Paralelismo a nivel de instrucciones (*ILP*):

Pipelining, multiple issue, multithreading

Arquitectura de Computadores – IIC2343

Hans Löbel, Yadran Eterovic (yadran@uc.cl)

2025-2

Paralelismo —hacer varias 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— **esto es lo que vamos a ver a continuación**

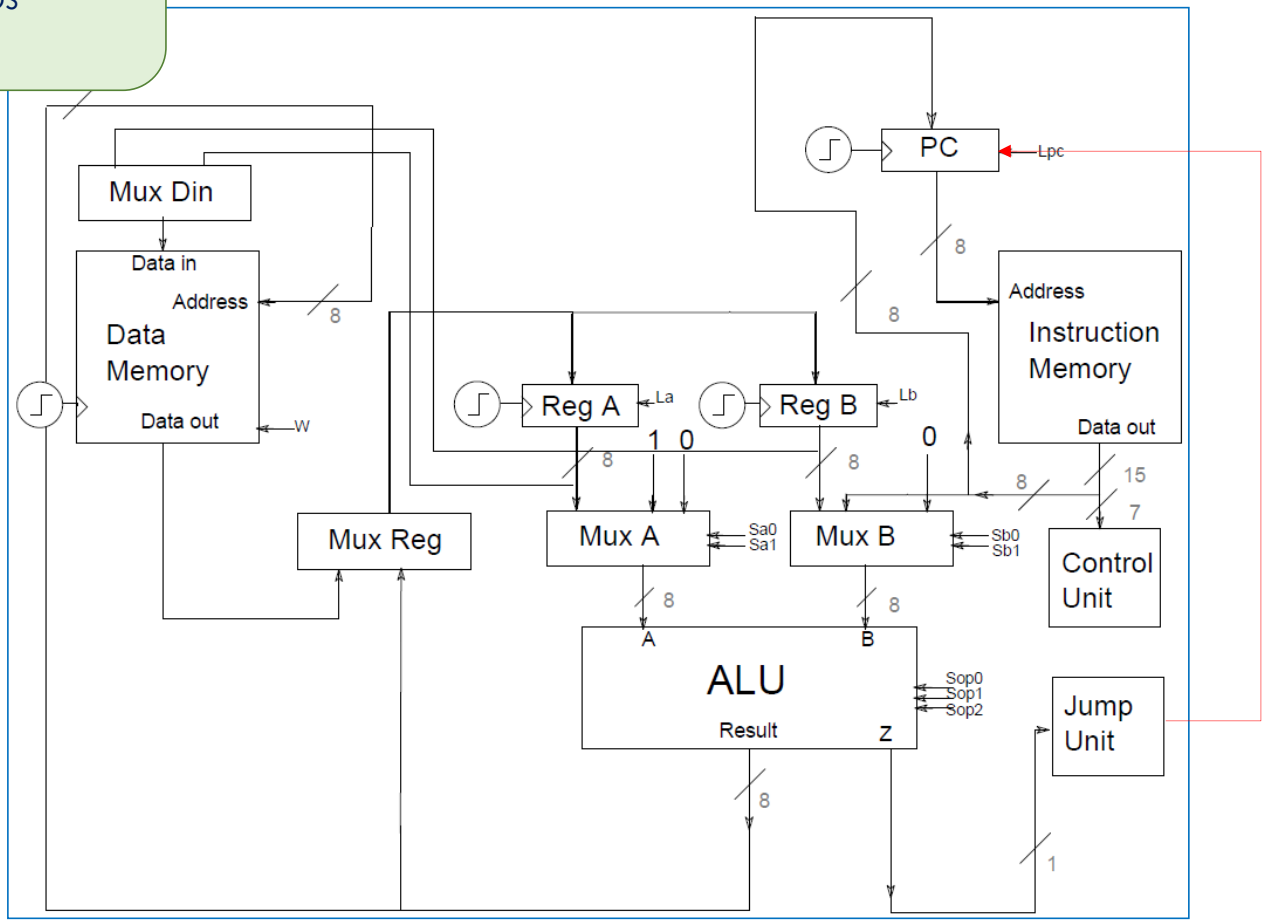
- paralelismo a nivel de procesadores —múltiples CPUs comparten memoria— esto lo vimos la clase pasada

La CPU ejecuta cada instrucción siguiendo una secuencia de 5 pasos

… llamada el ciclo ***fetch-decode-execute***:

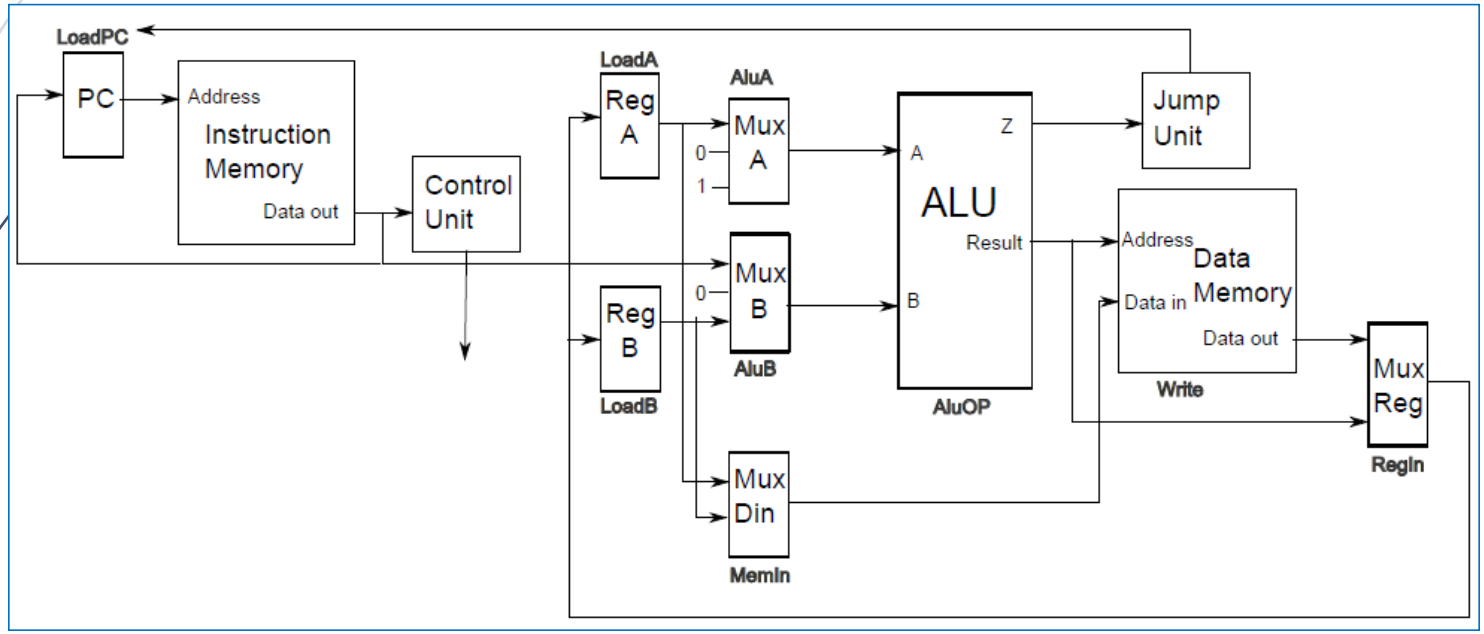
1. Traer (*fetch*) la instrucción desde la memoria (la cache de instrucciones)
2. Decodificar la instrucción y leer los registros involucrados
3. Ejecutar la operación o calcular una dirección de memoria
4. Tener acceso a un operando en la memoria (la cache de datos), si es necesario
5. Escribir el resultado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

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

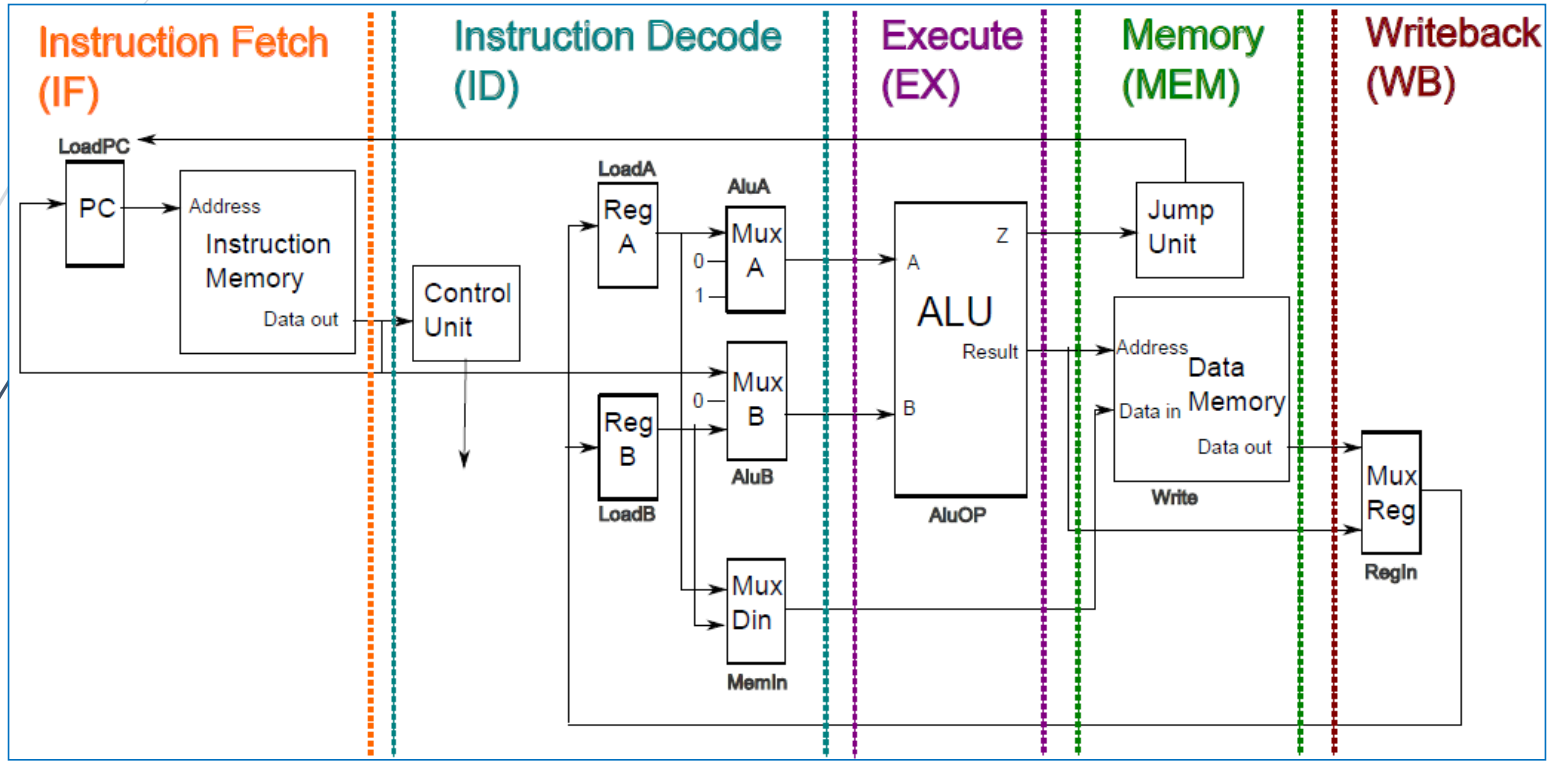


Tomemos esta versión simplificada del computador básico:
Data Memory, *Instruction Memory*, *PC*, ALU, registros *A* y *B*, *Control Unit* y *Jump Unit*

… y 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 (diap. 3)



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 operar en paralelo

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

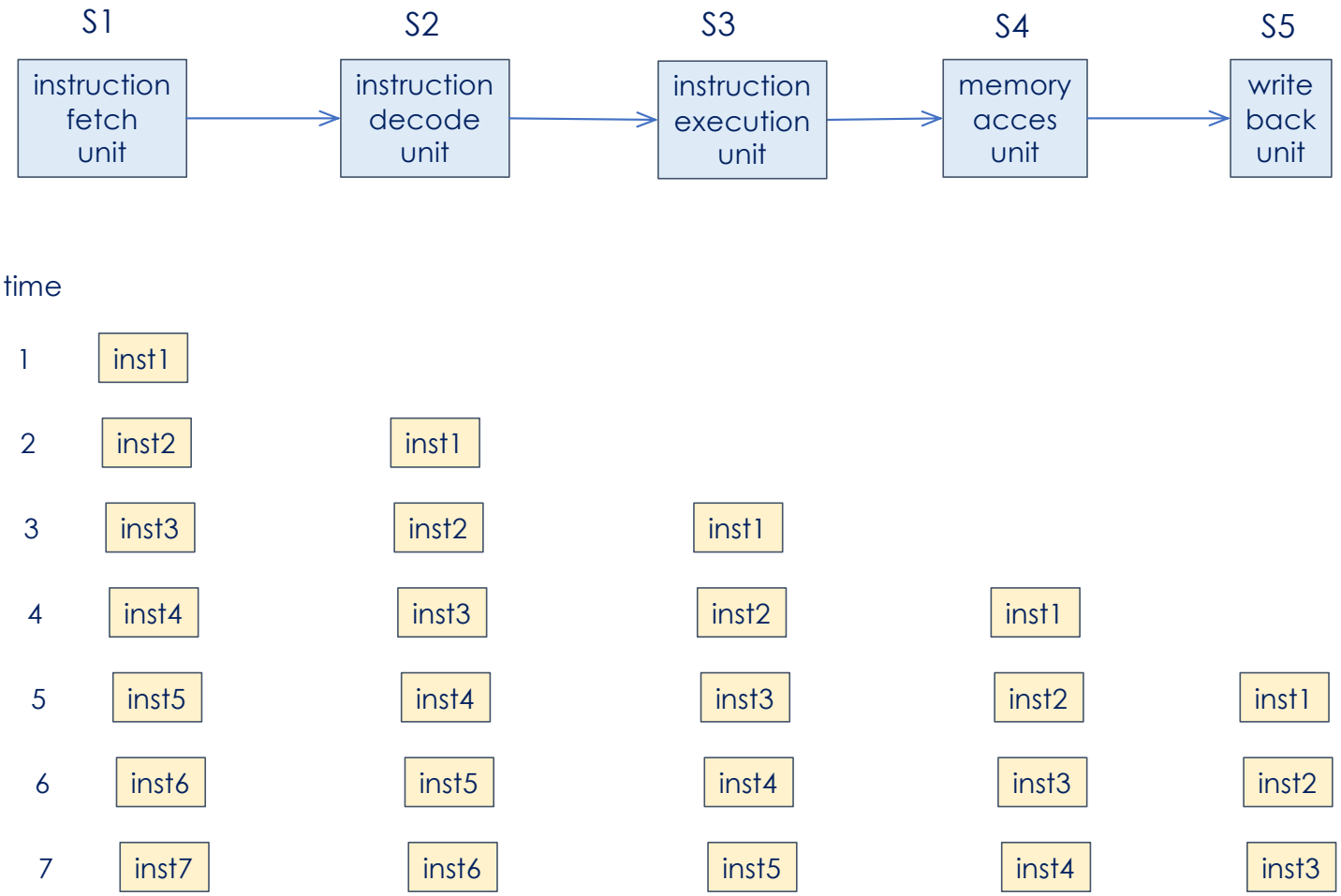
- en un mismo ciclo del reloj las distintas instrucciones ocupan distintas unidades del pipeline

