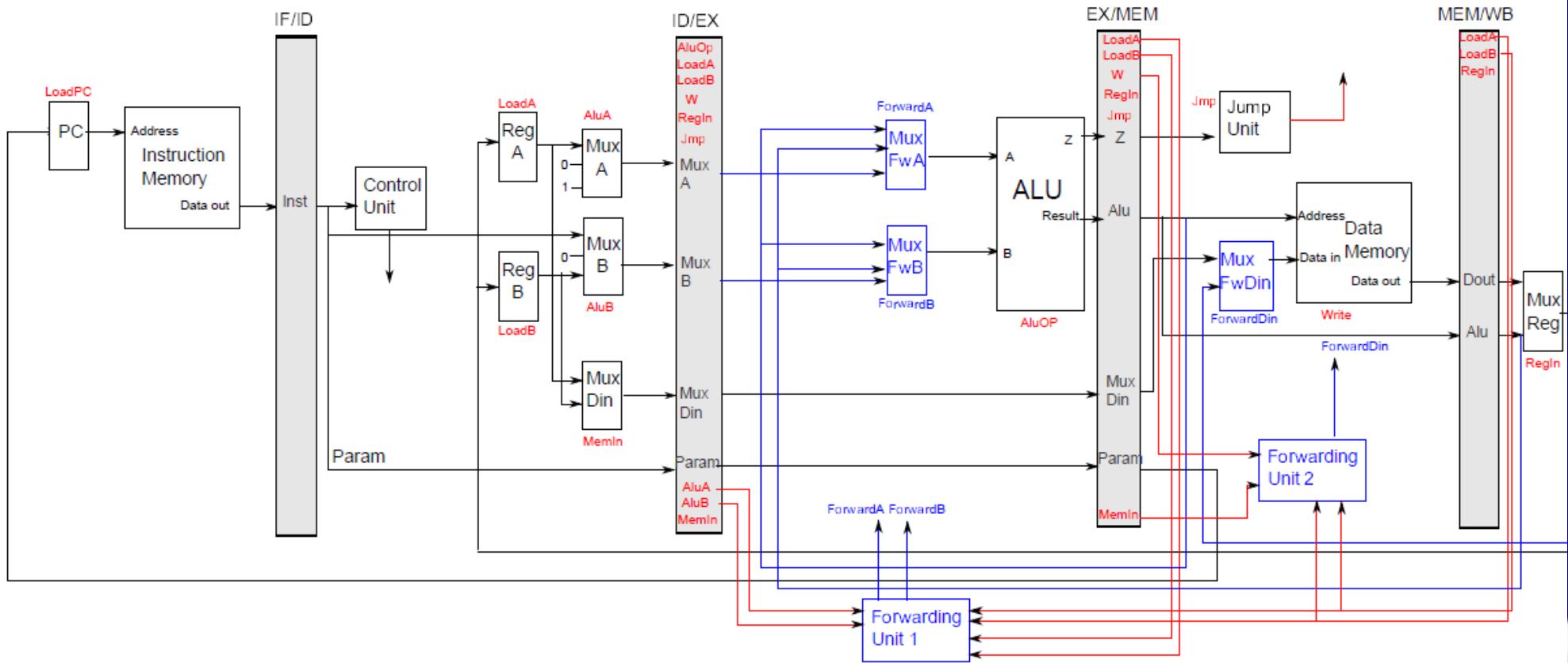
Otro ej., en que ADD A,B es ahora seguida por MOV (var),A:

- MOV (var),A necesita el valor (actualizado) del registro A cuando pasa por etapa *ID* en el ciclo 3; pero A, de nuevo, sólo es actualizado en el ciclo 5
- si bien ADD A,B almacena la suma en el ciclo 3 en el registro *EX/MEM*, ésta puede ser, y es, propagada al registro *MEM/WB* en el ciclo 4
- así, cuando MOV (var),A pasa por su etapa *MEM*, en el ciclo 5, el valor que necesita está disponible en el registro *MEM/WB* y puede usarlo (la flecha azul) → **forwarding**, nuevamente
- lo que necesitamos es otra **forwarding unit**, ahora en la etapa *MEM*, que detecte esta dependencia de datos y la resuelva



