

Tema 2. Segmentación Básica

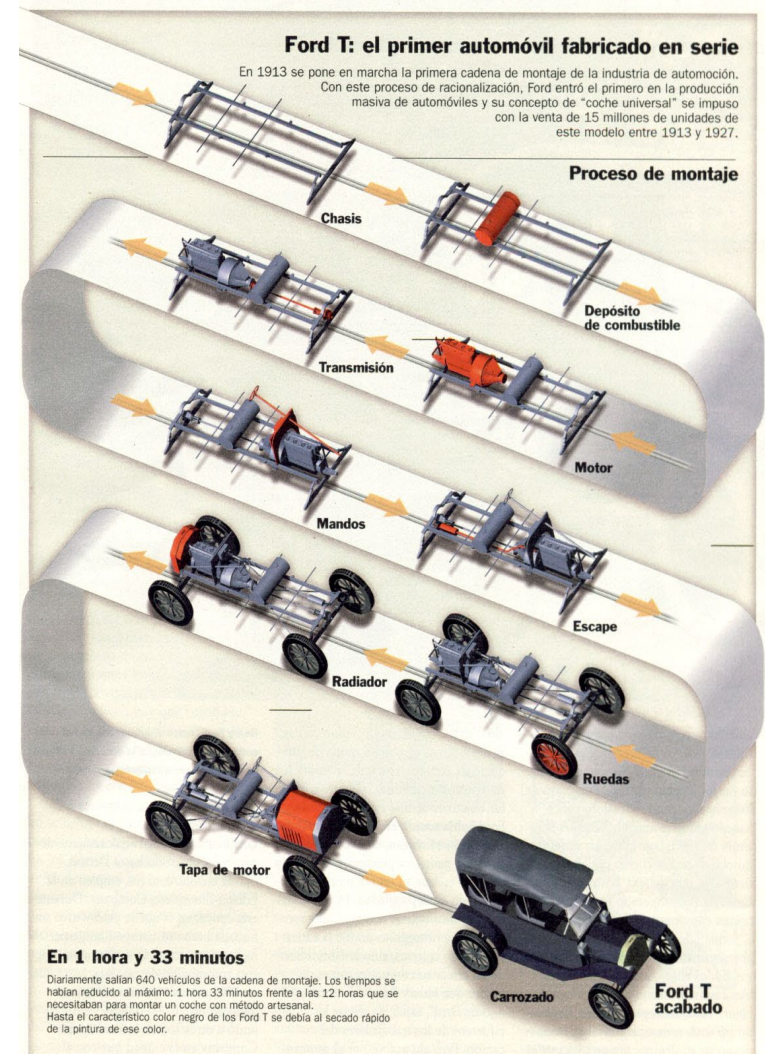
Departamento de Ingeniería y
Tecnología de Computadores

Índice

- Introducción
 - La Arquitectura DLX
 - Implementación de DLX sin segmentación
- Segmentación para la ejecución de instrucciones
- Problemas de la segmentación: los riesgos
 - Riesgos estructurales
 - Riesgos de datos
 - Riesgos de control

Introducción

- La **segmentación** es una técnica que consiste en solapar la ejecución de las instrucciones
- El cauce de ejecución estará dividido en varias etapas o segmentos
- Aprovecha el paralelismo existente entre las acciones necesarias para ejecutar las instrucciones
- Hoy en día es una técnica fundamental para conseguir CPUs más rápidas



Introducción

- Las etapas se conectan una a la siguiente formando el cauce (*pipeline*)
- Las instrucciones entran por un extremo del cauce, pasan por las distintas etapas y salen por el otro extremo
- Como las etapas del cauce están conectadas en cascada, todas deben de estar listas para avanzar en el mismo instante
- Definimos la **productividad** (*throughput*) como la frecuencia de salida de instrucciones
- El tiempo necesario para que una instrucción avance una etapa en el cauce se llama **ciclo máquina**, y coincidirá con el tiempo necesario para hacer la etapa más lenta

Introducción

- Por tanto, el tiempo por instrucción en el procesador segmentado en el caso ideal sería:

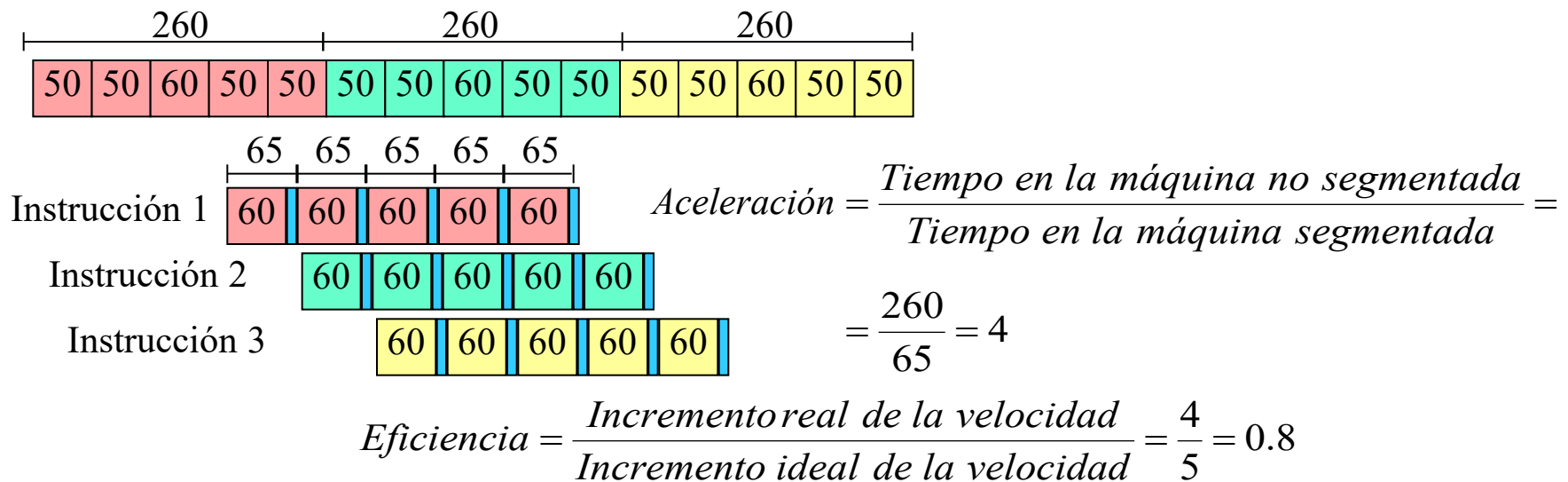
$$\frac{\text{Tiempo por instruc. en una máquina no segmentada}}{\text{Número de etapas en el cauce}}$$

- Así, en el caso ideal se obtiene una aceleración igual al número de etapas del cauce
- Normalmente estas etapas no estarán perfectamente balanceadas, y la segmentación suele tener un coste, con lo que el tiempo por instrucción no suele ser el mínimo posible, aunque sí se acerca
- La segmentación consigue aprovecharse del paralelismo existente entre las instrucciones, y no es visible al programador

Introducción

- Depende de cómo queramos verlo, la segmentación va a conseguir:
 - Disminuir el número de ciclos de reloj por instrucción (CPI)
 - Disminuir el tiempo del ciclo de reloj.

Ejemplo: Supongamos un procesador monociclo con un tiempo de ciclo de 260 u.t., sabiendo que la segmentación tiene una sobrecarga de 5 u.t., ¿qué beneficio se obtiene cuando segmentamos?

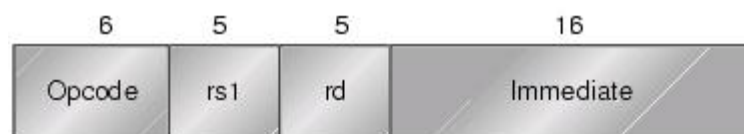


La Arquitectura DLX

- DLX es una arquitectura RISC de carga-almacenamiento parecida a MIPS y a la que usan la mayoría de procesadores actuales
- 32 registros de propósito general de 32 bits llamados R0, R1,R31 y otros 32 de punto flotante que pueden usarse como 32 registros de simple precisión (F0, F1, ... F31) o 16 de doble (F0, F2, ... F30)
- También tiene los siguientes registros especiales:
 - un registro de estado de punto flotante (fp)
 - el registro contador de programa (PC)
 - un registro de direcciones de interrupción (IAR)
- Los tipos de datos pueden ser bytes, medias palabras (16 bits), 32 bits para las palabras enteras y los datos en punto flotante de precisión simple, y 64 bits para la doble precisión
- Memoria direccionable por bytes con una dirección de 32 bits (capacidad de direccionamiento de 4 GBytes)

La Arquitectura DLX

- Modos de direccionamiento: registro, inmediato, base+desplazamiento relativo al PC y pseudodirecto
- Sólo tiene tres formatos de instrucciones, todas de 32 bits:
 - Instrucciones con inmediato
 - cargas y almacenamientos
 - todas las operaciones con inmediato
 - instrucciones de salto condicional
 - bifurcación a registro, bifurcación y enlace a registro
 - Instrucciones entre registros
 - operaciones ALU registro-registro
 - registros especiales de lectura/escritura y transferencias
 - Instrucciones de salto
 - Salto incondicional (j)
 - Salto incond. y enlace (jal)
 - Trap y retorno tras excepción



La Arquitectura DLX

- Repertorio de Instrucciones
 - Transferencia de datos
LB/LBU/SB, LH/LHU/SH, LW/SW, LF/SF, LD/SD
MOVI2S/MOVS2I, MOVF/MOVD, MOVFP2I/MOVI2FP
 - Aritmético-Lógicas
ADD/ADDI/ADDU/ADDUI, SUB/SUBI/SUBU/SUBUI
MULT/MULTU, DIV/DIVU
AND/ANDI, OR/ORI, XOR/XORI, LHI
SLL, SRL, SRA, SLLI, SRLI, SRAI
SLT/SLTI, SGT/SGTI, SLE/SLEI, SGE/SGEI, SEQ/SEQI, SNE/SNEI
 - Control
BEQZ, BNEZ, BFPT, BFPF, J, JR, JAL, JALR, TRAP, RFE
 - Punto Flotante
ADDD/ADDF, SUBD/SUBF, MULTD/MULTF, DIVD/DIVF
CVTF2D/CVTD2F, CVTF2I/CVTD2I, CVTI2F/CVTI2D
LTD/LTF, GTD/GTF, LED/LEF, GED/GEF, EQD/EQF, NED/NEF

Implementación de DLX sin segmentación

- Veamos primero una implementación mult ciclo de DLX
- Concretamente, vamos a implementar un subconjunto de DLX que consta de las siguientes instrucciones: carga y almacenamiento de palabras, saltos y operaciones enteras de la ALU
- Cada instrucción puede ejecutará en **5 etapas**:
 1. Búsqueda de instrucción (IF)
 2. Decodificación de la Instrucción/Leer Registros (ID)
 3. Ejecución / Dirección Efectiva (Ex)
 4. Acceso a Memoria / Finalización de Saltos (Mem)
 5. Postescritura (WB)