- usada desde 1985

**Pipeline*: Proceso secuencial de varias etapas independientes*

Supongamos el pipeline de 5 etapas (diap. 3):

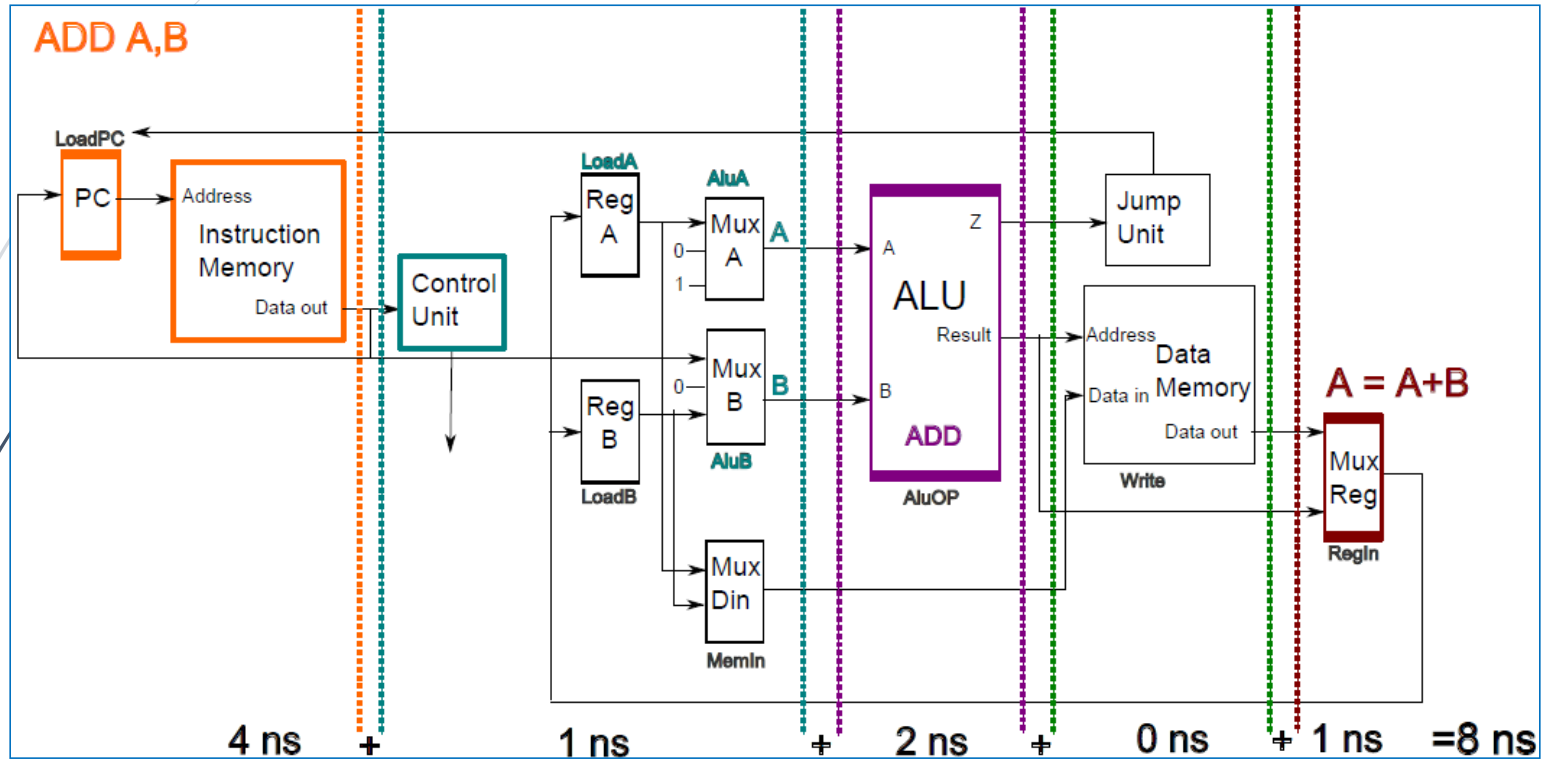
1. Traer la instrucción desde la cache de instrucciones —*instruction fetch*
2. Decodificar la instrucción en la unidad de control —*instruction decode*
3. Ejecutar la instrucción en ALU —*instruction execution*
4. Tener acceso a cache de datos —*memory acces*
5. Escribir el resultado en un registro —*writeback*



Para concretar las ideas, volvamos al computador básico
Consideremos las siguientes instrucciones y su división en etapas:

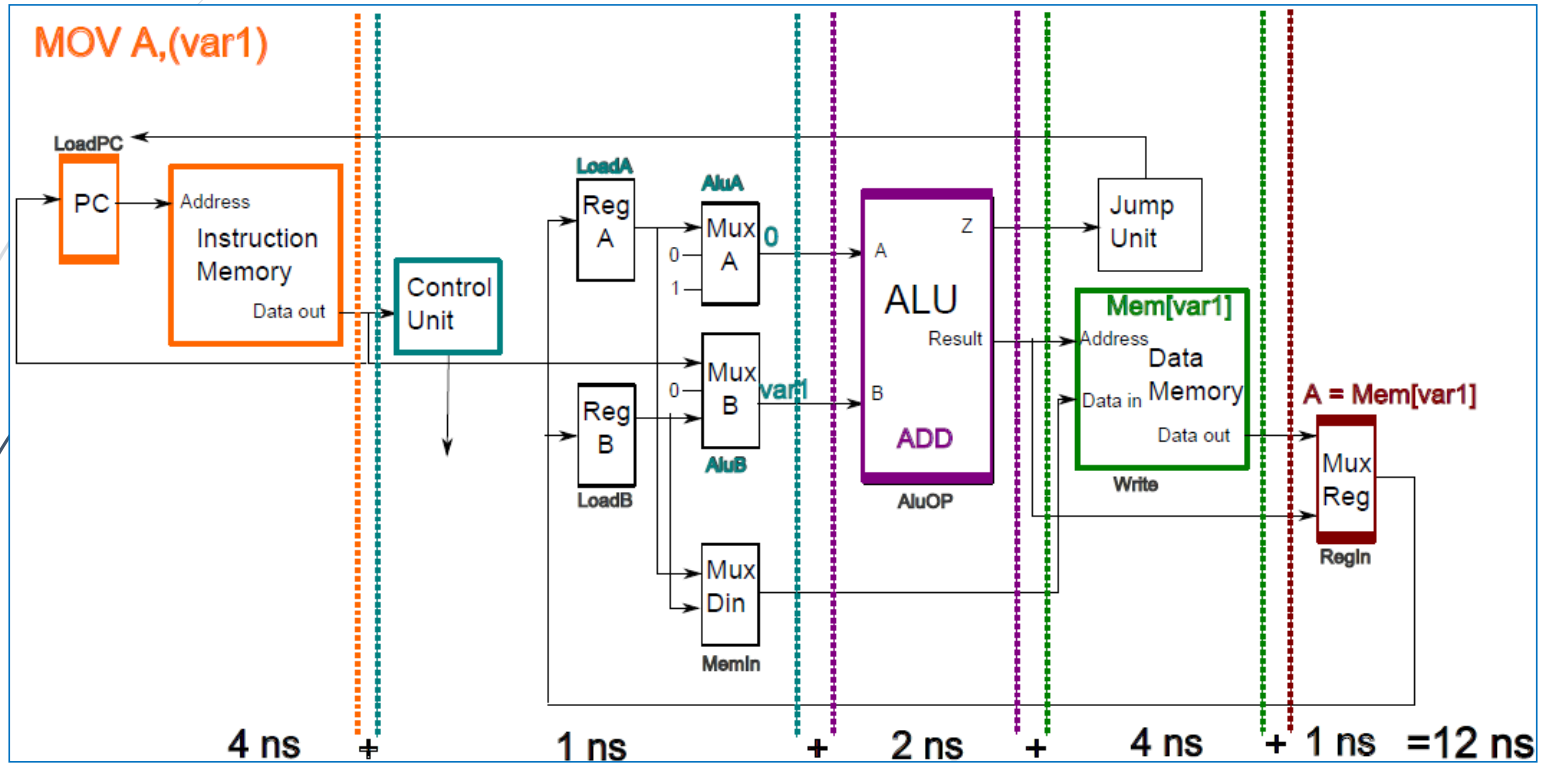
`ADD A,B`	`MOV A,(var)`	`MOV (var),A`	`JEQ label`
1. Traer instruccción	1. Traer instrucción	1. Traer instrucción	1. Traer instrucción
2. Decodificar instrucción y leer registros *A* y *B*	2. Decodificar instrucción	2. Decodificar instrucción y leer registro *A*	2. Decodificar instrucción y leer registros *A* y *B*
3. Calcular en *ALU*	3. Calcular en *ALU*	3. Calcular en *ALU*	3. Calcular en *ALU*
4. Escribir en registro *A*	4. Leer de *Data Memory*	4. Escribir en *Data Memory*	4. Decidir en *Jump Unit*
	5. Escribir en registro *A*		

Etapas que participan en la ejecución de **ADD A,B** y sus duraciones
(en un procesador moderno, los tiempos pueden ser 10 veces menores **)**



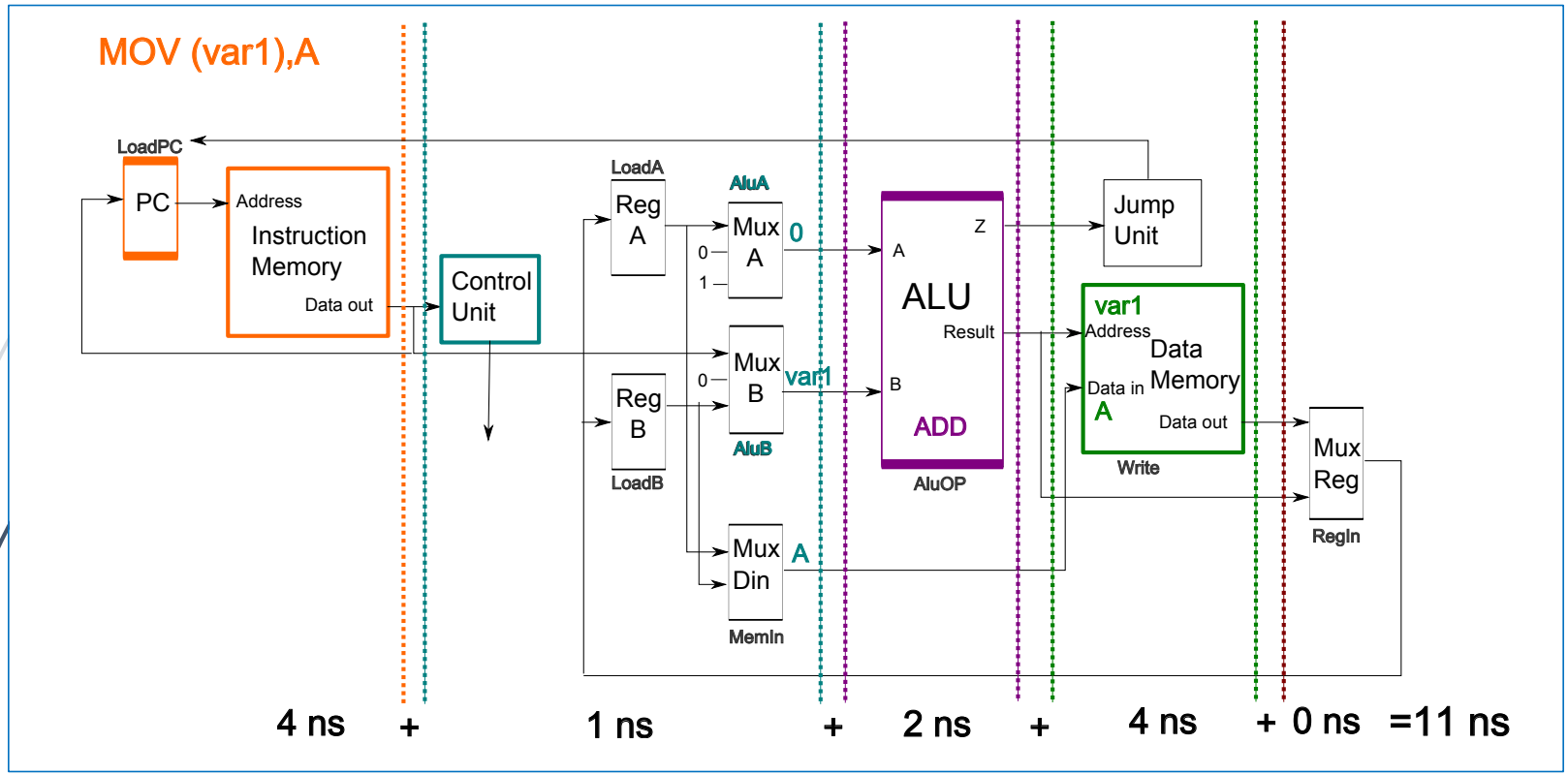
ADD A,B no hace acceso a memoria

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



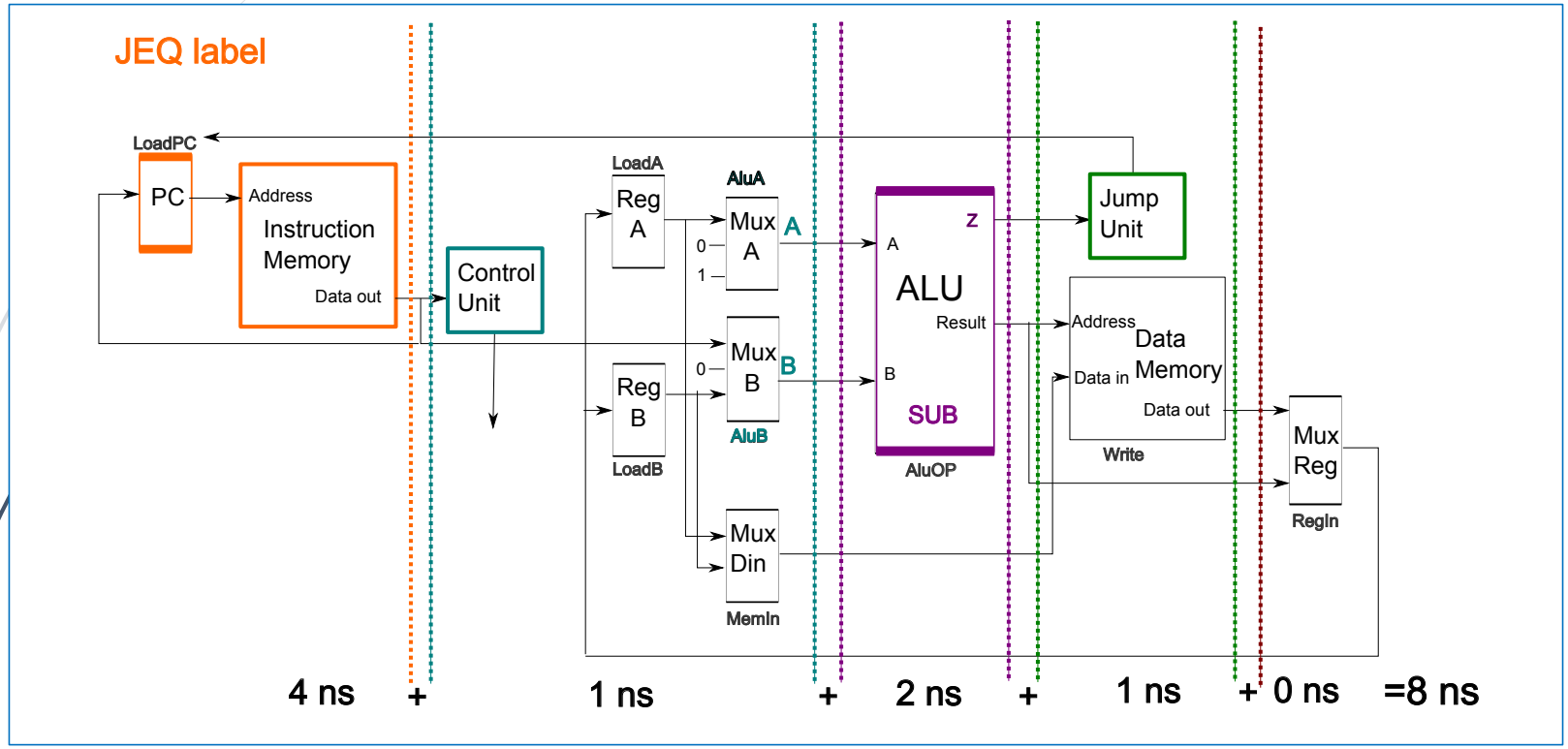
MOV A,(var1) hace uso de **todas** las etapas