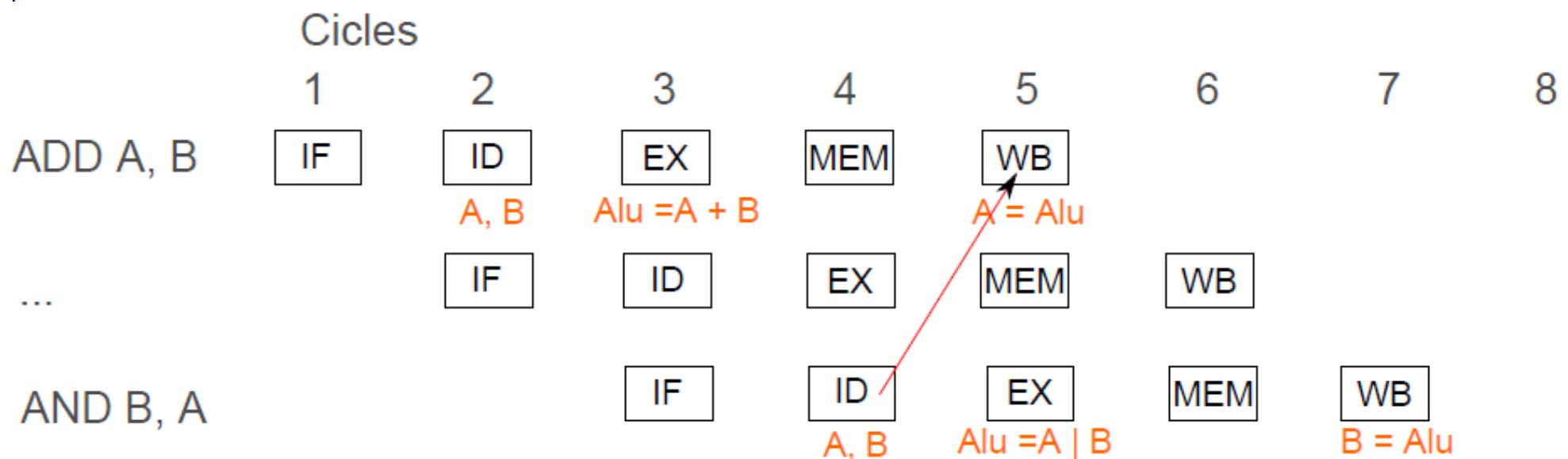
Las *forwarding units* reciben como inputs las señales de control que ya conocemos, sólo que provenientes de los registros interetapas

... y producen como output nuevas señales de control que controlan los nuevos multiplexores, en las entradas A y B de la ALU, y en la entrada *Data in* de la *Data Memory*



Ej. de *hazard* de datos en una secuencia de tres instrucciones:

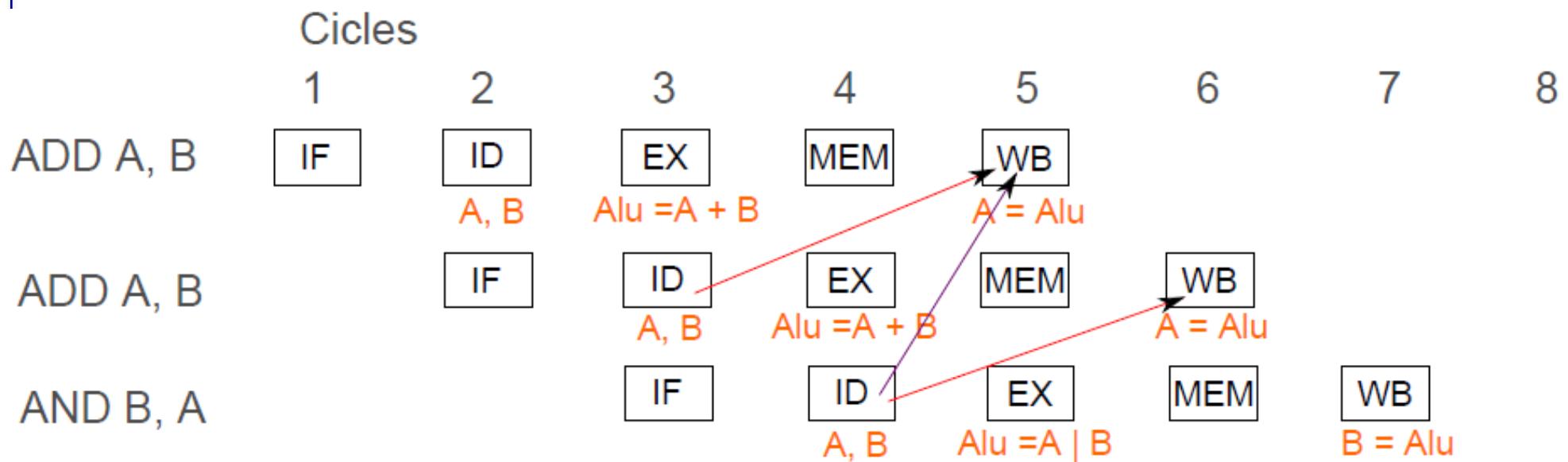
- puede ocurrir que la dependencia de datos está en la tercera instrucción (con respecto a la primera), la cual va a ejecutar su etapa *EX* en el ciclo 5 (y no en el ciclo 4, como en el ejemplo de la diap. #27)
- ... → el *forwarding* tiene que hacerse desde el registro *MEM/WB* (en lugar del registro *EX/MEM*, como en la diap. #28)



¿Y si la dependencia de datos de una instrucción es con respecto a las dos instrucciones anteriores?

- si la segunda instrucción también cambia el valor del registro A

... entonces la dependencia de datos de la tercera instrucción pasa a ser con respecto a ésta segunda instrucción y deja de serlo con respecto a la primera
→ el *forwarding* va a provenir ahora del registro EX/MEM

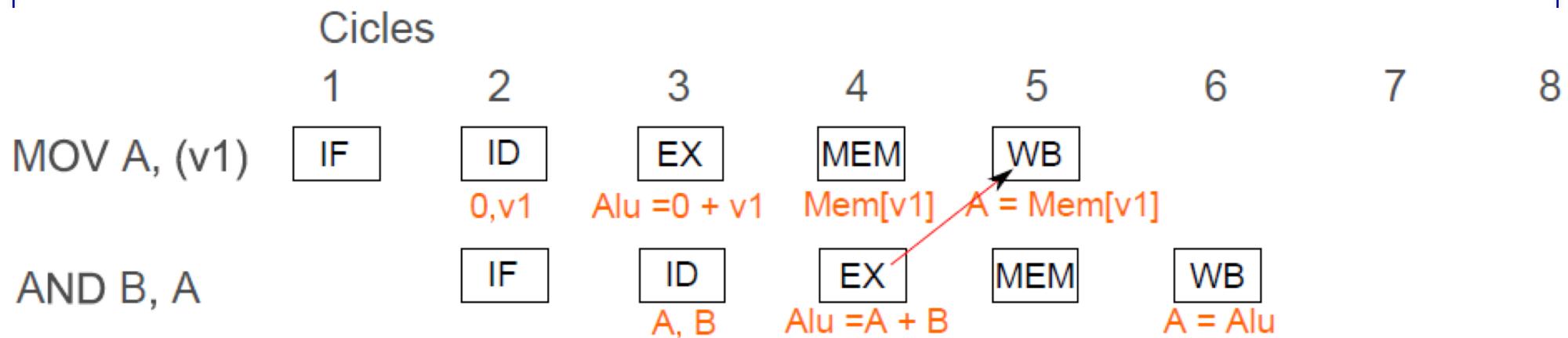


Veamos un último caso de *data hazard*:

MOV A, (v1)