1. Búsqueda de instrucción o Instruction Fetch (IF)

- Busca la siguiente instrucción en la memoria y la trae al registro de instrucción (IR)
- Incrementa el PC en 4 y se almacena en el registro NPC

$$IR \leftarrow Mem[PC]$$

$$NPC \leftarrow PC + 4$$

Implementación de DLX sin segmentación

2. Decodificación de la Instrucción/Leer Registros (ID)

- Decodifica la instrucción y accede al banco de registros para leer dos registros
- La salida de los registros se almacena en dos registros intermedios (A y B) para usarla en los ciclos posteriores
- Se extiende el signo a los 16 bits menos significativos de la instrucción y se almacena en un registro intermedio (Imm)

$$\begin{aligned} A &\leftarrow \text{Regs}[IR_{6..10}] \\ B &\leftarrow \text{Regs}[IR_{11..15}] \\ \text{Imm} &\leftarrow ((IR_{16})^{16} \# \# IR_{16..31}) \end{aligned}$$

3. Ejecución / Dirección Efectiva (Ex)

- Dependiendo del tipo de instrucción, la ALU realiza (con los operandos obtenidos en la etapa anterior) lo siguiente:
 - A) Referencia a Memoria: Suma los operandos para formar la dirección efectiva. El resultado se deja en el registro ALU_{Output}

$$ALU_{\text{Output}} \leftarrow A + \text{Imm}$$

Implementación de DLX sin segmentación

B) Instrucción ALU Registro-Registro: Realiza la operación especificada por el código de función con los operandos situados en los registros A y B. El resultado se guarda en ALU_{Output}

$$ALU_{Output} \leftarrow A \text{ func } B$$

C) Instrucción ALU Registro-Inmediato: Realiza la operación especificada por el código de operación con los operandos situados en los registros A e Imm. El resultado se guarda en ALU_{Output}

$$ALU_{Output} \leftarrow A \text{ op } Imm$$

D) Salto:

- Calcula la dirección efectiva del salto sumando el registro NPC al valor del inmediato con signo que tenemos en Imm

$$ALU_{Output} \leftarrow NPC + Imm$$

$$Cond \leftarrow (A \text{ op } 0)$$

- Se chequea el registro A, leído en la etapa anterior, para determinar si el salto es tomado (se cumple la condición) o no
- La operación de comparación *op* es el operador relacional especificado en la instrucción de salto, por ejemplo, *op* es "==" para la instrucción BEQZ

Implementación de DLX sin segmentación

4. Acceso a Memoria / Finalización de Saltos (Mem)

- Las únicas instrucciones DLX activas en esta etapa son:

A) Referencia a Memoria: puede ser una carga o un almacenamiento

- En una carga el dato extraído de la memoria se almacena en el registro LMD
- En un almacenamiento el dato que está en el registro B se escribe en memoria
- En ambos casos, la dirección a usar es la calculada en la etapa anterior y almacenada en el registro ALUOutput

$$LMD \leftarrow Mem[ALU_{Output}]$$

ó

$$Mem[ALU_{Output}] \leftarrow B$$

B) Saltos: si la condición es tal que debe tomarse el salto, el PC se reemplaza por la dirección de destino del salto. Si no debe tomarse el PC se reemplaza con el PC incrementado que está almacenado en el registro NPC.

$$if(cond) PC \leftarrow ALU_{Output} \quad else \quad PC \leftarrow NPC$$

Implementación de DLX sin segmentación

5. Postescritura o Write-back(WB)

- Escribe el resultado en el banco de registros:
 - Para una instrucción de carga, el resultado está en LMD
 - Para una instrucción ALU, el resultado está en ALU_{Output}
- El campo que indica el registro destino también puede estar en dos posiciones distintas dependiendo del código de función indicado

- Instrucción ALU Registro-Registro:

$$\text{Regs}[IR_{16..20}] \leftarrow ALU_{Output}$$

- Instrucción ALU Registro-Inmediato

$$\text{Regs}[IR_{11..15}] \leftarrow ALU_{Output}$$

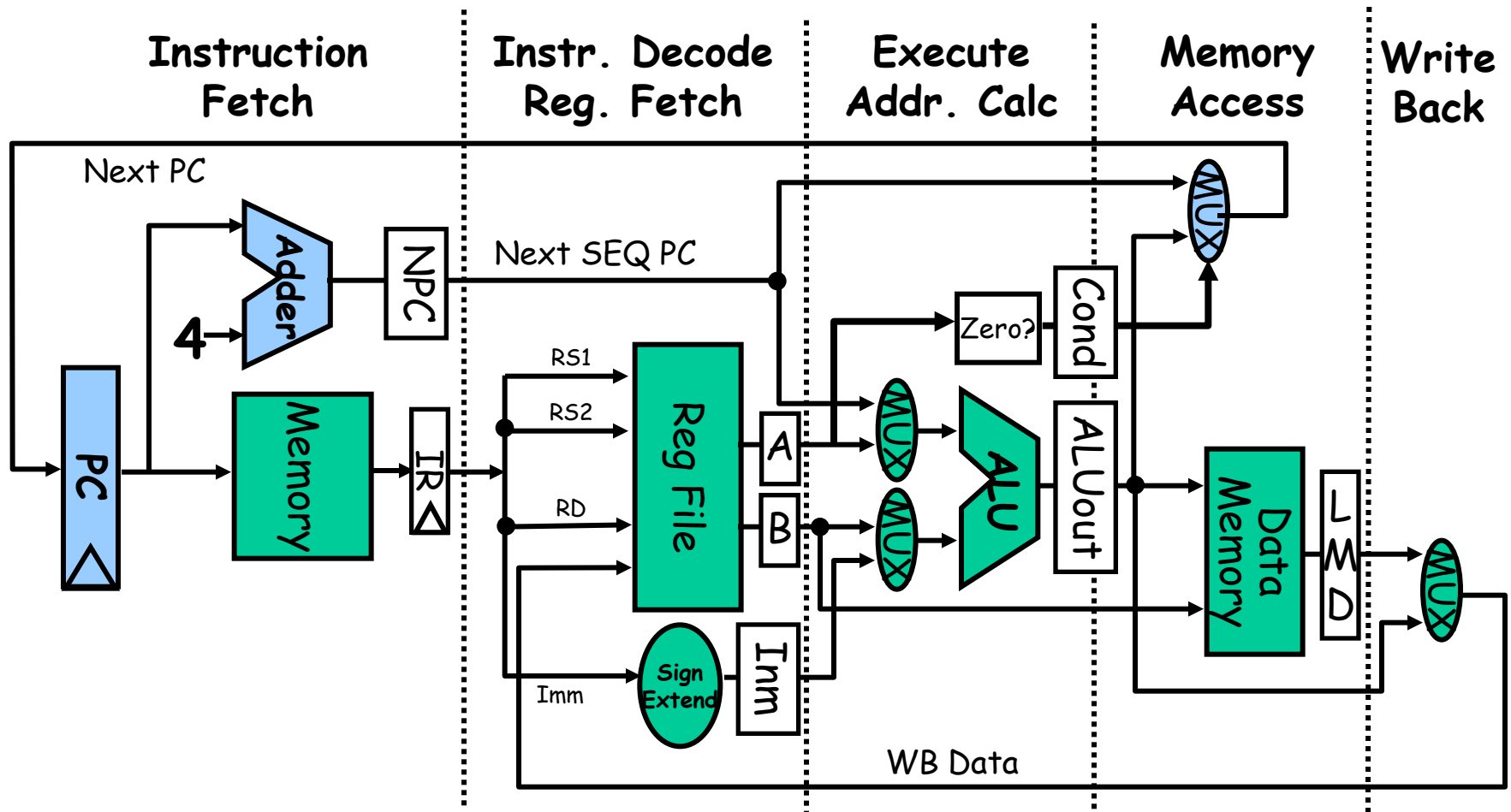
- Instrucción de carga

$$\text{Regs}[IR_{11..15}] \leftarrow LMD$$

¿Por qué el WB de un *ADD* se hace en el ciclo 5 en lugar de en el ciclo 4?

Implementación de DLX sin segmentación

- Al final de cada ciclo de reloj, lo calculado en una etapa se almacena en registros intermedios. Se necesitan 4 ciclos para saltos y almacenamientos, 5 para el resto

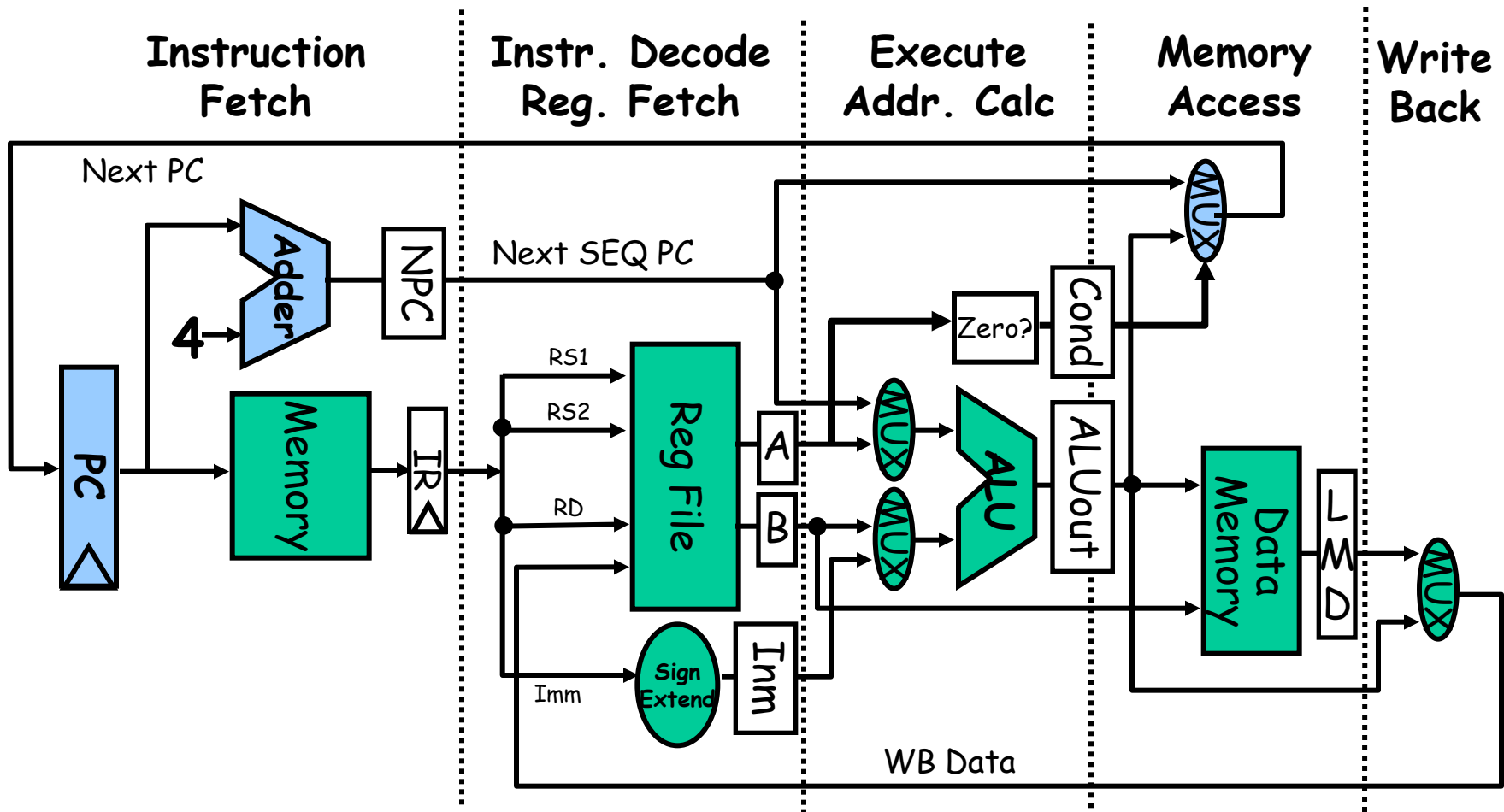


Índice

- Introducción
 - La Arquitectura DLX
 - Implementación de DLX sin segmentación
- Segmentación para la ejecución de instrucciones
- Problemas de la segmentación: los riesgos
 - Riesgos estructurales
 - Riesgos de datos
 - Riesgos de control