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



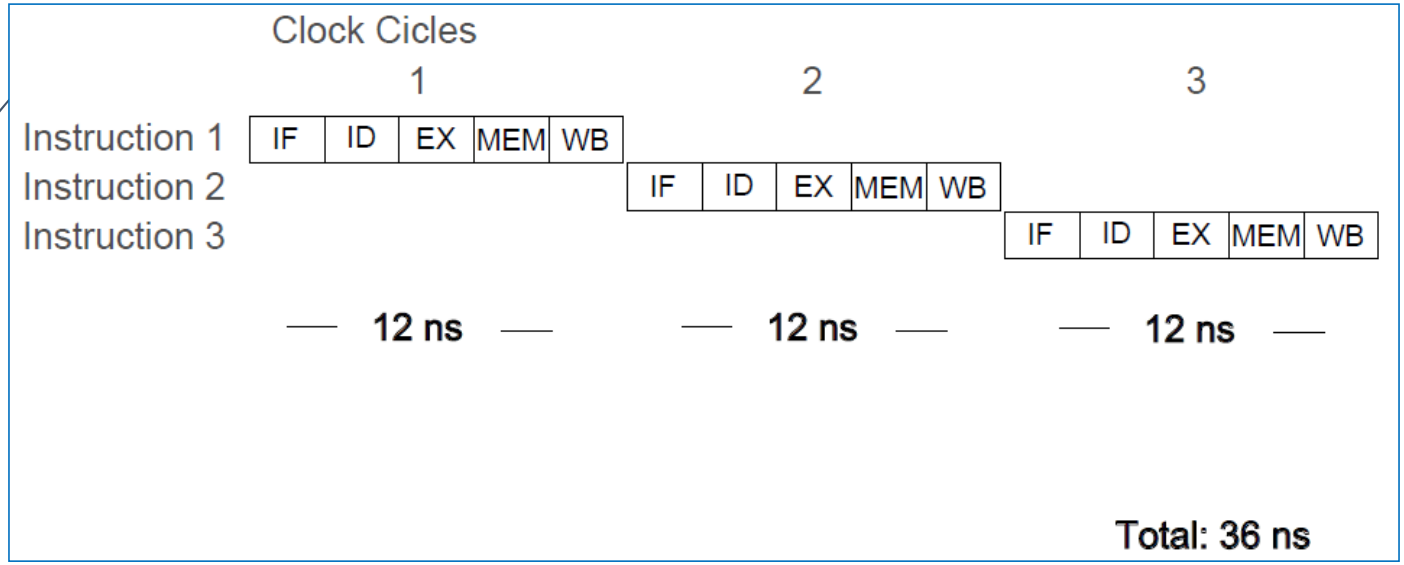
MOV (var1),A no escribe en los registros

Etapas que participan en la ejecución de **`JEQ label`** y sus duraciones



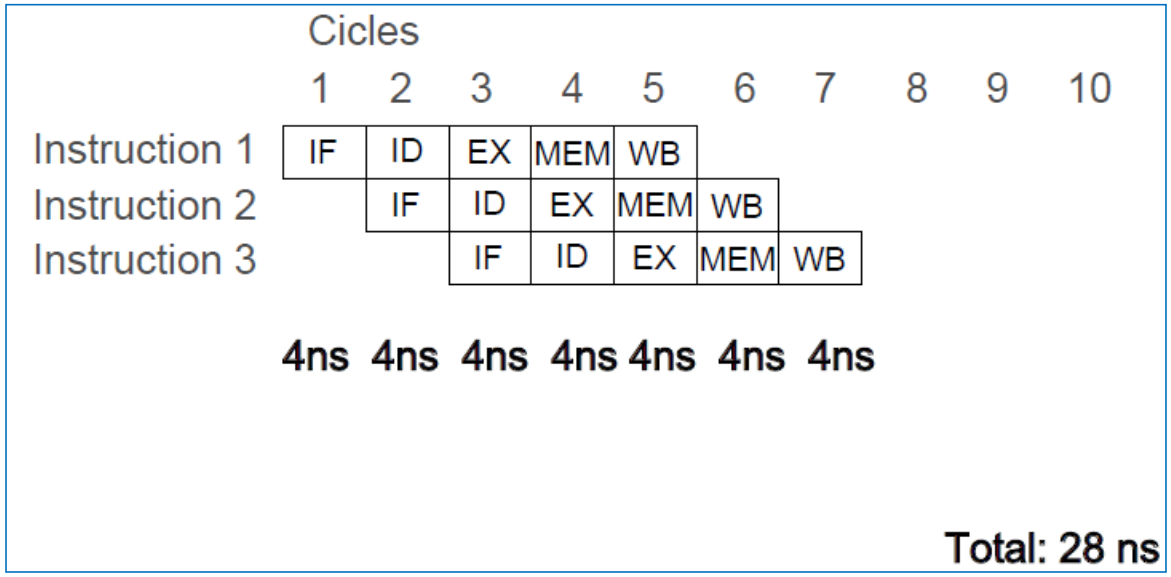
`JEQ label` no escribe en los registros

Cuando ejecutamos una instrucción tras otra, esperando 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**



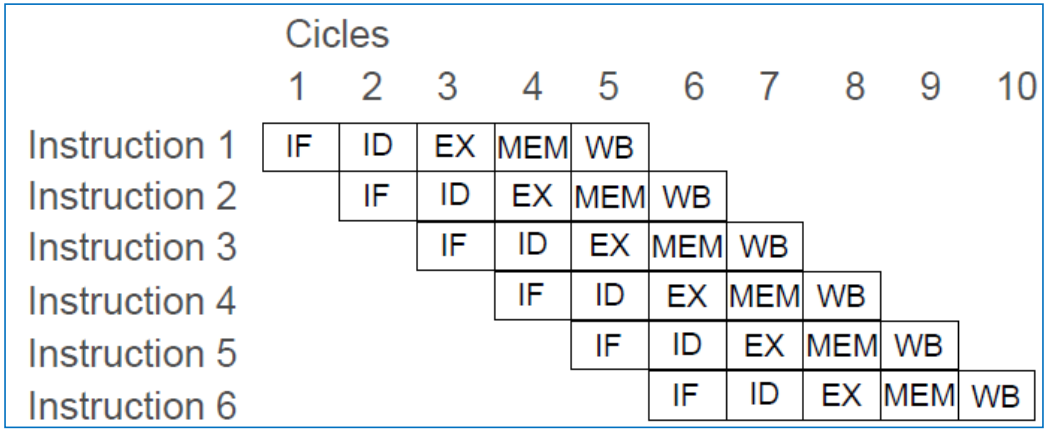
En cambio, si aplicamos *pipelining* y traslapamos las ejecuciones de varias instrucciones, de modo que cada una ejecute una etapa distinta, el ciclo del reloj puede ser tan rápido como la **etapa más lenta**:

- 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**



La disminución del tiempo total va a ser 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**



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

- pareciera que el computador corre a 50 MIPS

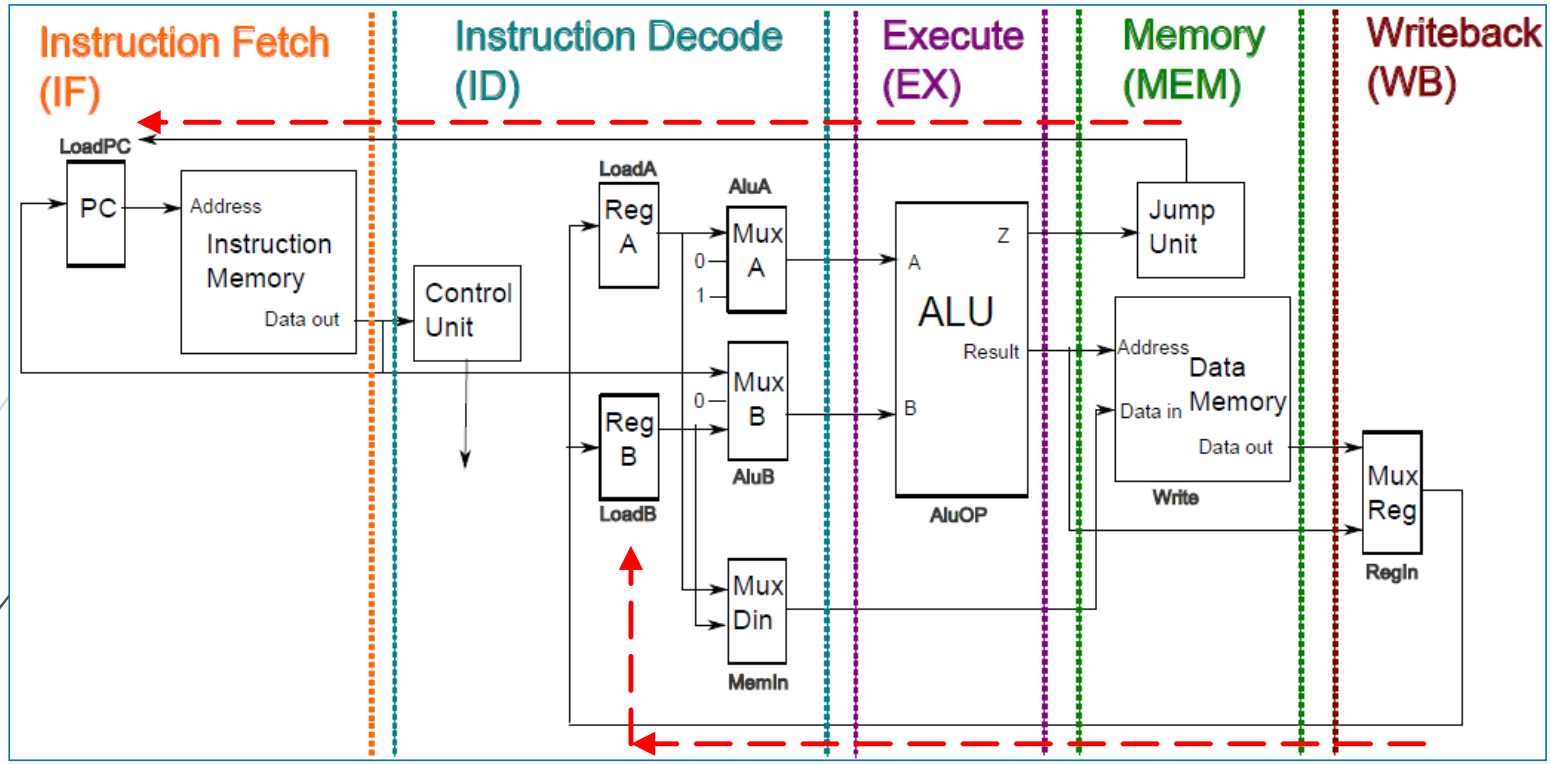
  … pero en realidad como cada 4 ns se termina de ejecutar una nueva instrucción

  … la tasa de procesamiento es 250 MIPS

millones de instruc-ciones por segundo

Pipelining mejora el desempeño por la vía de *aumentar la tasa de ejecución de instrucciones*

... en vez de disminuir el tiempo de ejecución de una instrucción individual

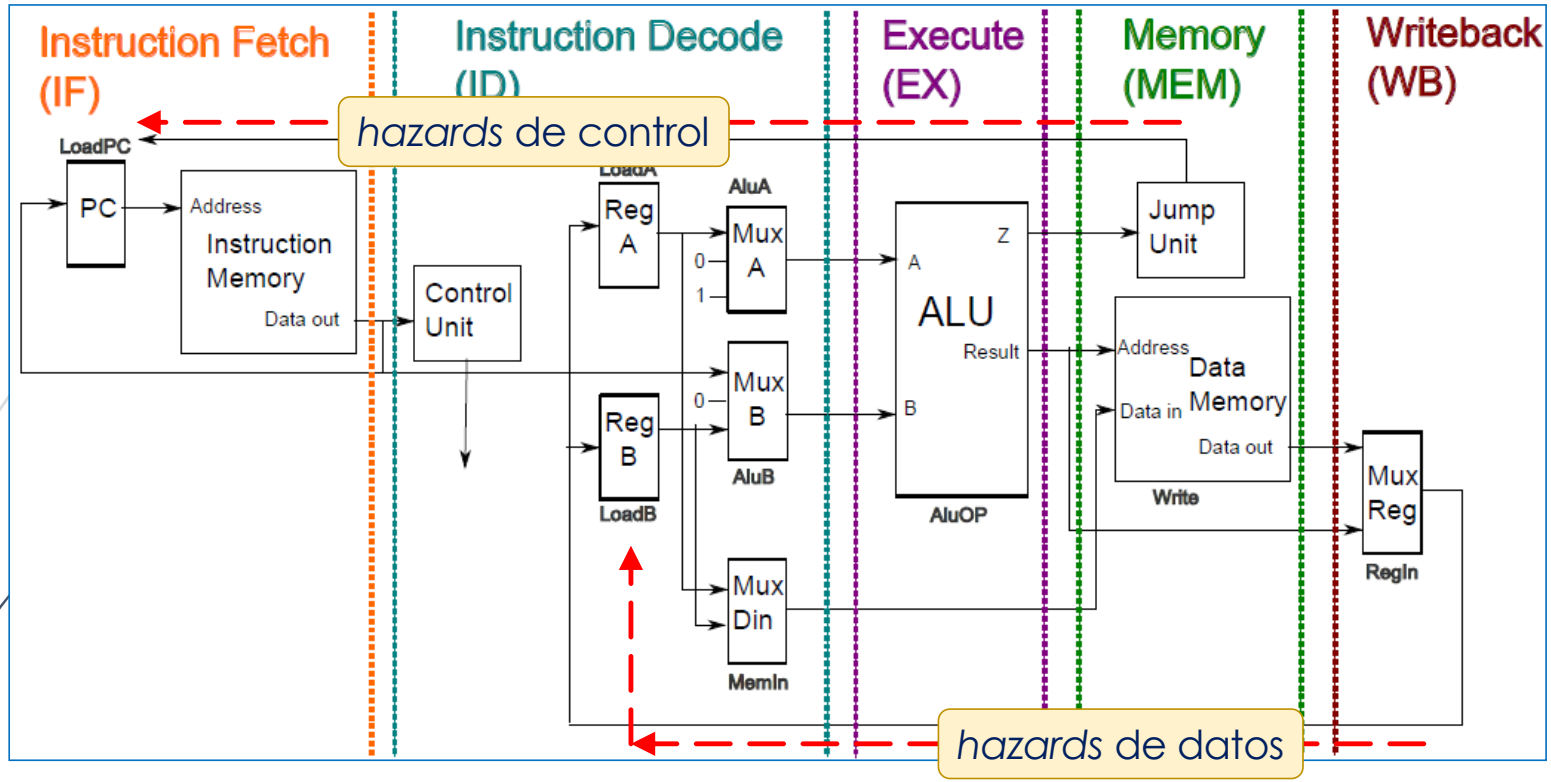


Instrucciones y datos se mueven de izquierda a derecha a lo largo de las 5 etapas