AND B, A

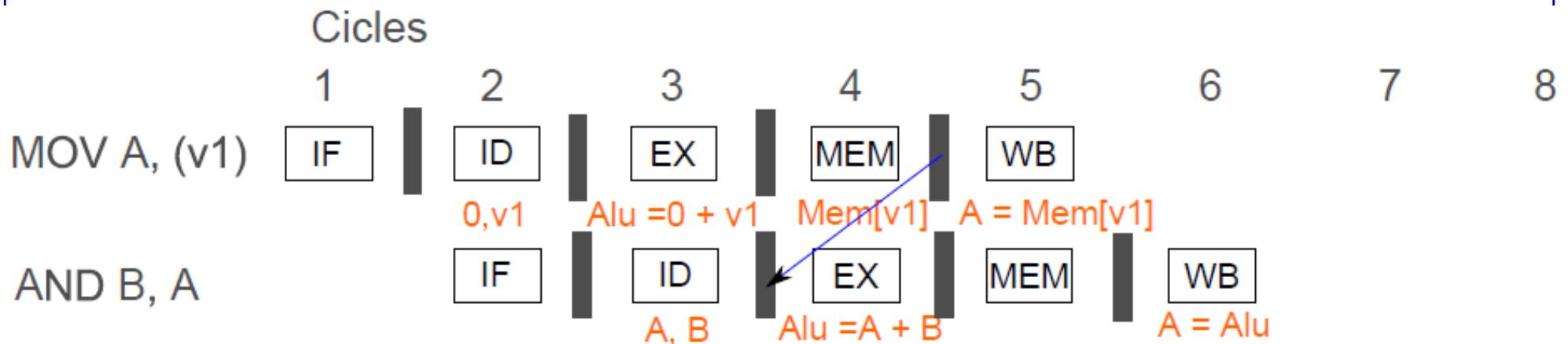
- la instrucción AND trata de leer un registro, A, justo después de que la instrucción MOV “carga” (escribe) el registro A con un dato traído desde la *Data Memory*
- el problema es que en el mismo ciclo 4 en que el dato está aún siendo leído desde la *Data Memory* (en la etapa *MEM* de MOV), la ALU tiene que realizar la operación correspondiente a AND (en la etapa *EX* de esta instrucción)



Forwarding por sí solo no resuelve este *hazard*:

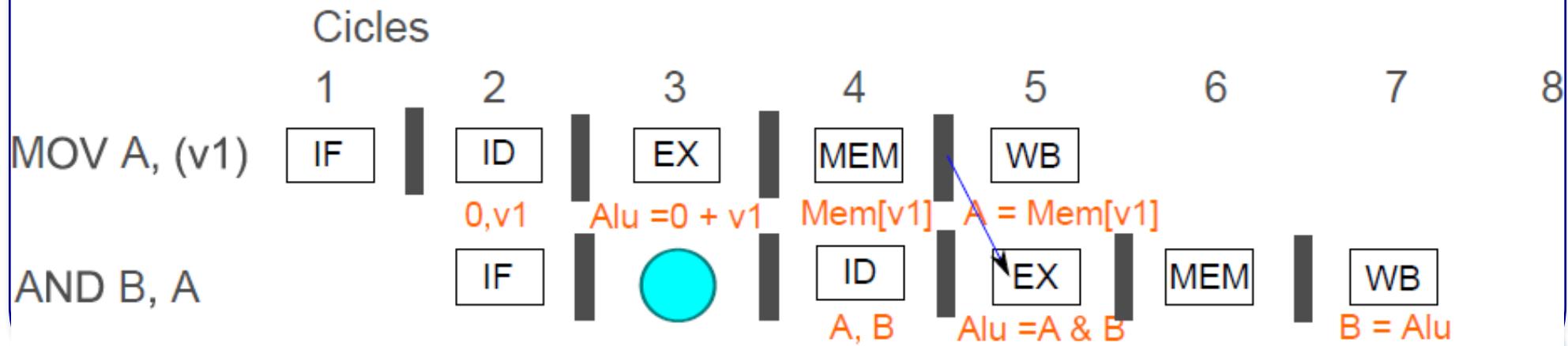
- aunque la etapa *MEM* de *MOV* almacene el dato traído desde la *Data Memory* en el registro *MEM/WB* en el ciclo 4 (podemos pensar que es hacia el final del ciclo 4)
- ... la etapa *EX* de *AND** necesita ese dato en el mismo ciclo 4 (podemos pensar que es al inicio del ciclo 4)

* en este diagrama y en el de la diap. anterior, la operación ejecutada por la ALU en la etapa *EX* debe ser $A \& B$ (y no $A+B$, aunque no hace diferencia para el problema que estamos ilustrando)



