Arquitectura de Computadoras para Ingeniería

(Cód. 7526)
1° Cuatrimestre 2016

Dra. Dana K. Urribarri
DCIC - UNS

Pipelining

Clasificación de los pipeline

De acuerdo a su nivel de procesamiento

- Pipeline aritmético:
  - Las operaciones aritméticas se segmentan para permitir el procesamiento en pipeline.
- Pipeline de instrucciones:
  - Las instrucciones se procesan en pipeline, permitiendo el solapamiento.
- Pipeline de procesadores:
  - Procesadores en cascada procesan el mismo flujo de datos. Cada uno realiza una tarea específica.

Clasificación de los pipeline

De acuerdo a su configuración y estrategia de control

- Unifuncional vs. Multifuncional:
  - Un pipeline *unifuncional* realiza una única tarea fija.
  - Un pipeline *multifuncional* puede realizar varias tareas diferentes, en simultáneo o no.
- Estático vs. Dinámico:
  - Un pipeline *estático* asume una única configuración por vez. *Un pipeline unifuncional es estático.*
  - Un pipeline *dinámico* permite varias configuraciones en simultáneo. *Un pipeline dinámico es multifuncional.*
- Escalar vs. Vectorial:
  - Un pipeline *escalar* procesa operandos escalares (enteros, punto flotante o valor lógico).
  - Un pipeline *vectorial* procesa instrucciones vectoriales sobre operandos vectoriales.

Pipeline de instrucciones

Diseño de la microarquitectura

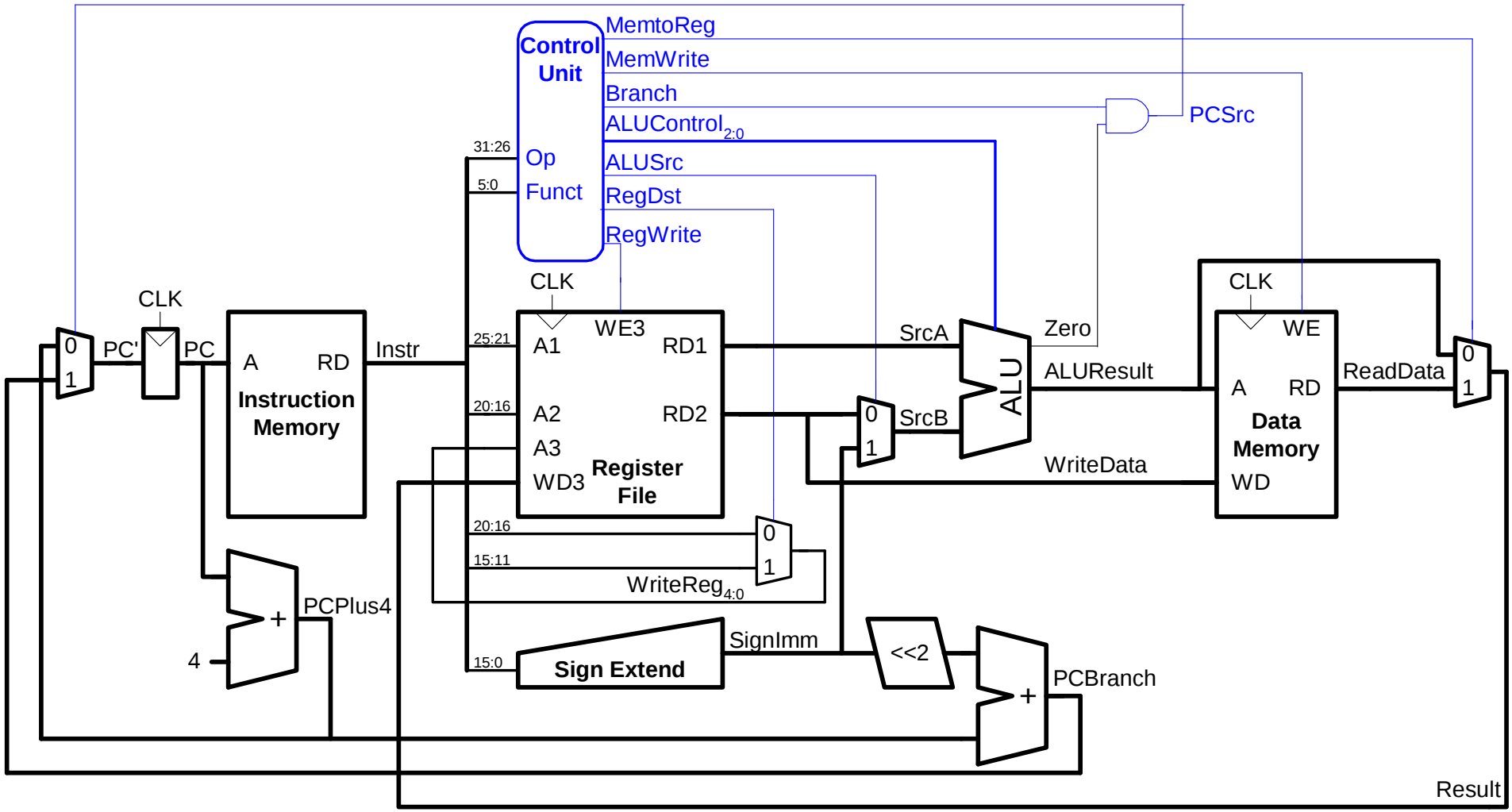
- Tres posibles microarquitecturas para MIPS:
  - **Diseño único ciclo:**

    Ejecuta la instrucción completa en un único ciclo.
  - **Diseño multi-ciclo:**

    Ejecuta las instrucciones en una serie de ciclos cortos.
  - **Diseño en pipeline:**

    Aplica pipeline a la microarquitectura de un único ciclo.

Microarquitectura único ciclo



Microarquitectura en pipeline

- Se divide la microarquitectura único ciclo en 5 etapas.
- Pueden haber 5 instrucciones ejecutando simultáneamente.
- Idealmente, la lógica de cada etapa es 1/5 de la lógica total.
  - La frecuencia del reloj puede ser 5 veces mayor.
- Si bien la latencia de cada instrucción es la (≈) misma, el *throughput* es 5 veces mayor.

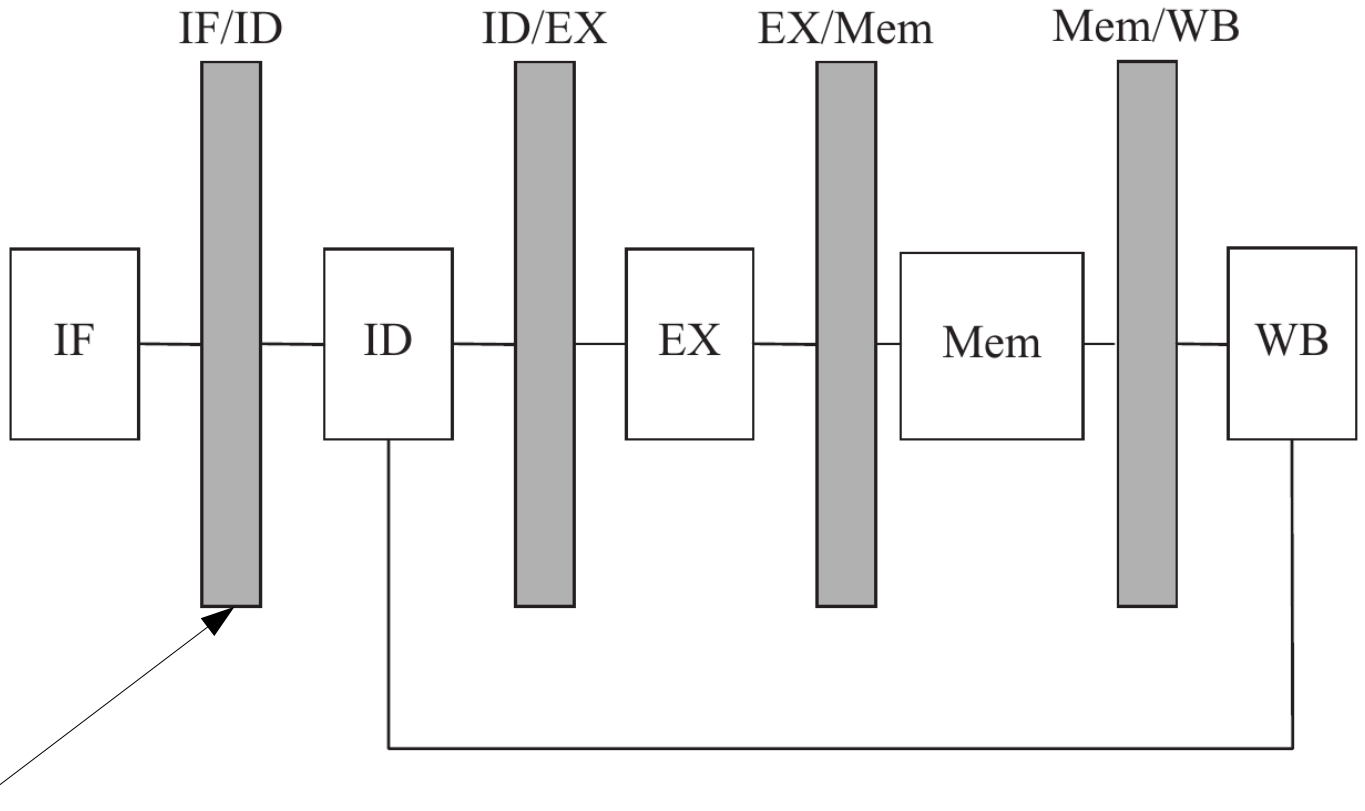
Microarquitectura en pipeline

- Las tareas con más retardos son:
  - Lecturas y escrituras en memoria
  - Operaciones en la ALU
- Las 5 etapas del pipeline se diseñan de forma que cada una involucre sólo una de esas tareas.
  1) **Instruction fetch** → Lectura en memoria
  2) **Instruction decode** → Lectura de registros
  3) **Execution** → ALU
  4) **Memory access** → Lectura/escritura en memoria
  5) **Writeback** → Escritura de registros

Microarquitectura en pipeline

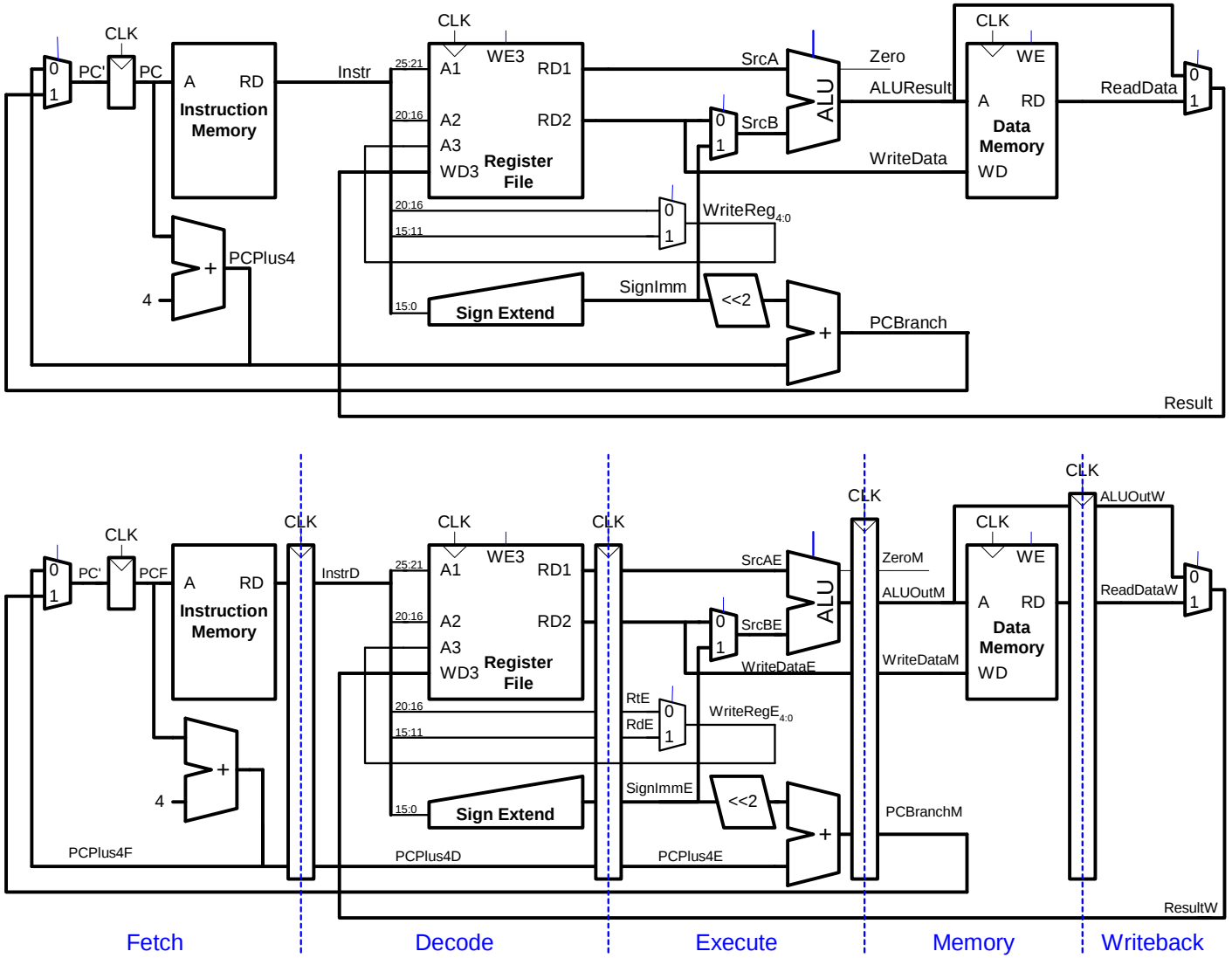
1) **Instruction fetch (IF).** Se trae de memoria la instrucción en la posición indicada por el PC. Se asume que la instrucción no es un *branch* y se incrementa el PC.
2) **Instruction decode (ID)**. Se decodifica la instrucción y se reconoce su tipo. Se buscan los operandos.
3) **Execution (EX)**. Se ejecuta la instrucción. Difiere si es una operación aritmética, un acceso a memoria o un *branch*.
4) **Memory access (Mem)**. Se accede efectivamente a la memoria en caso de que la instrucción sea un acceso a memoria.
5) **Writeback (WB)**. Si la instrucción no es un *branch* o un *store*, el resultado de la operación (o del *load*) se almacena en el registro resultado.