… **excepto** (las dos flechas rojas 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*



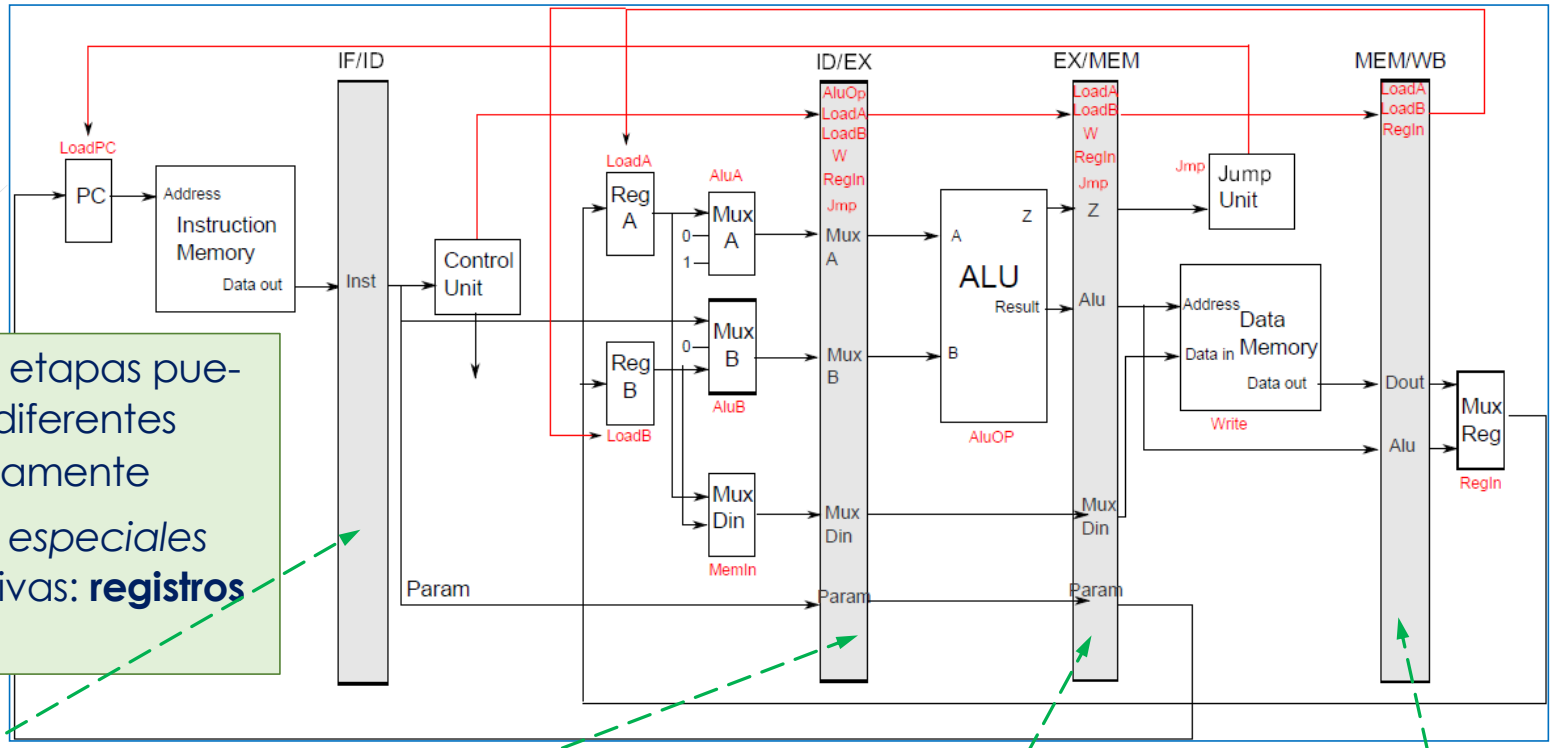
Estas dos excepciones no afectan la ejecución de la instrucción vigente

… pero pueden afectar a instrucciones posteriores en el *pipeline*

… dando origen a dos tipos de problemas o *hazards*

Para que las diferentes etapas puedan estar ejecutando diferentes instrucciones simultáneamente

… agregamos *registros especiales* entre etapas consecutivas: **registros del pipeline**



Registro ***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*)

Registro ***ID/EX***

Almacena los valores de ambos registros, y posiblemente un *offset* constante, todos especificados en los operandos de la instrucción; y también la dirección *PC*+4

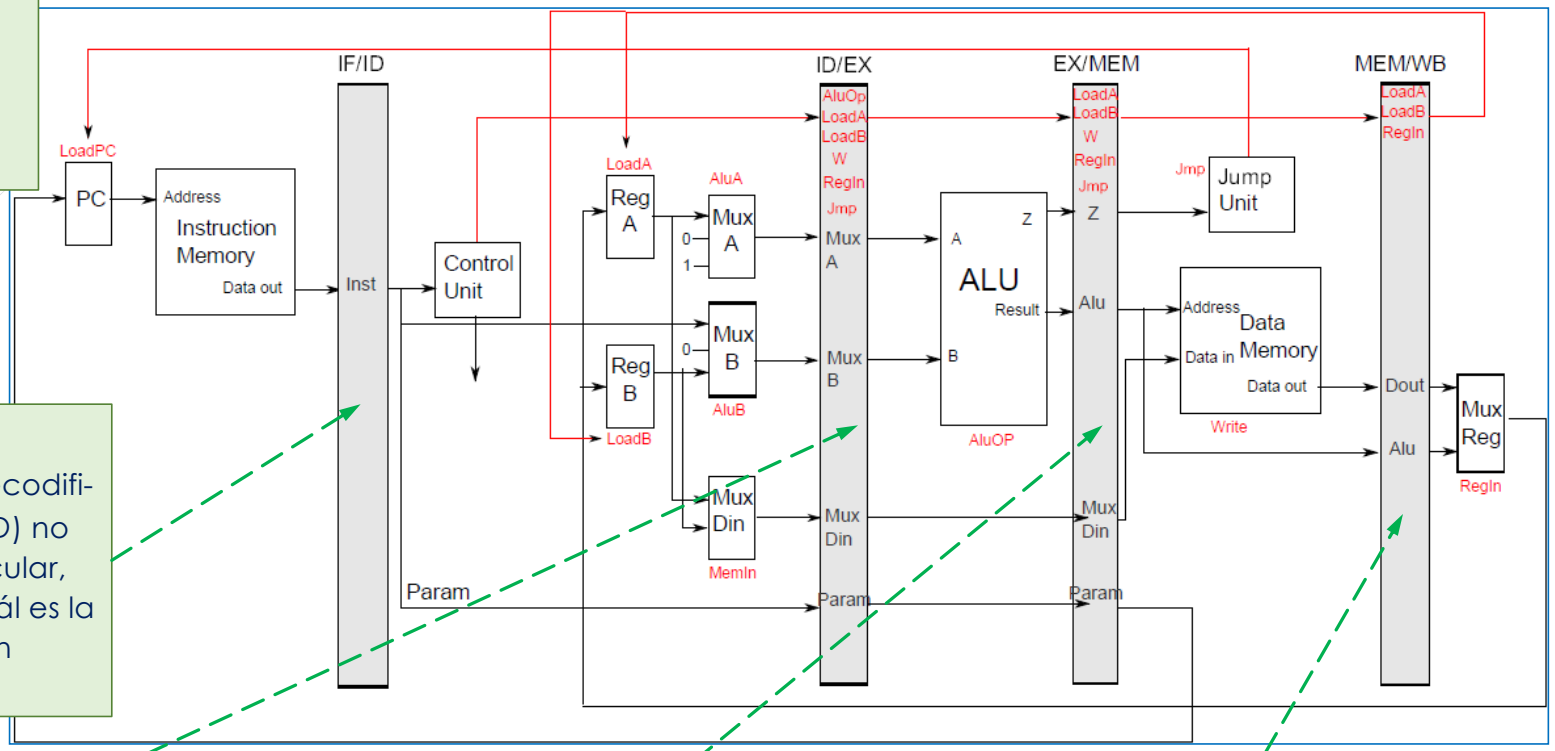
Registro ***EX/MEM***

Almacena el resultado de la operación ejecutada en la ALU, que puede ser una dirección de memoria, y posiblemente el valor de uno de los registros

Registro ***MEM/WB***

Almacena el dato leído desde la memoria

Los registros del pipe-line también almace-nan información de control (líneas rojas)



IF/ID

Durante la lectura (*IF*) y decodifi-cación de la instrucción (*ID*) no se controla nada en particular, ya que aún no se sabe cuál es la instrucción: *IF* e *ID* ejecutan siempre lo mismo

ID/EX

Almacena información de control creada durante la decodificación de la instrucción (*ID*); algunas de estas líneas son usadas durante la ejecución de la instrucción (*EX*) …

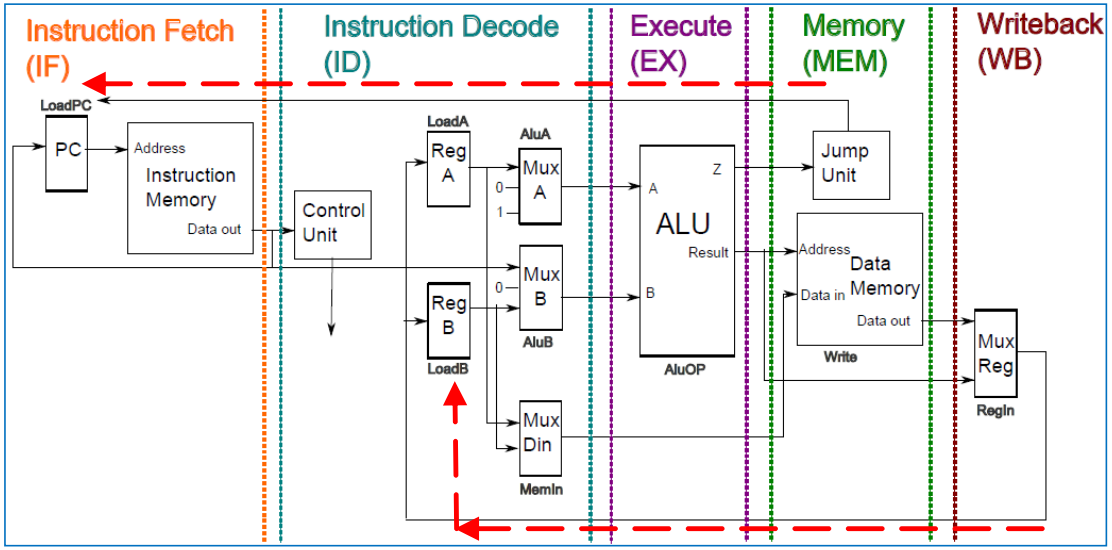
EX/MEM

… las otras son almacenadas en este registro; similarmente, algu-nas de estas últimas líneas son usadas en el acceso a memoria (*MEM*) …

MEM/WB

… las restantes son almacenadas en este registro para ser usdas durante el *writeback* (*WB*) del resultado en uno de los registros *A* o *B*

Al usar *pipelining* se producen situaciones —***hazards***— en que la próxima instrucción no se puede ejecutar en el siguiente ciclo de reloj



***Hazards* de control** (diaps. 37 – 48):

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 instrucciones) no es el que el pipeline esperaba

***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

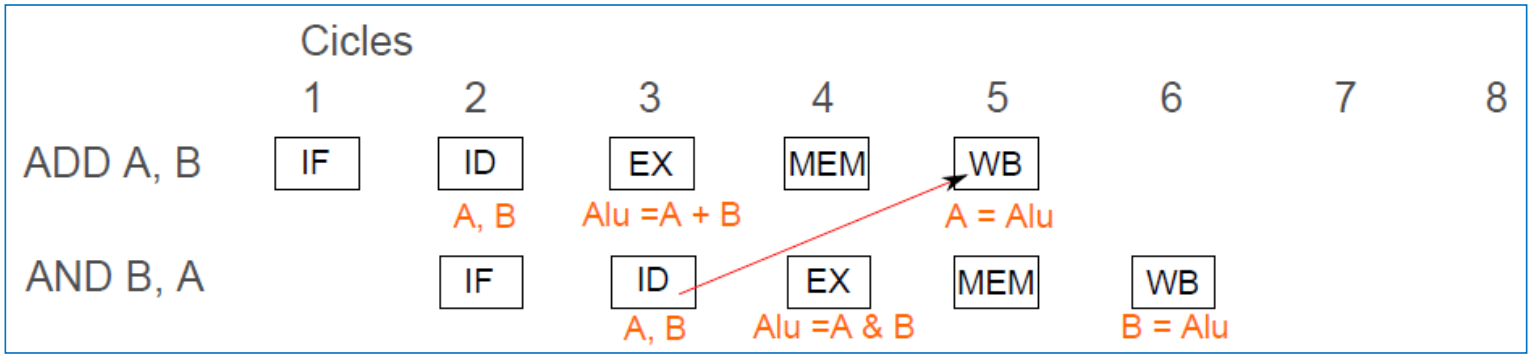
***Hazards* de datos** (diaps. 24 – 36):

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

Ej. de ***hazard* de datos**: ejecución de la instrucción `ADD A,B`
… seguida por la ejecución de `AND B,A` :

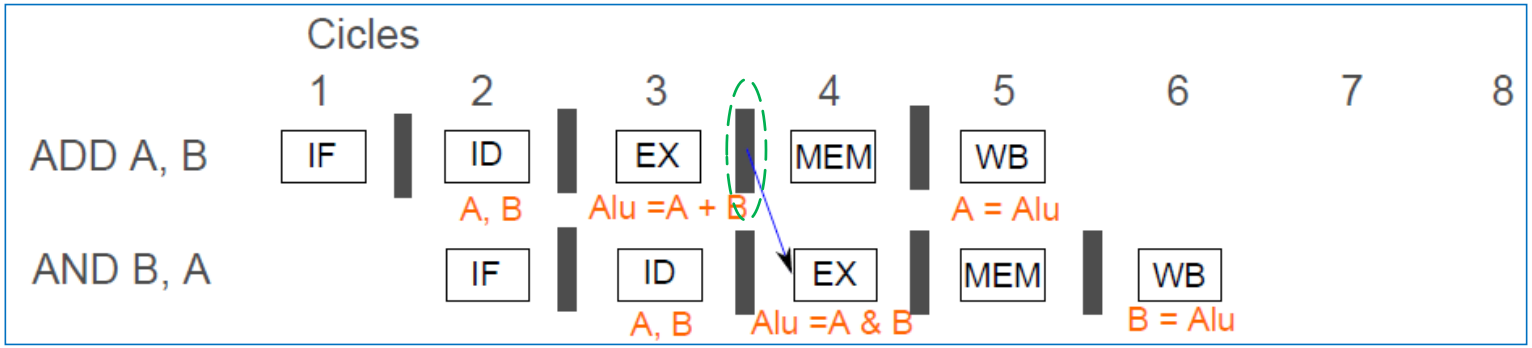
- naturalmente, la ejecución de `AND B,A` espera que el valor del registro *A* que va a usar sea el valor actualizado, después de la ejecución de `ADD A,B`
- sin embargo, como se ve en el diagrama, `AND B,A` necesita los valores de los registros cuando pasa por la etapa *ID*, durante el ciclo 3

  … mientras que 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)



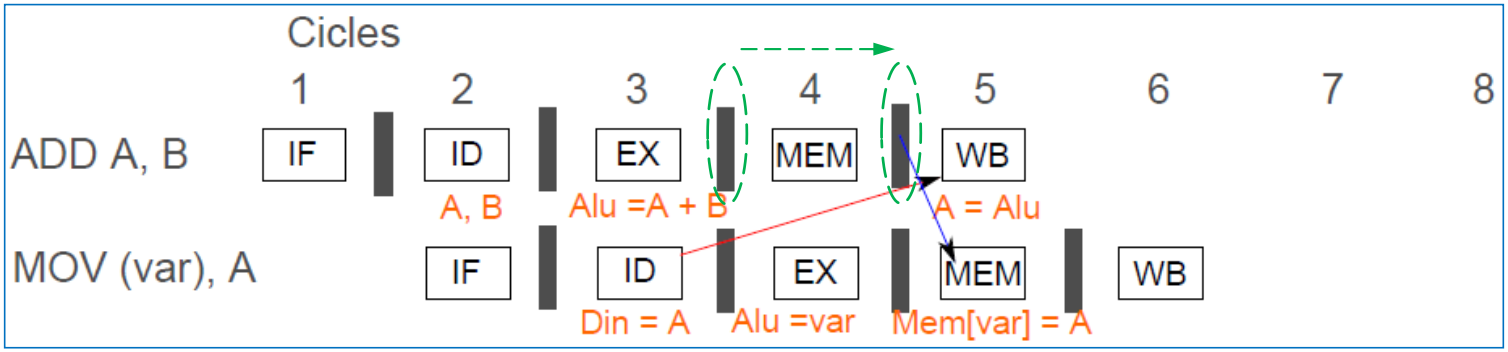
La mayoría de los *hazard* de datos tiene una solución simple:

- incluyamos en el diagrama los registros del pipeline y consideremos el registro *EX/MEM*
- cuando **`ADD A,B`** pasa por su etapa *EX*, en el ciclo 3, almacena en *EX/MEM* el 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 *EX/MEM* (aunque aún no en *A*) 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*



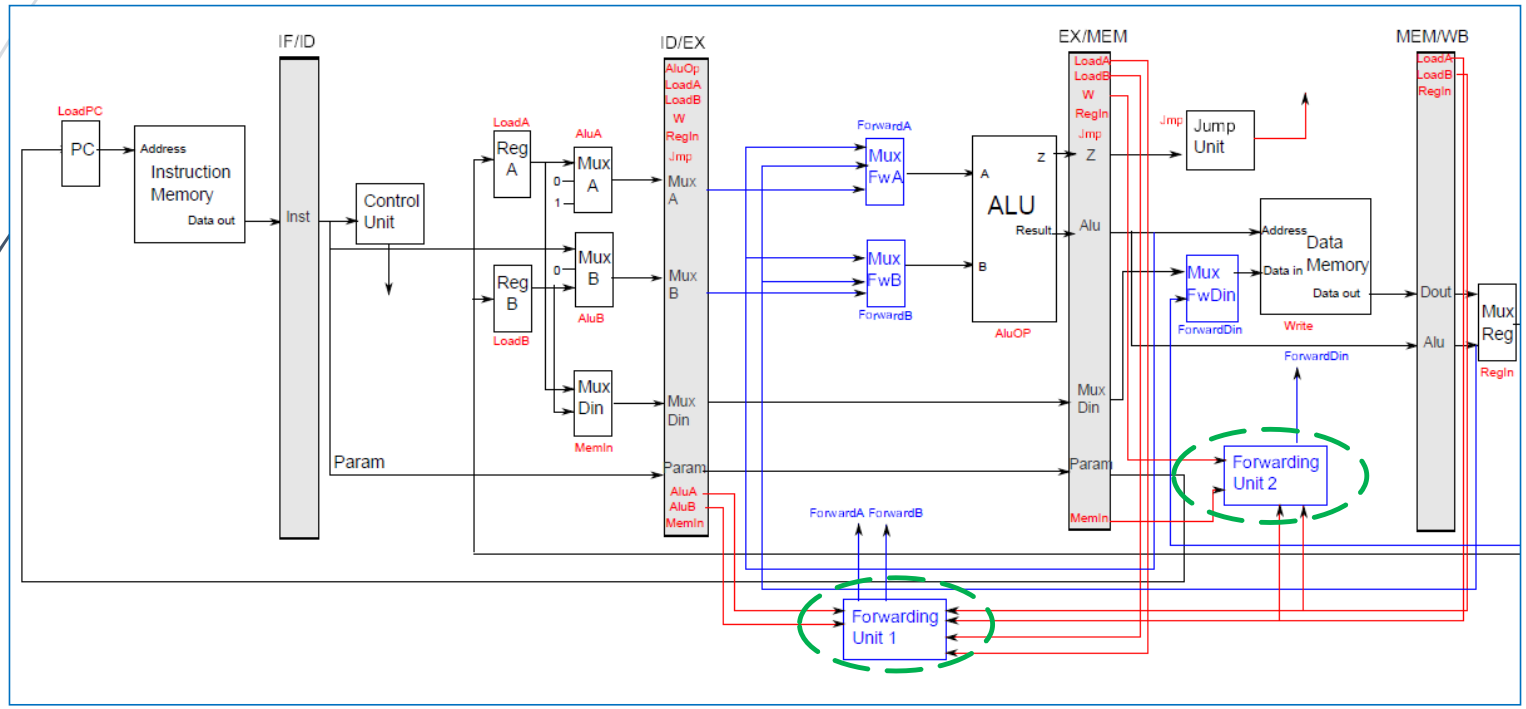
Otro ej., en que `ADD A,B` es ahora seguida por `MOV (var),A`:

- `MOV (var),A` necesita el valor (actualizado) del registro *A* cuando pasa por etapa *ID* en el ciclo 3; pero *A* sólo es actualizado en el ciclo 5
- `ADD A,B` almacena la suma, ejecutada en el ciclo 3, en *EX/MEM*, quien la propaga a *MEM/WB* en el ciclo 4
- así, cuando `MOV (var),A` pasa por su etapa *MEM*, en el ciclo 5, el valor que necesita ya está disponible en *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 del pipeline

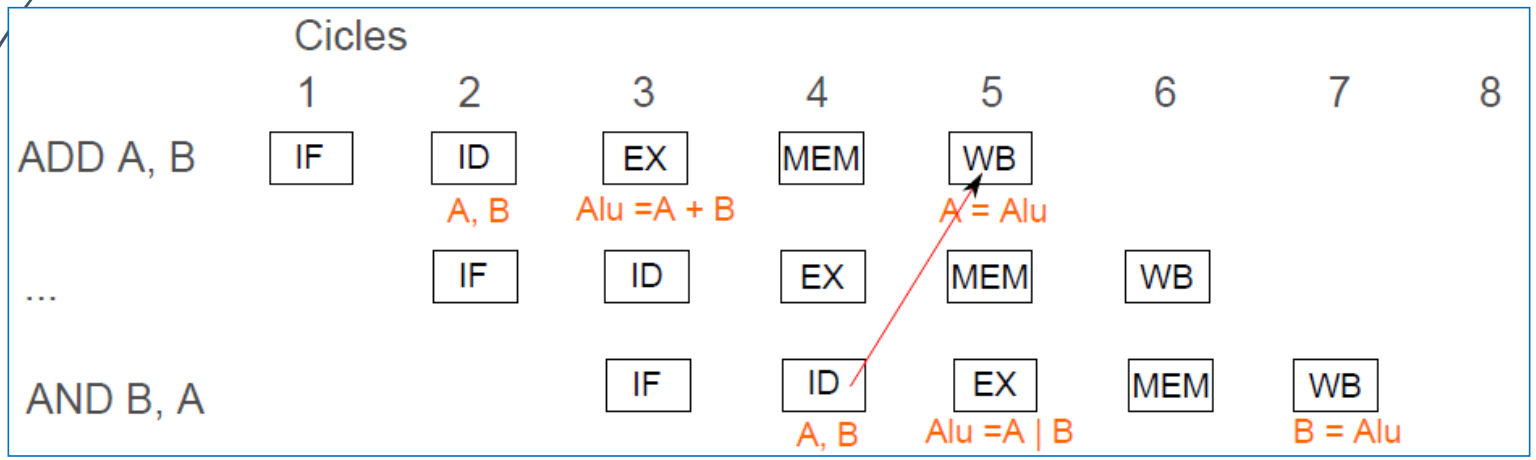
… y producen como output nuevas señales de control que controlan los nuevos multiplexores, en las entradas *A* y *B* de la ALU, y en la entrada *Data in* de la *Data Memory*



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 anterior)

  …⟶ el *forwarding* tiene que hacerse desde *MEM/WB* (en lugar de *EX/MEM*, como en la diap. 26)

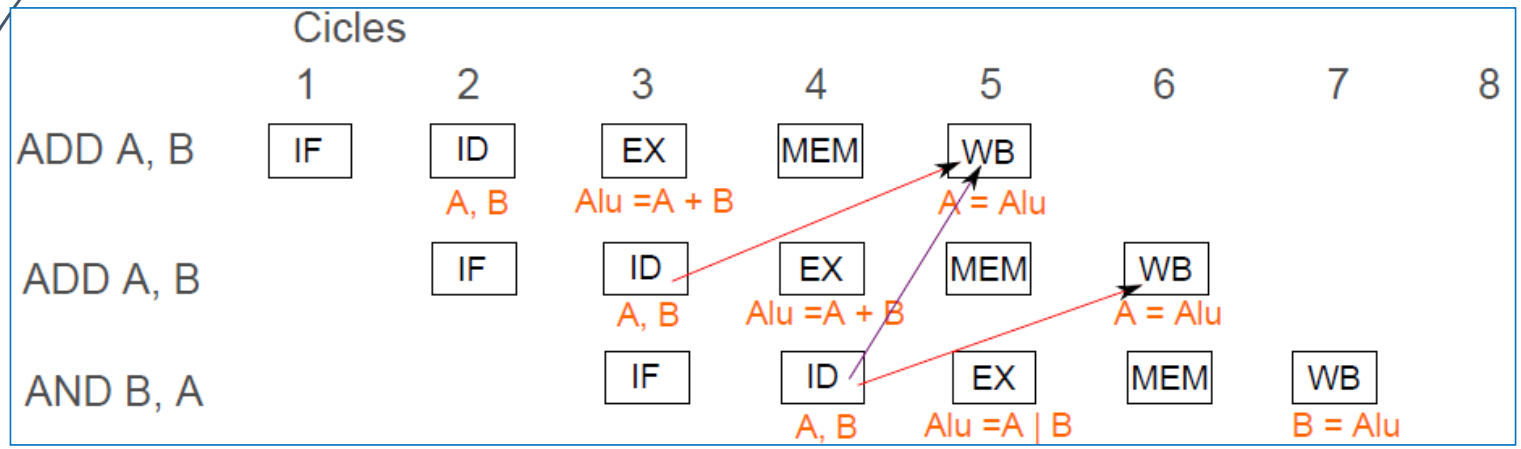


¿Qué pasa si la dependencia de datos de una instrucción es con respecto a las dos instrucciones anteriores?

- 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 la segunda instrucción y no con respecto a la primera

  ⟶ el *forwarding* va a provenir ahora de *EX/MEM*



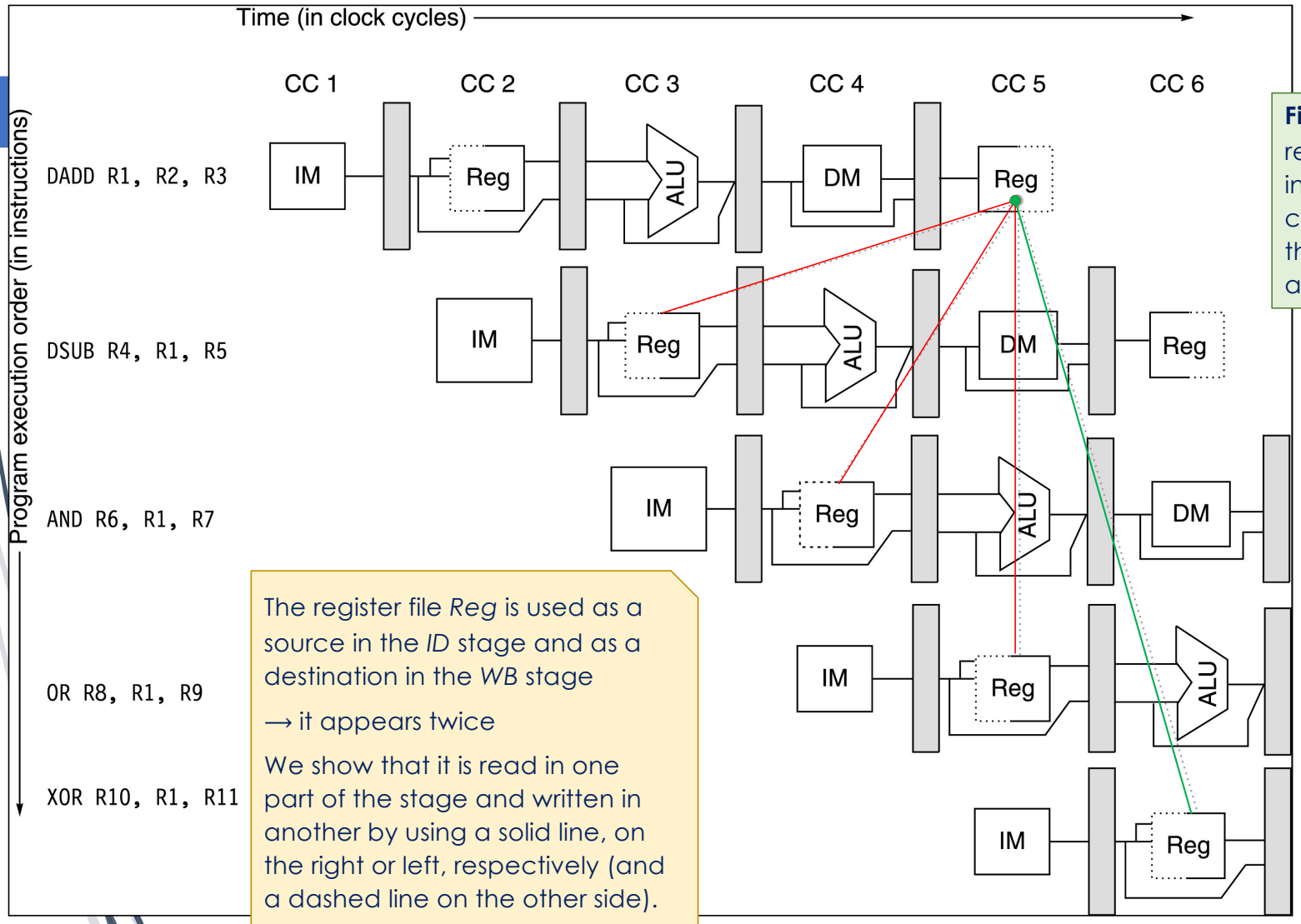


Figure C.4 The use of the result of the **DADD** instruction in the next three instructions causes a *hazard*, because the register is not written until after those instructions read it.