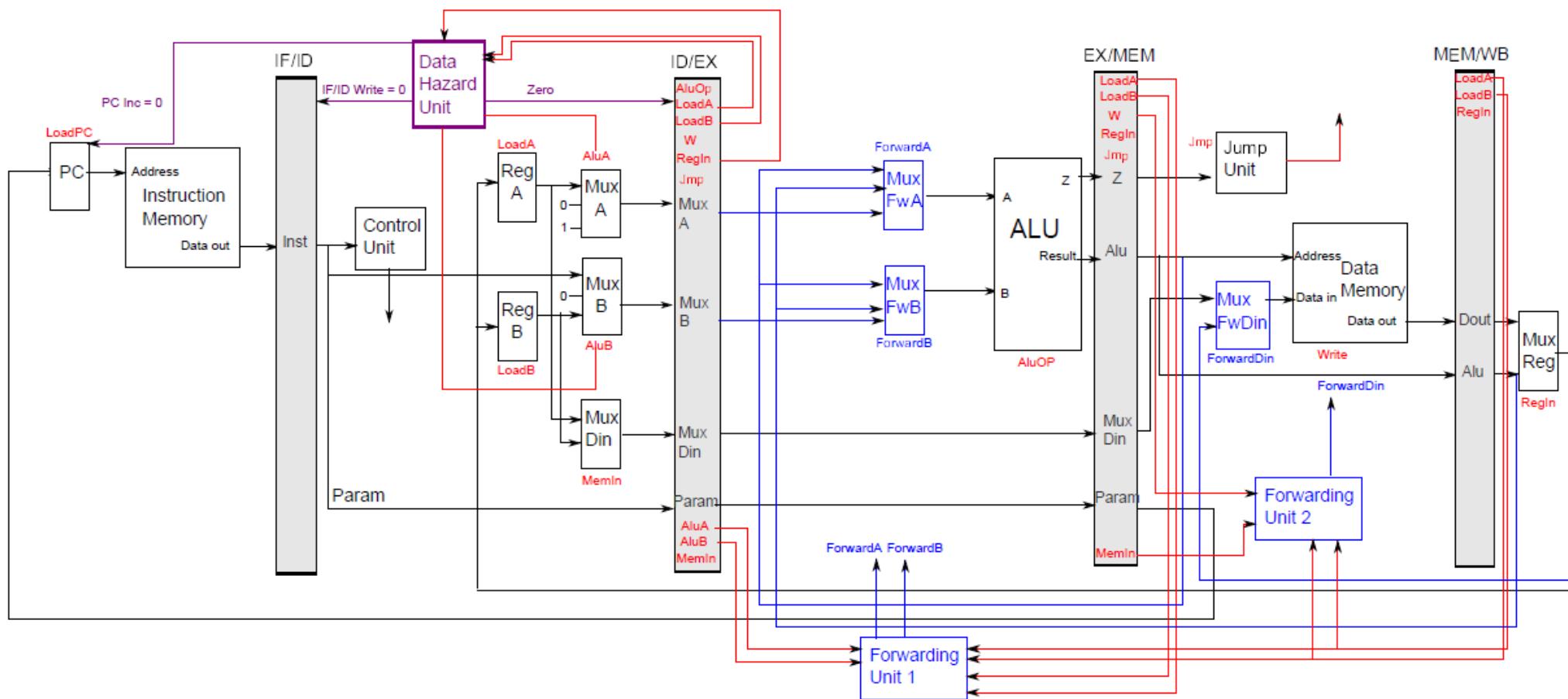
Es necesario detener (*stall*) el pipeline cuando una instrucción *load* (“carga” un registro con un dato traído desde la memoria) es seguida por una instrucción que usa el valor del registro recién cargado:

- 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 correctamente la dependencia y la ejecución sigue



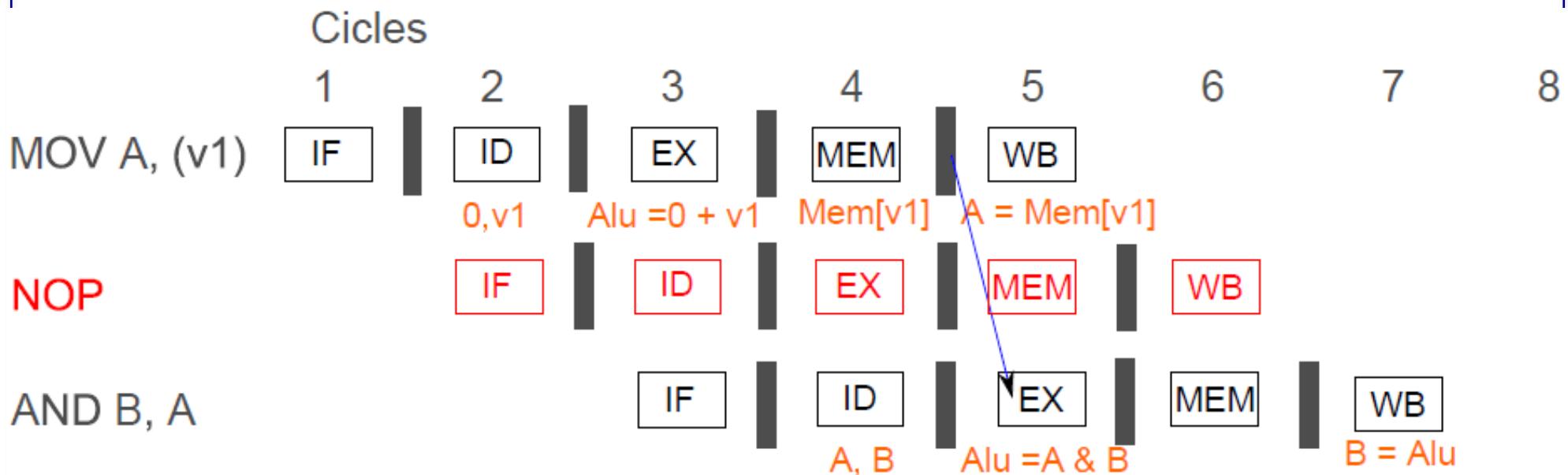
Al detener una instrucción en la etapa *ID*, se hace necesario detener también la próxima instrucción en la etapa *IF*:

- la *Data Hazard Unit* debe evitar que el registro *PC* y el registro *IF/ID* cambien



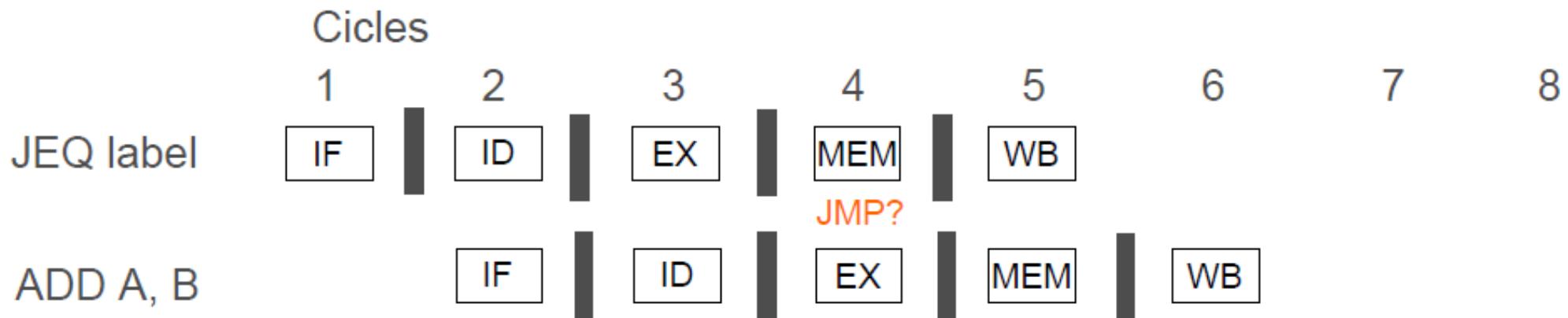
El caso anterior —una instrucción *load* seguida por una instrucción que lee el registro recién cargado— es detectable por el compilador cuando está generando el código *assembly*:

- → una solución alternativa a *data hazard unit + stallings* es que el compilador inserte una instrucción **NOP** —que no hace nada— entre ambas instrucciones



Ej. de *hazard* de control — JEQ *label* seguida por ADD A, B:

- al ejecutarse la instrucción JEQ *label*, hay que esperar hasta la etapa MEM (ahí está la *Jump Unit*), en el ciclo 4, para saber si el salto a la instrucción etiquetada *label* se va a tomar o no
- la pregunta es, ¿qué hacemos con las tres instrucciones siguientes a JEQ (en el diagrama sólo mostramos una, ADD A, B)?
 - ... ¿las ingresamos al pipeline una tras otra en los tres ciclos siguientes?
 - ... y si las ingresamos, ¿qué pasa si finalmente se toma el salto?

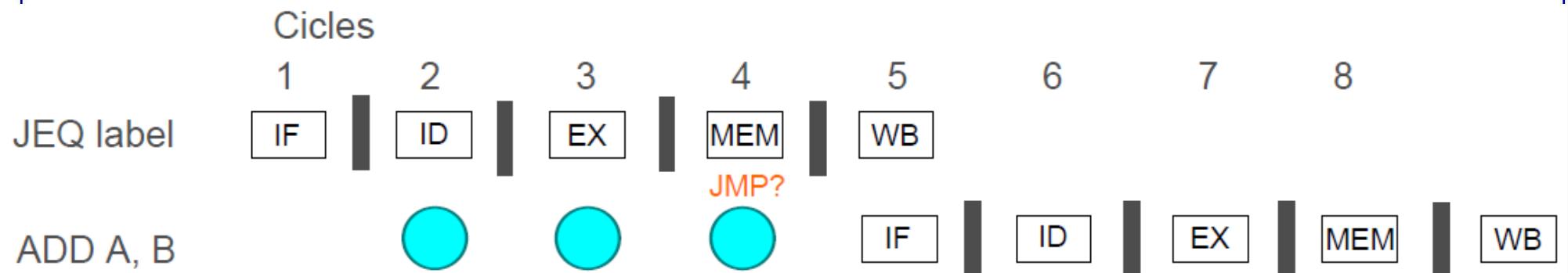


Para los *hazards* de control no hay nada tan eficaz como *forwarding* es eficaz para los *hazards* de datos:

- felizmente, los hazards de control ocurren menos frecuentemente que los hazards de datos

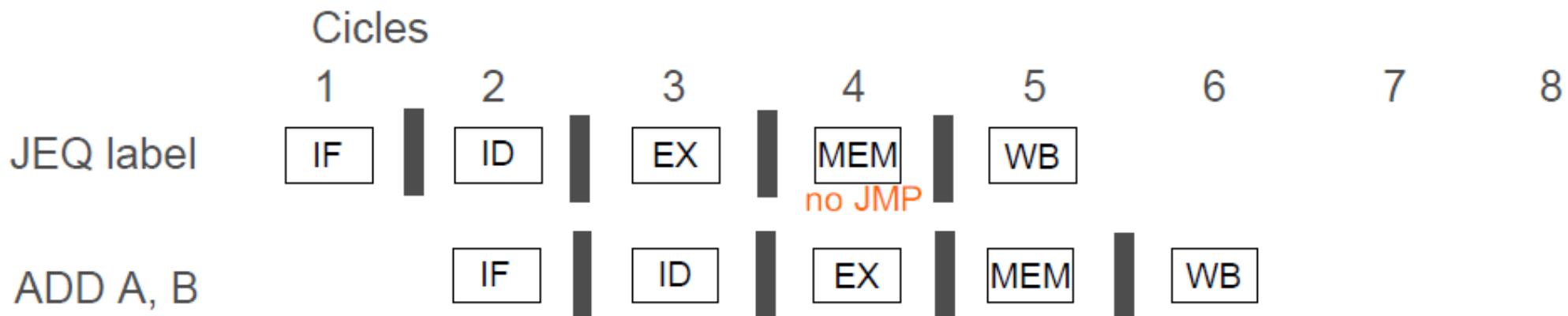
Una solución simple es hacer *stalling* del pipeline durante los ciclos de reloj que sean necesarios, en el ej., 3:

- ... pero esta es una solución muy costosa, en términos de desempeño, para la mayoría de los computadores
 - ... 3 ciclos de reloj en los que el computador no hace nada, y esto es cada vez que hay una instrucción *jump* en el programa



Una solución más sofisticada es emplear **predicción de saltos**:

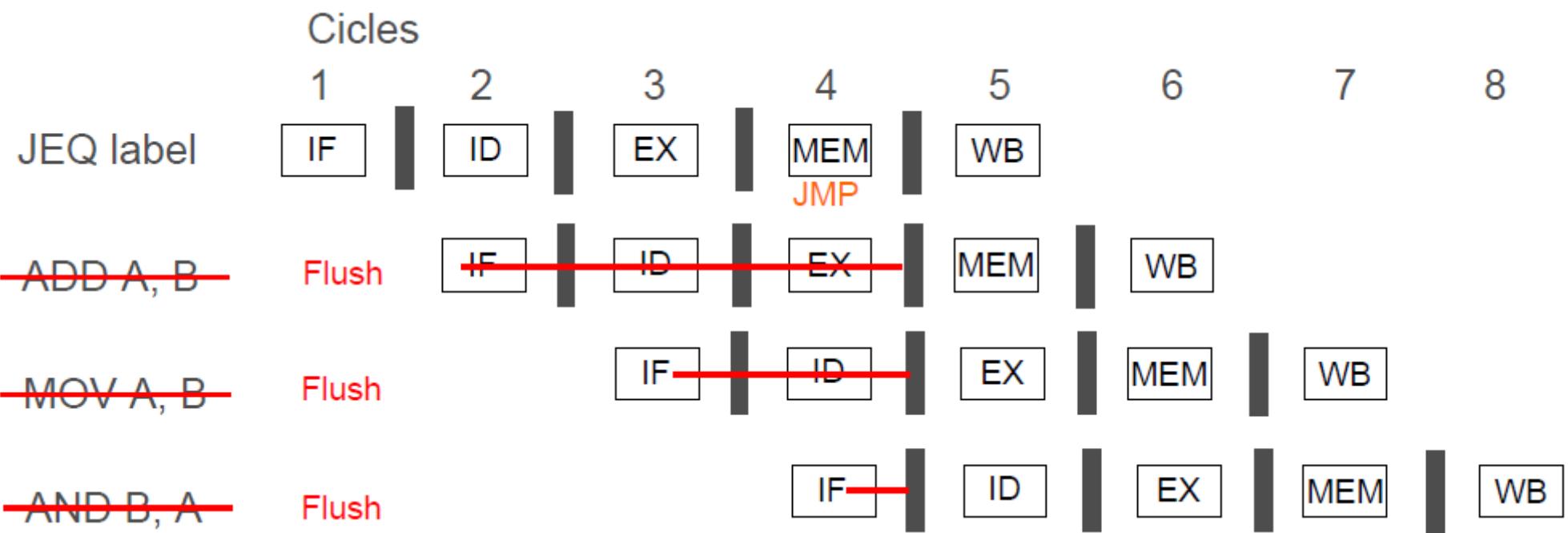
- p.ej., predecir que el salto no va a ser tomado y simplemente continuar con la ejecución secuencial de las instrucciones a continuación de *JEQ label*
... por supuesto, si finalmente el salto es tomado, va a haber que descartar estas instrucciones y continuar con la instrucción en la dirección *label*
- si cuesta poco descartar instrucciones, esta solución reduce el costo de los *hazards* de control en la misma proporción en que la predicción es correcta



¿Y qué hacemos cuando hay que descartar instrucciones que ya ingresaron al pipeline?:

- agregamos una **Control Hazard Unit** a la salida de la *Jump Unit*

... si finalmente el salto es tomado, hay que descartar estas instrucciones y continuar con la instrucción en la dirección *label*



La **Control Hazard Unit**, a la salida de la *Jump Unit* en la etapa *MEM*, envía una señal *flush* a los registros *IF/ID*, *ID/EX* y *EX/MEM*:

- la señal *flush* básicamente pone en 0 los bits que identifican a la instrucción en ejecución en esa etapa