Abstracción del pipeline



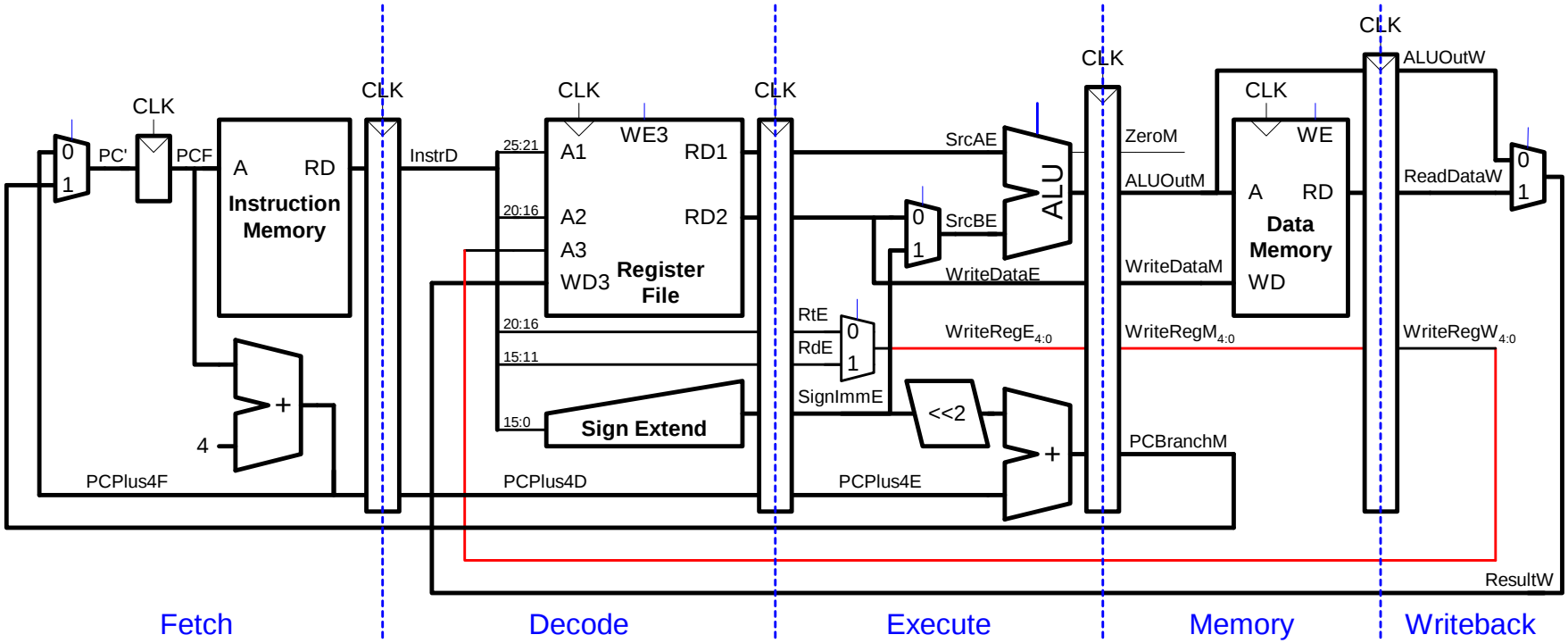
Registros interetapas.
Almacenan la información necesaria para comenzar con la siguiente etapa.

Datapath único ciclo & en pipeline

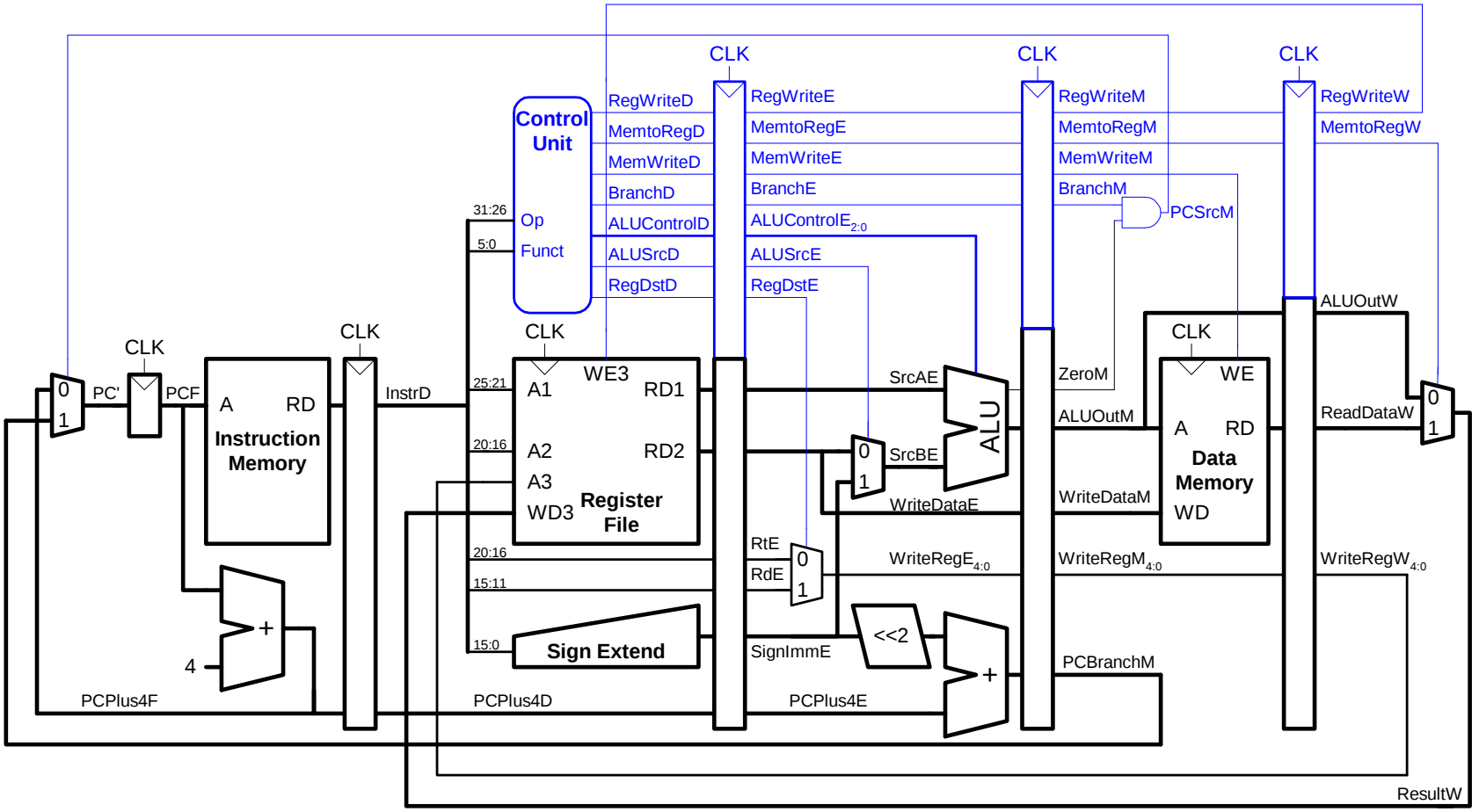


Datapath en pipeline corregido

La señal de escritura en registros debe llegar junto con el dato.

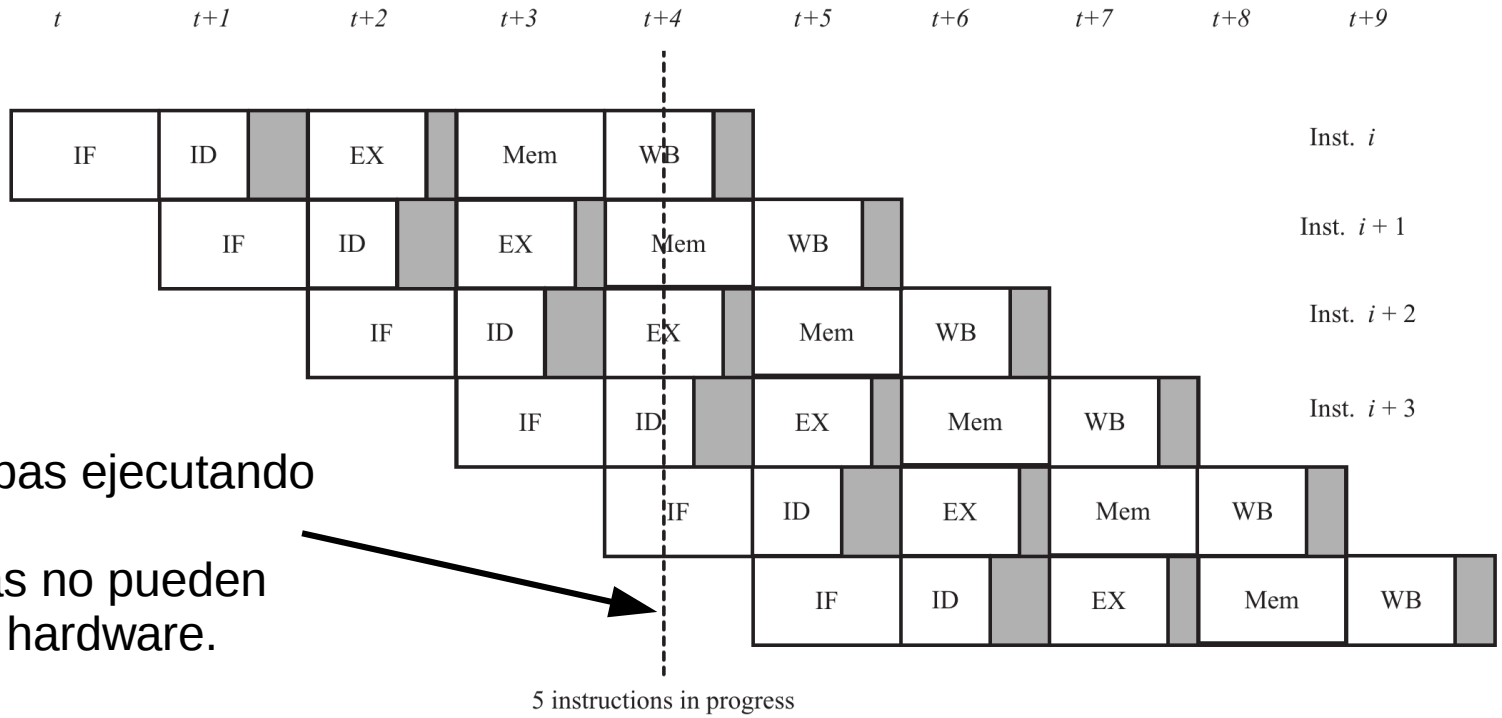


Datapath & control del pipeline



Superposición de etapas

- Si asumimos que acceder a memoria de datos e instrucciones es lo que más demora.
  - Las etapas no demoran exactamente lo mismo, aumenta la latencia de cada instrucción.