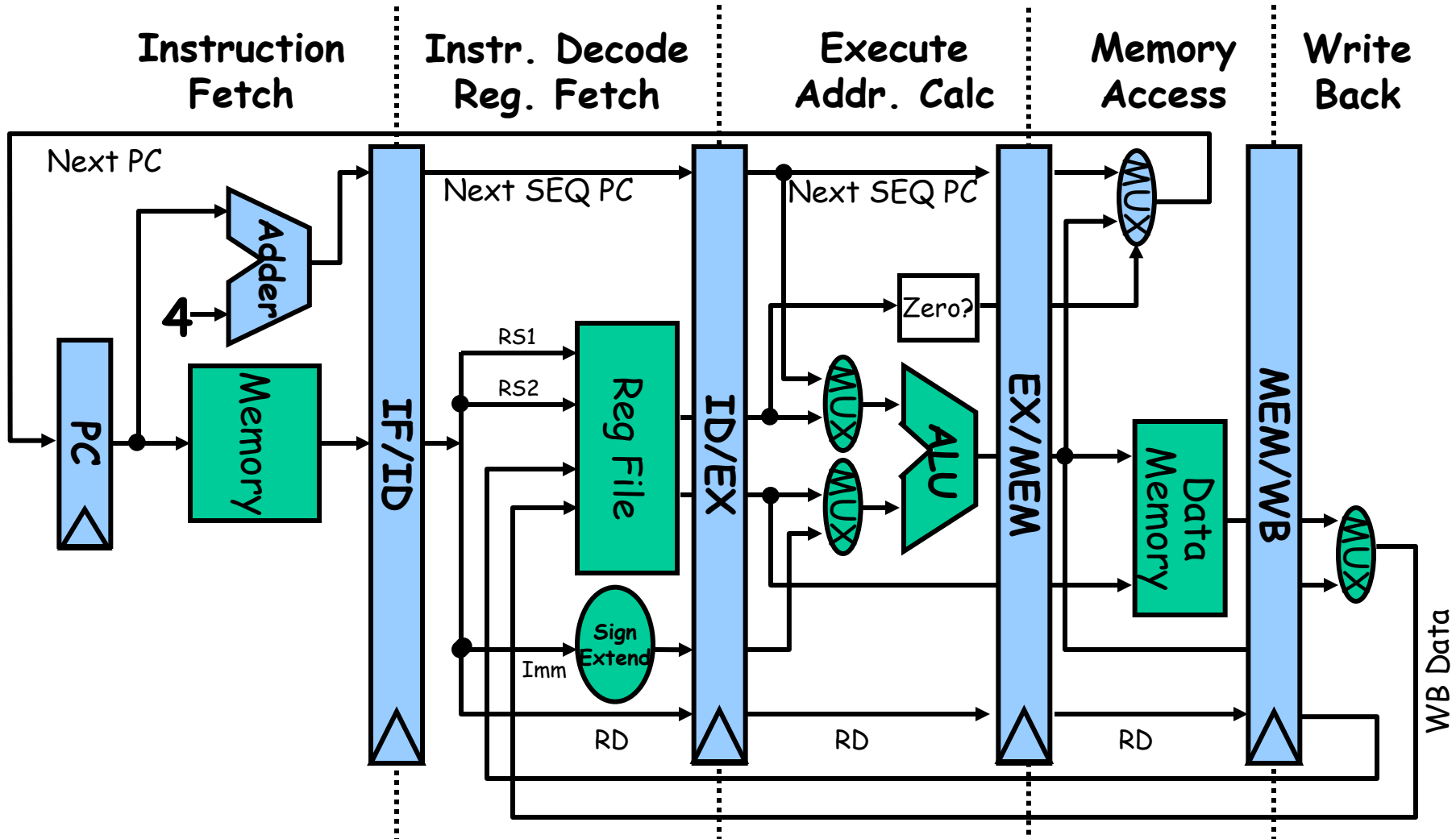
Segmentación para la ejecución de instr.

- Podemos segmentar el cauce de ejecución de instrucciones de DLX casi sin cambios de forma que comience una nueva instrucción en cada ciclo de reloj:



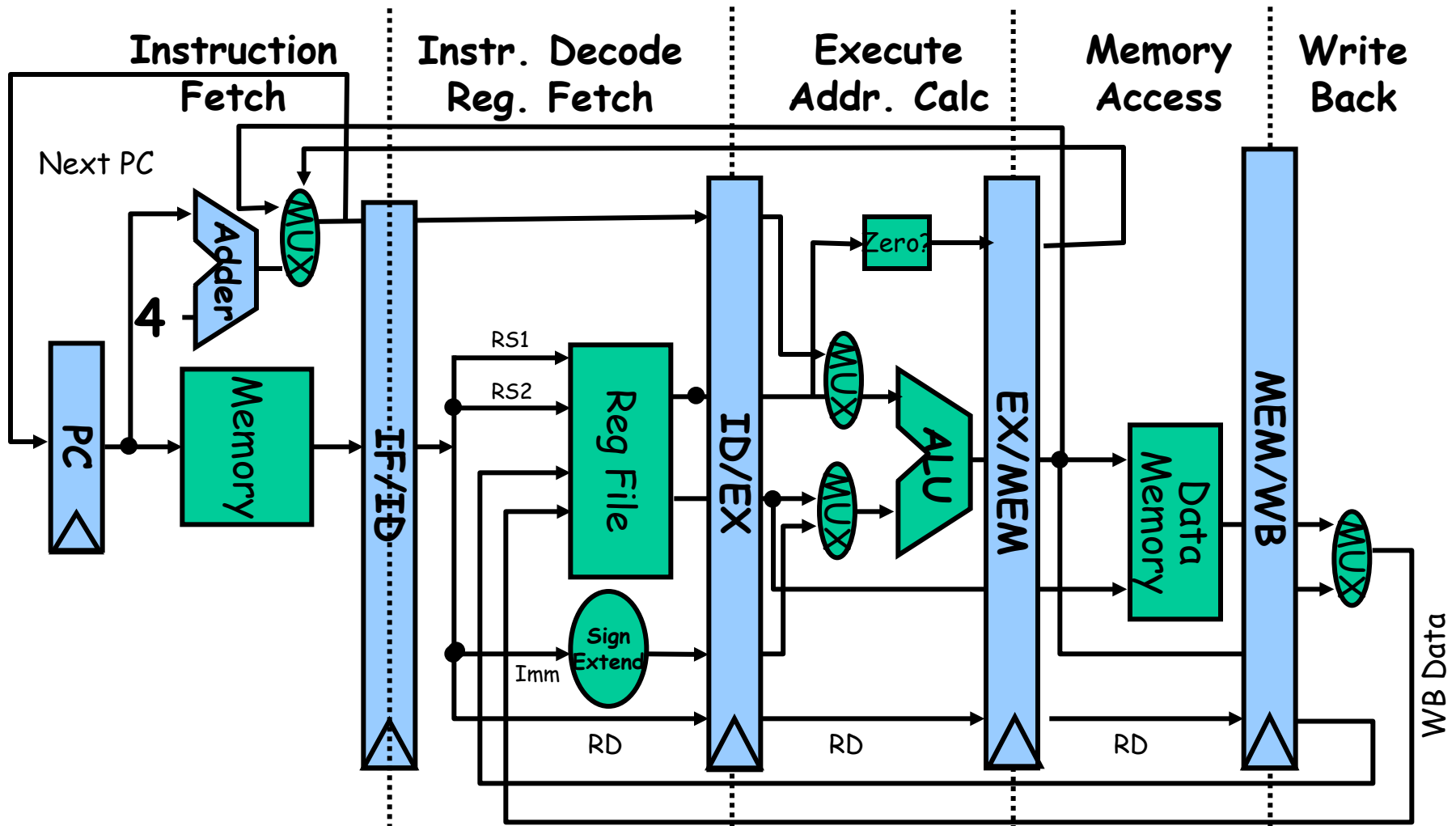
Segmentación para la ejecución de instr.

- Añadimos **los registros de la segmentación** que aíslan las distintas etapas, permitiendo que estas trabajen en paralelo:



Segmentación para la ejecución de instr.

- Modificamos el cauce para que se calcule la dirección de la siguiente instrucción en la etapa IF:



Segmentación para la ejecución de instr.

- Como cada etapa está activa en cada ciclo, cada una debe completar sus acciones en dicho ciclo
- Los registros de la segmentación (***pipeline registers*** o *pipeline latches*) **aíslan las etapas** del cauce:
 - Estos registros se llaman y se etiquetan con el nombre de las etapas que separan
 - Estos registros se usan para pasar datos y el control de una etapa a la siguiente:
 - cualquier valor que pueda ser necesario en una etapa posterior debe propagarse a través de estos registros, hasta que ya no sea necesario
- Una instrucción está activa en una única etapa en cada ciclo: las acciones se realizan entre dos registros de la segmentación

Segmentación para la ejecución de instr.

- De esta forma cada instrucción seguiría tardando cinco ciclos, pero ahora se estarían haciendo cinco instrucciones al mismo tiempo, estando cada una de ellas en una etapa distinta
- El patrón de ejecución sería algo así:

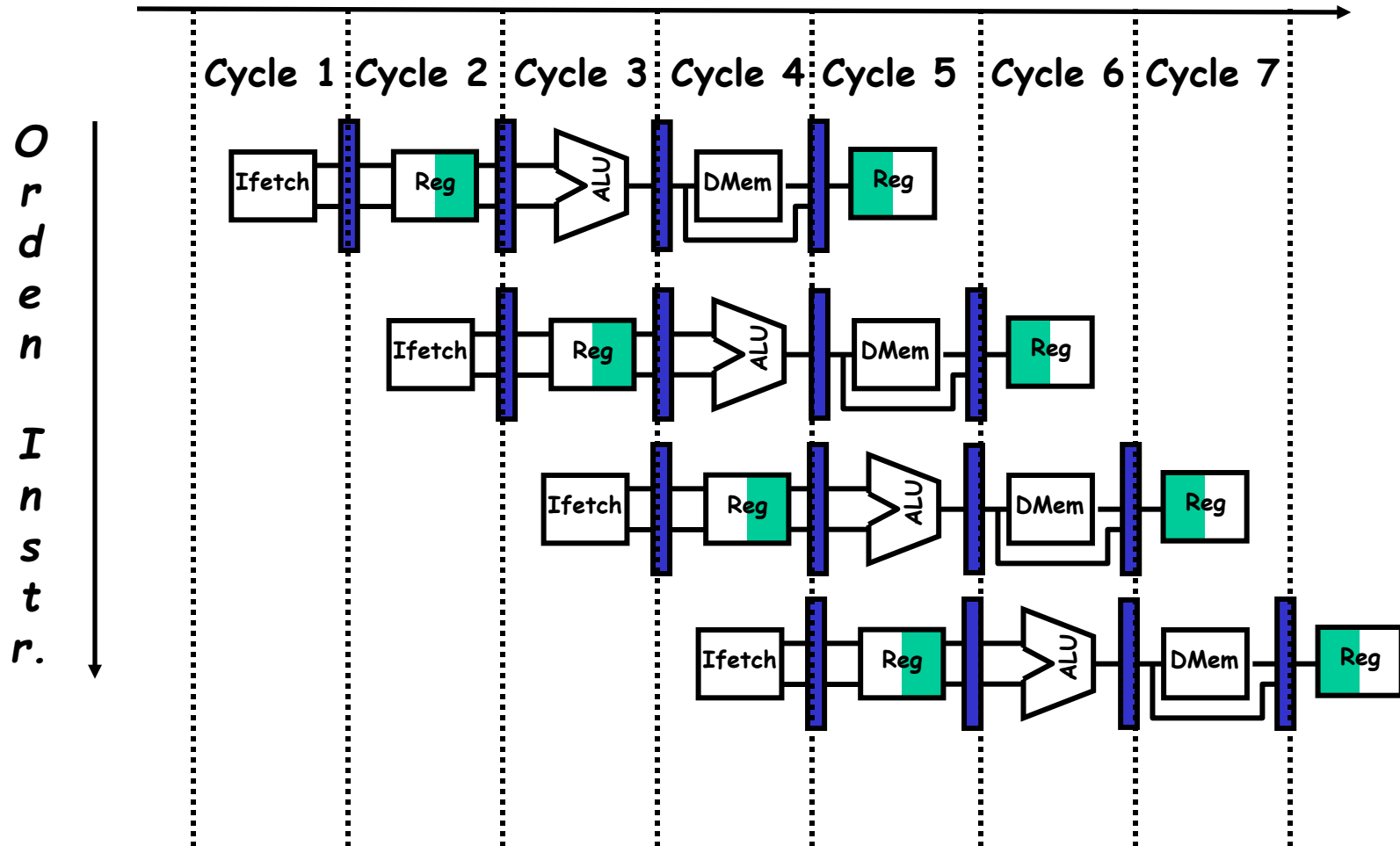
Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

- Hay que tener en cuenta que dos instrucciones distintas no pueden hacer uso de ningún recurso común al mismo tiempo, por lo que se debe asegurar que la segmentación no cause ese conflicto

Segmentación para la ejecución de instr.

- Los recursos que las diferentes instrucciones de cada etapa están usando serían:

Tiempo (ciclos de reloj)



Segmentación para la ejecución de instr.

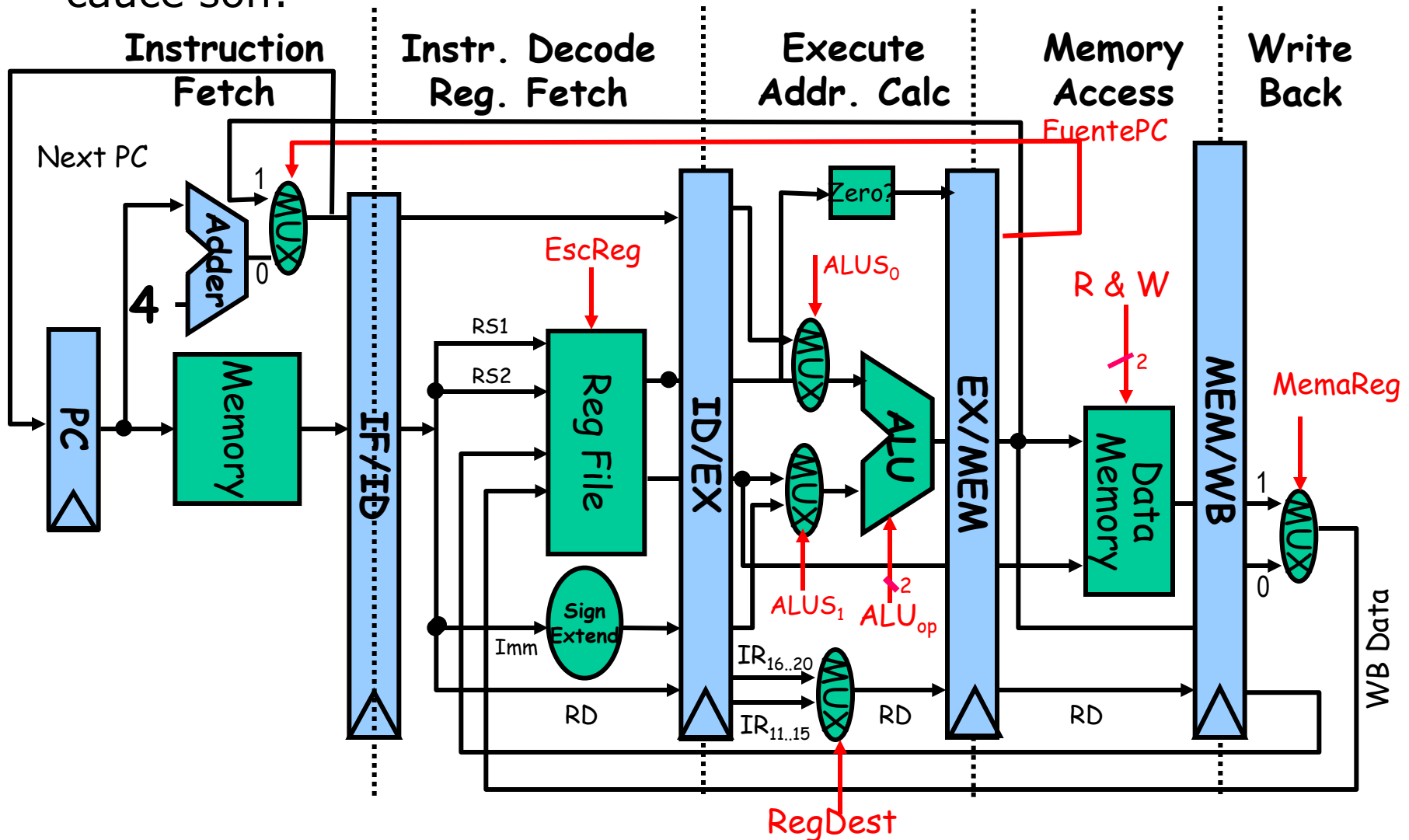
- Se usan memorias **cache separadas** para instrucciones y para datos, con lo que se puede acceder a la vez a ambas
- Si el ciclo de reloj es el mismo que el de la máquina sin segmentar, el **ancho de banda** del sistema de memoria debe ser cinco veces mayor
- El **banco de registros** se usa tanto en la etapa ID para la lectura de los dos registros, como en WB para la escritura del registro resultado
- Para poder empezar una nueva instrucción cada ciclo de reloj debemos incrementar el PC cada ciclo

Segmentación para la ejecución de instr.

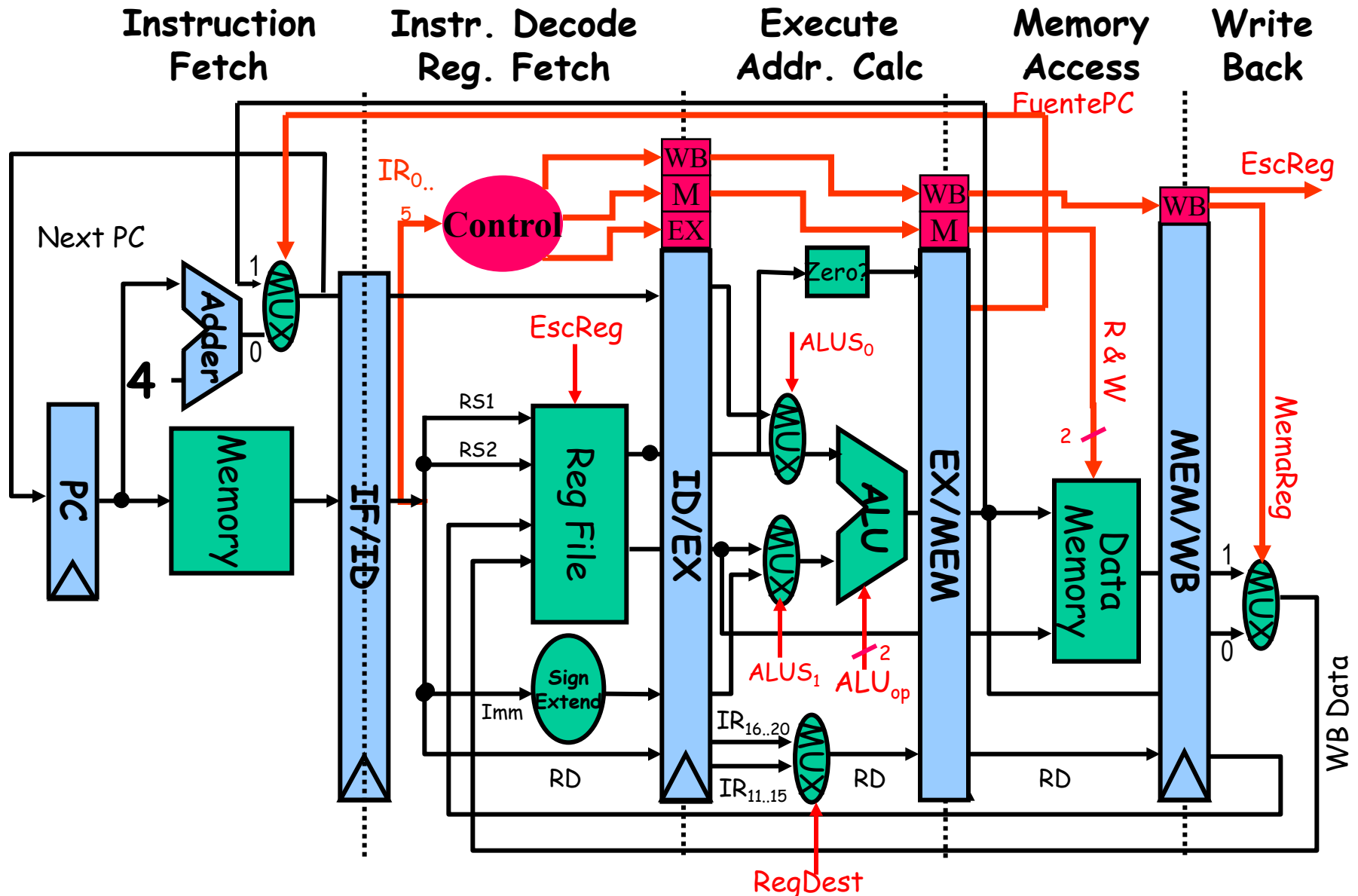
- La segmentación **incrementa la productividad** de instrucciones de la CPU, pero no reduce el tiempo de ejecución de cada una de las instrucciones
- De hecho, la segmentación suele incrementar ligeramente el tiempo de ejecución de cada instrucción debido a la sobrecarga que supone
- El incremento en la productividad de instrucciones significa que un programa se ejecuta más rápido y tiene un tiempo de ejecución total menor, aunque sus instrucciones tarden más en ejecutarse
- El hecho de que el tiempo de ejecución de cada instrucción no disminuya pone una cota práctica en la profundidad de la segmentación

Segmentación para la ejecución de instr.

- Las **señales de control** que la segmentación introduce en el cauce son:



Segmentación para la ejecución de instr.