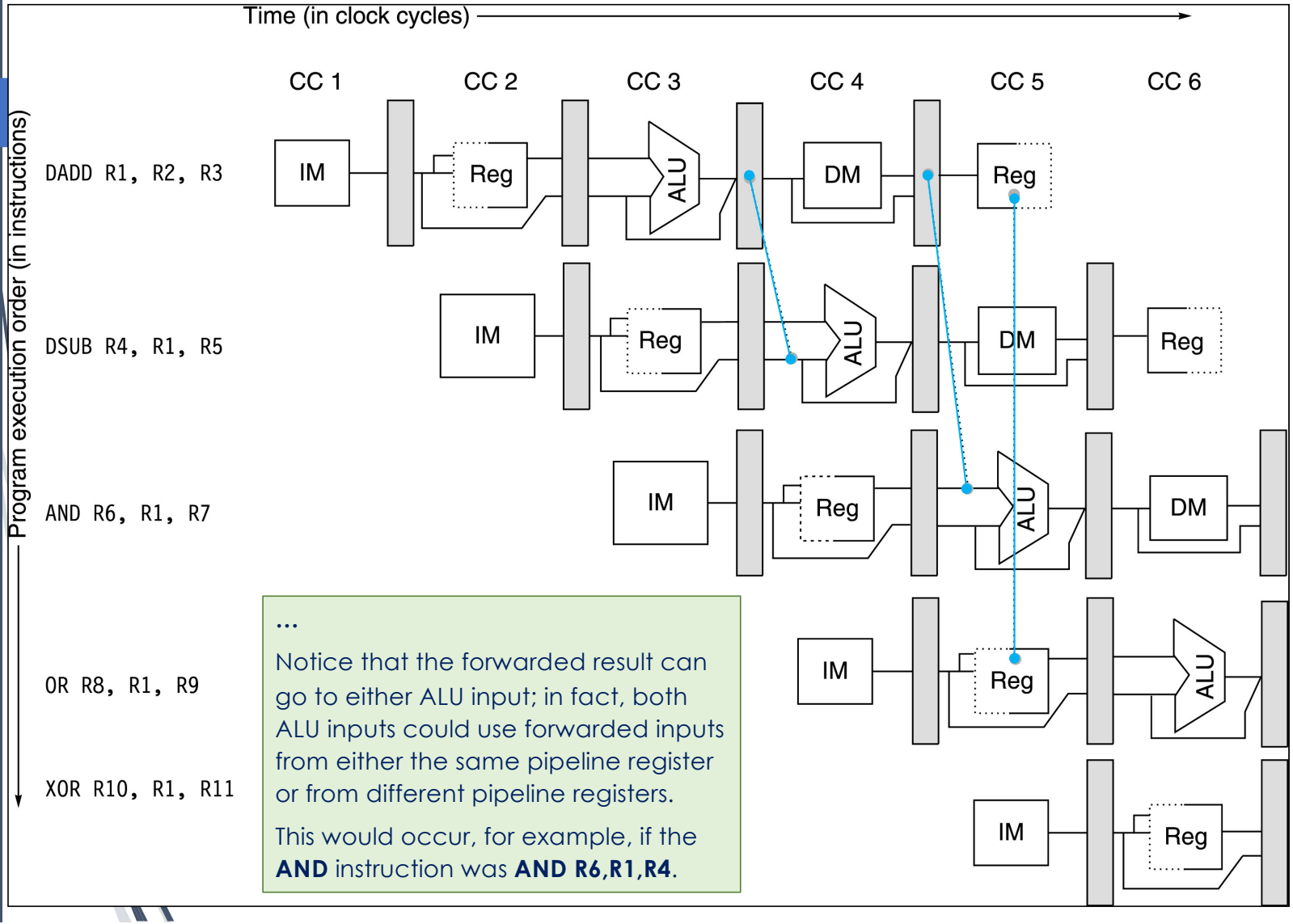
The register file *Reg* is used as a source in the *ID* stage and as a destination in the *WB* stage

→ it appears twice

We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively (and a dashed line on the other side).

Las mismas situaciones ya descritas, pero con ej. tomado de J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach" (6th ed.) Morgan Kaufman 2019

La solución para **DSUB** y **AND** basada en *forwarding* se muestra en la próx. diap.



...

Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers.

This would occur, for example, if the **AND** instruction was **AND R6,R1,R4**.

Figure C.5 A set of instructions that depends on the **DADD** result uses forwarding paths to avoid the data hazard.

The inputs for the **DSUB** and **AND** instructions forward from the pipeline registers to the first ALU input.

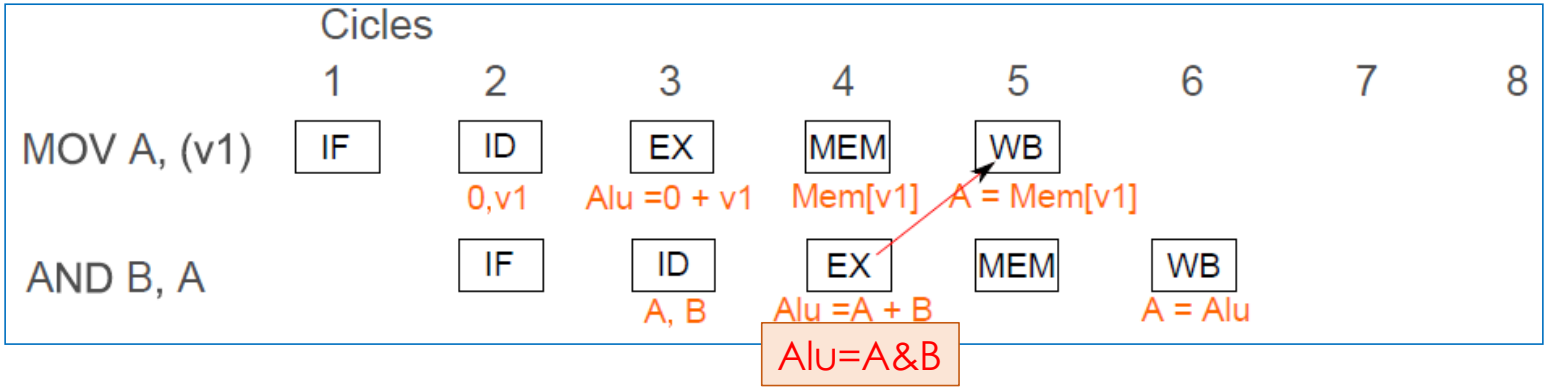
The **OR** receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate.

...

Veamos un último caso de *data hazard*:

```
MOV A,(v1)
AND B,A
```

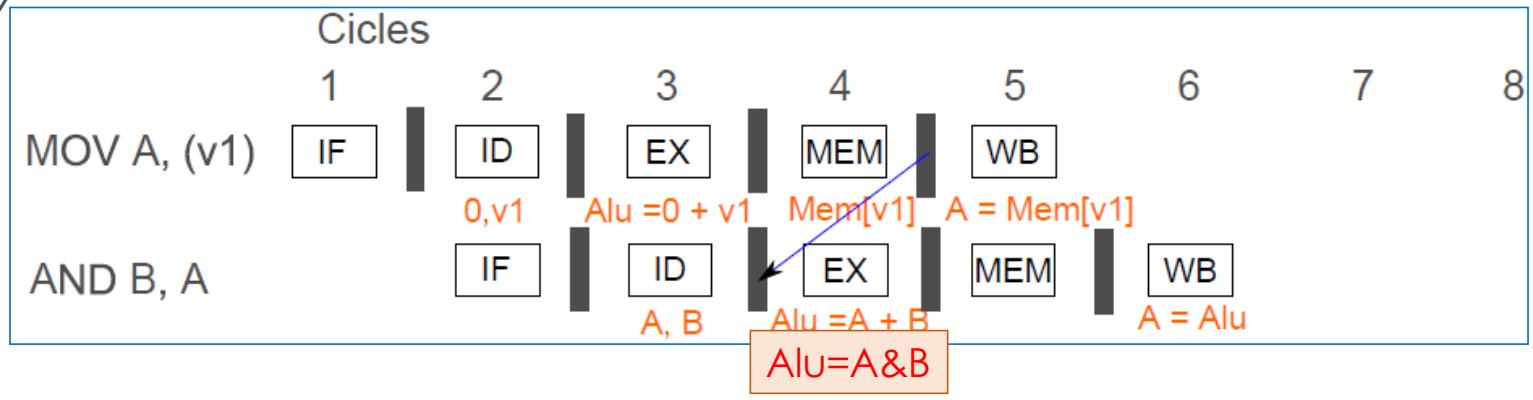
- la instrucción **AND** lee el registro, *A*, justo después de que la instrucción **MOV** escribe el registro *A* con un dato traído desde la *Data Memory*
- el problema es que en el mismo ciclo 4 en que el dato está aún siendo leído desde la *Data Memory* (en la etapa *MEM* de **MOV**), la ALU tiene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



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 (hacia el final del ciclo 4)
- … la etapa *EX* de `AND`* necesita ese dato en el mismo ciclo 4 (al inicio del ciclo 4)

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

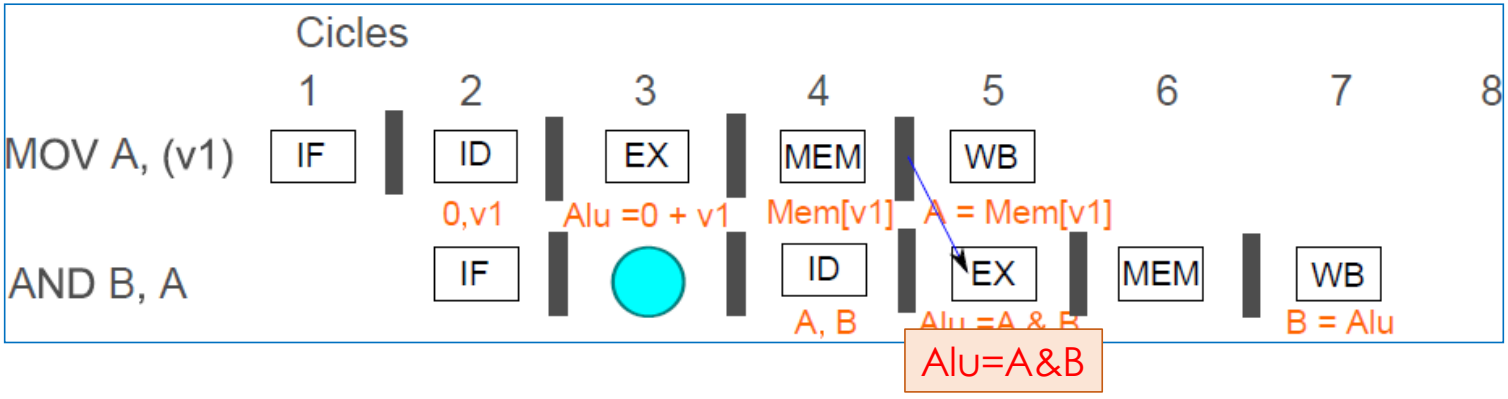


Es necesario detener (*stall*) el pipeline cuando una instrucción de tipo *load* ("carga" un registro con un dato traído desde la memoria) es seguida por una instrucción que usa el valor del registro recién cargado:

- agregamos una ***Data Hazard Unit*** en la etapa *ID*

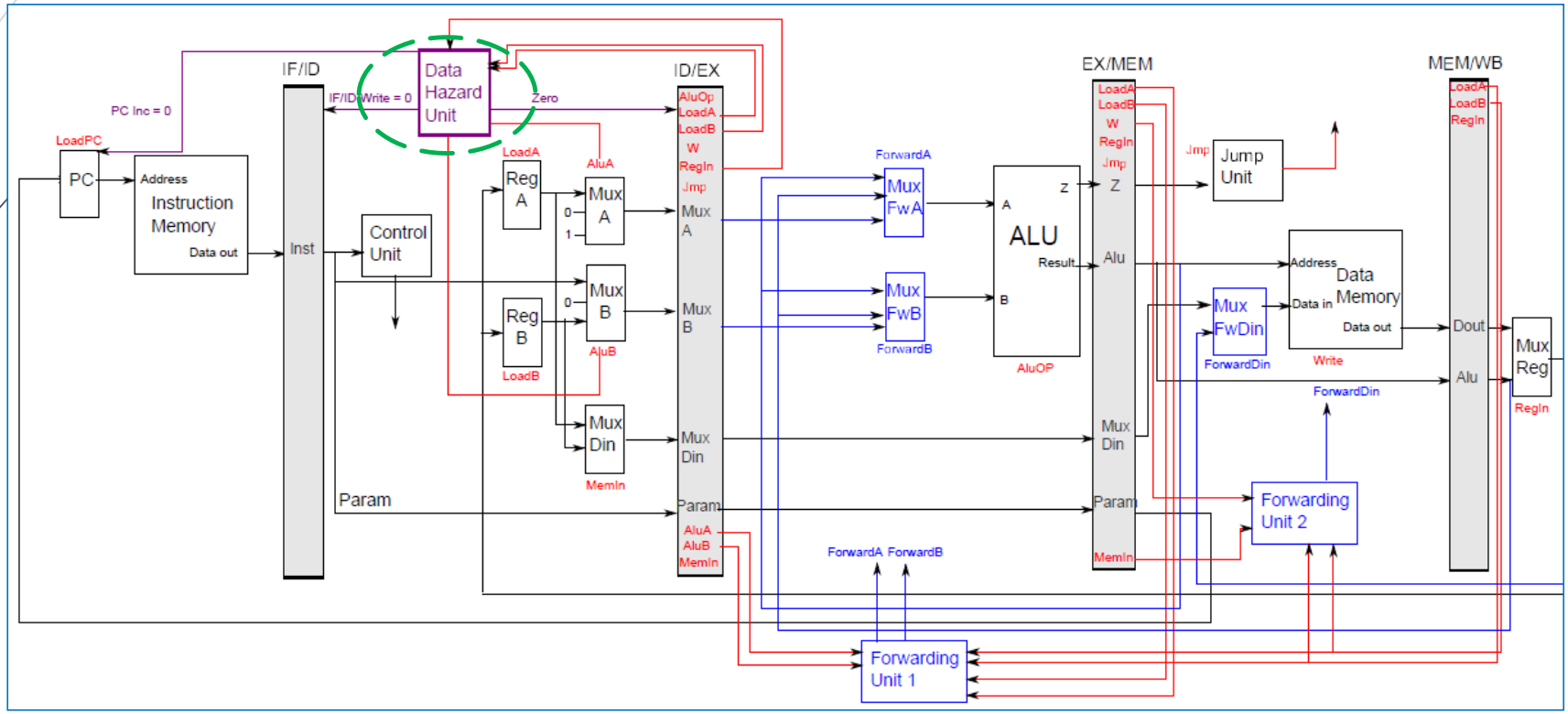
  … que produce un *stall* de un ciclo (el círculo verde en la fig.) en la ejecución de la segunda instrucción

- luego del *stall*, la lógica de *forwarding* maneja correctamente la dependencia, y la ejecución puede proseguir



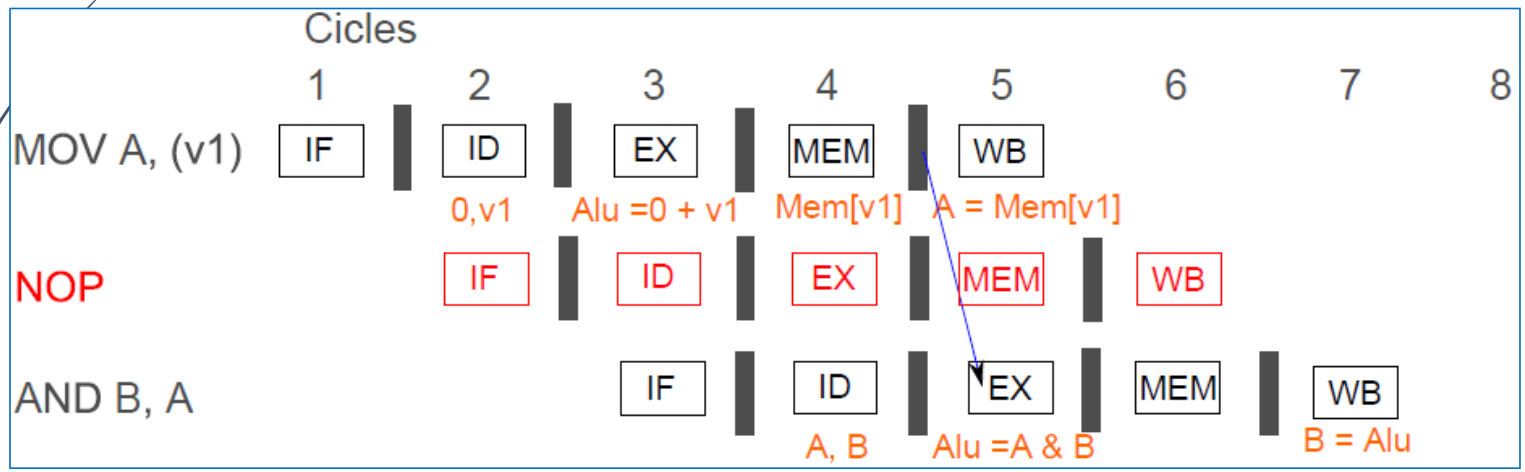
Al detener una instrucción en la etapa *ID*, se hace necesario detener también la próxima instrucción en la etapa *IF*:

la *Data Hazard Unit* debe evitar que los registros *PC* e *IF/ID* cambien



El caso anterior —una instrucción de tipo *load* seguida por una instrucción que lee el registro recién cargado— también es detectable por el compilador mientras está generando el código *assemly*:

→ una solución alternativa a *Data Hazard Unit + stall* es que el compilador inserte una instrucción **NOP** —que no hace nada— entre ambas instrucciones

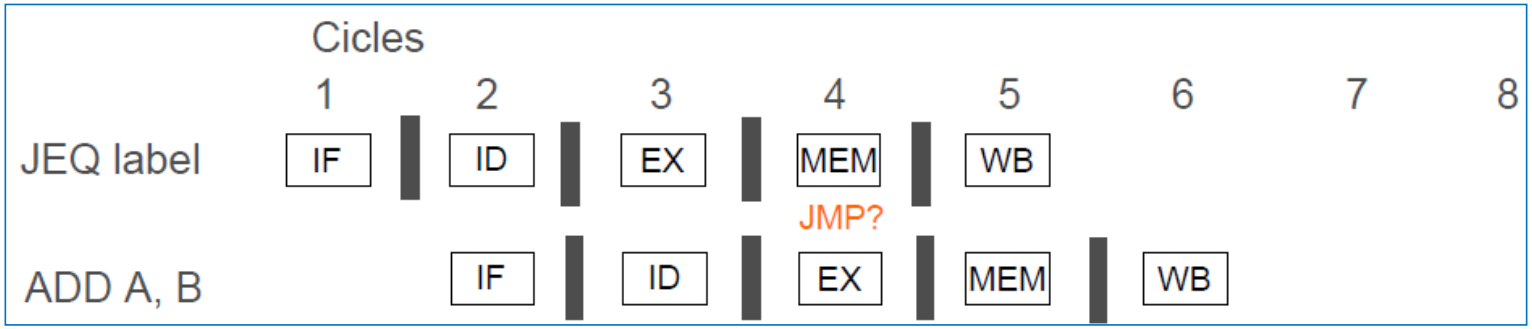


Ej. de ***hazard* de control** —`JEQ` *label* seguida por `ADD A,B`:

- al ejecutarse `JEQ` *label*, hay que esperar hasta la etapa *MEM*, en el ciclo 4 (recién ahí la ALU ha calculado el resultado de la comparación), para saber si el salto a la instrucción en la dirección *label* (de la Instruction Memory) 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, como si el salto no se fuera a tomar?

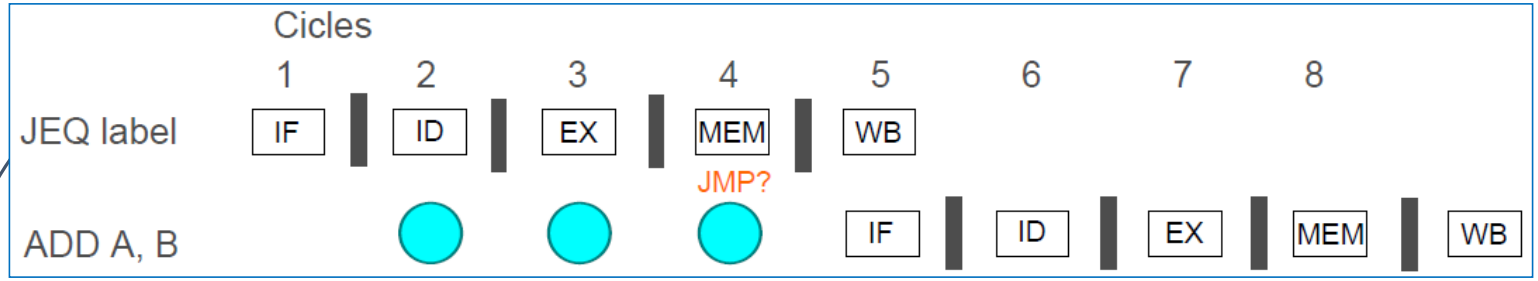
  … y si las ingresamos, ¿qué pasa si finalmente se toma el salto?



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

… solución muy costosa, en términos de desempeño, para la mayoría de los computadores

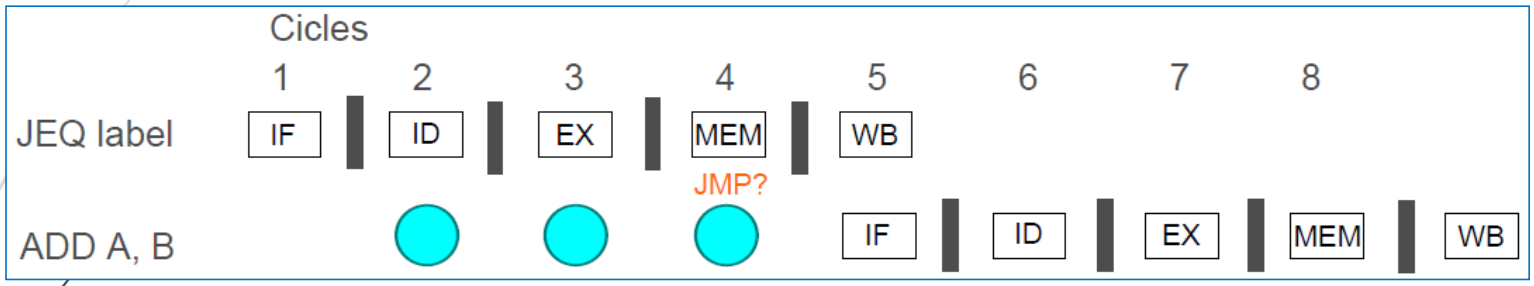
… tres ciclos de reloj en los que el computador no hace nada, y esto es cada vez que hay una instrucción *jump* en el programa (~ 20% en los procesadores Intel)



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

… solución muy costosa, en términos de desempeño, para la mayoría de los computadores

… tres ciclos de reloj en los que el computador no hace nada, y esto es cada vez que hay una instrucción *jump* en el programa (~ 20% en los procesadores Intel)



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

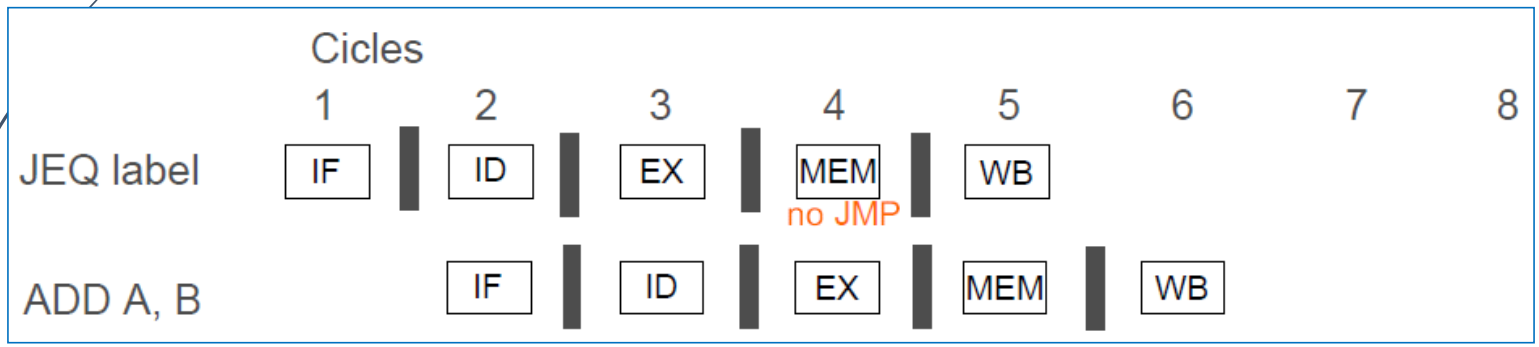
… felizmente, los hazards de control ocurren menos frecuentemente

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*

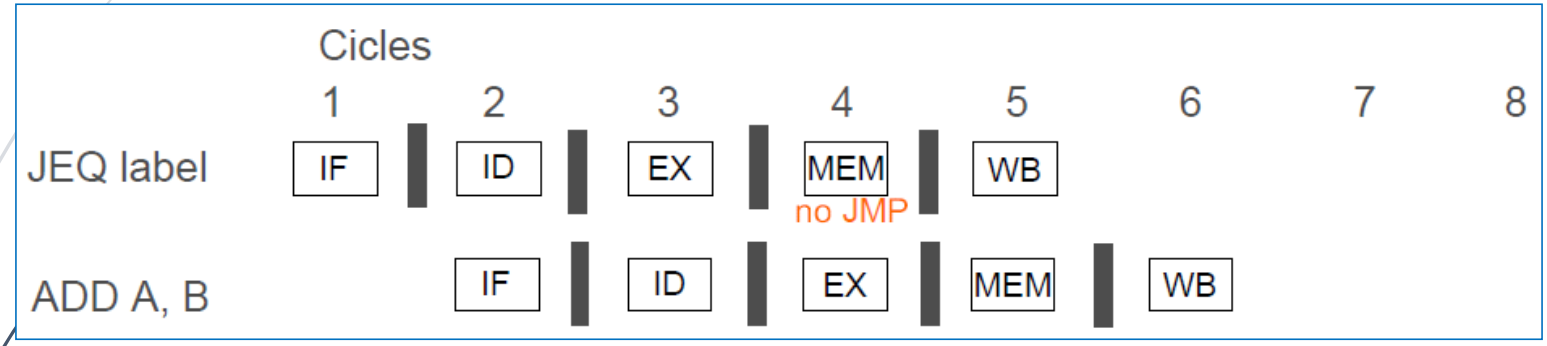
  … por supuesto, si el salto es tomado, entonces va a haber que hacer algo con estas instrucciones y continuar con la ejecución de la instrucción en la dirección *label*

- **(** alternativamente, se podría predecir que el salto sí va a ser tomado **)**



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 el salto es tomado, entonces va a haber que hacer algo con estas instrucciones y continuar con la ejecución de la instrucción en la dirección *label*
- **(** alternativamente, se podría predecir que el salto va a ser tomado **)**

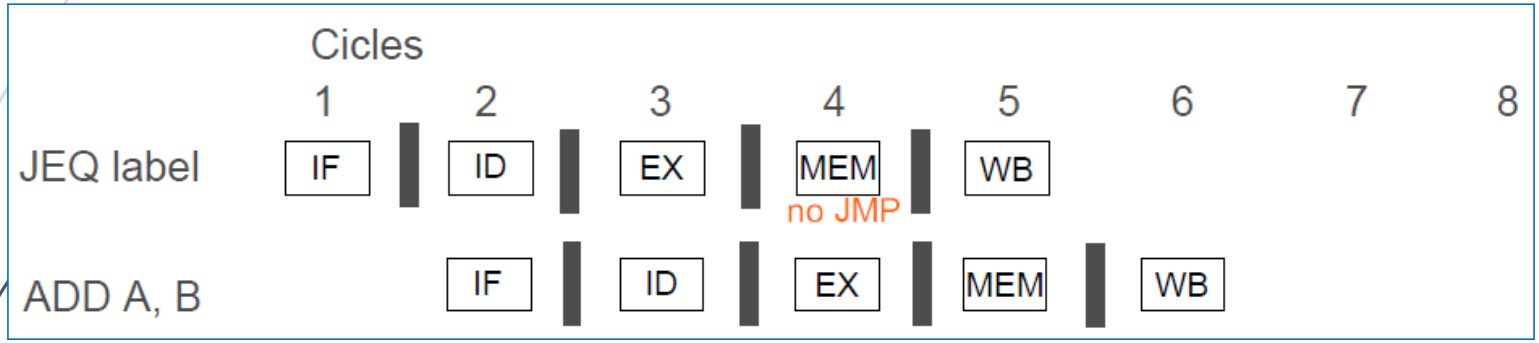


P.ej., predecir siempre que el salto no va a ser tomado es una forma de **predicción estática**:

relativamente barata en cuanto a hardware

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 el salto es tomado, entonces va a haber que hacer algo con estas instrucciones y continuar con la ejecución de la instrucción en la dirección *label*
- **(** alternativamente, se podría predecir que el salto va a ser tomado **)**



P.ej., predecir siempre que el salto no va a ser tomado es una forma de **predicción estática**:
relativamente barata en cuanto a hardware

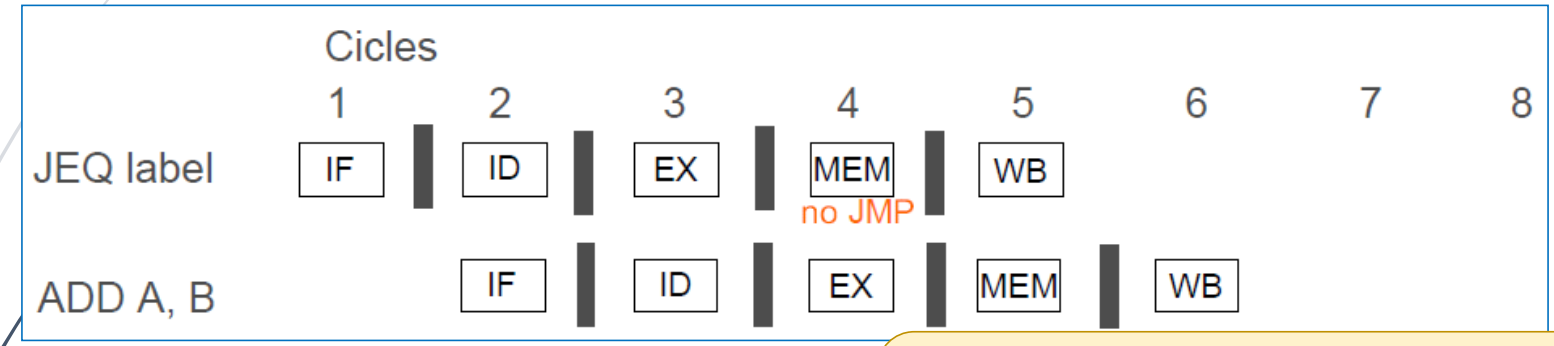
Otra forma de predicción estática:

los saltos hacia atrás se toman

los saltos hacia adelante no se toman

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 el salto es tomado, entonces va a haber que hacer algo con estas instrucciones y continuar con la ejecución de la instrucción en la dirección *label*
- **(** alternativamente, se podría predecir que el salto va a ser tomado **)**



Tasa de éxito de predicción estática:

80% – 90% para operaciones de números enteros

85% – 95% para operaciones de punto flotante

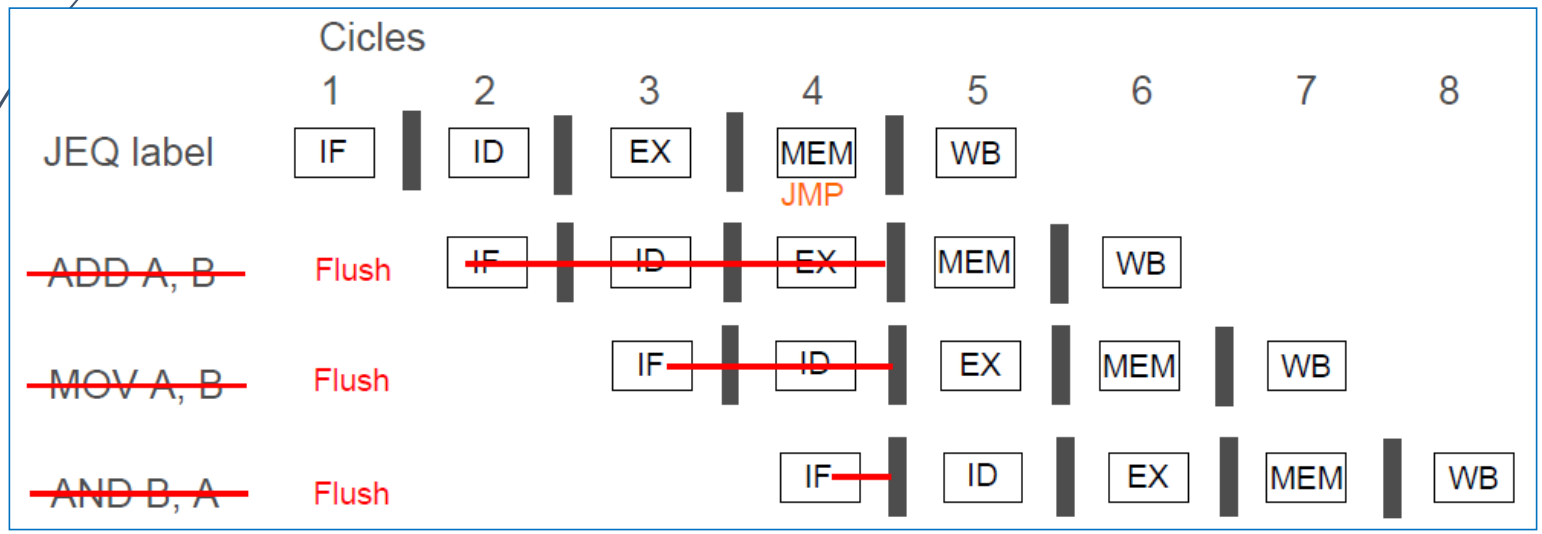
P.ej., predecir siempre que el salto no va a ser tomado es una forma de **predicción estática**:
relativamente barata en cuanto a hardware

Otra forma de predicción estática:
los saltos hacia atrás se toman
los saltos hacia adelante no se toman

Si la predicción resultó errada ⟹ hay que descartar del pipeline las instrucciones que ingresaron después de `JEQ` *label* y hasta que detectamos el error:

- el error lo detectamos en la etapa *MEM* de la ejecución de `JEQ` *label* —ahí es cuando tenemos el resultado de la comparación (por eso, ahí está la *Jump Unit*)

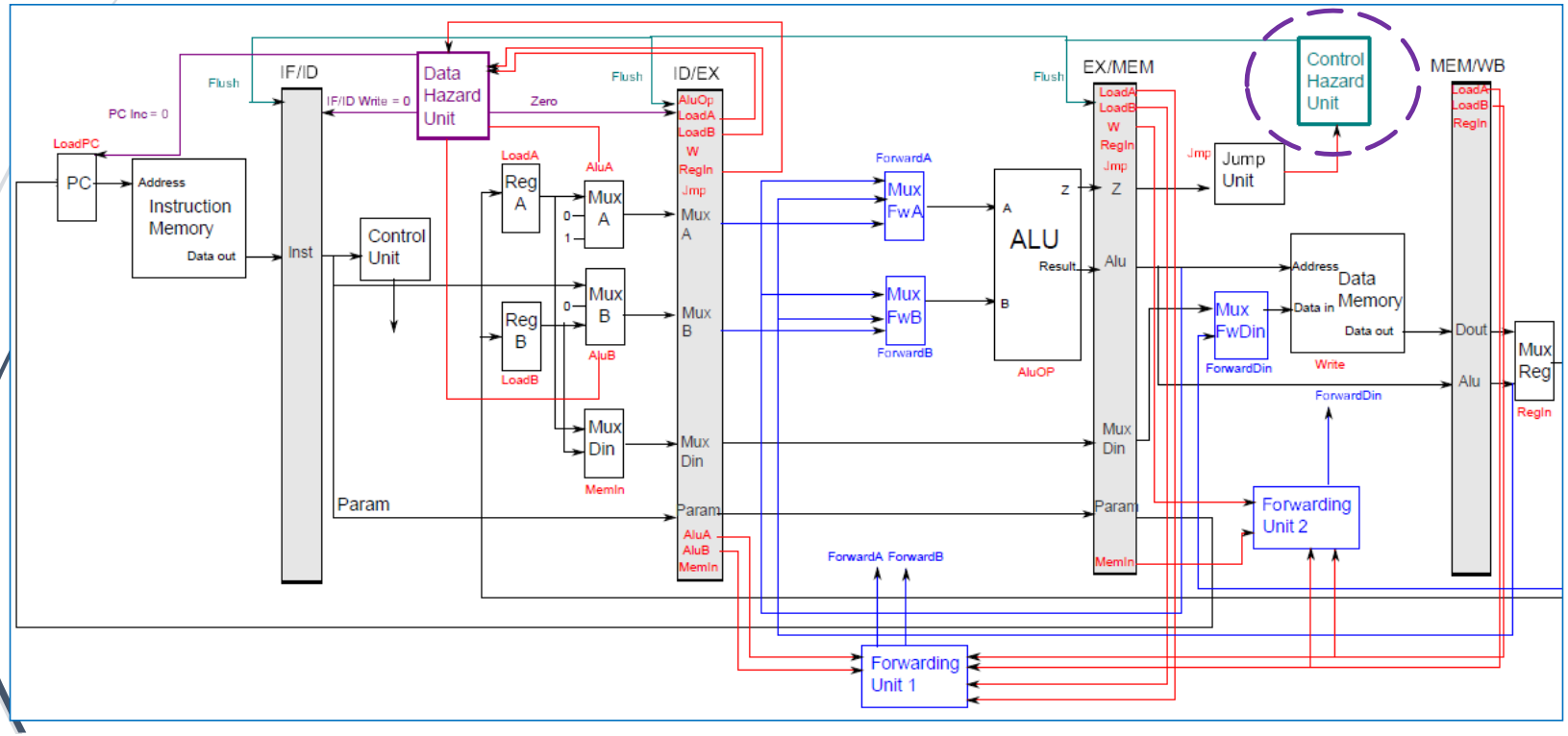
  ⟹ hay que descartar las instrucciones que en ese momento están en sus etapas *EX*, *ID* e *IF*



Agregamos una **Control Hazard Unit** a la salida de la *Jump Unit*:

si esta *Unit* detecta que la predicción ha sido errada ⟹ envía una señal *flush* a los registros *IF/ID*, *ID/EX* y *EX/MEM*

… que pone en 0 los bits que identifican a las instrucciones que están en ejecución en esas etapas



Los procesadores más recientes emplean **predicción dinámica de saltos**:

a partir de información producida durante la ejecución del propio programa

Versión simple

Mantienen una tabla* con las direcciones de las instrucciones a las cuales se salta (los *labels*)

… y para cada una hay **un bit que dice si la última vez el salto a esa instrucción fue tomado o no**

… así, la próxima vez que se ejecute un *jump* condicional a esa instrucción la predicción es igual a lo que ocurrió esa última vez

… por supuesto, si esta predicción resulta errónea, entonces se cambia el bit

* Implementar esta tabla es un desafío grande para el hardware; en la práctica, es una *caché*

Los procesadores recientes emplean **predicción dinámica de saltos**:

a partir de información producida durante la ejecución del propio programa

Versión simple

Mantienen una tabla* con las direcciones de las instrucciones a las cuales se salta (los *labels*)

… y para cada una hay **un bit que dice si la última vez el salto a esa instrucción fue tomado o no**

… así, la próxima vez que se ejecute un *jump* condicional a esa instrucción la predicción es igual a lo que ocurrió esa última vez

… por supuesto, si esta predicción resulta errónea, entonces se cambia el bit

* Implementar esta tabla es un desafío grande para el hardware; en la práctica, es una *caché*

Versión más sofisticada

Se usan dos bits (próx. diap.)

Ahora **la predicción tiene que resultar errónea dos veces seguidas para que se cambie ambos bits** y la próxima vez la predicción sea la opuesta:

la primera vez que la predicción resulta errónea, se cambia un bit, pero no la predicción

y si la predicción es correcta la próxima vez, entonces el bit se vuelve a su valor anterior

Tasa de éxito de predicción dinámica:

85% – 95% para operaciones de números enteros

90% – 99% para operaciones de punto flotante

J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach" (6th ed.) Morgan Kaufman 2019

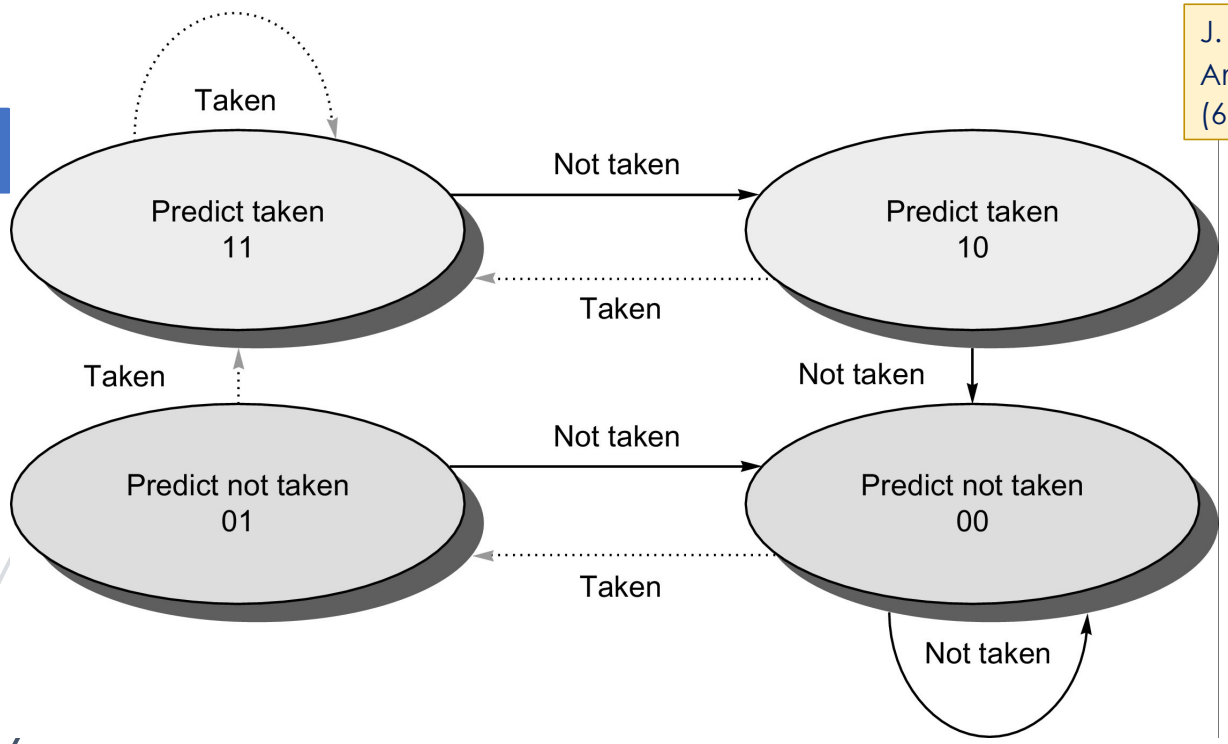


Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system.

The 2-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of *n*-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general *n*-bit predictors.