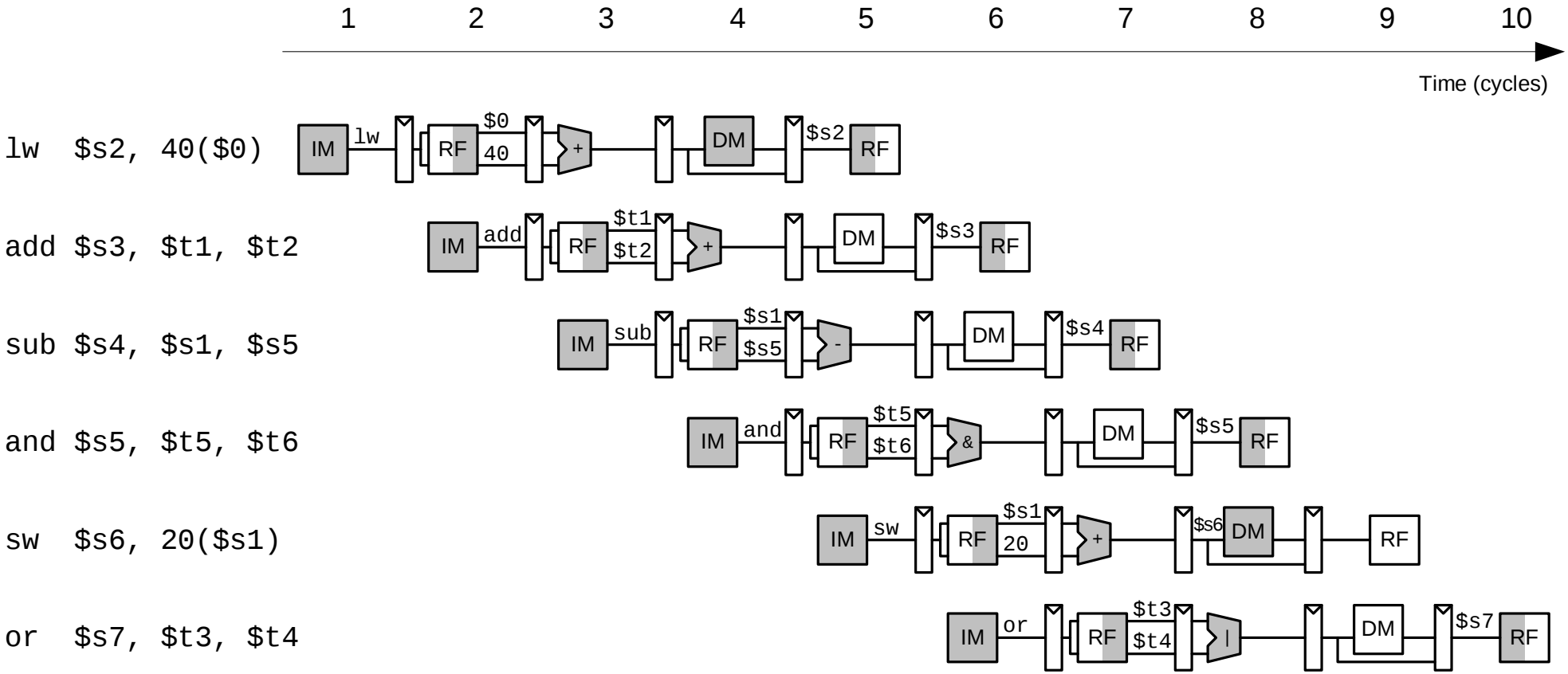


Las 5 etapas ejecutando a la vez.
Las etapas no pueden compartir hardware.

Requerimientos de hardware

- Para superponer Fetch y Memory
  - Memorias (cache) separadas para instrucciones y datos.
- Para superponer Fetch y Execute
  - Dos ALUs. Una para incrementar el PC y otra para ejecutar la instrucción.
- Para superponer WriteBack y Decode
  - Los registros se escriben en la primera mitad del ciclo y se leen en la segunda.

Uso de las etapas del pipeline



- Dependiendo de la instrucción, algunas etapas pueden estar ociosas.
- Todas involucran Fetch → Decode → Execute

Conflictos

- En condiciones ideales, el pipeline termina una instrucción por ciclo.
- Las condiciones no siempre son ideales:
  - conflictos (*hazards*)
- Los conflictos se detectan en la etapa de decode.

Tipos de conflictos

- Estructurales:
  - La instrucción necesita de un recurso que está ocupado.
- De datos:
  - La ejecución de una instrucción depende de la ejecución de una instrucción previa y ambas instrucciones están concurrentemente ejecutándose en el pipeline.
- De control:
  - Dado que el PC se incrementa en cada ciclo, el pipeline funciona bien cuando no hay cambios en el flujo secuencial de las instrucciones (branch, jump, call, return)

Conflictos estructurales

- La instrucción necesita de un recurso que está ocupado.
- Soluciones:
  - Duplicar el recurso
  - Implementar la unidad en pipeline
  - Cola de esperas a la unidad funcional (D-virtuales)

Conflictos de datos

- La ejecución concurrente debe coincidir con la ejecución secuencial. (*i* < *j*)
- Read after Write (RAW)

  Si la instrucción *j* lee su dato fuente antes de que la instrucción *i* lo escriba, *j* operaría con un valor incorrecto.

- Write after Read (WAR)

  La instrucción *j* escribe el dato antes de que la instrucción *i* lo lea.

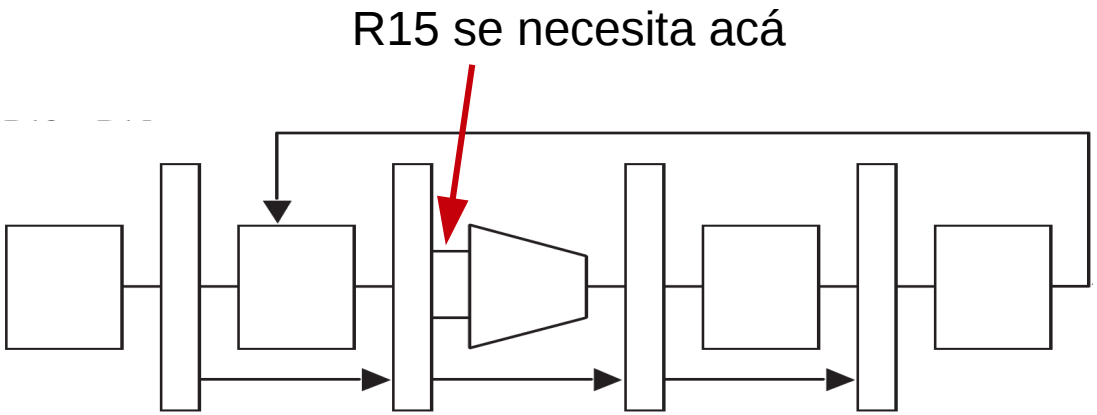
- Write after Write (WAW)

  Dos instrucciones tienen el mismo registro de salida

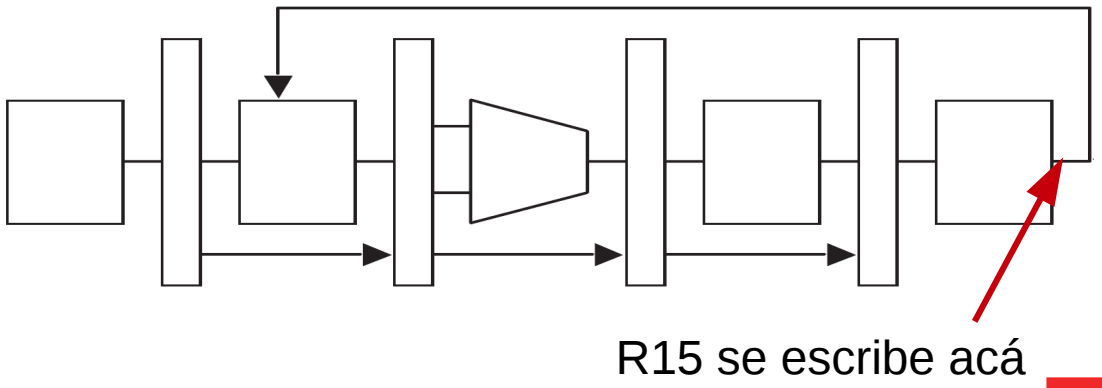
Write after Read (antidependiencia)

- i : R7 ← R12 + R15
- i+1 : R15 ← R7 – R12

i : R7 ← R12 + R15



i+1 : R15 ← R7 – R12

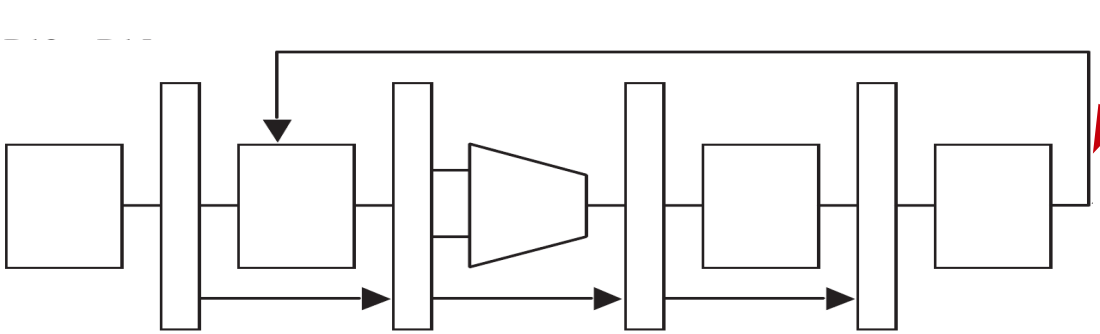


Write after Write

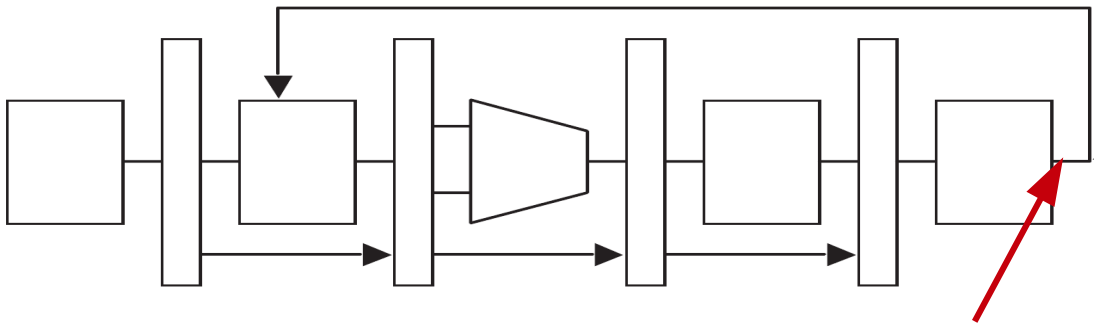
- i : R7 ← R12 + R15
- i+1 : R7 ← R4 – R12

R7 se escribe acá

i : R7 ← R12 + R15



i+1 : R7 ← R4 – R12

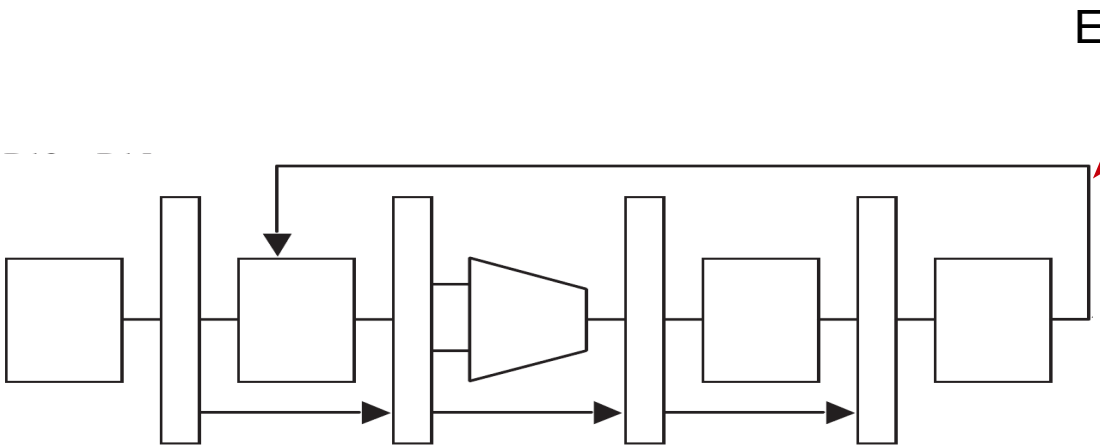


R7 se vuelve a escribir acá

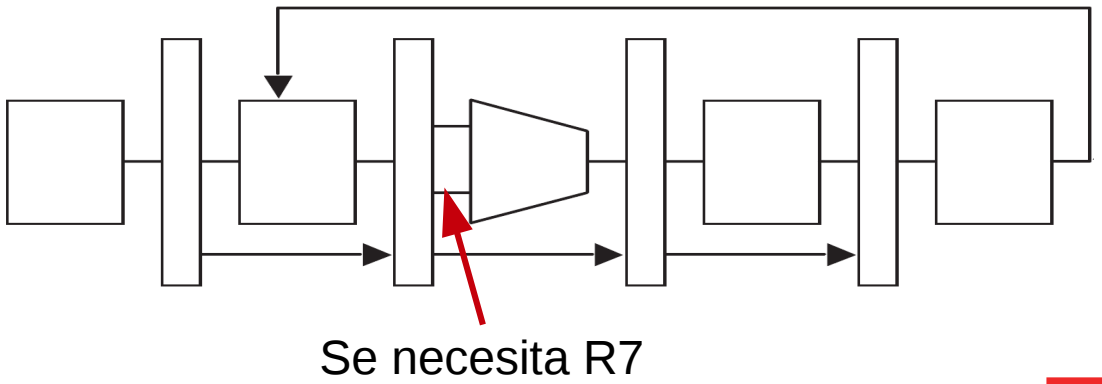
Read after Write

- i : R7 ← R12 + R15
- i+1 : R15 ← R7 – R12

i : R7 ← R12 + R15



i+1 : R15 ← R7 – R12



Conflictos en el pipeline

- En la implementación simple del pipeline la ejecución es en orden y todas las etapas demoran lo mismo. Por lo tanto evita:
  - Conflictos estructurales
  - Conflictos de datos
    - WAR
    - WAW
- Solamente tenemos conflictos:
  - RAW → *Dependencia de procedimiento*
  - De control