Índice

- Introducción
 - La Arquitectura DLX
 - Implementación de DLX sin segmentación
- Segmentación para la ejecución de instrucciones
- Problemas de la segmentación: los riesgos
 - Riesgos estructurales
 - Riesgos de datos
 - Riesgos de control

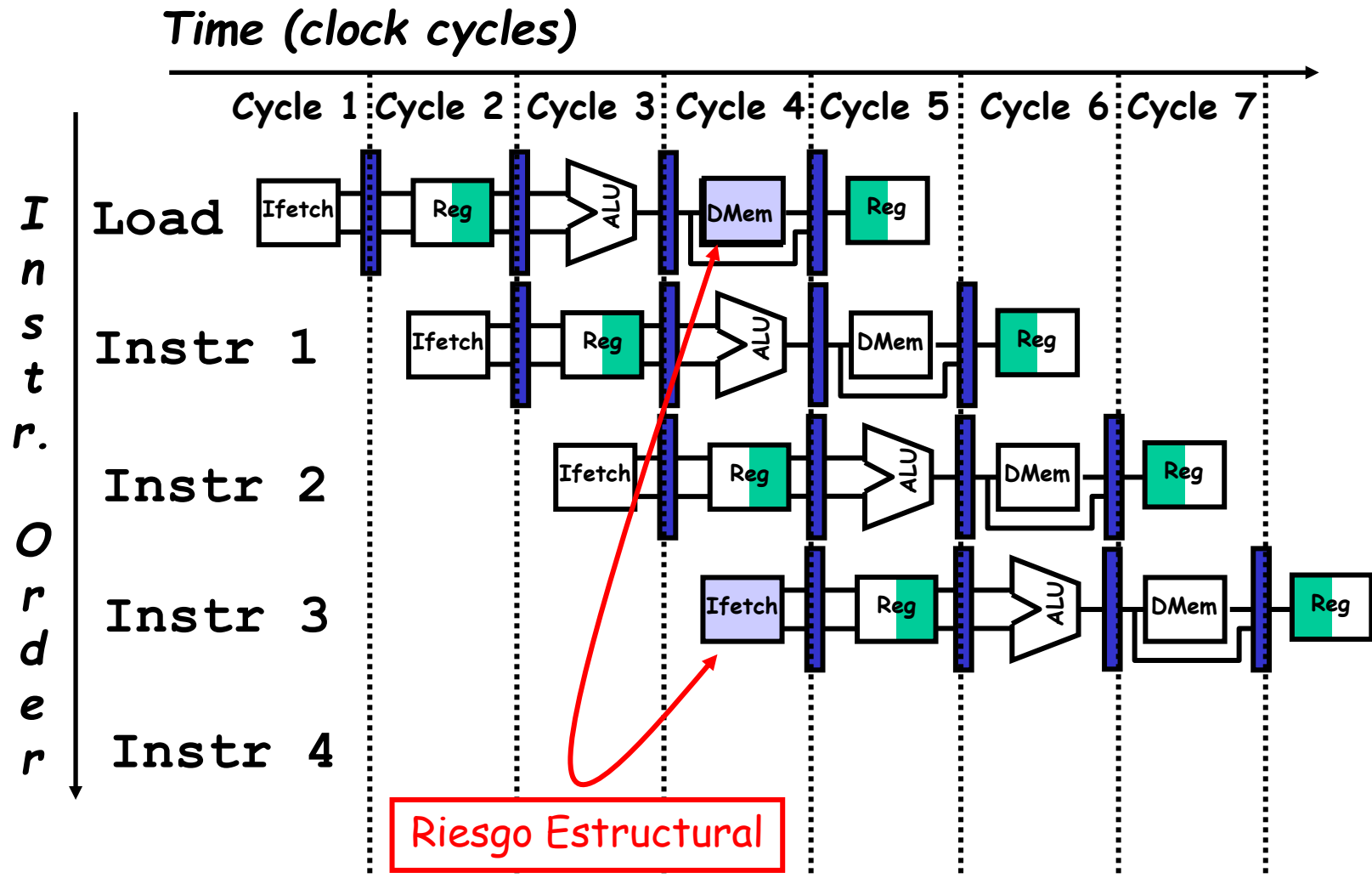
Problemas de la segmentación: los riesgos

- Hay situaciones, llamadas **riesgos** (*hazards*), en las que una instrucción no puede avanzar a la siguiente etapa del cauce
- Los riesgos provocan degradación del rendimiento
- Podemos distinguir tres clases de riesgos:
 - **Estructurales**: causados por un conflicto en el uso de un recurso hardware si hay combinaciones de instrucciones no soportadas
 - **Datos**: cuando una instrucción depende del resultado de otra previa, y la ejecución simultánea de ambas instrucciones puede provocar problemas
 - **Control**: provocados por la segmentación de los saltos y otras instrucciones que modifican el PC
- Los riesgos pueden ser **resueltos**:
 - A nivel **hardware**: detectando la situación y parando la instrucción problemática y las siguientes mientras avanzan las emitidas anteriormente hasta su finalización
 - A nivel **software**: añadiendo en el ISA una instrucción (NOP) que será empleada por el compilador/programador para evitar los riesgos

Riesgos estructurales

- Surgen cuando dos o más instrucciones necesitan **usar el mismo recurso** hardware para cosas distintas
- Esta situación ocurre cuando un **recurso no se ha replicado lo suficiente** (o no se ha segmentado) para permitir la combinación de algunas instrucciones
- **Ejemplos:**
 1. Banco de registros con un puerto de escritura y dos instrucciones necesitaran escribir un registro al mismo tiempo
 2. En máquinas con una única memoria para datos e instrucciones, cuando una instrucción tiene una referencia a memoria provoca un conflicto en el acceso a la memoria
 3. Si una unidad funcional necesita varios ciclos para realizar una operación y no ha sido segmentada, dos instrucciones consecutivas que quieran usarla no podrán hacerlo

Riesgos estructurales



Riesgos estructurales

- **Posibles soluciones:**

1. Dedicar **más hardware** al problema:

- Podría no ser posible por el incremento en el costo que supondría
- Incluso no merecería la pena en algunos casos: cuando se segmentan unidades funcionales que no se usan mucho

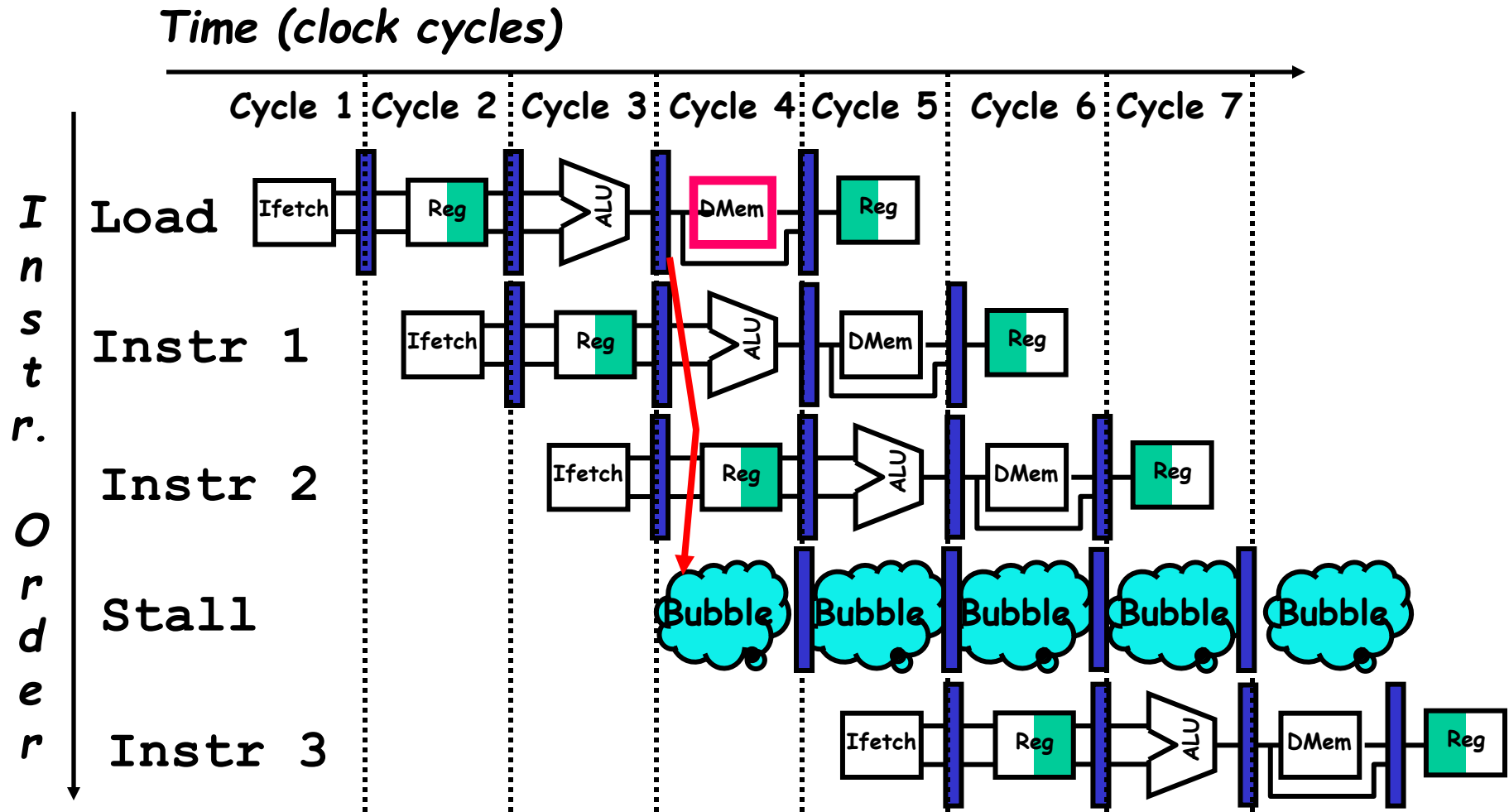
2. **Detener el cauce** durante un ciclo de reloj:

- Debemos detectar el riesgo
- Debemos tener un mecanismo para detener el cauce

3. Que **sea el software** (programador o compilador) el que evite el riesgo estructural:

- Por ejemplo, que separe en el programa dos instrucciones que requieran usar la misma unidad funcional lo suficiente

Riesgos estructurales



Riesgos estructurales

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

- La detención se muestra indicando el ciclo en el que no ocurre ninguna acción y se retrasa la emisión de la *instrucción $i+3$* hasta el ciclo siguiente, desplazándose en la traza
- El conjunto de instrucciones (ISA) debe permitir detectar los riesgos estructurales de forma sencilla:
 - Debe ser sencillo saber los recursos usados por una instrucción, pues el código de operación lo dice todo
 - Uniformidad en la utilización de recursos: las instrucciones DLX se pueden ejecutar de forma balanceada en 5 etapas

Riesgos estructurales

- Si el resto de factores son iguales, una máquina sin riesgos estructurales siempre tendrá un CPI menor:

$$CPI_{\text{Riesgos}} = CPI_{\text{Ideal}} + \text{Ciclos de reloj de detención por instrucción}$$

Ejemplo. Dado que las instrucciones de referencia a memoria constituyen el 40% de las instrucciones ejecutadas y que el CPI ideal, ignorando los riesgos estructurales, es 1, ¿cuánto más rápida es la máquina con caches separadas para datos e instrucciones que aquella que emplea una única cache? Suponer que el hecho de tener que detectar el riesgo estructural implica un ciclo de reloj que es 1.05 veces mayor.

$$T_{\text{CPU}\$i+\$d} = NI \times CPI_{\text{ideal}} \times T_{\text{CICLO}} = NI \times 1 \times T_{\text{CICLO}}$$

$$\begin{aligned} T_{\text{CPU}\$id} &= NI \times CPI_{\$id} \times 1.05 \times T_{\text{CICLO}} = NI \times (1 + 0.4 \times 1) \times 1.05 \times T_{\text{CICLO}} = \\ &= NI \times 1.47 \times T_{\text{CICLO}} \end{aligned}$$

Luego la máquina sin el riesgo estructural es un 47% más rápida !