Gestión de los conflictos de datos

En tiempo de compilación:

- Insertar NOPs en el código
- Reordenar el código

En tiempo de ejecución

- Adelantar los datos (forwarding)
- Frenar el pipeline (stall)

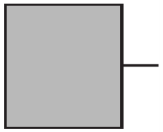
Read after Write

- Ejemplo 1:
  - i : R7 ← R12 + R15
  - i+1 : R8 ← R7 – R12
  - i+2 : R15 ← R8 + R7

Read after Write

i : R7 ← R12 + R15

be R7



Se necesita R7

i+1 : R8 ← R7 – R12

Se necesitan R7 y R8

i+2 : R15 ← R8 + R7

Read after Write

i : R7 ← R12 + R15

i+1: NOP

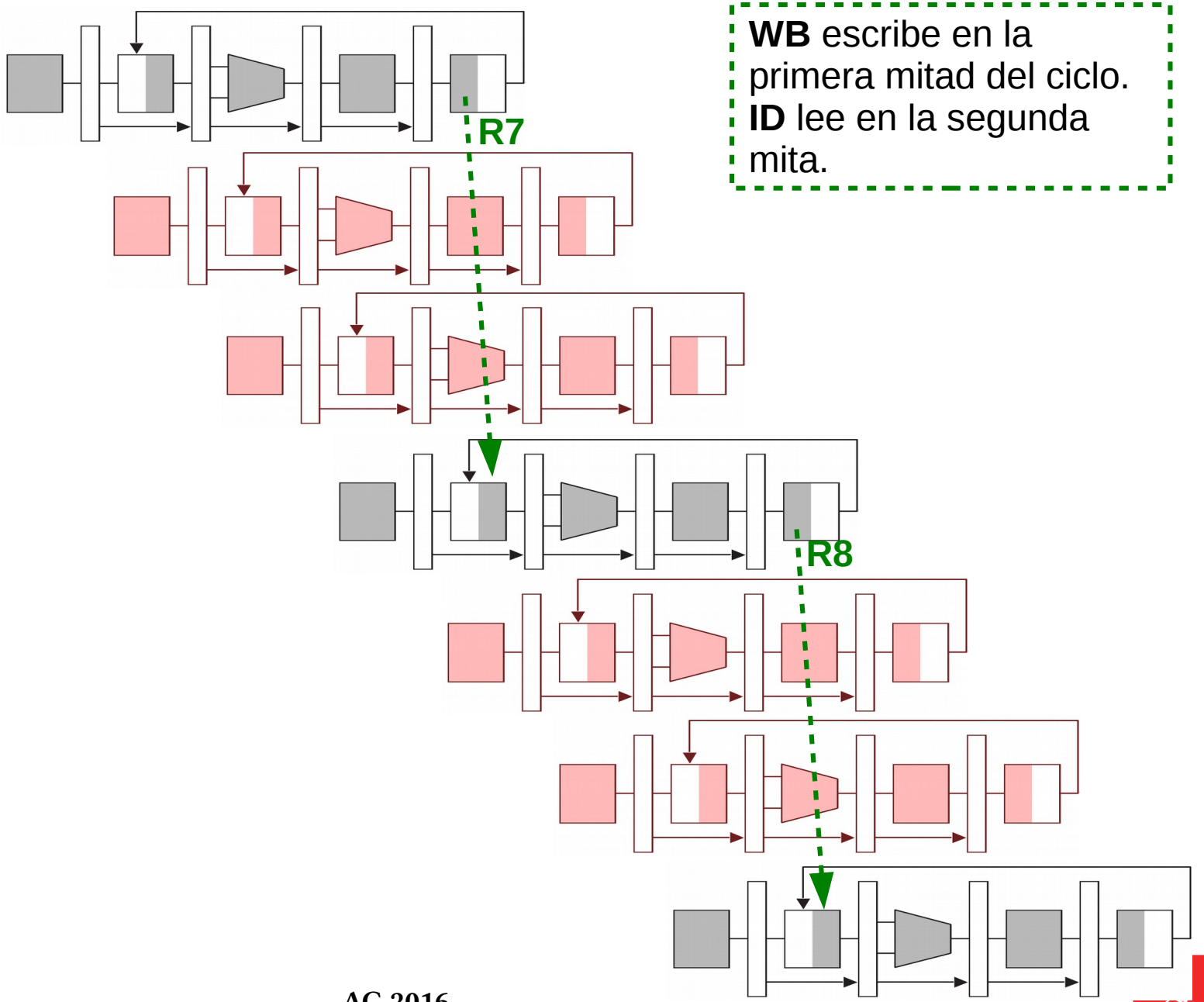
i+2: NOP

i+3 : R8 ← R7 – R12

i+4: NOP

i+5: NOP

i+6 : R15 ← R8 + R7

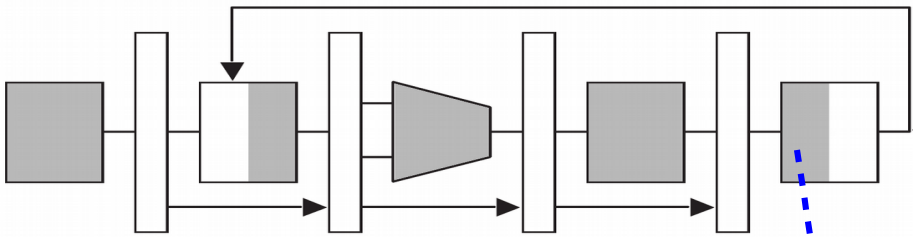


WB escribe en la primera mitad del ciclo. **ID** lee en la segunda mita.

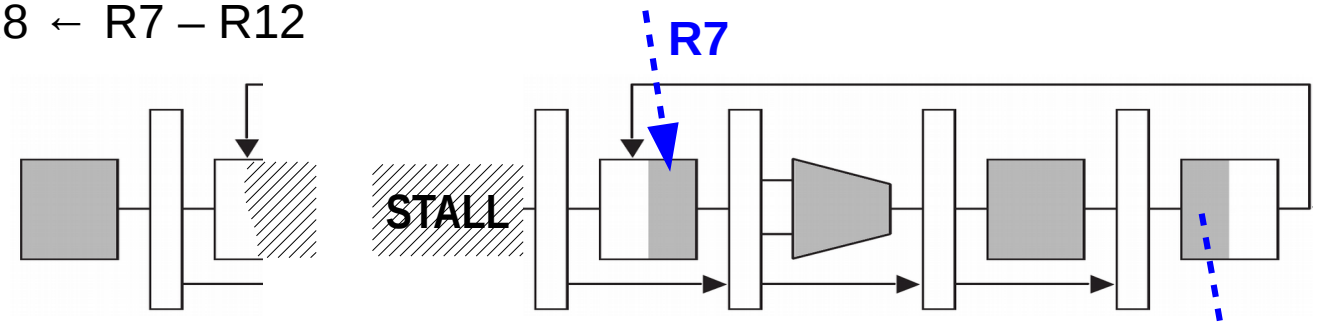
Read after Write

WB escribe en la primera mitad del ciclo.
ID lee en la segunda mita.
Los Stall se detectan en el decode.