Riesgos de datos

- Antes de explicar los **riesgos de datos** vamos a ver el concepto de **dependencia**
- Tipos de dependencias
 - **Dependencias de datos**
 - Existe flujo de información entre una instrucción productora de un dato y otra instrucción que consume dicho dato
 - **Dependencias de nombre**
 - No hay flujo real de información entre las instrucciones
 - Antidependencias
 - Dependencias de salida
 - **Dependencias de control**
 - Debidas a instrucciones de control o saltos

Dependencias de datos

- La instrucción **j** depende de **i**
 - i** produce un resultado que usa **j**
 - j** depende de **k** y **k** depende de **i**
- También llamadas **dependencias verdaderas**

```

Loop:  LD      F0,0(R1)
        ADDD   F4,F0,F2
        SD     0(R1),F4
        SUBI   R1,R1,#8
        BNEZ   R1,Loop
  
```

- Dependencias de datos para SUBI:

```

Loop:  LD      F0,0(R1)
        ADDD   F4,F0,F2
        SD     0(R1),F4
        SUBI   R1,R1,#8
        LD     F6,0(R1)
        ADDD   F8,F6,F2
        SD     0(R1),F8
        SUBI   R1,R1,#8
        LD     F10,0(R1)
        ADDD   F12,F10,F2
        SD     0(R1),F12
        SUBI   R1,R1,#8
        LD     F14,0(R1)
        ADDD   F16,F14,F2
        SD     0(R1),F16
        SUBI   R1,R1,#8
        BNE    R1,R2,Loop
  
```

Dependencias de nombre

- Antidependencia
 - La instrucción **j** escribe registro o posición de memoria que **i** lee
- Dependencia de salida
 - Las instrucciones **i** y **j** escriben en el mismo registro o posición de memoria

Loop: **LD** **F0,0(R1)**
 ADDD **F4,F0,F2**
 SD **0(R1),F4**
 LD **F0,-8(R1)**
 ADDD **F4,F0,F2**
 SD **-8(R1),F4**
 LD **F0,-16(R1)**
 ADDD **F4,F0,F2**
 SD **-16(R1),F4**
 LD **F0,-24(R1)**
 ADDD **F4,F0,F2**
 SD **-24(R1),F4**
 SUBI **R1,R1,#32**
 BNE **R1,R2,Loop**

SOLUCIÓN:
 renombrar
 registros

Loop: **LD** **F0,0(R1)**
 ADDD **F4,F0,F2**
 SD **0(R1),F4**
 LD **F6,-8(R1)**
 ADDD **F8,F6,F2**
 SD **-8(R1),F8**
 LD **F10,-16(R1)**
 ADDD **F12,F10,F2**
 SD **-16(R1),F12**
 LD **F14,-24(R1)**
 ADDD **F16,F14,F2**
 SD **-24(R1),F16**
 SUBI **R1,R1,#32**
 BNE **R1,R2,Loop**

Dependencias de control

- Cuando la ejecución de una instrucción depende de si otra instrucción de control (salto) ha sido tomado o no

	ADDU	R3,R3,R4
	BEQZ	R2,Etq
<hr/>		
	SUBU	R1,R5,R6
Etq:	OR	R7, R1,R8

```
If P1 { S1; };
```

```
If P2 { S2; };
```

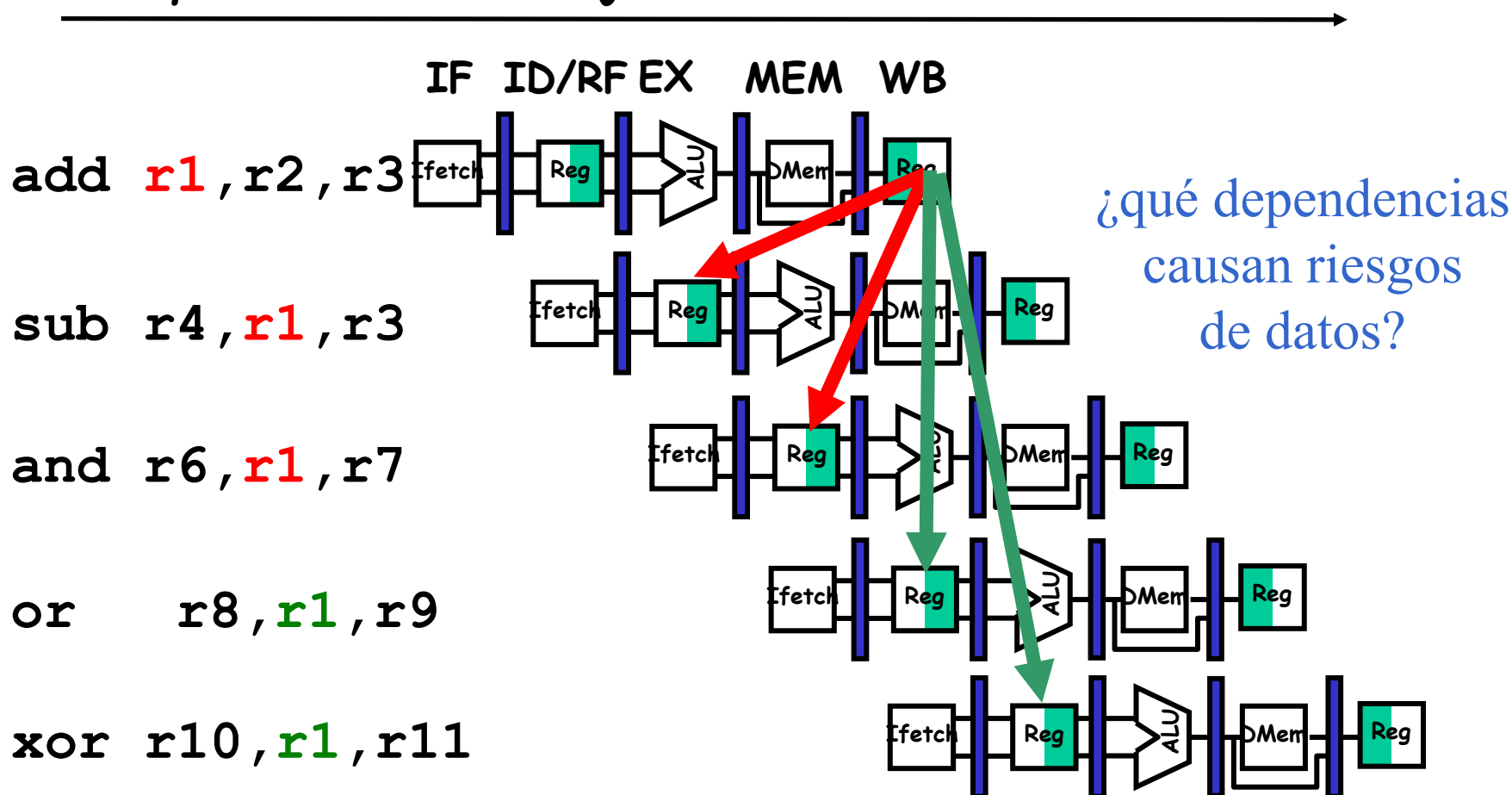
S1 depende de **P1** y **S2** de **P2** pero **NO** de **P1**.

- Determinan el flujo de ejecución de las instrucciones y el grado de reordenación que se puede realizar en el programa
 - Una instrucción dependiente de un salto (**posterior al salto**) no puede moverse a una posición anterior al salto
 - Una instrucción no dependiente de un salto (**anterior al salto**) no puede moverse a una posición posterior al salto

Riesgos de datos

- Los riesgos de datos ocurren cuando la segmentación cambia el orden de los accesos de lectura/escritura a los operandos que se daría en la versión secuencial sin segmentación


Tiempo en ciclos de reloj



Riesgos de datos

- Se produce un riesgo si hay una **dependencia** entre instrucciones, y estas están suficientemente cerca para que la segmentación cambie el orden de acceso a los operandos
- Puede darse en el **acceso a los registros**, pero también en el **acceso a una posición de memoria**
- En DLX, al tener sólo un canal de acceso a memoria y caché bloqueante, los accesos se hacen en orden
- Dependiendo del orden entre lecturas y escrituras, podemos clasificar los riesgos de datos


- **RAW** (*read after write*): una instrucción posterior (J) quiere leer un operando antes de que lo escriba una instrucción anterior (I)

 I: add **r1**, r2, r3
J: sub r4, **r1**, r3

- Viene provocado por una “**dependencia de datos**” y aparece por una **necesidad real** de comunicación

Riesgos de datos

- **WAR** (*write after read*): una instrucción posterior (J) trata de escribir su resultado antes de una instrucción anterior (I) haya leído el registro o posición de memoria que pretende escribir



 I: sub r4, **r1**, r3
 J: add **r1**, r2, r3
 K: mul r6, r1, r7

- Provocado por una “**anti-dependencia**” que se produce por una reutilización de R1 (un caso de **dependencia de nombres**)
- Los compiladores resuelven las anti-dependencias mediante el **renombramiento de registros**
- Esto **NO puede pasar en DLX** pues todas las lecturas se hacen en ID y todas las escrituras en WB

IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB

Riesgos de datos

- **WAW** (*write after write*): una escritura posterior (J) se produce antes que otra escritura anterior en el mismo destino (I)


 I: sub r1, r4, r3
 J: add r1, r2, r3

- Causado por una “**dependencia de salida**” (otro tipo de **dependencia de nombres**)
- Esto sólo es posible si:
 - varias etapas del cauce pueden escribir sus resultados, o
 - se permite que otra instrucción continúe mientras otra está detenida
- Luego en DLX no se van a producir estos riesgos

IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB

Riesgos de datos

- **¿Cómo resolver el problema?**

- **Solución HW:** detectándolo e **insertando ciclos de parada:**

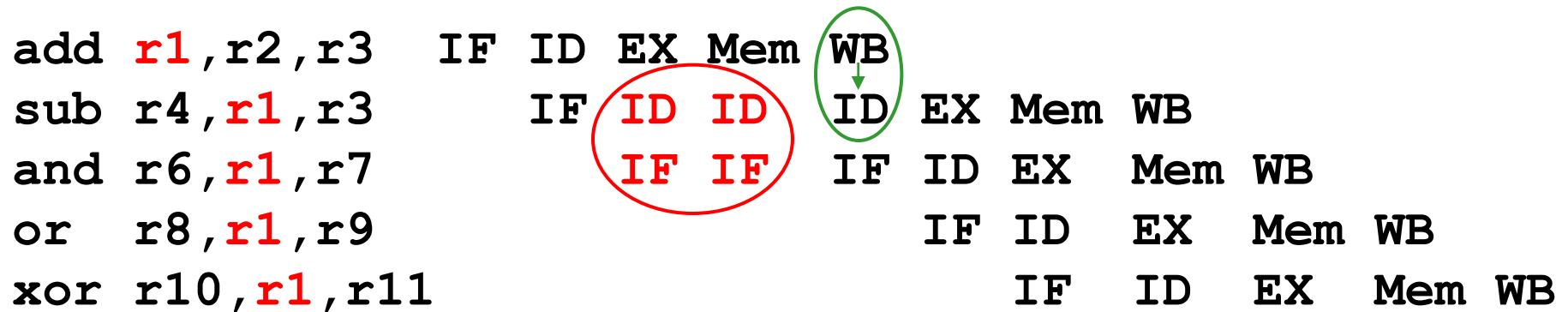
- Habría que añadir hardware para detectar el riesgo de datos y provocar una detención
- La instrucción que debe producir el dato (en el ejemplo el *add*) continuará, mientras que la que necesita el dato y las siguientes esperarán hasta que este esté disponible
- El CPI de la instrucción detenida aumenta en tantos ciclos como dure la detención

- **Solución SW:** técnicas de **compilación:**

- El compilador podría evitar el riesgo insertando instrucciones NOP (*no-operation*) entre la instrucción que produce el dato (el *add*) y la que lo consume
- Incrementa el número de instrucciones del programa

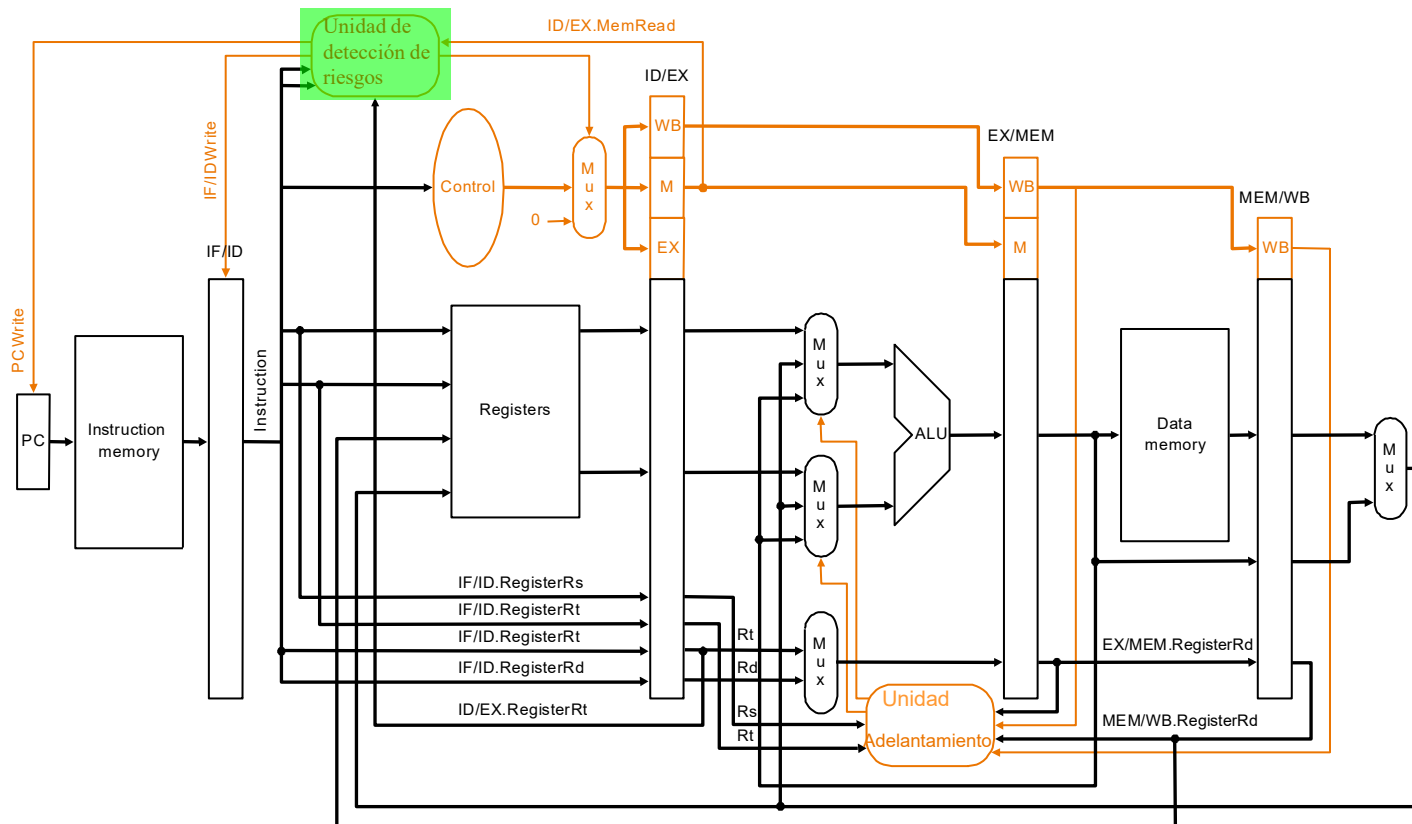
Riesgos de datos: solución HW

- La unidad de detección de riesgos detecta el problema en ID e inserta ciclos de parada (dos en este caso):



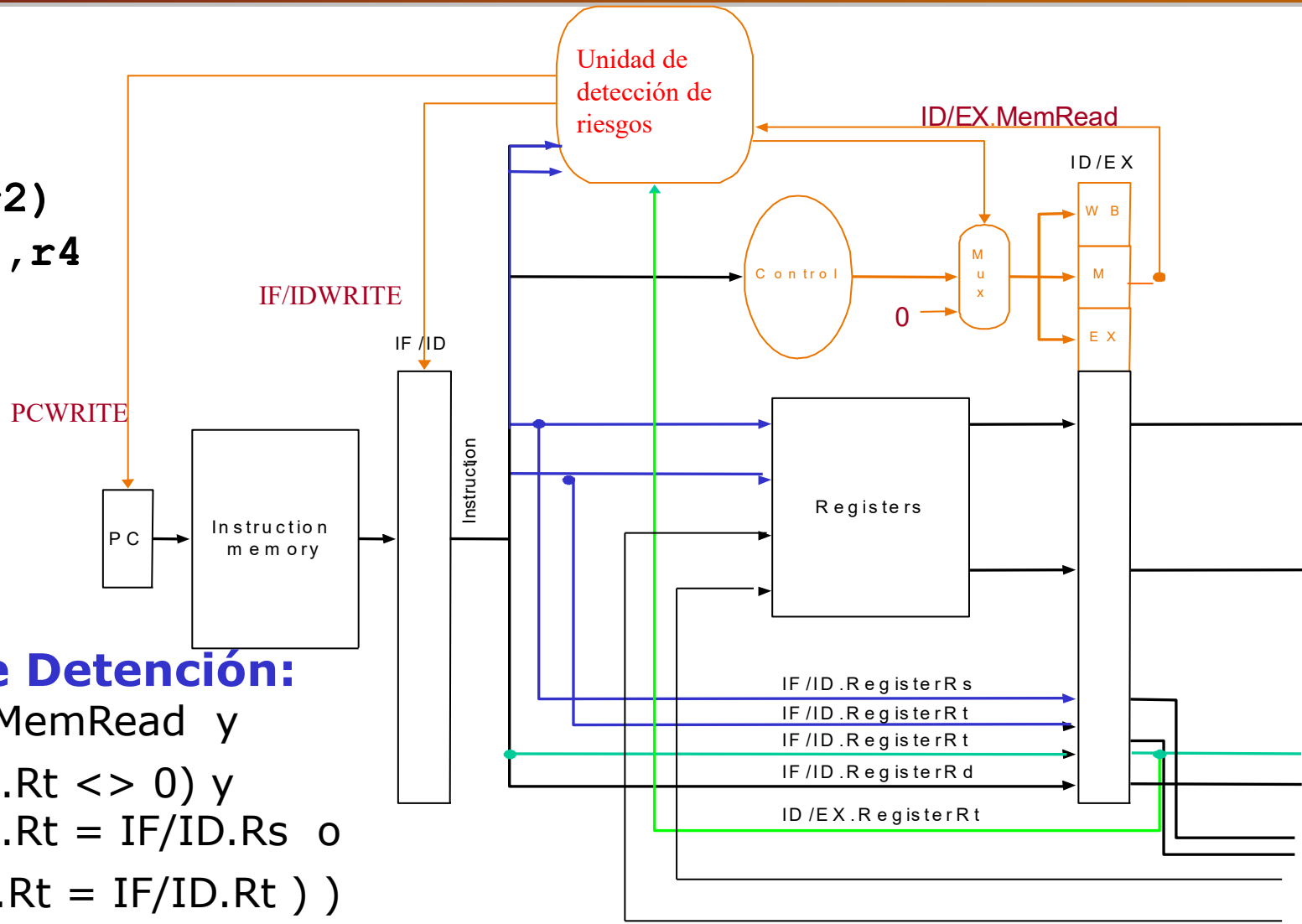
Riesgos de datos: solución HW

- **Unidad de detección de riesgos:** los riesgos por dependencias de datos se analizan en la etapa ID y si aparecen la instrucción no es emitida, siendo parada hasta que el riesgo desaparezca



Riesgos de datos: solución HW

```
lw r1, 0(r2)
add r3, r1, r4
```



Ejemplo de Detención:

```

si (ID/EX.MemRead y
    ( ID/EX.Rt <> 0) y
    ( ID/EX.Rt = IF/ID.Rs o
        ID/EX.Rt = IF/ID.Rt ) )
    bloquear pipeline;

```

Pero hay que considerar todas las situaciones posibles

Riesgos de datos: solución HW

Ejemplo: Suponer que el 30% de las instrucciones son cargas, y que la mitad de las veces la instrucción que sigue a una carga depende del resultado de dicha carga ¿Cuántos ciclos de parada ocasiona ese riesgo? ¿Cuánto más rápida es la máquina ideal frente a ésta con detenciones?

El número de ciclos de parada que se introducirían sería 2, de forma que coincidan en el tiempo la etapa WB de la instrucción de carga y la etapa ID de la instrucción consumidora.

La máquina ideal será tanto más rápida como el ratio de sendos CPIs. El CPI para las instrucciones que siguen a una carga sería **$1 + 0.5 \times 2 = 2$** (puesto que la mitad de las veces la instrucción que sigue a la carga necesita el dato producido por ésta).

Como las cargas son el 30% del total, el CPI efectivo de la máquina con detenciones será **$(0.7 \times 1 + 0.3 \times 2) = 1.3$** frente al CPI ideal de 1.

Así pues, la máquina ideal es un 30% más rápida.

La introducción de ciclos de parada incrementa el CPI

Riesgos de datos: solución SW

- Se emplea la **instrucción NOP** (*no-operation*) para separar las instrucciones dependientes de forma que no haya riesgo de datos:

add r1, r2, r3	IF	ID	EX	Mem	WB															
nop		IF	ID	EX		Mem	WB													
nop			IF	ID		EX		Mem	WB											
sub r4, r1, r3				IF	ID	EX		Mem	WB											
and r6, r1, r7					IF	ID	EX		Mem	WB										
or r8, r1, r9						IF	ID	EX		Mem	WB									
xor r10, r1, r11							IF	ID	EX		Mem	WB								

- Solución **más sencilla** desde el punto de vista del hardware, ya que evita tener que disponer de la unidad de detección de riesgos y la lógica de inserción de ciclos de parada
- Sin embargo, complica la vida del compilador (programador en última instancia) e impide la portabilidad de binarios

Riesgos de datos: solución SW

Ejemplo: Suponer que el 30% de las instrucciones son cargas, y que la mitad de las veces la instrucción que sigue a una carga depende del resultado de dicha carga. ¿Cómo se resolvería el riesgo utilizando instrucciones NOP? ¿Cuánto más rápida es la máquina ideal frente a ésta que inserta instrucciones extra?