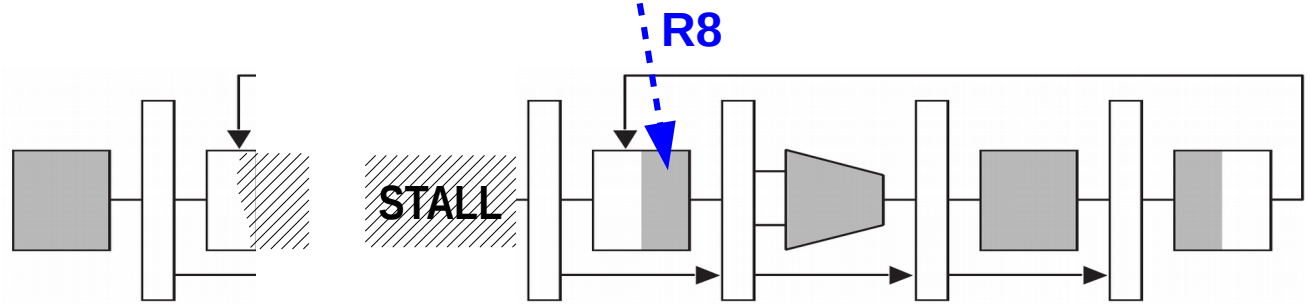
i : R7 ← R12 + R15



i+1 : R8 ← R7 – R12



i+2 : R15 ← R8 + R7

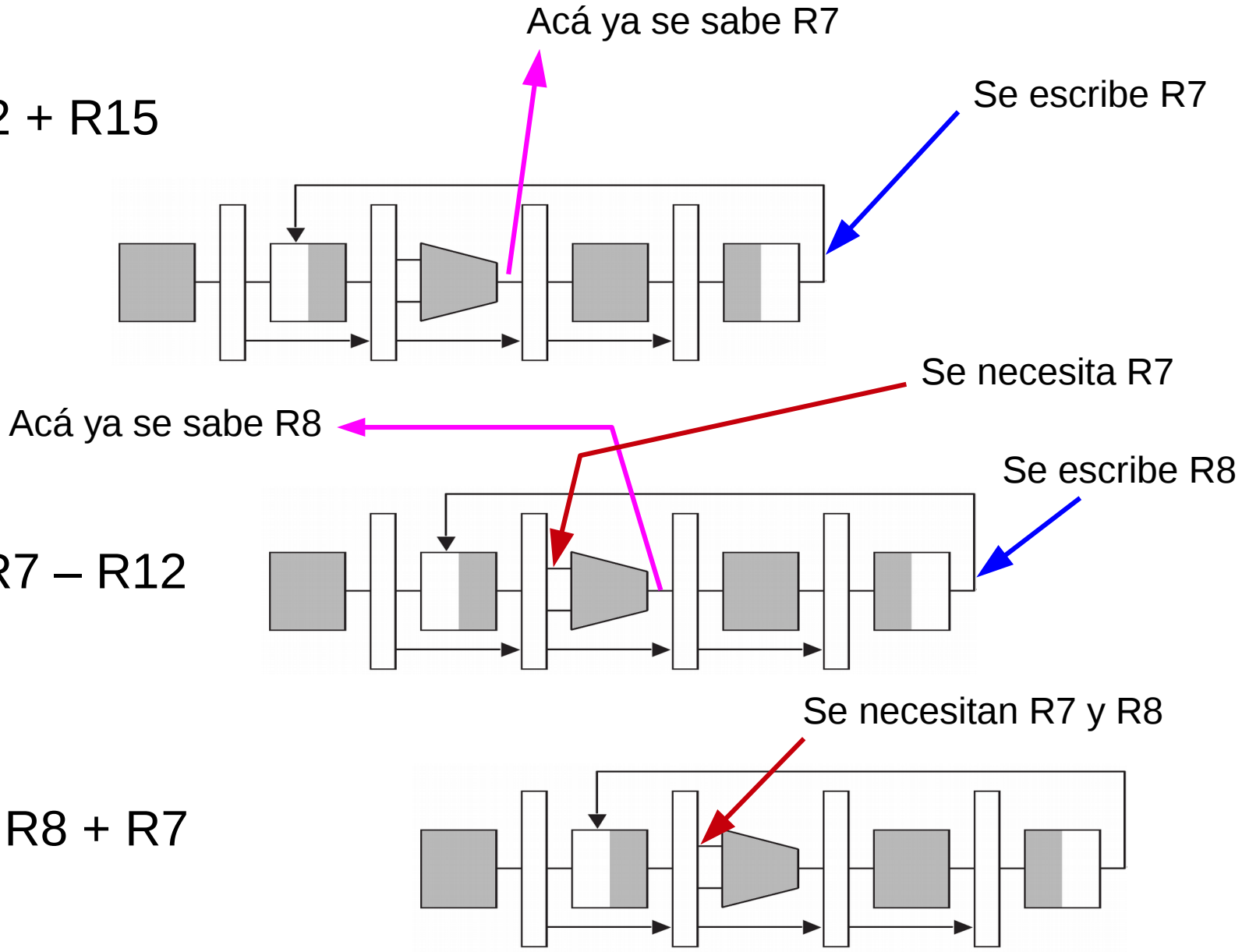


Read after Write

i : R7 ← R12 + R15

i+1 : R8 ← R7 – R12

i+2 : R15 ← R8 + R7

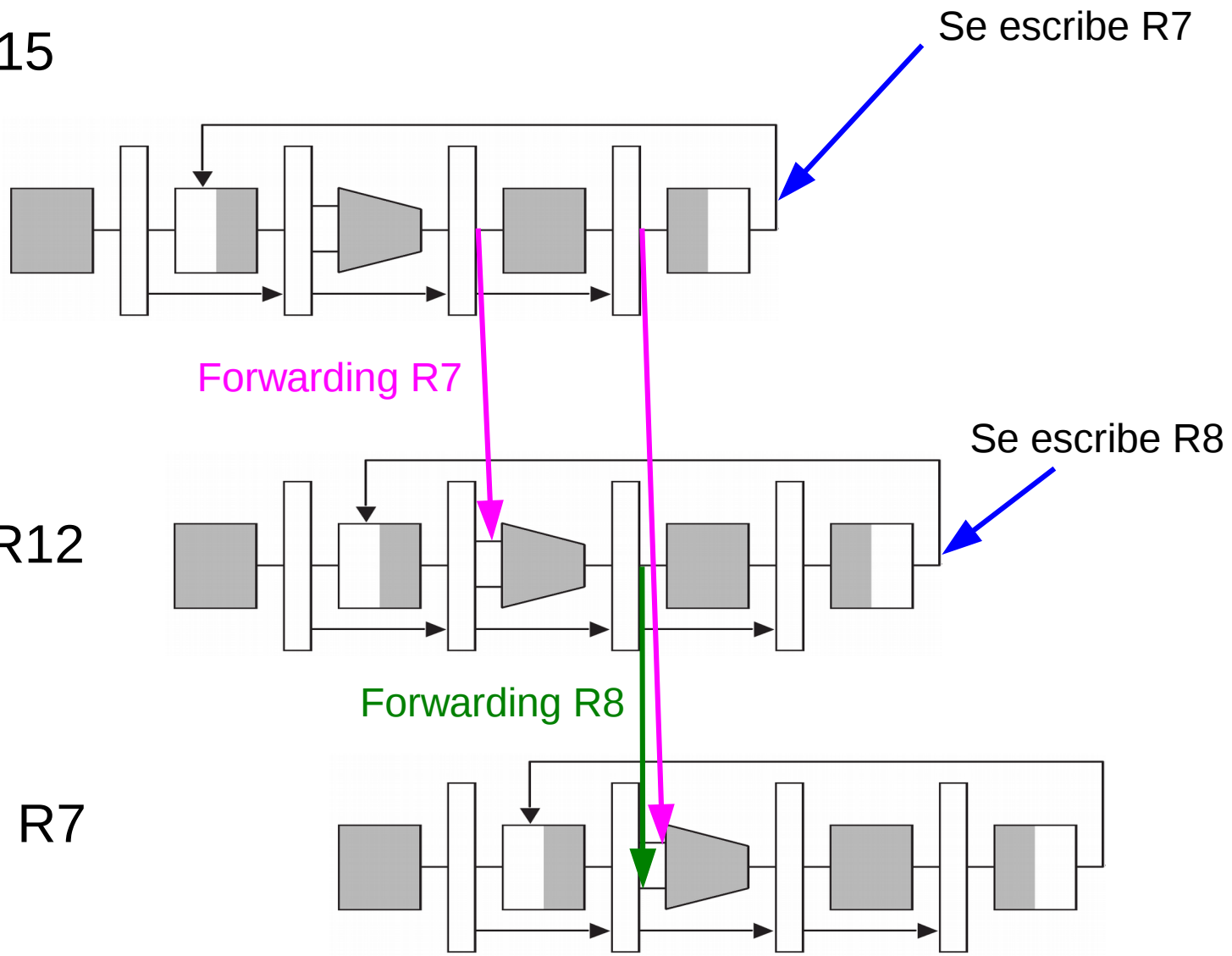


Read after Write

i : R7 ← R12 + R15

i+1 : R8 ← R7 – R12

i+2 : R15 ← R8 + R7



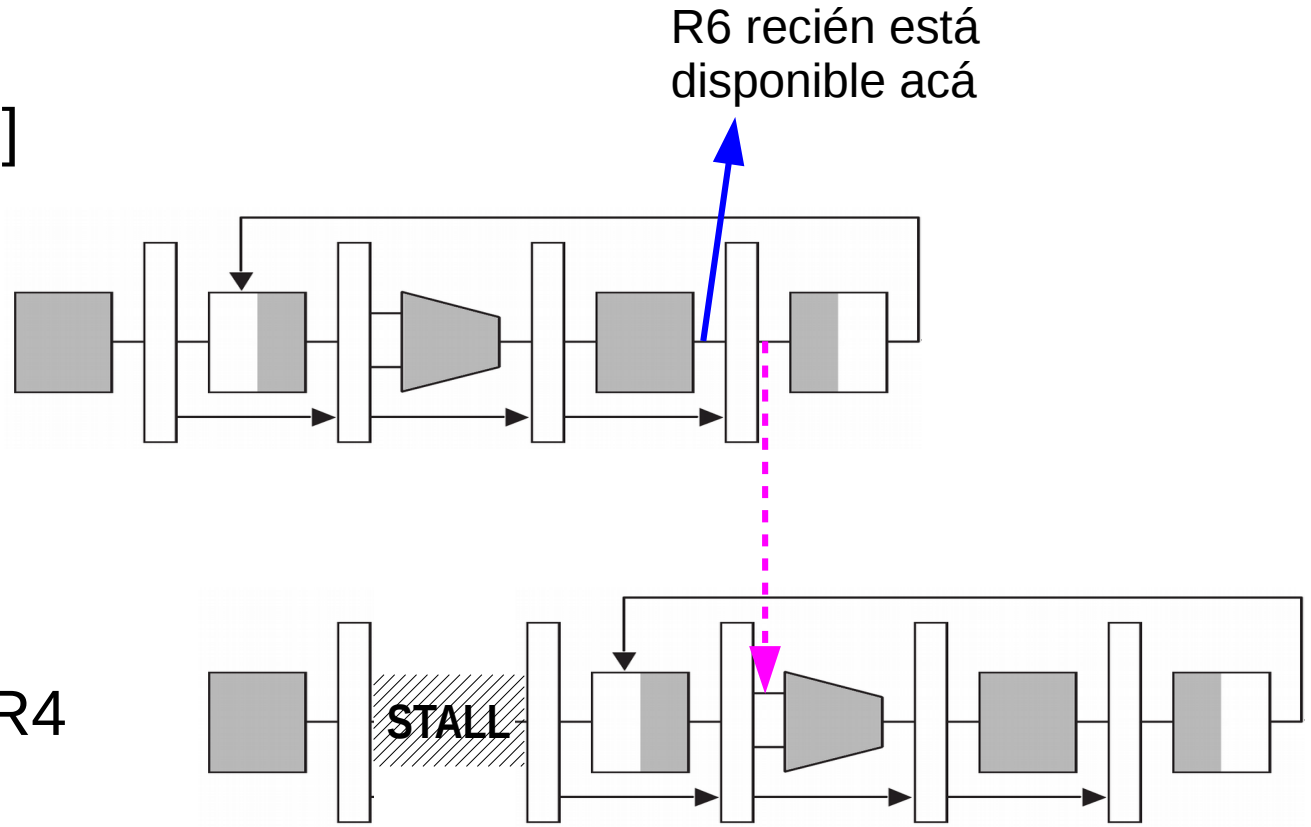
Read after Write

- Ejemplo 2:
  - i: R6 ← Mem[R2]
  - i+1: R7 ← R6 + R4

Read after Write

i : R6 ← Mem[R2]

i+1 : R7 ← R6 – R4



Hasta después de **MEM** no se conoce R6.
No hay otra opción que retardar 1 ciclo la instrucción.
Hay *forwarding* pero 1 ciclo más tarde

Read after Write

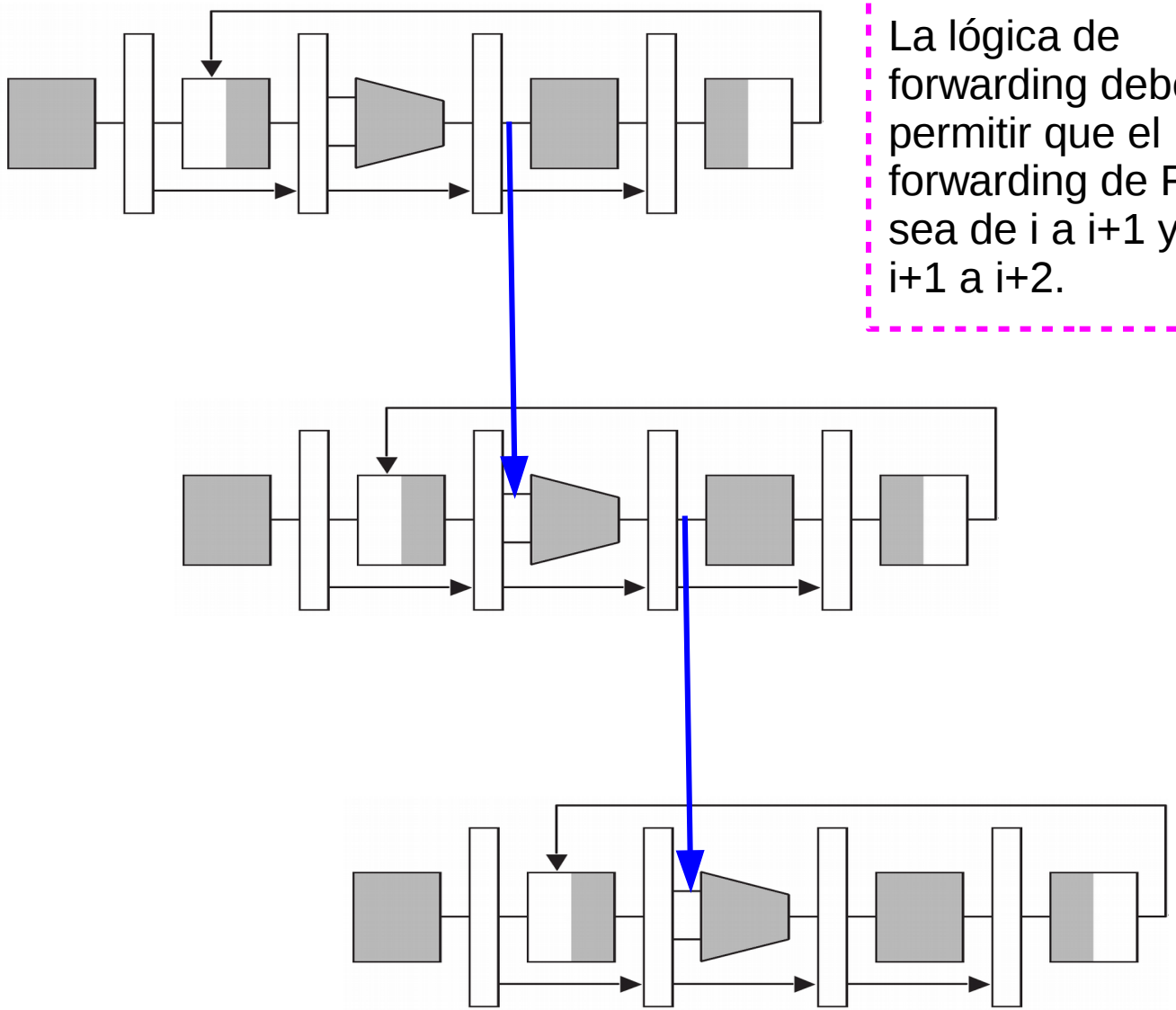
- Ejemplo 3:
  - i: R10 ← R4 + R5
  - i+1: R10 ← R10 + R4
  - i+2: R8 ← R10 + R7

Read after Write

i: R10 ← R4 + R5

i+1: R10 ← R10 + R4

i+2: R8 ← R10 + R7



La lógica de forwarding debe permitir que el forwarding de R10 sea de i a i+1 y de i+1 a i+2.

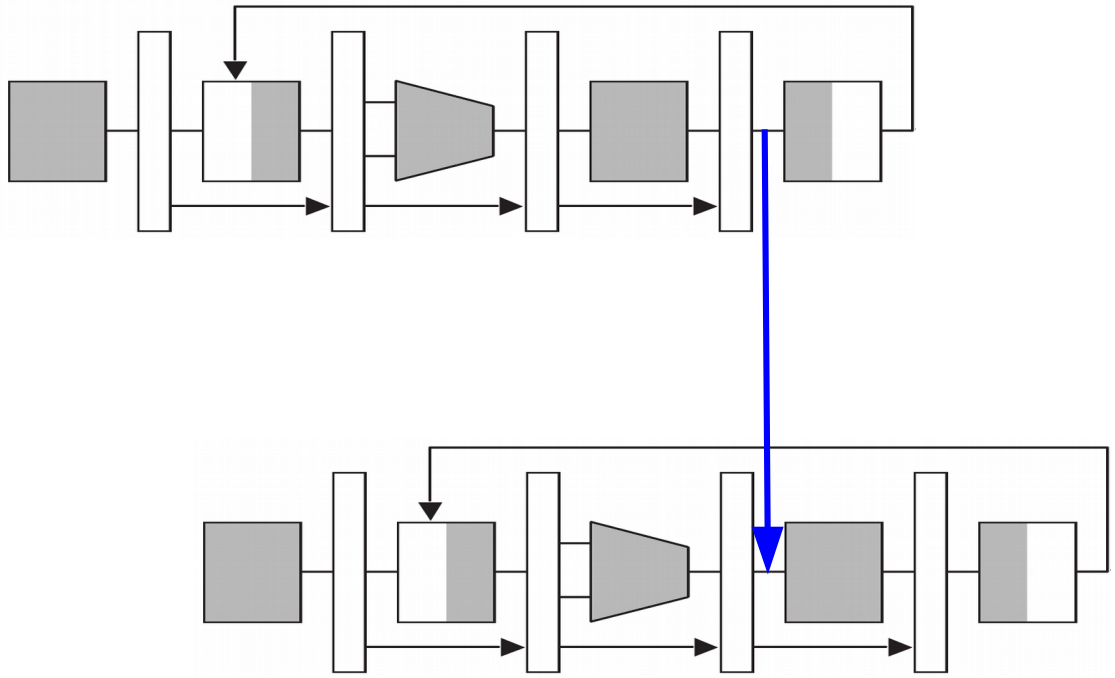
Read after Write

- Ejemplo 4:
  - i: R5 ← Mem[R6]
  - i+1: Mem[R8]← R5

Read after Write

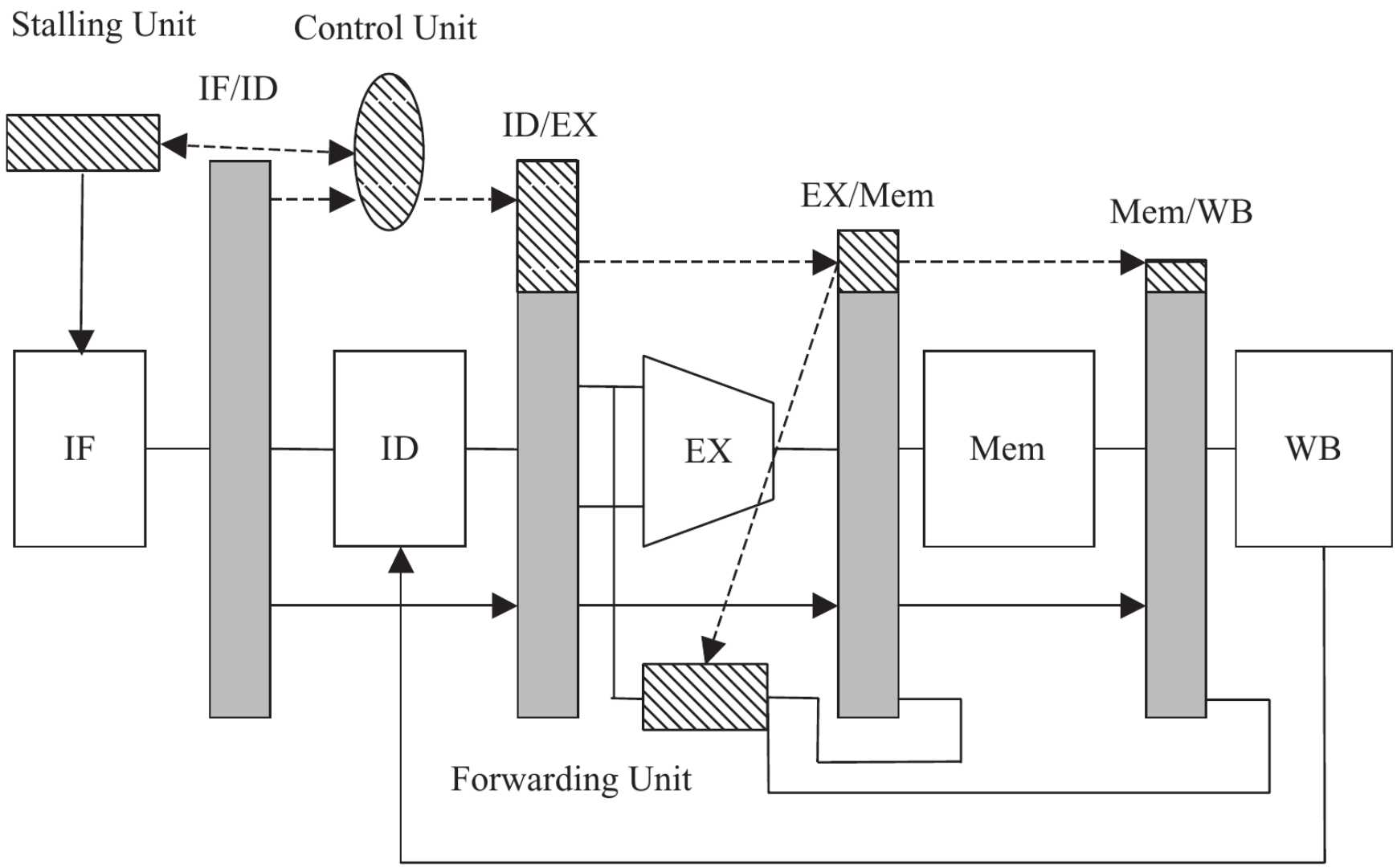
i: R5 ← Mem[R6]

i+1: Mem[R8] ← R5

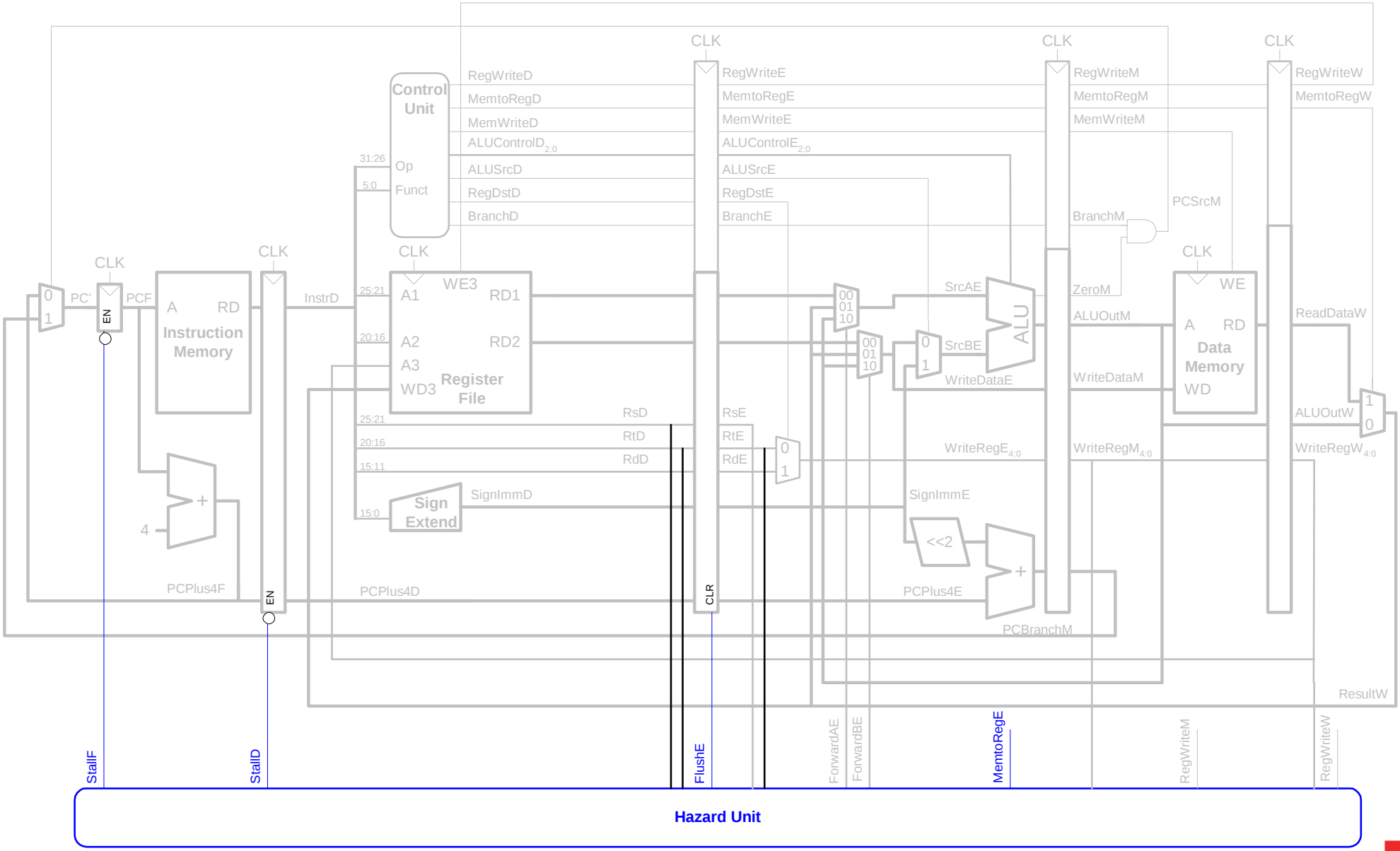


La lógica de forwarding debe reconocer que el valor a escribir en memoria es que se acaba de leer en la instrucción anterior y no el valor del registro R5.

Unidades de Stall y Forwarding



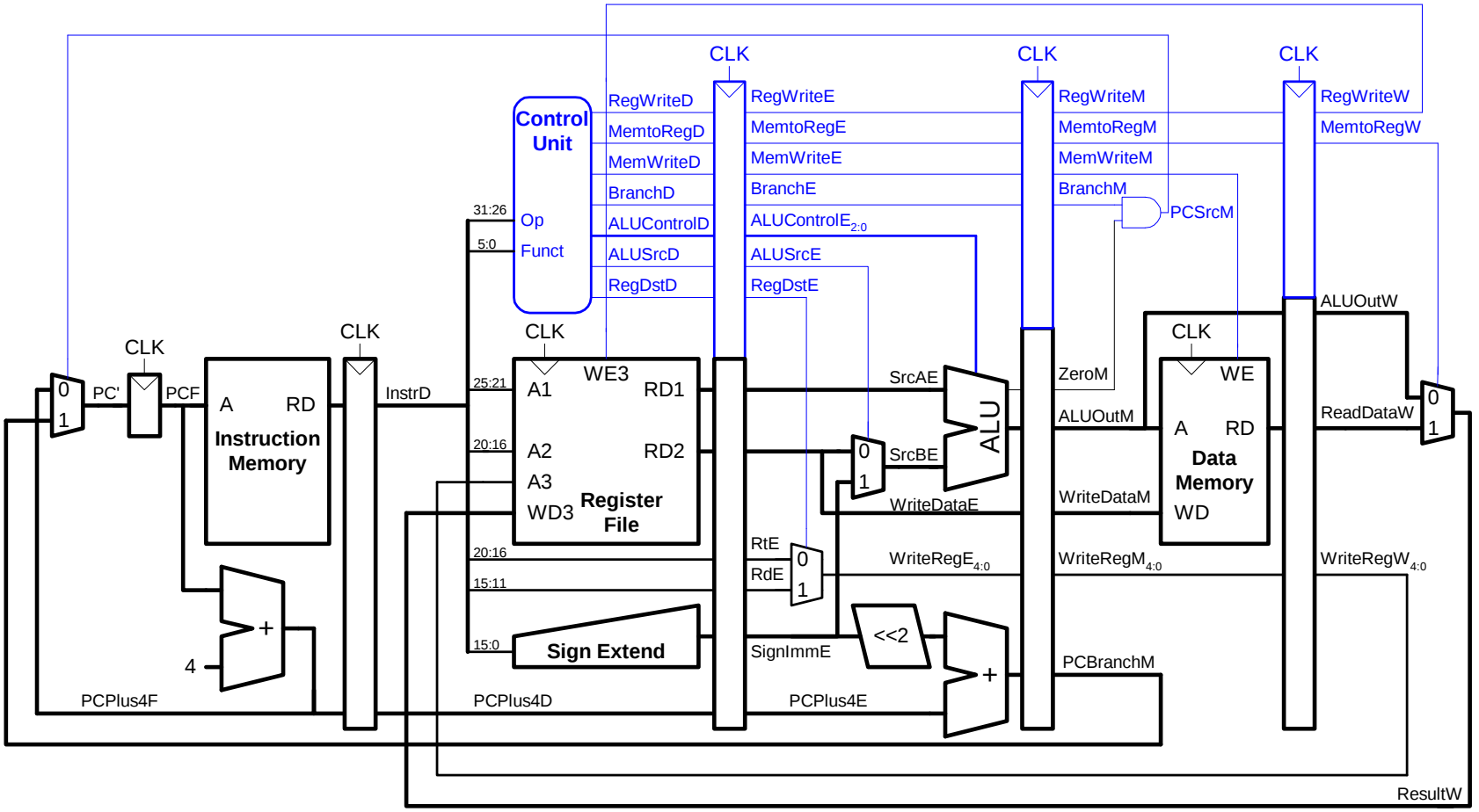
Unidad de conflictos



Microarquitectura en pipeline

1) **Instruction fetch (IF).** Se trae de memoria la instrucción en la posición indicada por el PC. Se asume que la instrucción no es un *branch* y se incrementa el PC.
2) **Instruction decode (ID)**. Se decodifica la instrucción y se reconoce su tipo. Se buscan los operandos.
3) **Execution (EX)**. Se ejecuta la instrucción. Difiere si es una operación aritmética, un acceso a memoria o un *branch*.
4) **Memory access (Mem)**. Se accede efectivamente a la memoria en caso de que la instrucción sea un acceso a memoria.
5) **Writeback (WB)**. Si la instrucción no es un *branch* o un *store*, el resultado de la operación (o del *load*) se almacena en el registro resultado.

Datapath & control del pipeline



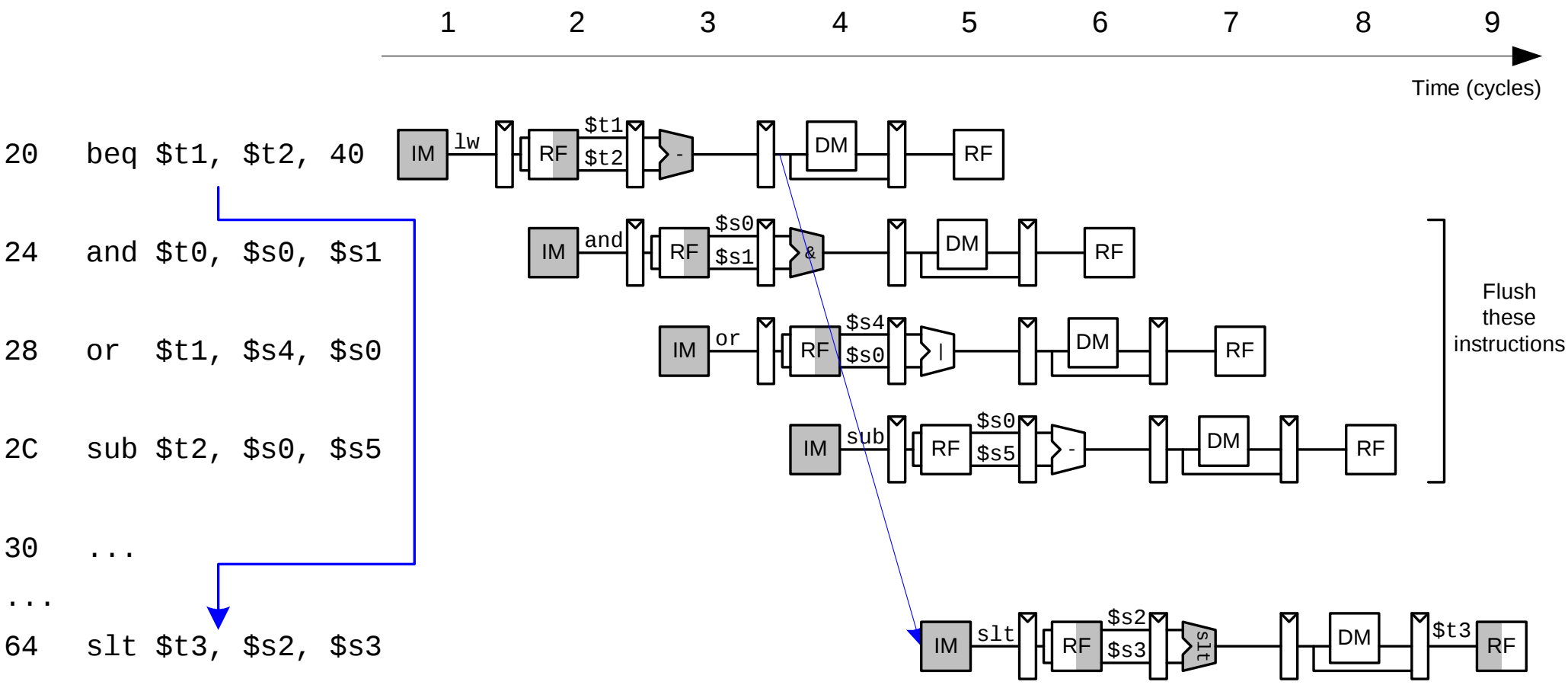
Conflictos de control

- El PC se incrementa en el Fetch.
- El pipeline trabaja bien cuando no hay transferencia de control.
- Instrucciones *branch*, *jump*, *call* y *return* interrumpen el flujo secuencial del programa y crean conflictos de control.
- Tanto la dirección de salto como la condición (branch) se calculan en *Execute*.
- Por lo tanto, recién se decide y se modifica el PC en *Memory*.

Branch

- La condición del *branch* y el PC destino no se saben hasta la 4ta etapa del pipeline.
- Hasta ese momento se hizo el fetch de las tres instrucciones siguientes al *branch*.
- Pero si hay que saltar esas instrucciones deberán descartarse.

Branch


1
2
3
4
5
6
7
8
9
Time (cycles)
20 beq $t1, $t2, 40
24 and $t0, $s0, $s1
28 or $t1, $s4, $s0
2C sub $t2, $s0, $s5
30 ...
...
64 slt $t3, $s2, $s3
IM
lw
RF
$t1
$t2
-
DM
RF
IM
and
RF
$s0
$s1
&
DM
RF
IM
or
RF
$s4
$s0
|
DM
RF
IM
sub
RF
$s0
$s5
-
DM
RF
IM
slt
RF
$s2
$s3
slt
DM
$t3
RF
Flush these instructions

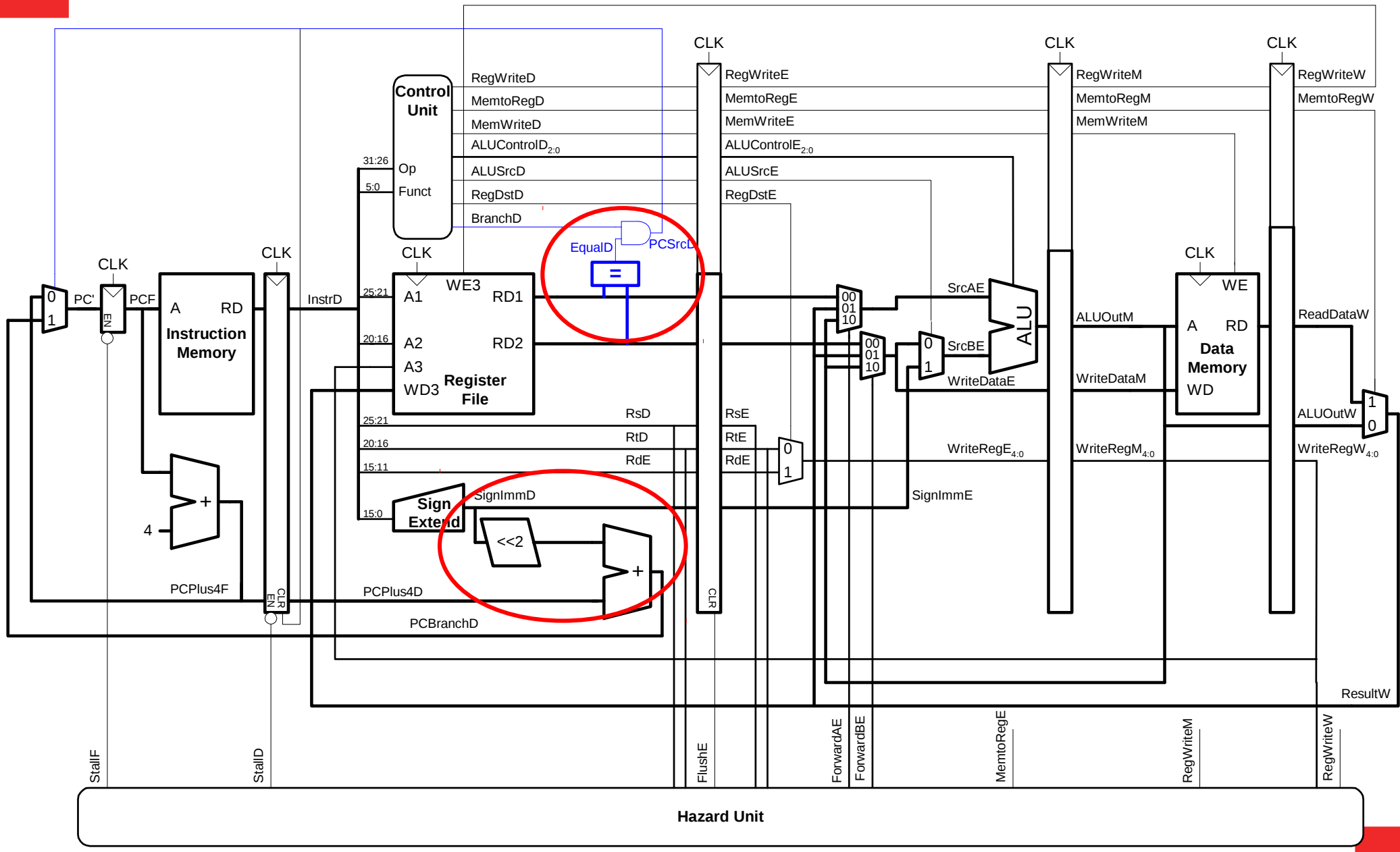
Penalización del branch

- Es la cantidad de instrucciones que se descartan cuando se toma el *branch*.
- Puede reducirse determinando el *branch* antes (cuando sea posible).
- *Branch on equal*:
  - Un comparador de igualdad es mucho más rápido que restar y detectar resultado cero.
  - Si es lo suficientemente rápido se podría mover al decode y determinar el PC en el decode.

Jump

- El *jump* (*call* y *return*) es un salto incondicional.
- En el pipeline básico el PC destino no se saben hasta la 4ta etapa del pipeline.
- Tiene la misma penalización que el *branch*.
- Puede reducirse adelantando el cálculo de la dirección destino (al decode).

Determinación temprana del salto



Gestión de conflictos de control

- A nivel compilador
  - Insertar NOPs → Disminuye el rendimiento del procesador
  - Reordenar el código (*branch* retardado) → No siempre es posible encontrar código útil para retardar el *branch*.
- A nivel hardware
  - Frenar el pipeline → Disminuye el rendimiento del procesador
  - Predicción del *branch*
    - Estática
    - Dinámica

Predicción del branch

- En un procesador en pipeline el CPI ideal es 1.
- La penalización del *branch* es un factor importante en el incremento del CPI.
- Predictor de *branch*:
  - Predecir si el *branch* será tomado o no.

- *La implementación sin predicción, equivale a predecir que será no tomado.*

Predicción estática del *branch*

- No depende de la historia del programa.
- Al final de los bucles (for, while) hay un salto para volver al principio y eso se repite varias veces.
- Los saltos hacia atrás son generalmente tomados.

- La predicción más simple controla la dirección del salto y predice tomado para los saltos hacia atrás.

Predicción dinámica del *branch*

- La predicción de los saltos hacia adelante necesitan más información del programa.
- Los predictores dinámicos usan la historia de ejecución del programa para predecir si el *branch* será tomado o no.
- Se mantiene una tabla de las últimas cien (o mil) instrucciones *branch* ejecutadas.
- *Branch target buffer*: incluye destino del branch e historia pasada.

Predicción dinámica del *branch*

- La predicción debe hacerse en la etapa IF para determinar qué instrucción ejecutar en el próximo ciclo.
- Cuando predice que debe saltar, la próxima instrucción es la indicada en el *branch target buffer*.
- Si en el *branch target buffer* se tiene información de branch y jump, se evita la penalización de los jumps.

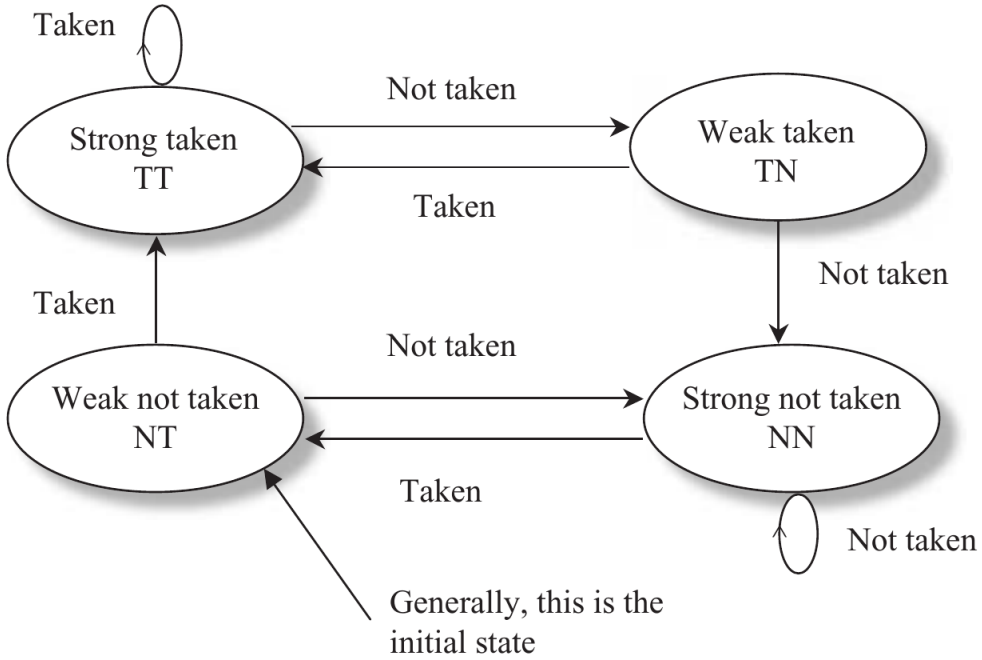
Predictor dinámico de 1 bit

- Recuerda qué ocurrió la última vez y predice lo mismo.
  - Mientras se repite el bucle, recuerda que el beq no salta y predice correctamente.
  - Luego del último ciclo, el *branch* salta, pero predice no tomado.
  - Si el bucle vuelve a ejecutarse, el predictor recuerda no tomado y predice erróneamente en la primera iteración.

```
R0 ← 0
R1 ← 10
For:
    Beq R0, R1, done
    …
    Add R0, R0, 1
Done:
```

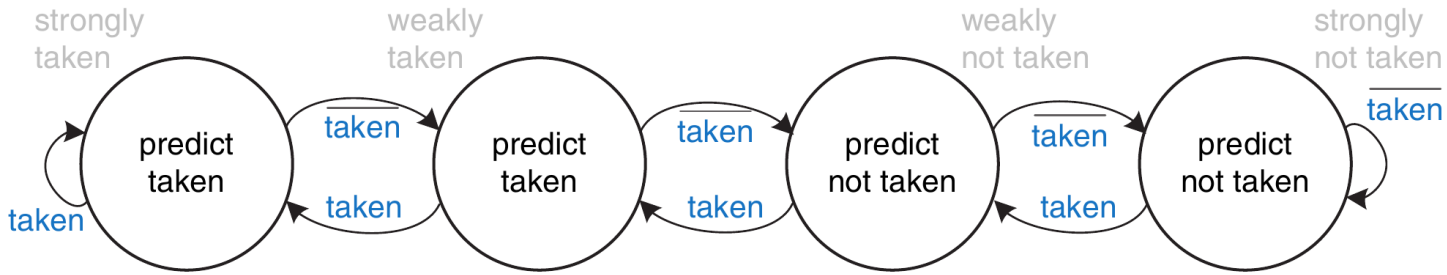
Predictor dinámico de 2 bits

- Para realizar la predicción distingue entre 4 estados.



Varían en cuantas veces debe ocurrir:

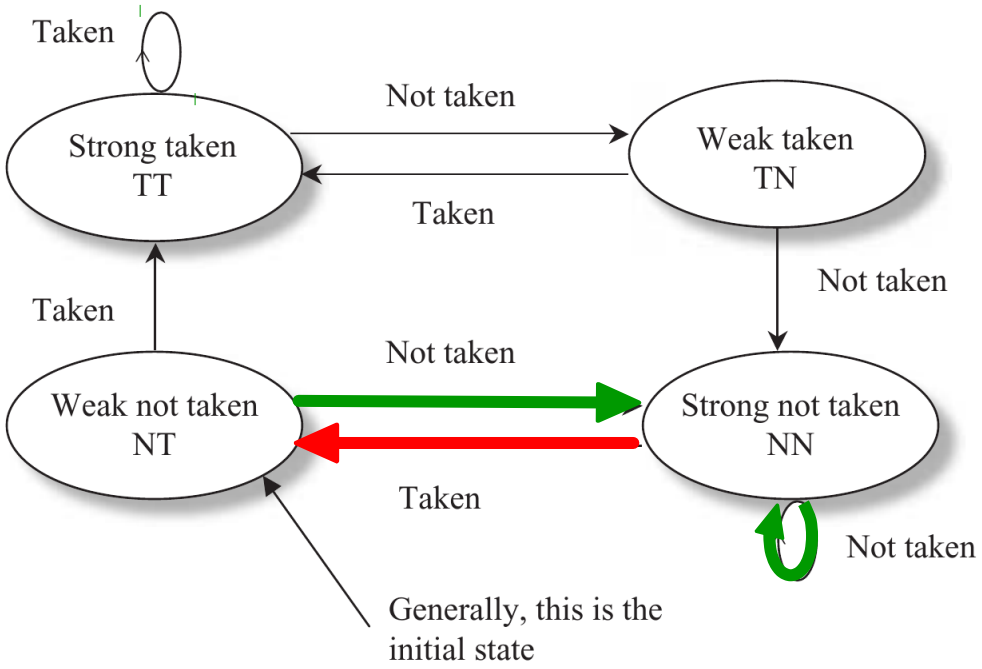
- *no-tomado* para que pase de *strongly-taken* a *strongly-no-taken*.
- *tomado* para que pase de *strongly-no-taken* a *strongly-taken*.



Predictor dinámico de 2 bits

- Para realizar la predicción distingue entre 4 estados.

```
R0 ← 0
R1 ← 10
For:
    Beq R0, R1, done
    …
    Add R0, R0, 1
Done:
```



La primera vez, predice no tomado, y efectivamente no debía saltar.
Sigue prediciendo no tomado, incluso luego de la última iteración.
Pero en la última iteración debía saltar.

Análisis de rendimiento

$T_1 =$ Tiempo de procesamiento sin pipeline

$T_k =$ Tiempo de procesamiento en pipeline con k etapas

$$Speedup_{Pipeline} = \frac{T_1}{T_k}$$

$Av_1 =$ Tiempo promedio de procesamiento sin pipeline

$Av_k =$ Tiempo promedio de procesamiento en k etapas

$$Speedup_{Pipeline} = \frac{Av_1}{Av_k}$$

Análisis de rendimiento

- Ciclos por instrucción (pipeline de k etapas)
  - 1 instrucción → k ciclos
  - n instrucciones → $k + n - 1$ ciclos
  - En promedio

Profundidad del pipeline

$$\text{1 instrucción demanda: } \frac{n}{k+n-1}$$

  - Idealmente

$$CPI_{\text{ideal}} = \lim_{n \to \infty} \frac{n}{k+n-1} = 1$$

  - CPI = CPI_{ideal} + Ciclos Stall por instrucción
    = 1 + Ciclos Stall por instrucción

Análisis de rendimiento

- Los *stalls* causan que baje el rendimiento del pipeline.

$$Av_1 = \text{CPI}_{\text{único ciclo}} \times \text{Período del reloj}_{\text{único ciclo}}$$
$$Av_k = \text{CPI}_{\text{pipeline}} \times \text{Período del reloj}_{\text{pipeline}}$$
$$Speedup_{Pipeline} = \frac{Av_1}{Av_k}$$

- $\text{Período del reloj}_{\text{único ciclo}} = k \times \text{Período del reloj}_{\text{pipeline}}$
- $\text{CPI}_{\text{único ciclo}} = 1$

Análisis de rendimiento

$$Av_1 = 1 \times k \times \text{Período del reloj}_{\text{pipeline}}$$

$$Av_k = (1 + \text{Ciclos Stall por instrucción}) \times \text{Período del reloj}_{\text{pipeline}}$$

$$Speedup_{Pipeline} = \frac{Av_1}{Av_k} = \frac{k}{1 + \text{Ciclos Stall por instrucción}}$$

- Si no hay Stalls, el Speedup del pipeline es igual a la profundidad *k* del pipeline.

Análisis de rendimiento

Pipeline de 5 etapas.

- Branch sin predicción. Branch y jump actualizan el PC en Memory. → Se descartan 3 instrucciones.
- 25% load
- 11% branch
- 2% jump
- 40% de los load están seguidos de una instrucción que usa el dato → requiere un stall
- 60% de los *branch* se toman → requiere descartar tres instrucciones.

Análisis de rendimiento

- Ciclos Stall por instrucción
  - + 25% load * 40% * 1
  - + 11% branch * 60% * 3
  - + 2% jump * 3

    ≈ 0.36

$$Speedup_{Pipeline} = \frac{5}{1+0.36} \approx 3.68$$

Análisis de rendimiento

Pipeline de 5 etapas.

- Predicción de branch. Branch y jump se resuelven en ID → Se descarta una instrucción.
- 25% load
- 11% branch
- 2% jump
- 40% de los load están seguidos de una instrucción que usa el dato → requiere un stall
- 25% de los *branch* se predicen mal → requiere descartar una instrucción.

Análisis de rendimiento

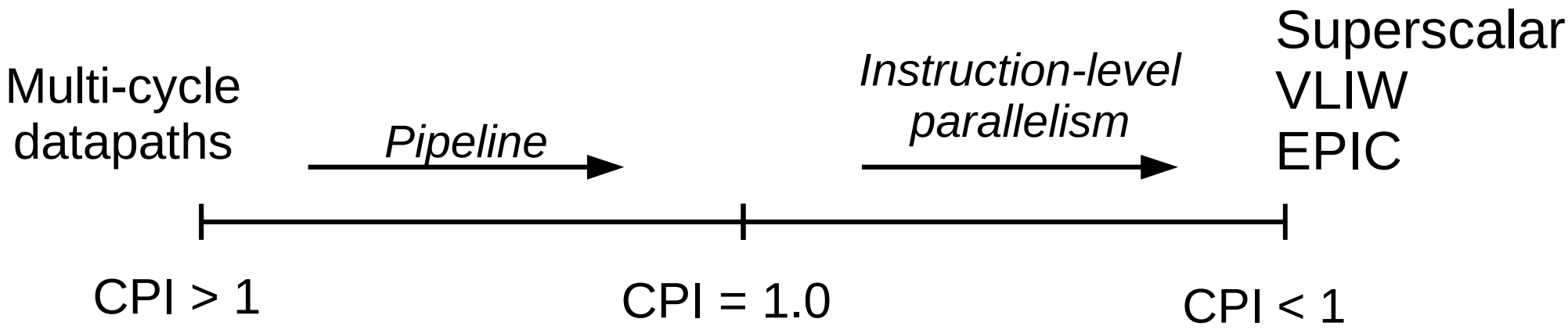
- Ciclos Stall por instrucción
  - + 25% load * 40% * 1
  - + 11% branch * 25% * 1
  - + 2% jump * 1

    ≈ 0.15

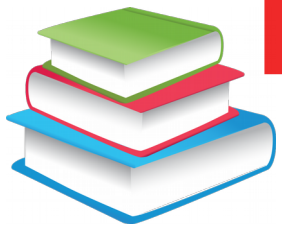
$$Speedup_{Pipeline} = \frac{5}{1+0.15} \approx 4.35$$

Instruction-level parallelism

- El objetivo es aprovechar al máximo el paralelismo a nivel de instrucciones para ejecutar múltiples instrucciones por ciclo.



Bibliografía



- *Capítulo 2*. *Multiprocessor Architecture. From simple pipelines to chip multiprocessor*. Jean-Loup Baer. Cambridge University Press. 2010.
- *Capítulo 7.* David Money Harris & Sarah L. Harris. *Digital Design and Computer Architecture*. Elsevier. 2013, 2da Ed.

Suplementaria

- *Apéndice C.* John L. Hennessy & David A. Patterson. *Computer Architecture. A Quantitative Approach*. Elsevier Inc. 2012, 5ta Ed.
- *Capítulo 4.* David A. Patterson & John L. Hennessy. *Computer Organization and Design. The Hardware/Software Interface.* Elsevier Inc. 2014, 5ta Ed.