Ahora tras cada instrucción de carga que es seguida por una instrucción dependiente hay que insertar 2 instrucciones NOP.

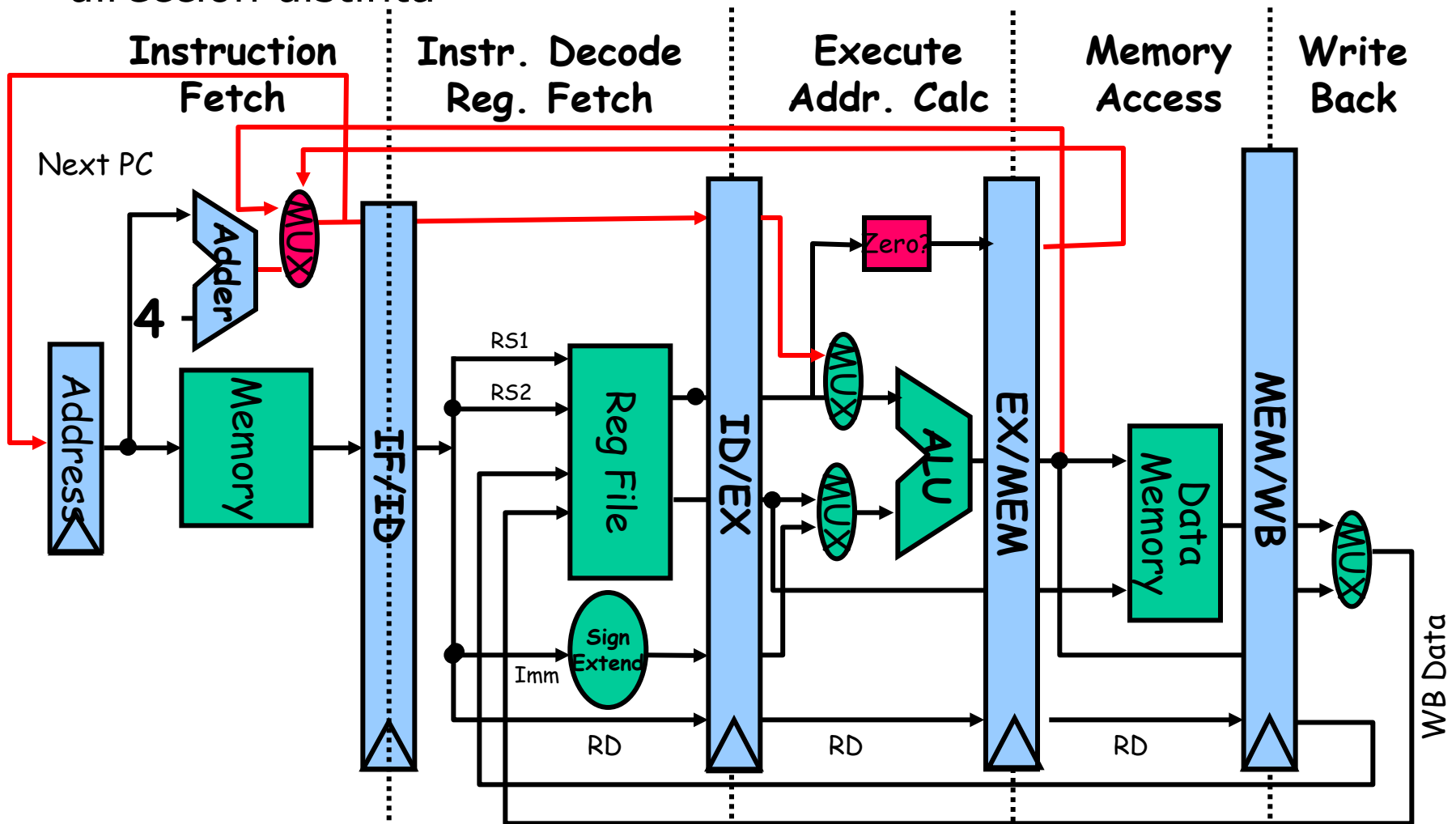
Ambas máquinas (ideal y con el riesgo) tienen ahora el mismo CPI (1) dado que no hay ciclos de parada. La diferencia está en las instrucciones NOP que se ejecutan en la máquina con el riesgo. En este caso, el número de instrucciones sería

$NI + 0.3 \times NI \times 0.5 \times 2 = 1.3 \times NI$. Dado que CPI y Tciclo son idénticos en ambas máquinas, la ideal es un 30% más rápida.

**La introducción de NOPs incrementa el número de inst.
¿Es lo mismo desde el punto de vista del rendimiento
insertar NOPs que ciclos de parada?**

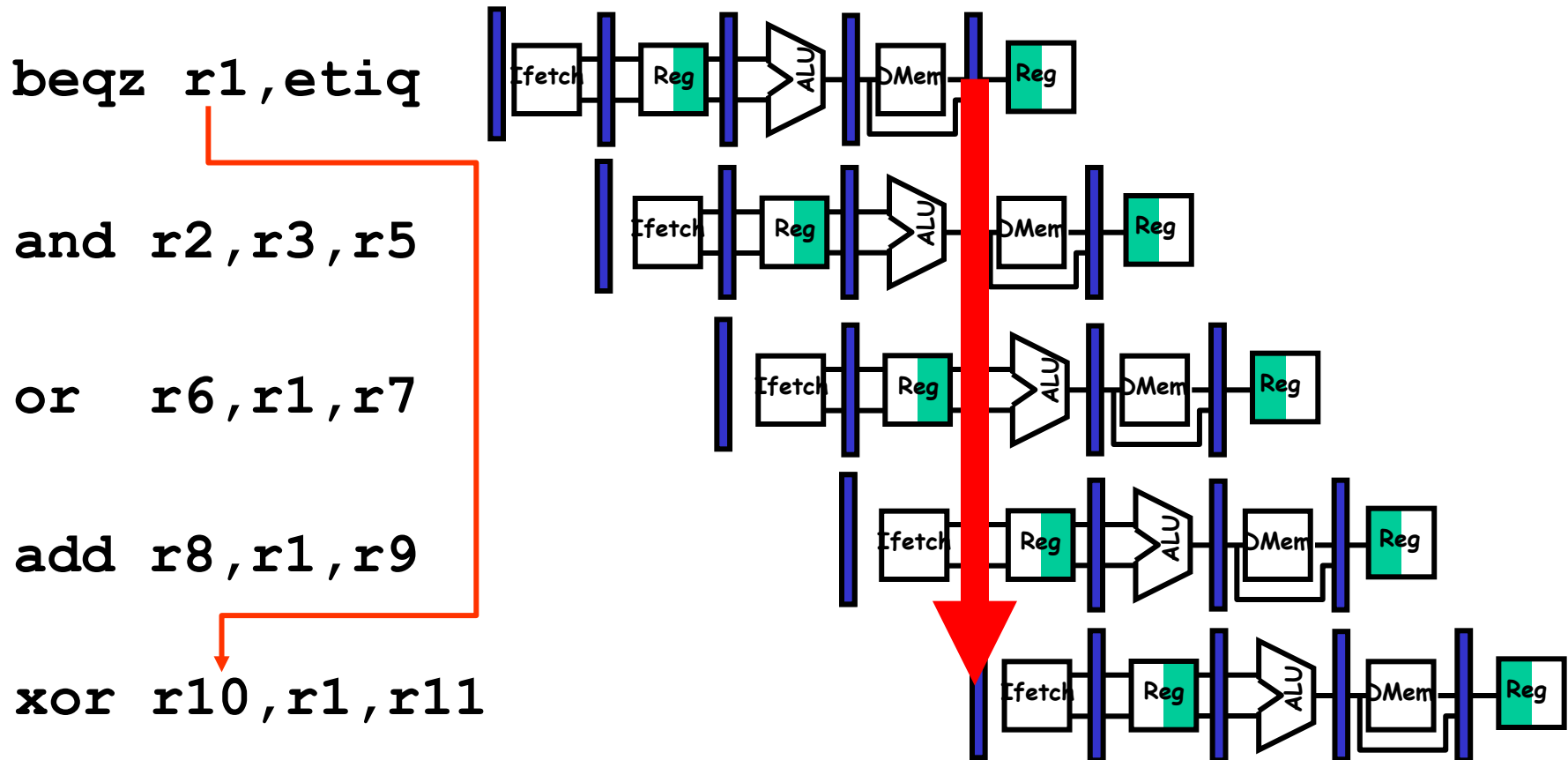
Riesgos de control

- Cuando se ejecuta una instrucción de salto, dependiendo de si se toma el salto o no, se cambiará el PC a PC+4 o a otra dirección distinta



Riesgos de control

- En nuestro DLX básico, este cambio del PC no se lleva a cabo hasta el final de la etapa MEM cuando se haya completado el calculo de la dirección y la comparación.



Riesgos de control: *detención del cauce*

- La solución HW más básica consiste en ampliar la unidad de detección de riesgos para que cuando se detecte que un salto (en ID) **se detenga el cauce** hasta saber si el salto es **T** o **NT**

Branch instruction	IF	ID	EX	MEM	WB				
Branch successor	IF	stall	stall	IF	ID	EX	MEM	WB	
Branch successor + 1					IF	ID	EX	MEM	WB
Branch successor + 2						IF	ID	EX	MEM
Branch successor + 3							IF	ID	EX
Branch successor + 4								IF	ID
Branch successor + 5									IF

- Como en el DLX básico el salto se resuelve en la etapa MEM será necesario introducir **3 ciclos de parada** para darle tiempo al salto a que llegue hasta dicha etapa.
- En cauces más profundos, en los que los saltos pasarían por más etapas hasta resolverse, habría más ciclos de parada y se degradaría de forma significativa el rendimiento.

Riesgos de control

- Los riesgos de control pueden causar una mayor pérdida de rendimiento que los riesgos de datos
 - Suponiendo que tuviéramos un 20% de instrucciones de salto y un CPI ideal de 1, la máquina con saltos tan sólo alcanzaría $0.8 \times 1 + 0.2 \times 4 = 1.6$, es decir, **!!! un 60% más lenta que la original !!!**
- La latencia de un salto puede reducirse en dos pasos:
 - Averiguando si el salto se va a tomar o no en una etapa anterior
 - Calculando antes la dirección efectiva del salto
- Mejora para DLX:
 - Mover la **unidad detectora** de ceros a **ID**
 - Calcular la **dirección de salto** sea cual sea el resultado de la comparación **en ID**. Esto obliga a añadir un sumador, ya que la ALU no está disponible en esta etapa
 - Con este hardware añadido podemos tomar la decisión del salto en la etapa ID, con lo que sólo tendríamos **un ciclo de parada** para los saltos

Riesgos de control

- El cauce DLX modificado para reducir la penalización de los saltos a **un único ciclo de parada** sería:

