TEMA 4. SEGMENTACIÓN

1.1. INTRODUCCIÓN

1.1.1. Concepto de segmentación

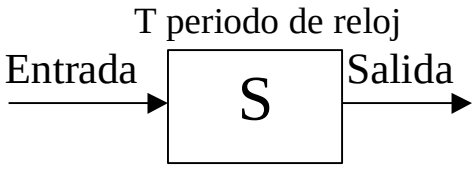
Estudiaremos en este tema los ***computadores segmentados***, el **procesamiento encauzado**, y los **pipelines**.

La idea de segmentación es **análoga a la de cadena de montaje industrial**. La ejecución de un proceso se **divide en etapas**, especializando cada elemento de la cadena en realizar una operación concreta.

El resultado es un **aumento de la productividad** aunque el tiempo de realización de una tarea aislada es el mismo.

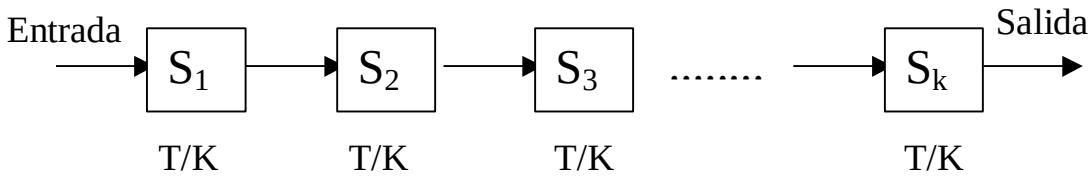
El pipeline **explota el paralelismo temporal**, ya que **aunque opera de forma serie** cuando nos referimos a una pieza determinada, lo hace en paralelo sobre las diferentes piezas en las sucesivas etapas.

Sin segmentación



E_k				I_{1k}			...	I_{2k}			...	I_{3k}
....			...	...	...	...	...		...	...	...	
E_2		I_{12}		...	...	I_{22}			...	I_{32}		
E_1	I_{11}			...	I_{21}				I_{31}			
	1	2	3	k			...					tiempo ciclos

Con segmentación



E_k				I_{1k}	I_{2k}	I_{3k}	I_{4k}				I_{nk}
....			...	...	...	...	...	...	...	...	...
E_2		I_{12}	I_{22}	I_{32}	I_{42}				I_{n2}		
E_1	I_{11}	I_{21}	I_{31}	I_{41}				I_{n1}			
	1	2	3	4			...	n			

tiempo
ciclos

La importancia de la descomposición equilibrada.

Un factor determinante es la **descomposición de la tarea** a realizar en subtareas para conseguir una **distribución uniforme del tiempo** ya que en caso contrario la etapa más lenta actúa como **cuello de botella**.

El **caso ideal** es aquel en el que el ciclo de reloj de la tarea se distribuye uniforme entre todas las subtareas. **Clk=T/K**.

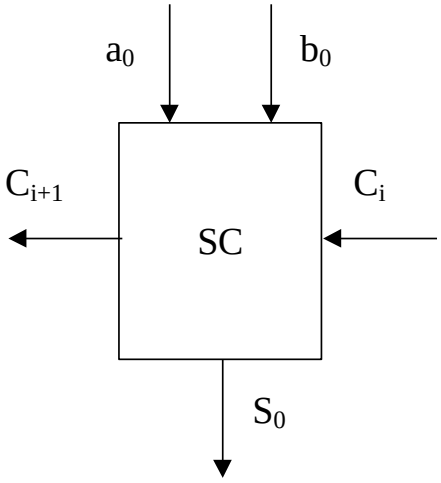
En la práctica si no es posible esta distribución equitativa del tiempo entre las subtareas, **se ajusta** el ciclo de reloj al de la **etapa más lenta.**

Existe por lo tanto un **control centralizado** con un **único pulso de reloj.**

Ejemplo de cauce aritmético:

Sumador secuencial **de tres bits con propagación de acarreo**, caso secuencial frente al segmentado.

Estructura del sumador completo

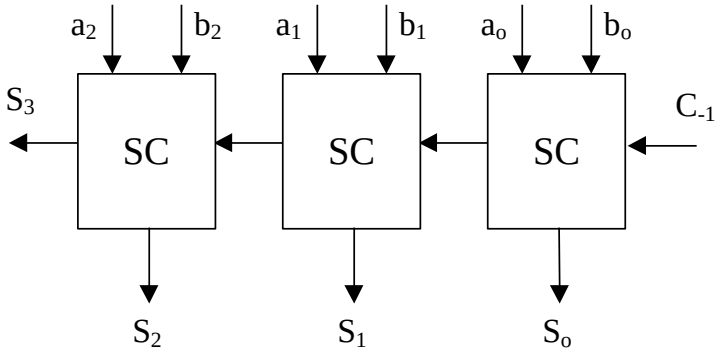


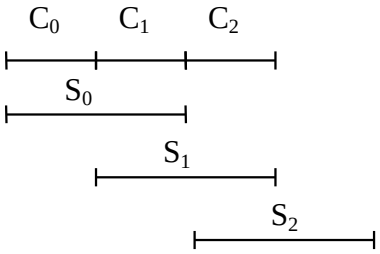
$$S = a \oplus b \oplus c$$

$$C_{i+1} = a \cdot b + a \cdot C_i + b \cdot C_i$$

T_c = Tiempo de acarreo
T_s = Tiempo de suma
$T_s > T_c \quad T_s \cong 2T_c$

Caso secuencial. Construimos el sumador a base de sumadores completos





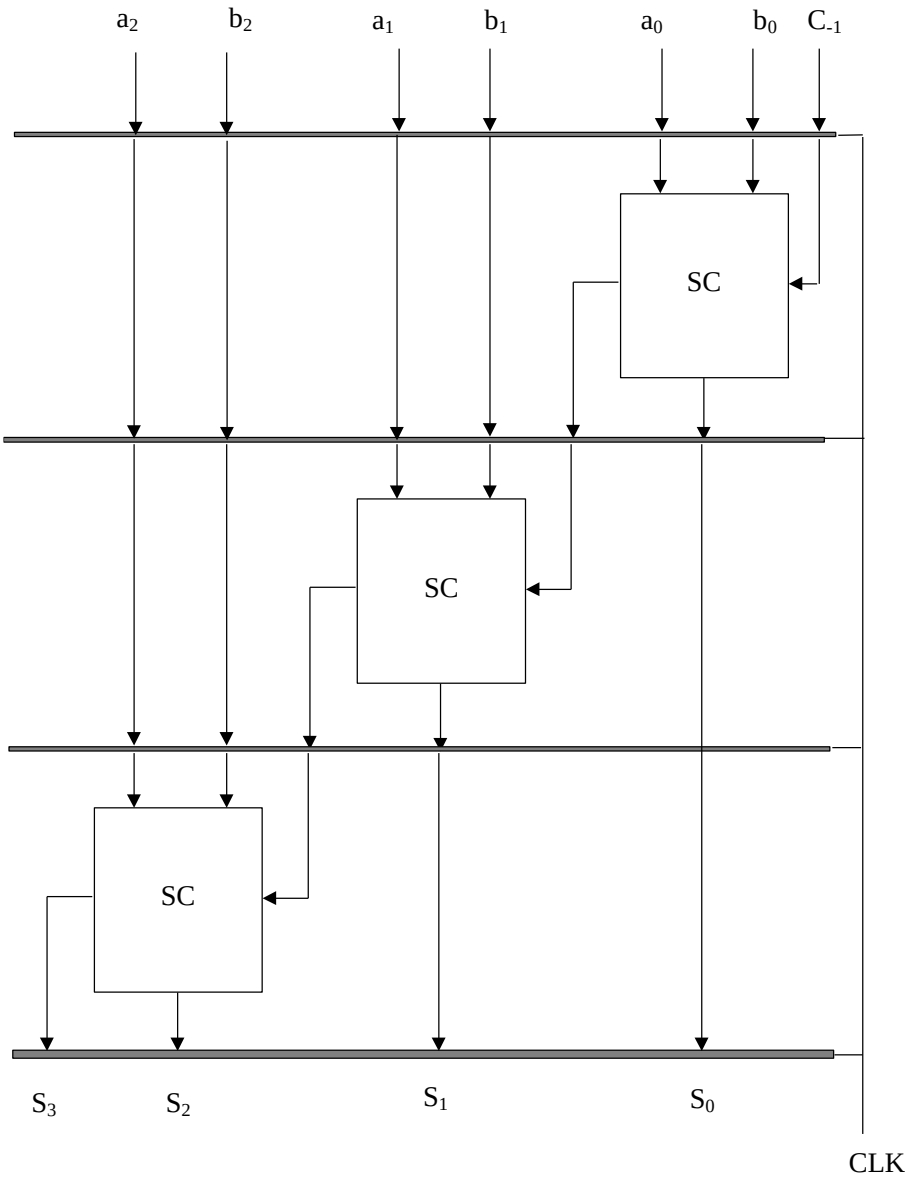
$$T_{suma} = 2T_c + T_s$$

Caso segmentado

En el circuito segmentado hacemos que **clk** sea el **tiempo de la etapa más lenta**.

El **coste temporal** para realizar una **suma aislada** es incluso **mayor**.

$$T_{suma} = 3\left(T_s + retardos\ registros\ \text{int}\ ermedios\right)$$



El **aumento de velocidad** aparece **cuando tenemos que procesar más de un dato**. Por **ejemplo** para sumar **100 números de tres bits**.

Tiempo del **secuencial**

$$T_{Secuencial} = 100(2T_c + T_s)$$

Tiempo del **segmentado**

La **primera suma tarda** $3T_s$ pero **una vez lleno el cauce** se obtiene **1 resultado cada pulso de reloj**.

$$T_{segmentado} = 3T_s + 99T_s = 102T_s$$

E3			I_{13}	I_{23}	I_{33}	I_{43}				I_{n3}
E2		I_{12}	I_{22}	I_{32}	I_{42}				I_{n2}	
E1	I_{11}	I_{21}	I_{31}	I_{41}				I_{n1}		
	1	2	3	4			...	100		

T_s tiempo ciclos

Para el estudio de unidades segmentadas es **frecuente** la **utilización** de **diagramas espacio-tiempo** como los mostrados. Nos dan el flujo de datos a través del tiempo y las diferentes unidades que se van atravesando.

Ventajas de la segmentación

- Bajo coste de implementación.
- Facilidad de control.
- Amplio espectro de aplicación.

1.1.2. Niveles de aplicación clasificación

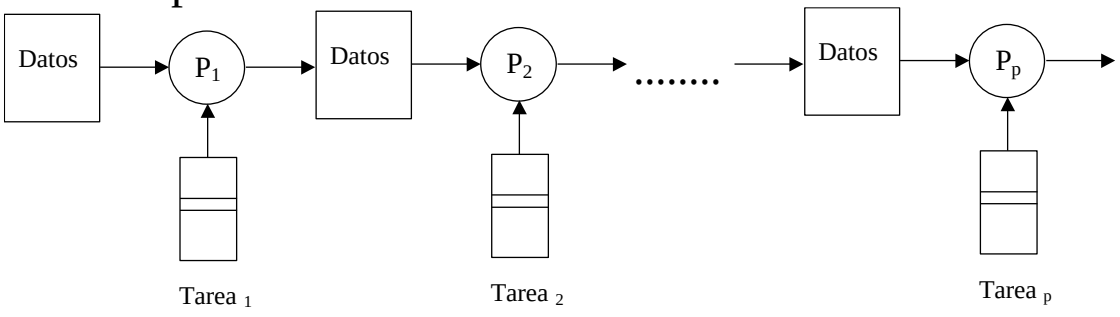
Tenemos **tres** posibles **niveles de aplicación**:

a) **Segmentación aritmética**: Se **descompone** cada **operación aritmética en distintas etapas** y se realiza un **diseño encauzado de la ALU**. Técnica empleada por la mayoría de los **computadores vectoriales** Cray-I (14 Etapas), Start-100 (4 etapas), TI-ASC (8 etapas).

b) **Segmentación de instrucciones**: La **instrucción** a ejecutar **se descompone en diferentes fases** comunes (búsqueda instrucción, decodificación, búsqueda de operando, ejecución, almacenamiento de resultado) **y se encauzan**, de manera que mientras estamos en fase de ejecución de una instrucción podemos realizar la decodificación de otra distinta de forma solapada. Se emplea **en los RISC para la UC**.

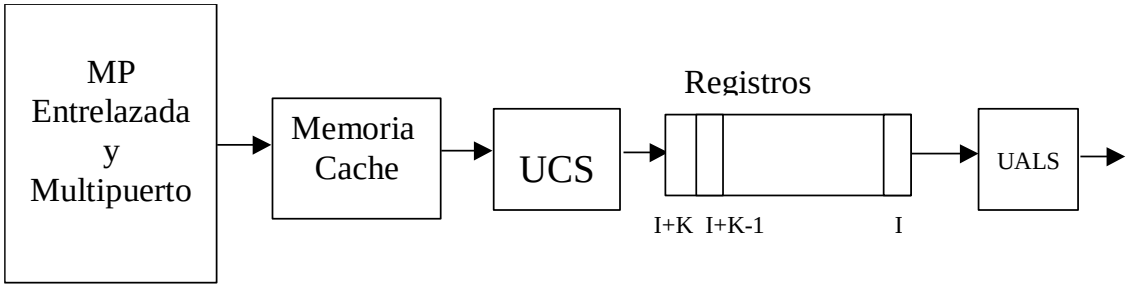
AR					AR	AR	AR	AR	AR
EJ				EJ	EJ	EJ	EJ	EJ	
BO			BO	BO	BO	BO	BO		
DI		DI	DI	DI	DI	DI			
BI	BI	BI	BI	BI	BI				
	I_1	I_2	I_3	I_4	I_5				

c) **Segmentación de procesadores**: Las etapas del cauce están formadas por distintos procesadores que realizan, distintas operaciones sobre el flujo de datos. Propio de la computación sistólica



Estos niveles de segmentación no son excluyentes sino que se complementan en un **computador completamente encauzado.**

- CPU con UC segmentada
- ALU segmentada



A su vez **esta CPU puede formar parte de un sistema multiprocesador** encauzado a nivel de procesadores.

Clasificación:

1. Atendiendo a la **funcionalidad**

   1.1. **Unifunción** (sumador)
   1.2. **Multifunción** (CPU segmentada)

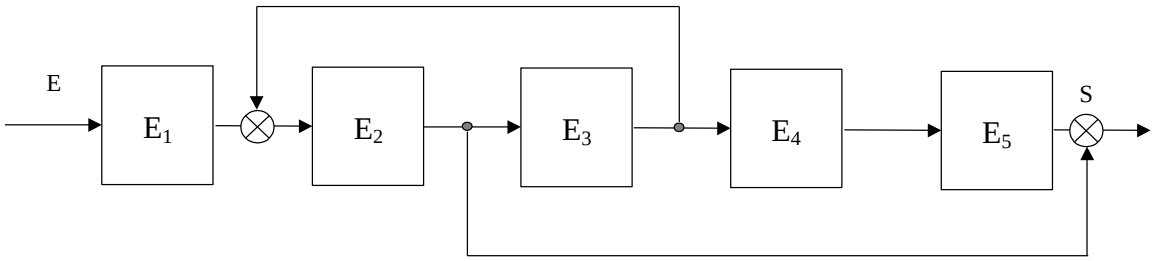
2. Según la **configuración del cauce y estrategia de control**

   2.1. **Estática** (siempre se configuran antes de iniciar el proceso).
   2.2. **Dinámica** (se permite cambiar su función en cualquier momento) (mecanismos de control complejos)

3. Según tenga **caminos de realimentación**

3.1. **Lineales**. T se descompone en tareas T_i de forma que si i<j la subtarea T_j no puede iniciarse hasta que T_i no haya terminado.

3.2. **No lineales**. Existe **realimentación**



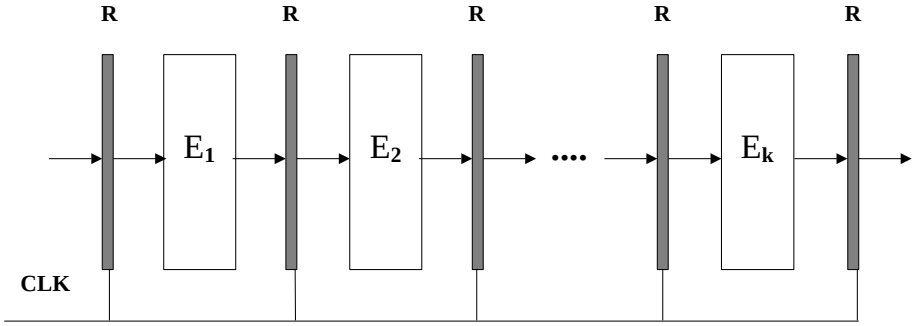
4. Según el **control del flujo de datos**

4.1. **Síncrono**. Una **única señal de control** que dirige el flujo de datos.

4.2. **Asíncrono**. No existe control centralizado, sino que cada etapa funciona de forma autónoma. Se plantea algún tipo de **protocolo para la comunicación** entre etapas (semáforos, paso de mensajes).

1.1.3. Análisis de prestaciones. (Unidad segmentada lineal)

Vamos a evaluar los **parámetros para el siguiente cauce de K etapa**s.



UNIDAD SEGMENTADA LINEAL DE K ETAPAS

R = **Registros** de almacenamiento intermedio Buffers.
E_i = **circuitos combinacionales** que desarrollan operaciones aritméticas o lógicas.
CLK = Señal de **reloj** que controla el flujo de datos.
t_i = **Retardo temporal** de cada etapa E_i
t_r = Es el **retardo** de cada **registro R.**

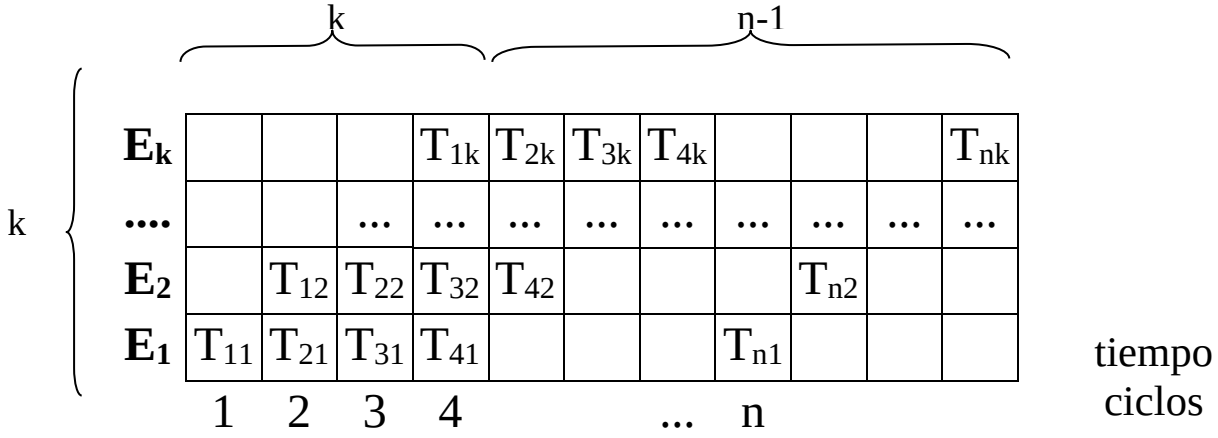
Definimos el **periodo de reloj del cauce** con k etapas como:

$$CLK = max\{t_i\}_{i=1}^{k} + t_r$$

En teoría el pulso CLK llega a todos los R simultáneamente, en la práctica hay un pequeño retardo (**clock skewing**) de forma que CLK llega a las etapas con un **offset s**.

$$CLK \geq max\{t_i\}_{i=1}^{k} + t_r + s$$

Diagrama espacio temporal del cauce:



Tiempo necesario **para procesar n tareas** en el cauce lineal de k etapas.

$$T_{SEG} = \underbrace{k \cdot CLK}_{\text{Tiempo para procesar la primera T. Llenado del cauce}} + \underbrace{(n-1)CLK}_{\text{Tiempo para procesar las n-1 T restantes. A razón de una tarea por pulso}}$$

Tiempo equivalente para un **computador no encauzado**:

$$T_{SEC} = \underbrace{K \cdot CLK}_{\text{Tiempo para procesar una Tarea.}} \cdot n$$

Ganancia de velocidad (Speed-up): **Incremento de velocidad** de un cauce de k etapas respecto del proceso secuencial.

$$G_k = \frac{T_{SEC}}{T_{SEG}} = \frac{n \cdot k \cdot CLK}{(k+n-1)CLK} = \frac{n \cdot k}{k+n-1}$$

Cuando $n \to \infty$ G_k crece hasta alcanzar un máximo teórico:

$$\lim_{n\to\infty} G_k = K$$

En la práctica diversos condicionantes hacen que $G_k << k$:

- **Imposibilidad de la descomposición óptima** de la tarea en k subtareas.
- **Dependencias entre datos** e instrucciones.
- **Bifurcaciones** en el programa.

Eficiencia (Eficiency): Es la **relación entre el número de tramos temporales ocupados y el número total de tramos** en dicho periodo de tiempo T. Nos ofrece idea del **grado de utilización del Hardware**.

$$E_k = \frac{k \cdot n \cdot CLK}{k(k+n-1)CLK} = \frac{n}{k+n-1} = \frac{G_k}{k}$$

Relación entre la G real y la ideal

Cuando $n \to \infty$, la eficiencia crece hacia su límite teórico de 1 (100%).

$$\lim_{n\to\infty} E_k = 1$$

Productividad: **Número de datos** o **instrucciones que puede procesar por unidad de tiempo.**

$$P_k = \frac{n}{(k+n-1)CLK} = \frac{E_k}{CLK}$$

La **cota máxima** coincide con la **frecuencia de funcionamiento** y corresponde a la **aparición de un resultado cada periodo de reloj**:

$$\lim_{n\to\infty} P_k = \frac{1}{CLK}$$

Ejemplo de unidad aritmética segmentada: Sumador en coma flotante.

$$\left.\begin{aligned} A &= a \times 2^{p} \\ B &= b \times 2^{q} \end{aligned}\right\}$$ a,b mantisas normalizadas

p,q exponentes

$$C = A + B = C \times 2^{r} = \underbrace{d \times 2^{s}}_{\text{normalizado}}$$

$$r = max(p, q)$$

Paso 1. Comparación exponentes para detectar el mayor

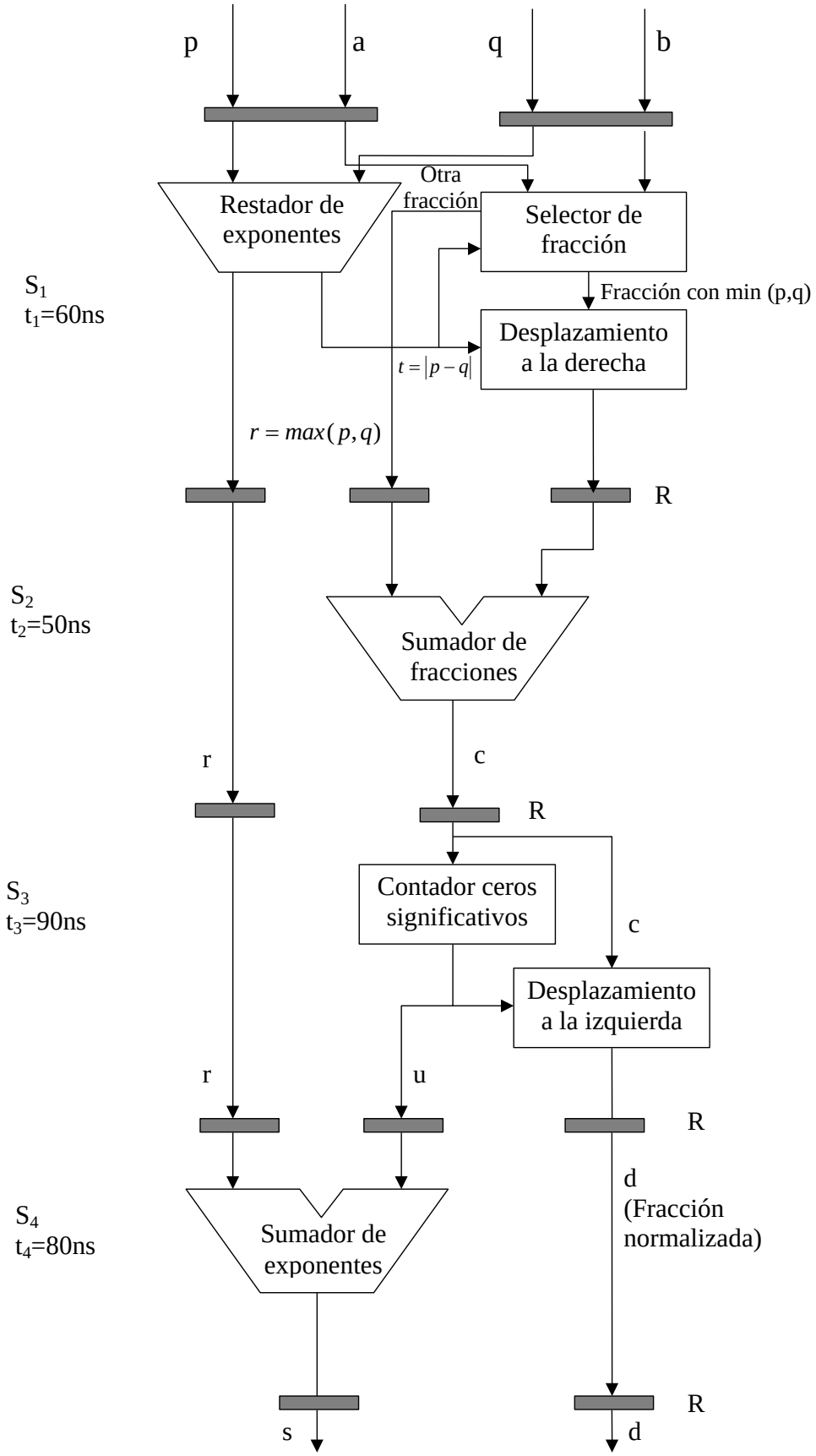
$$r = max(p, q) \qquad t = |p - q|$$

Paso 2. Desplazamiento de t bits a la derecha de la fracción de menor exponente. Igualar exponente.

Paso 3. Suma de las mantisas para obtener la fracción suma intermedia C.

Paso 4. Normalización de la mantisa. Desplazamiento u bits a la izquierda (para que el bit más significativo sea 1).
Normalización del exponente

$$s = r - u$$


p
a
q
b
Otra fracción
Restador de exponentes
Selector de fracción
Fracción con min (p,q)
S1
t1=60ns
Desplazamiento a la derecha
t = |p − q|
r = max(p, q)
R
S2
t2=50ns
Sumador de fracciones
r
c
R
S3
t3=90ns
Contador ceros significativos
c
Desplazamiento a la izquierda
r
u
R
d
(Fracción normalizada)
S4
t4=80ns
Sumador de exponentes
R
s
d

A partir del sumador en coma flotante encauzado del ejemplo anterior, calcula la frecuencia del cauce, la ganancia, la eficiencia, y la productividad, suponiendo que van a ser procesadas 100 sumas.

Clk=90+10=100ns

$$f = \frac{1}{Clk} = \frac{1}{100ns} = \frac{1}{100 \cdot 10^{-9}} = 10MHz$$

n=100

k=4

$$G_k = \frac{n \cdot k}{k + n - 1} = \frac{100 \cdot 4}{4 + 100 - 1} = 3{,}883$$

$$E_k = \frac{G_k}{k} = \frac{n}{k + n - 1} = 0{,}97$$

$$P_k = \frac{E_k}{Clk} = \frac{0{,}97}{100 \cdot 10^{-9}} = 9{,}7MOPS$$

1.2. SEGMENTACIÓN DE CPUs

La **mayoría de las CPUs** en la actualidad utilizan **técnicas** de **segmentación** para **aumentar la productividad**.

Vamos a **utilizar la arquitectura DLX** como caso de estudio para analizar los principios de la segmentación de CPUs.

1.2.1. Una implementación simple de DLX

Los **cinco ciclos de reloj** de las instrucciones DLX

1. Ciclo de búsqueda de instrucción

$IR \leftarrow MEM\ [PC]$

$NPC \leftarrow PC + 4$

Operación: Transfiere el PC y ubica la instrucción de memoria en el registro de instrucciones. Incrementa el PC en 4 para apuntar la siguiente instrucción de la secuencia.

2. Ciclo de decodificación de la instrucción/carga de registros

$A \leftarrow Regs\ [IR_{6..10}]$

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

$Inm \leftarrow ((IR_{16})^{16}$ ## $IR_{16..31})$

Operación: Decodifica la instrucción y accede al banco de registro para leer los registros. Las salidas de los dos registros de propósito general se cargan en dos registros temporales (A y B) para su uso posterior. Se extiende el signo a los 16 bits bajos del IR y se almacena el resultado en el registro temporal Inm para uso posterior.

La **decodificación** puede hacerse **en paralelo con** la **lectura de los registros**, lo que es posible porque estos campos ocupan posiciones fijas en el formato de instrucción DLX. Esta técnica se conoce como **decodificación de campo fijo**. Observa que podemos leer un registro que no se usa, lo que no ayuda pero tampoco perjudica. Como el inmediato se sitúa en posiciones idénticas en todos los formatos de instrucción DLX, el inmediato de signo extendido se calcula también en este ciclo en caso de que se requiera en el siguiente ciclo.

3. Ciclo de ejecución / dirección efectiva

La **ALU opera** sobre los operandos preparados en el paso anterior, **realizando una de cuatro funciones**, dependiendo del tipo de instrucción DLX.

Referencia a memoria

ALUoutput ← A + Inm

Operación: La ALU suma los operandos para formar la dirección efectiva.

Instrucción ALU registro-registro

ALUoutput ← A func B

Operación: La ALU realiza la operación especificada por el código de operación sobre el valor de A y sobre el valor de B. El resultado se coloca en el registro temporal ALUoutput.

Instrucción ALU registro-inmediato

ALUoutput ← A op Inm

Operación: La ALU realiza la operación especificada por el código de operación sobre el valor de A y sobre el valor del registro Inm. El resultado se coloca en el registro temporal ALUoutput.

Salto/ bifurcación

ALUoutput ← NPC + Inm

cond ← (A op 0)

Operación: La ALU suma el NPC al valor inmediato de signo extendido para calcular la dirección destino del salto. El registro A, que ha sido leído en el ciclo anterior, se examina para decidir si se realiza el salto o no. La operación de comparación op es el operador relacional determinado por el código de operación, por ejemplo, op es == para la instrucción BEQZ.

La **arquitectura de carga almacenamiento** de DLX significa que la dirección efectiva y **los pasos de ejecución se pueden combinar en un solo paso**, ya que **ninguna instrucción necesita calcular una dirección y**, además, **realizar una operación sobre los datos**.

No hemos contemplado los saltos de tipo incondicional ya que son similares a los condicionales.

4. Paso de acceso a memoria / completar salto:

Las únicas instrucciones DLX activas en este paso son cargas, almacenamientos y saltos.

Referencia a memoria

LMD ← MEM [ALUoutput] o

MEM [ALUoutput] ← B

Operación: Accede a memoria si es necesario. Si la instrucción es una carga, devuelve el dato desde memoria y se carga en LMD (load memory data); si es un almacenamiento, entonces escribe el dato del registro B en memoria. En cualquier caso la dirección utilizada es la calculada durante el paso anterior y almacenada en el registro ALUoutput.

Salto

If (cond) PC ← ALUoutput else PC ← NPC

Operación: Si la instrucción salta, el PC es sustituido por la dirección destino del salto del registro ALUoutput, en caso contrario, se reemplaza con el contador de programa incrementado en el registro NPC.

5. Paso de postescritura (write-back)

Instrucciones ALU registro-registro:
Regs [$IR_{16..20}$] ← ALUoutput

Instrucciones ALU registro-inmediato:
Regs [$IR_{11..15}$] ← ALUoutput

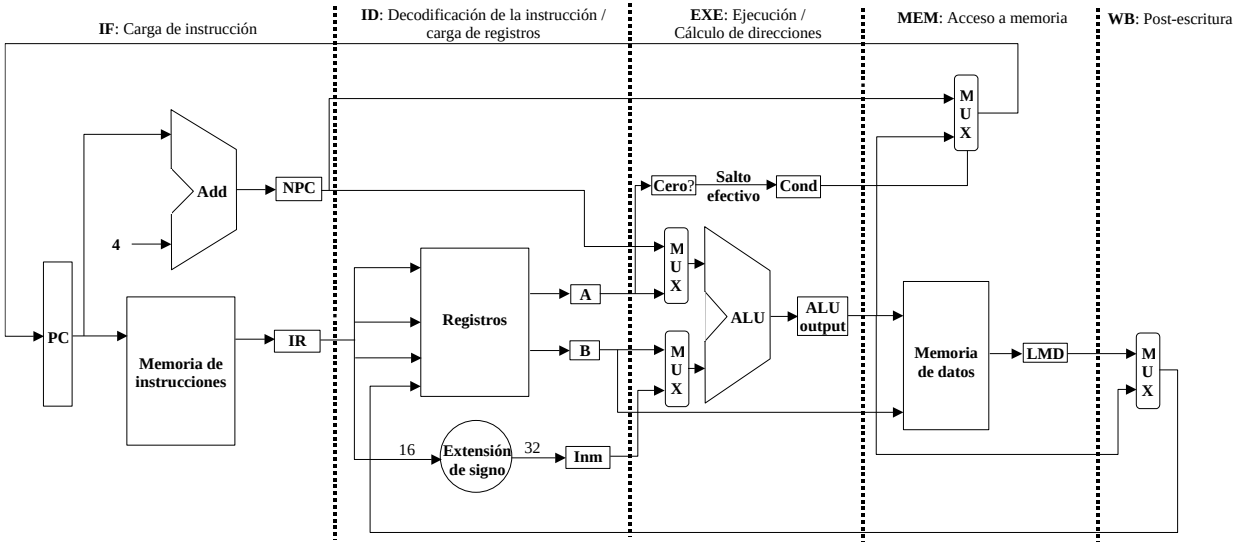
Instrucciones load
Regs [$IR_{11..15}$] ← LMD

Operación: Escribir el resultado en el fichero de registros tanto si proviene del sistema de memoria (en LMD) como de la ALU (en ALUoutput). El campo del registro destino puede estar en dos posiciones dependiendo del código de operación.

La figura muestra como evoluciona una instrucción en el camino de datos.

Al final de cada ciclo de reloj, todos **los valores computados** durante ese ciclo y requeridos en un ciclo posterior, s**e escriben en un dispositivo de almacenamiento**, que puede ser memoria, un registro de propósito general, el PC o un registro temporal (LMD, Inm, A, B, IR, NPC, ALUoutput o Cond).

Los **registros temporales mantienen valores entre ciclos** para una misma instrucción, mientras que **los otros elementos de almacenamiento mantienen valores entre instrucciones sucesivas**.



La implementación del camino de datos DLX permite que **cualquier instrucción** se ejecute **en cuatro o cinco ciclos**.

El PC se observa en la parte del camino de datos en que se usa **durante la carga de la instrucción** y **los registros se muestran** en la parte del camino de datos donde se utilizan **en la decodificación/carga** de registros.

Aunque observamos estas unidades funcionales en el ciclo de reloj en el que son leídas, **el PC se escribe durante** el ciclo de **acceso a memoria** y **los registros se escriben durante el ciclo de post-escritura**. En ambos casos **las escrituras en etapas posteriores se indican mediante multiplexores en las salidas** (en acceso a memoria y post-escritura) que retornan resultados al PC o los registros.

Estas señales que fluyen hacia atrás introducen gran parte de la complejidad de la segmentación y las estudiaremos más detenidamente.
En esta implementación**, las instrucciones de salto y almacenamiento requieren 4 ciclos** y las demás cinco.

1.2.2. Segmentación básica para DLX

Podemos **segmentar el camino de datos** sin casi ningún cambio, **sin más que iniciar una nueva instrucción en cada ciclo de reloj**.

Cada uno de los **ciclos de reloj** estudiados **se convierte en una etapa del cauce**. Esto **conduce al patrón de ejecución típico de un cauce** que se muestra en la figura.

Número de instrucción	**1**	**2**	**3**	**4**	**Ciclo 5**	**Reloj 6**	**7**	**8**	**9**
Inst i	IF	ID	EX	MEM	WB				
Inst i+1		IF	ID	EX	MEM	WB			
Inst i+2			IF	ID	EX	MEM	WB		
Inst i+3				IF	***ID***	EX	MEM	WB	
Inst i+4					IF	ID	EX	MEM	WB

Durante un ciclo de reloj **el hardware ejecuta alguna parte de cinco instrucciones diferentes**.

Si comienza una instrucción nueva cada ciclo de reloj **el rendimiento mejora hasta cinco veces con respecto a una máquina no encauzada**.

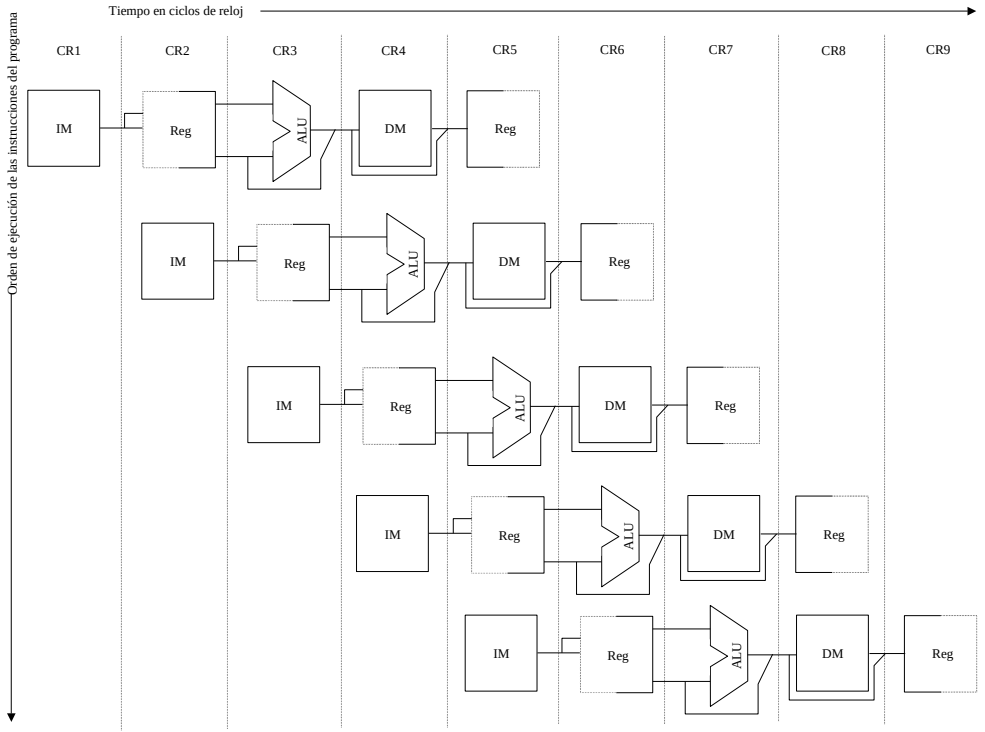
La segmentación no es tan sencilla como esto, el hecho de iniciar una nueva instrucción cada ciclo **introduce problemas que trataremos**.

Para empezar, tenemos que determinar que ocurre en cada ciclo de reloj de la máquina y **asegurarnos que no estamos intentando realizar dos operaciones diferentes con el mismo recurso** del camino de datos en el mismo ciclo de reloj.

> Por **ejemplo**, a **una simple ALU no** se le puede pedir que **realice el cálculo de una dirección efectiva y una operación de resta al mismo tiempo**. Por lo tanto, debemos asegurarnos que el solapamiento de las instrucciones en el cauce no está causando este tipo de conflictos.

Evaluación de conflictos entre recursos para el cauce DLX

La simplicidad del conjunto **de** instrucciones **DLX hace la evaluación de los recursos** relativamente **sencilla**.
En la figura **vemos** una **versión simplificada del camino de datos DLX** dibujada **en forma de cauce**.
Las principales unidades funcionales se usan en diferentes ciclos de reloj y por lo tanto **solapar la ejecución de múltiples instrucciones introduce pocos conflictos**.



El cauce puede entenderse como una **serie de caminos de datos solapados en el tiempo**. **Esto muestra los solapamientos** entre las partes del camino de datos.
Como el banco de registros se usa como una fuente en la etapa ID y como un destino en la etapa WB aparece repetida. Se observa que **la lectura se produce en la primera parte del ciclo y la escritura en la segunda**.
Por otro lado **se observan memorias de instrucciones y datos** (**IM** = memoria de instrucciones, **DM** = memoria de datos).

Observaciones

1. **La memoria:**
   El camino de datos básico propuesto utiliza **memorias de datos e instrucciones separadas**, **que podemos implementar con caches separadas de datos e instrucciones**.

   Esto elimina el conflicto creado por el uso de una sola memoria entre los **accesos para cargar una instrucción y los accesos a datos**.

   Observar que si nuestra máquina encauzada tiene un ciclo de reloj como el de la máquina no encauzada, **el sistema de memoria debe proporcionar un ancho de banda cinco veces mayor**.
   Máquina no encuazada (2 accesos cada cinco ciclos)
   Máquina encauzada (2 accesos cada ciclo)

   Este **es uno de los costes del alto rendimiento**.

2. **El banco de registros**:
   El banco de registros **se usa en dos etapas**:
   Para **leer en ID**
   Para **escribir en WB**

   Tenemos que realizar dos lecturas y una escritura en cada ciclo de reloj.

   ¿**Qué ocurre si una lectura y una escritura son sobre el mismo registro**?. Dejaremos este punto para **más tarde.**

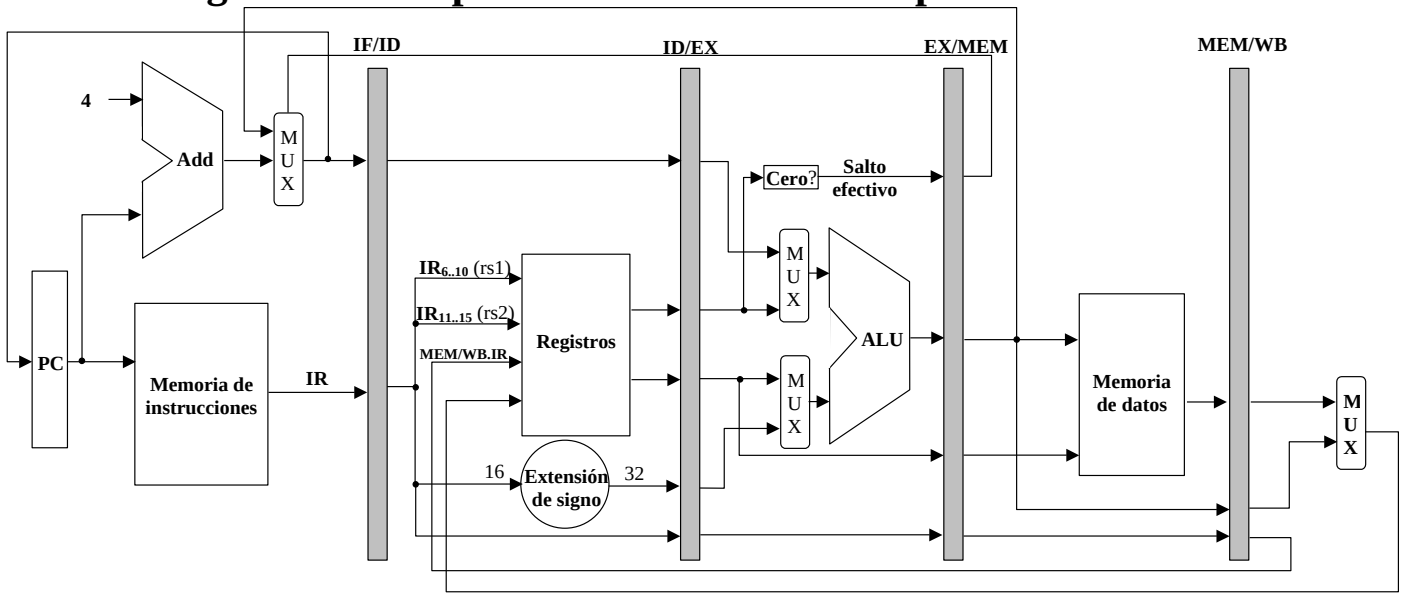
3. **El contador de programa:**
   La figura anterior no incluye el contador de programa **PC**.

   Para iniciar una nueva instrucción cada ciclo de reloj, **debemos incrementar** y guardar **el PC en cada ciclo**, y esto debe hacerse **durante la etapa IF para prepararlo para la siguiente instrucción**.

   El **problema** surge **cuando consideramos el efecto de los saltos**, **que** también **cambian el PC**, pero no lo hacen hasta **la etapa MEM**. Esto no era un problema en nuestro camino de datos multiciclo no segmentado, ya que el PC se escribe una vez en MEM.
   En el caso encauzado, organizaremos el camino de datos para e**scribir el PC en IF bien con el PC incrementado en 4 o el valor del destino del salto de una instrucción de salto ejecutada previamente**. Esto introduce la problemática de cómo debemos manejar los saltos, que trataremos también posteriormente.

Todas las operaciones de una etapa del cauce deben terminar en un ciclo de reloj y cualquier combinación de operaciones debe poder ocurrir en algún momento.
Además, **segmentar el camino** de datos **requiere** que los valores pasados de una etapa a la siguiente se coloquen en **registros intermedios**.

El cauce DLX con los registros correspondientes entre cada etapa.



Los registros intermedios

Los registros intermedios sirven para **transportar valores e información de control** desde una etapa a la siguiente.

Todos los registros necesarios para almacenar valores temporales entre ciclos de reloj en una instrucción, **se engloban dentro** de estos **registros intermedios**. Cualquier valor necesitado en una etapa posterior debe ser colocado en un registro y copiado desde un registro intermedio hasta el siguiente, hasta que no se necesite definitivamente.

> **Si intentamos usar sólo los registros temporales** q**ue teníamos en nuestro camino de datos no segmentado, se pueden sobrescribir valores antes de haberlos usado** por completo.
> **Ejemplo**, el **campo de IR** que **apunta** al **operando registro destino** de una operación load o ALU **se proporciona desde el registro intermedio MEM/WB en lugar de obtenerlo de IF/ID**. Esto es así, porque **nos interesa que** las **operaciones** load o ALU **escriban en el registro designado por esa instrucción que no coincide con la que evoluciona desde IF a ID en ese instante**. Este campo que apunta al registro destino se copia de un registro intermedio a otro hasta que se necesita durante la etapa WB.

Los multiplexores

Para controlar este camino de datos nos queda por determinar como fijar el **control de los multiplexores**.

Los dos multiplexores de la ALU:

Se fijan dependiendo del tipo de instrucción, que podemos encontrar en el campo IR del registro intermedio ID/EX.

El **multiplexor alto** de la ALU se fija **según la instrucción sea un salto o no**.
El **multiplexor bajo** se fija **según la instrucción sea ALU registro-registro o cualquier otro tipo de operación.**

El multiplexor en la etapa IF

El multiplexor que selecciona el valor que será **escrito en el PC se ha movido a la etapa IF**, de forma que **el PC sólo puede ser escrito durante la etapa IF**.

Selecciona el uso del **PC en curso o** el **valor de EX/MEM.ALUoutput** como dirección de instrucción. Este multiplexor **se controla mediante** el campo **EX/MEM.cond.**

El cuarto multiplexor

Se controla según la instrucción en la etapa WB sea una operación **ALU** o una **load**. Es necesario un multiplexor más para controlar si el campo de IR que expresa el registro destino está en una u otra posición según la instrucción sea ALU registro-registro frente a ALU inmediato o load.

Las dos primeras etapas

Las acciones de **las dos primeras etapas** son **independientes** del tipo de instrucción en curso, **esto debe ser así porque la instrucción no es decodificada hasta el final de la etapa ID**.

La actividad de la etapa **IF depende de que la instrucción en EX/MEM sea un salto efectivo**.

> **Si es así**, **el destino del salto se utiliza para cargar la instrucción** y computar el siguiente PC.
> **En caso contrario**, **el PC actual se utiliza para estas acciones**.

El flujo del camino de datos

La mayor parte del flujo del camino de datos es **de izquierda a derecha**.

Los movimientos de **derecha a izquierda**, que llevan la información para **post-escritura** del registro **y** la **información del PC en un salto**, **introducen complicaciones** que estudiaremos.

La codificación de campo fijo de los registros fuente es fundamental para permitir que los registros se carguen durante ID.

Etapa	Cualquier instrucción		
IF	**IF/ID.IR** ← Mem[PC]; **IF/ID.NPC, PC** ← (if EX/MEM.cond {EX/MEM.ALUoutput} else {PC+4})		
ID	**ID/EX.A** ← Regs [IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs [IF/ID.IR$_{11..15}$]; **ID/EX.NPC** ← IF/ID.NPC; **ID/EX.IR** ← IF/ID.IR **ID/EX.Inm** ← (IF/ID.IR$_{16}$)^{16} ## IF/ID.IR$_{16..31}$		
	Instrucción ALU	**Instrucción load o store**	**Instrucción de salto**
EX	**EX/MEM.IR** ← ID/EX.IR; **EX/MEM.ALUoutput** ← ID/EX.A func ID/EX.B; O **EX/MEM.ALUoutput** ← ID/EX.A op ID/EX.Inm; **EX/MEM.Cond** ← 0;	**EX/MEM.IR** ← ID/EX.IR; **EX/MEM.ALUoutput** ← ID/EX.A + ID/EX.Inm; **EX/MEM.cond** ← 0; **EX/MEM.B** ← ID/EX.B;	**EX/MEM.ALUoutput** ← ID/EX.NPC + ID/EX.Inm; **EX/MEM.cond** ← (ID/EX.A op 0);
MEM	**MEM/WB.IR**←EX/MEM.IR; **MEM/WB.ALUoutput** ← EX/MEM.ALUoutput;	**MEM/WB.IR**←EX/MEM.IR; **MEM/WB.LMD** ← Mem [EX/MEM.ALUoutput]; O **Mem[EX/MEM.ALUoutput]**← EX/MEM.B,	
WB	**Regs[MEM/WB.IR$_{16..20}$]** ← MEM/WB.ALUoutput; O **Regs[MEM/WB.IR$_{11..15}$]** ← MEM/WB.ALUoutput;	**Regs[MEM/WB.IR$_{11..15}$]** ← MEM/WB.LMD;	

IF: Además de **cargar la instrucción y calcular el nuevo PC**, **guardamos** el PC incrementado tanto en el PC como en como en el registro intermedio **NPC**, **para** posterior uso en **el computo del destino del salto**.

ID: Cargamos los registros, **extendemos el signo de los 16 bits bajos de IR** y **pasamos el IR y el NPC**.

EX: Realizamos una operación ALU o el **cálculo de una dirección**. **Pasamos el IR y el registro B** (si la instrucción es store). Además **ponemos el valor de cond a 1 si la instrucción es un salto efectivo.**

MEM: Accedemos a memoria. **Escribimos el PC si es necesario** y **pasamos los valores necesarios a la última etapa**.

WB: Actualizamos el banco de registros bien **con el valor de ALUoutput o** con el **valor cargado de memoria**.

P**or simplicidad siempre pasamos el registro IR entero de una etapa a otra**, si bien a medida que evolucionamos en el cauce se necesita menos información de IR.

Rendimiento segmentado

La segmentación **incrementa la productividad** de las instrucciones de la CPU (número de instrucciones completas por unidad de tiempo) p**ero no reduce el tiempo de ejecución de una instrucción individual**.
De hecho, **normalmente** se ve **incrementado** el **tiempo de ejecución de instrucciones individuales debido a la sobrecarga en el control de la segmentación**.
El incremento en la productividad de instrucciones **significa** que los **programas se ejecutan más rápido** y tienen menor tiempo total de ejecución, incluso con tiempos de ejecución de instrucciones individuales superiores.

Ejemplo. Considerando la máquina no segmentada de la sección previa. Supongamos que tiene un ciclo de reloj de 10ns y que usa cuatro ciclos para las operaciones ALU y saltos y cinco ciclos para las operaciones de memoria. Supongamos que las frecuencias relativas para estas operaciones son 40%,20% y 40% respectivamente. Suponemos que segmentar la máquina añade 1ns de recarga al ciclo de reloj. Ignorando cualquier impacto sobre la latencia, ¿Qué ganancia en el índice de ejecución de instrucciones se obtiene de la segmentación?

El tiempo de ejecución medio por instrucción de la máquina sin segmentación es:
Tiempo de ejecución medio por instrucción = Ciclo de reloj x CPI medio
= 10 ns x ((0,4+0,2)x4 + 0,4x5) = 10 ns x 4,4 = 44 ns.

En la implementación segmentada, el ciclo de reloj debe ir a la velocidad de la etapa más lenta más sobrecargas, que puede ser 10+1 = 11 ns; este es el tiempo de ejecución medio por instrucción. Por eso, la ganancia para la segmentación es:

$$G_s = \frac{tiempo\ medio\ instrucción\ \text{sin}\ segmentación}{tiempo\ medio\ instrucción\ con\ segmentación} = \frac{44ns}{11ns} = 4$$

Ejemplo. Supongamos que los tiempos requeridos para las cinco unidades funcionales, que operan en cada uno de los cinco ciclos son: 10ns, 8ns, 10ns, 10ns y 7 ns. Suponemos que la segmentación añade 1ns de sobrecarga. Buscar la ganancia frente a la versión monociclo.
Puesto que la máquina no segmentada ejecuta todas las instrucciones en un único ciclo de reloj, su tiempo medio por instrucción es simplemente el tiempo del ciclo de reloj. El tiempo del ciclo de reloj es igual a la suma de los tiempos para cada paso de la ejecución.
Tiempo de ejecución medio por instrucción = 10 + 8 + 10 + 10 + 7 = 45 ns
El tiempo del ciclo de reloj en la máquina segmentada debe ser el tiempo más largo de cada una de las etapas (10ns) más la sobrecarga de 1 ns, sumando un total de 11ns. Como el CPI es 1, esto conduce a un tiempo de ejecución medio por instrucción de 11ns.

$$G_s = \frac{tiempo\ medio\ instrucción\ \text{sin}\ segmentación}{tiempo\ medio\ instrucción\ con\ segmentación} = \frac{45ns}{11ns} = 4.1$$

La segmentación puede entenderse como una mejora del CPI, que es lo que típicamente entendemos o como una reducción del ciclo de reloj.

1.2.3. Riesgos de la segmentación

Hay situaciones de riesgo que impiden que se ejecuta la siguiente instrucción del flujo, **estos riesgos reducen la ganancia**.

Tres clases de riesgos

1. Riesgos estructurales

Surgen de conflictos de los recursos cuando **el hardware no puede soportar todas las combinaciones de instrucciones en ejecución**.

2. Riesgos por dependencias de datos

Surgen cuando **una instrucción depende de los resultados de una instrucción anterior.**

3. Riesgos de control.

Surgen de la segmentación de los saltos y otras instrucciones que cambian el PC.

Los riesgos de la segmentación **pueden hacer necesario detenerla**.

Cuando una instrucción se detiene las instrucciones posteriores a esta también lo hacen (**además no se buscan instrucciones nuevas**), **las anteriores pueden continuar**.

Una detención disminuye la ganancia con respecto a la ideal.

Rendimiento de la segmentación con detenciones

$$G_s = \frac{tiempo\ medio\ instrucción\ sin\ segmentación}{tiempo\ medio\ instrucción\ con\ segmentación} =$$
$$= \frac{CPI\ sin\ segmentación \cdot Ciclo\ de\ reloj\ sin\ segmentación}{CPI\ con\ segmentación \cdot Ciclo\ de\ reloj\ con\ segmentación} =$$
$$= \frac{Ciclo\ de\ reloj\ \text{sin}\ segmentación}{Ciclo\ de\ reloj\ con\ segmentación} \times \frac{CPI\ \text{sin}\ segmentación}{CPI\ con\ segmentación}$$

La **segmentación puede entenderse** como una **mejora del CPI** o como **una reducción del ciclo de reloj.**

Segmentación como una mejora del CPI

Vamos a **suponer que el CPI ideal de un cauce es 1**, de forma que **podemos calcular el CPI de una cauce:**

$$CPI\ con\ segmentación = CPI_{ideal} + Ciclos\ de\ reloj\ de\ \text{det}ención\ de\ la\ segmentación\ por\ instrución$$
$$= 1 + Ciclos\ de\ reloj\ de\ \text{det}ención\ de\ la\ segmentación\ por\ instrución$$

Ignorando el incremento potencial en el ciclo de reloj debido a la segmentación, y asumiendo que las etapas están equilibradas, **podemos igualar el ciclo de reloj de las dos máquinas:**

$$G_s = \frac{CPI\ \text{sin}\ segmentación}{1 + ciclos de reloj de\ \text{det}ención de la segmentación\ por instrucción}$$

Un caso simple e importante es aquel en el que **todas las instrucciones tardan el mismo número de ciclos que además es igual al número de etapas del cauce** (profundidad de la segmentación). **En este caso, el CPI sin segmentación es igual a la profundidad del cauce y por lo tanto:**

$$G_s = \frac{\text{Pr}ofundidad\ de\ la\ segmentación}{1 + ciclos de reloj de\ \text{det}ención de la segmentación\ por instrucción}$$

Si no hubiese detenciones esto llevaría al resultado intuitivo de que la segmentación puede mejorar el rendimiento en la magnitud de la profundidad de la segmentación.

Segmentación como una reducción del ciclo de reloj.

Alternativamente **si entendemos la segmentación como una mejora del ciclo de reloj**, entonces podemos suponer que **el CPI de la máquina no segmentada así como el de la segmentada vale 1**. Esto nos lleva a:

$$G_s = \frac{CPI \text{ sin } segmentación}{CPI \ con\ segmentación} \times \frac{Ciclo\ de\ reloj \text{ sin } segmentación}{Ciclo\ de\ reloj\ con\ segmentación}$$

$$= \frac{1}{1 + ciclos de \det ención\ por\ instrucción} \times \frac{Ciclo\ de\ reloj \text{ sin } segmentación}{Ciclo\ de\ reloj\ con\ segmentación}$$

En el caso de que las **etapas del cauce** estén **equilibradas y no exista sobrecarga**, **el ciclo de reloj en la máquina segmentada es menor que el ciclo de reloj de la máquina no segmentada en** un factor igual a **la profundidad del cauce.**

$$Ciclo\ de\ reloj\ con\ segmentación = \frac{Ciclo\ de\ reloj \text{ sin } segmentación}{profundidad\ de\ la\ segmentación}$$

$$profundidad\ de\ la\ segmentación = \frac{Ciclo\ de\ reloj \text{ sin } segmentación}{Ciclo\ de\ reloj\ con\ segmentación}$$

Esto nos lleva a

$$G_s = \frac{1}{1 + ciclos de \det ención\ por\ instrucción} \times \frac{Ciclo\ de\ reloj \text{ sin } segmentación}{Ciclo\ de\ reloj\ con\ segmentación}$$

$$= \frac{1}{1 + ciclos de \det ención\ por\ instrucción} \times profundidad\ de\ la\ segmentación$$

Esto nos llevaría, de nuevo, al resultado intuitivo de que la segmentación puede mejorar el rendimiento en la magnitud de la profundidad de la segmentación, si no hubiese detenciones.

1.2.4. Riesgos estructurales

Si **alguna combinación de instrucciones no puede darse** en el cauce **por conflicto de recursos**, se dice que **la máquina tiene un riesgo estructural**.

Cuando una máquina se segmenta, **la ejecución solapada** de instrucciones **necesita la segmentación de unidades funcionales** y la **duplicación de recursos** para permitir todas las posibles combinaciones de instrucciones en el cauce.

Dos formas de encontrar riesgos estructurales

Unidades funcionales que no están completamente segmentada
Los casos más comunes de riesgos estructurales surgen cuando alguna **unidad funcional no esta completamente segmentada**. En ese caso una secuencia de instrucciones usando esta unidad no segmentada no puede proceder a razón de una instrucción por ciclo.
Ejemplo: Unidad aritmética en punto flotante que no permite segmentación.

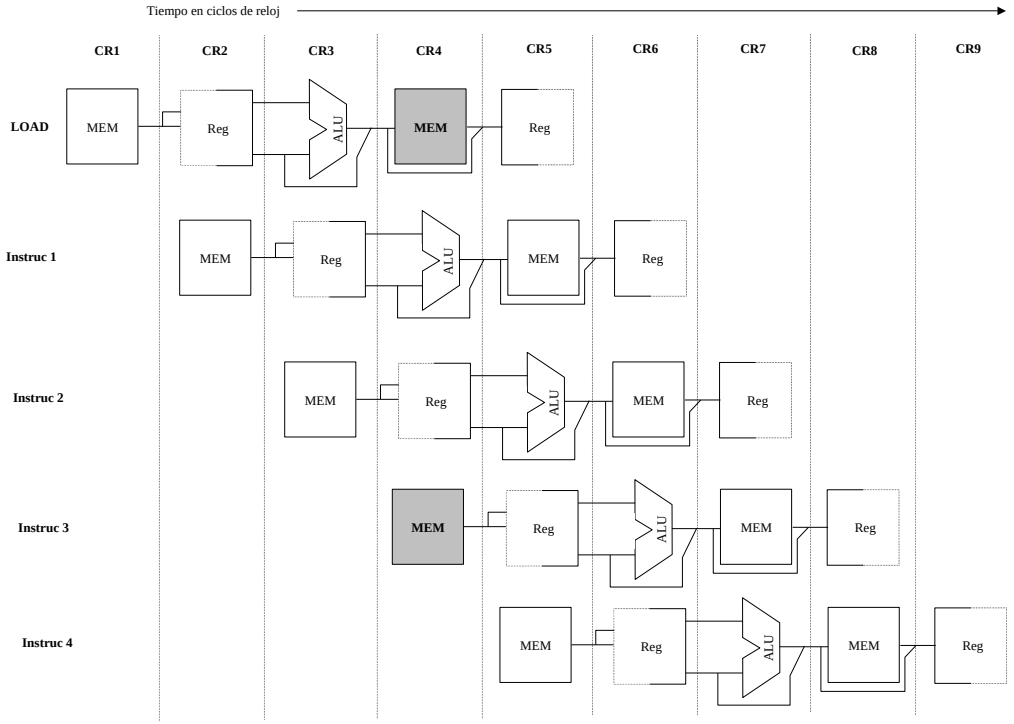
Recurso que no se ha duplicado lo suficiente
Otra forma habitual de encontrar riesgos estructurales es cuando **un recurso no se ha duplicado lo suficiente** para soportar todas las combinaciones.
Por **ejemplo**, una máquina puede tener **un solo puerto de escritura en el banco de registros**, pero en determinadas circunstancias, la segmentación puede necesitar realizar dos escrituras en un único ciclo. Esto generará un riesgo estructural.

Cuando una secuencia de instrucciones **encuentre este riesgo**, el cauce **detendrá** una de las instrucciones hasta que la unidad requerida este libre. Estas detenciones incrementarán el CPI ideal de valor 1.

Ejemplo: Algunas máquinas segmentadas comparten una **única memoria para datos e instrucciones**.

Como resultado de esto, **cuando una instrucción contiene una referencia a un dato de memoria**, **entrará en conflicto con la referencia a instrucción de una instrucción posterior**, como podemos ver en la figura.

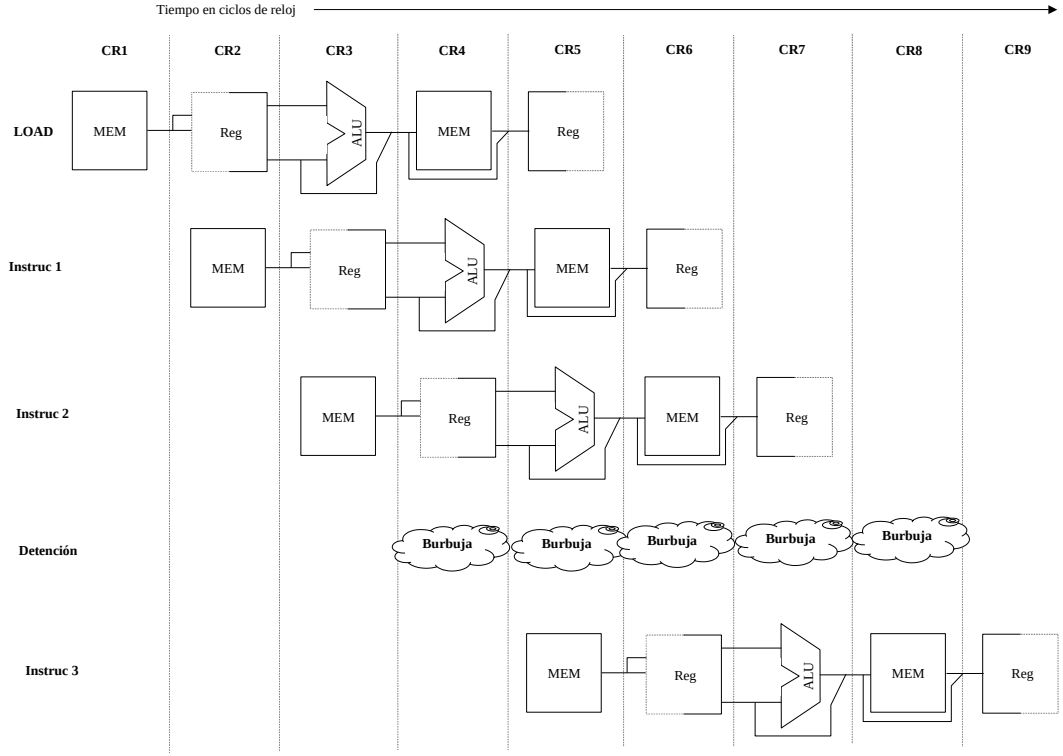
La instrucción **load usa la memoria para un acceso a datos al mismo tiempo que la instrucción 3 quiere ser cargada desde memoria.**



Detenemos la segmentación durante un ciclo de reloj cuando ocurre el acceso a los datos de memoria.

A las detenciones se les llama burbujas, ya que flotan en el cauce ocupando espacio pero sin realizar trabajo.

El efecto es que **ninguna instrucción terminará durante el ciclo de reloj 8,** instante de finalización normal de la instrucción 3.
Asumimos que la instrucción 1 no es load ni store, en ese caso la instrucción 3 no puede comenzar su ejecución.



En lugar de dibujar el camino de datos segmentado cada vez, los diseñadores a menudo indican el comportamiento de las detenciones **usando diagramas simples con los nombres de las etapas**, como en la figura.

La figura muestra la detención indicando el ciclo donde no ocurren acciones y desplazando la instrucción 3 a la derecha (lo que retrasa su inicio y finalización en un ciclo).

Nº ciclo de reloj Instrucción	1	2	3	4	5	6	7	8	9	10
Load	IF	ID	EX	MEM	WB					
Instrucción i+1		IF	ID	EX	MEM	WB				
Instrucción i+2			IF	ID	EX	MEM	WB			
Instrucción i+3				Deten	IF	ID	EX	MEM	WB	
Instrucción i+4						IF	ID	EX	MEM	WB
Instrucción i+5							IF	ID	EX	MEM
Instrucción i+6								IF	ID	EX

El efecto de la burbuja es **ocupar los recursos para el hueco de una instrucción como si esta cruzase todo el cauce**.

El **impacto** sobre el rendimiento es que **la instrucción 3 no se completa hasta el ciclo 9** y **ninguna instrucción se completa durante el octavo ciclo.**

En ocasiones este diagrama se dibuja con la detención ocupando una fila completa y la instrucción i+3 desplazada a la fila siguiente, en cualquier caso el efecto es el mismo, ya que la instrucción i+3 no comienza su ejecución hasta el ciclo 5.

Ejemplo. Veamos cuanto puede costar el riesgo estructural load. Supongamos que las referencias a datos constituyen un 40% de la mezcla, y que el CPI ideal de la máquina segmentada, ignorando el riesgo estructural es 1. Suponemos que la máquina con el riesgo estructural tiene una frecuencia de reloj que es 1.05 veces mayor que la frecuencia de reloj que la máquina sin el riesgo. Descartando otras perdidas de rendimiento. ¿Es la segmentación con el riesgo estructural más o menos rápida que sin el y en que magnitud?.

Calculando el tiempo medio por instrucción:

Tiempo medio por instrucción = CPI x Ciclo de reloj

El tiempo medio por instrucción para la máquina ideal es (al no tener detenciones)

Tiempo medio por instrucción $_{\text{ideal}}$ = Ciclo de reloj $_{\text{ideal}}$

El tiempo medio por instrucción para la máquina con el riesgo estructural es:

Tiempo medio por instrucción = CPI x Ciclo de reloj

$$= (1+0.4\cdot 1)\cdot \frac{Ciclo\,de\,reloj_{ideal}}{1.05} = 1.3\cdot Ciclo\,de\,reloj_{ideal}$$

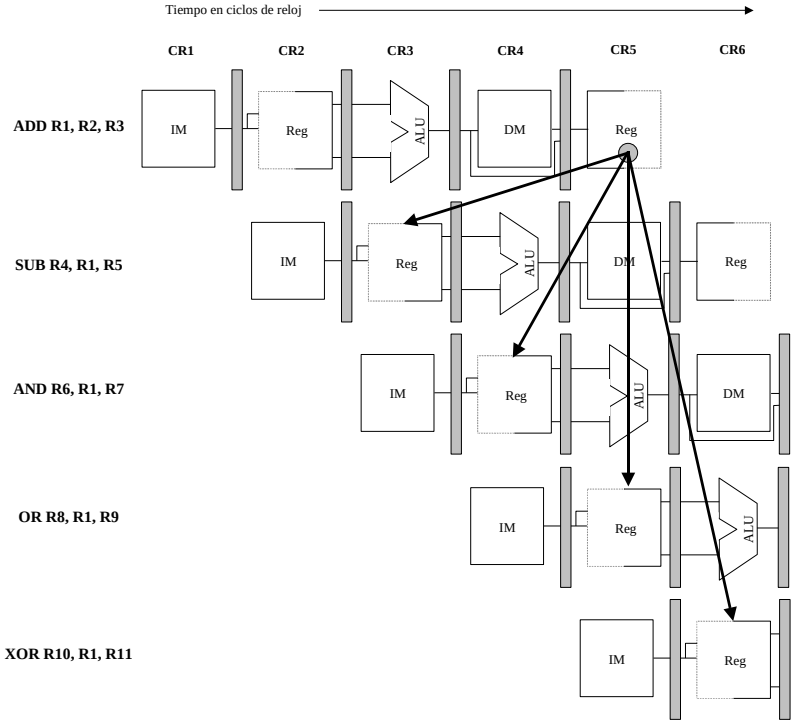
Claramente la máquina sin el riesgo estructural es 1.3 veces más rápida.

1.2.5. Riesgos por dependencia de datos

El **mayor efecto de la segmentación** es **cambiar la secuenciación relativa de las instrucciones** al solapar su ejecución. **Esto introduce** los **riesgos de datos y de control**.

Los **riesgos de datos ocurren** cuando **la segmentación cambia el orden de los accesos lectura/escritura a los operandos** y en consecuencia dicho orden difiere del secuencial en una máquina no segmentada.

Vamos a considerar la ejecución segmentada de las siguientes instrucciones:



Todas **las instrucciones tras la ADD usan el resultado que esta calcula**.

1. Riesgo de datos de la primera instrucción (SUB)

La instrucción ADD escribe el valor de **R1 en** la etapa **WB**, pero la **instrucción SUB lee el valor durante su etapa ID**. La instrucción SUB leerá el valor erróneo e intentará utilizarlo.

> **El valor usado por la instrucción SUB ni siquiera es determinístico**:
> Aunque podemos pensar que es lógico asumir que SUB siempre usa el valor de R1 que le fue asignado por una instrucción previa a la ADD, esto no siempre ocurre. Si ocurre una interrupción entre las instrucciones ADD y SUB, el estado WB de la instrucción ADD se completará, y el valor de R1 en este punto será el resultado de la ADD. Este comportamiento impredecible es evidentemente inaceptable.

2. Riesgo de datos de la segunda instrucción (AND)

La instrucción AND también se ve afectada por este riesgo. Como podemos ver en la figura, la escritura de R1 no se completa hasta el final del ciclo 5. Por lo tanto la instrucción AND que lee el registro durante el ciclo 4 recibirá resultados erróneos.

3. Riesgo de datos de la tercera instrucción (OR)

La instrucción OR puede hacerse funcionar sin que incurra en un riesgo con una técnica de implementación simple. La técnica propone la realización de las **lecturas del banco de registros en la segunda mitad del ciclo y las escrituras en la primera mitad**. Esta técnica, permite que la instrucción OR se ejecute perfectamente.

La instrucción **XOR funciona adecuadamente**, porque la lectura de sus registros ocurre en el ciclo 6, después de la escritura del registro.

1.2.5.1. La técnica del adelantamiento

El problema de **los riesgos de datos** puede ser **resuelto** con una **técnica hardware** simple **denominada adelantamiento** (forwarding, bypassing, short-circuiting).

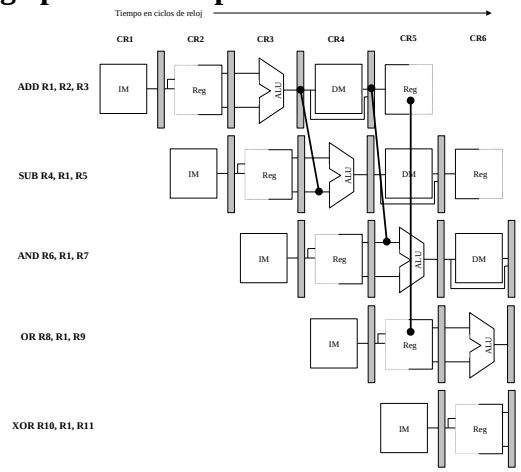
La clave del adelantamiento es que **el resultado no se necesita realmente en la instrucción SUB hasta después de que la ADD realmente los produce**.
Si el resultado **puede moverse desde donde lo produce la ADD** (**el registro EX/MEM**) **hasta donde lo necesita la SUB** (**los cerrojos de entrada de la ALU**) entonces la necesidad de detención puede evitarse.

Usando esta observación **el adelantamiento trabaja de la siguiente forma**:

1. El **resultado de la ALU** del registro EX/MEM **siempre se realimenta** a los cerrojos de entrada de la ALU.
2. **Si** el **hardware de adelantamiento detecta** que **la instrucción ALU previa ha escrito el registro correspondiente a una fuente de la operación ALU en curso**, la lógica de control selecciona el valor adelantado como valor de entrada a la ALU en lugar del valor leído en el banco de registros.

Completamente determinístico
Observar como con el adelantamiento, **si la SUB se detiene**, **la ADD se completará y el adelantamiento no será activado**. Esto es cierto también para el caso de una interrupción entre las dos instrucciones.

La figura siguiente muestra el ejemplo con las rutas de adelantamiento
Esta secuencia de código puede ser implementada sin detenciones.



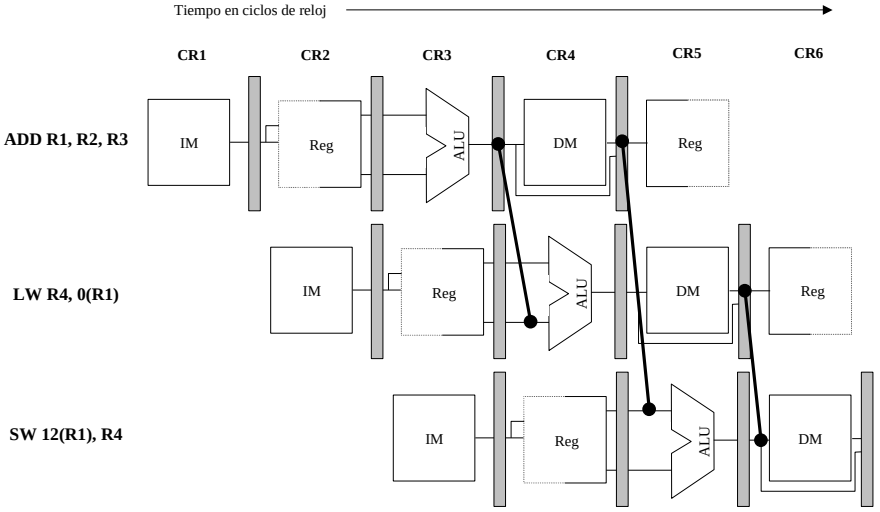
Las entradas **para las instrucciones SUB y AND se adelantan desde los registros intermedios EX/MEM y MEM/WB** respectivamente, **a la primera entrada de la ALU**.

La OR recibe el resultado mediante adelantamiento a través del banco de registros, situación fácilmente implementable sin más que leer el banco de registros en la segunda mitad del ciclo y escribirlo en la primera, como indican las líneas punteadas.

Observa que el resultado adelantado puede ir a cualquiera de las entradas de la ALU, de hecho ambas entradas de la ALU pueden usar entradas adelantadas tanto desde el mismo registro intermedio como desde diferente registro intermedio. Esto podría ocurrir, por ejemplo, si la instrucción AND fuese AND R6, R1, R4.

Generalización del adelantamiento

El adelantamiento puede generalizarse para incluir el paso de resultados directamente a la unidad funcional que los requiere: **un resultado se adelanta desde la salida de una unidad a la entrada de otra**, en lugar de permitir únicamente desde el resultado de una unidad a la entrada de la misma unidad.



Para prevenir una detención en esta secuencia, podemos necesitar **adelantar** los valores de **R1 y R4 desde** los **registros intermedios hasta** las **entradas** de la **ALU y** la **memoria de datos**.

La instrucción **STORE requiere un operando durante MEM**. **El resultado de la instrucción LOAD se adelanta desde la salida de la memoria en MEM/WB a la entrada de la memoria para ser almacenado.**

Además, la **salida de la ALU se adelanta a la entrada de la ALU** para el cálculo de la dirección **tanto en la instrucción LOAD como en la STORE** (esto no es diferente de adelantar a otro operación ALU).

En DLX, **podemos requerir una ruta de adelantamiento desde cualquier registro intermedio hacia la entrada de cualquier unidad funcional**.

Como tanto la ALU como la memoria de datos aceptan operandos, se **necesitan rutas de adelantamiento** hacia sus entradas desde los registros intermedios ALU/MEM y MEM/WB.

> Adicionalmente, DLX usa **una unidad de detección de cero** que opera durante el ciclo EX, y el adelantamiento a esa unidad puede ser necesario también. Más tarde en esta sección exploraremos todas las rutas de adelantamiento necesarias y el control de esas rutas.

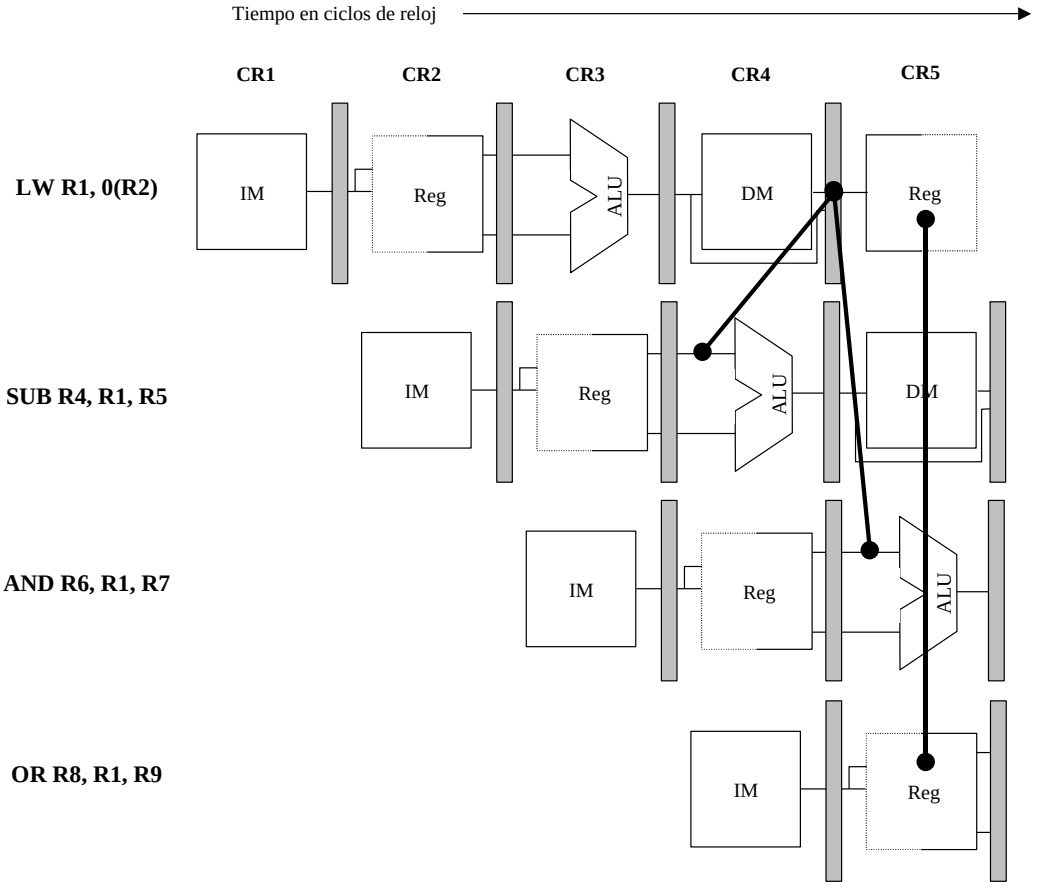
1.2.5.2. Riesgos de datos que requieren detenciones

No todos los riesgos de datos potenciales pueden tratarse mediante adelantamiento.

1ª instrucción posterior
La instrucción **LW no tiene el dato hasta** el **final del ciclo 4** (su ciclo MEM), mientras que **la instrucción SUB necesita el dato en el inicio de ese ciclo de reloj**. Por eso, la consecuencia de usar el resultado de una instrucción load no puede ser completamente eliminado simplemente con hardware. Como muestra la figura, una ruta de adelantamiento como esa tiene que operar hacia atrás en el tiempo.

2ª instrucción posterior
Podemos **adelantar** el **resultado** inmediatamente **a la ALU desde** los registros **MEM/WB** para usar en la operación AND, que comienza dos ciclos después de la load.

3ª instrucción posterior
Así mismo, la instrucción OR no tiene problema, ya que **recibe el valor a través** del **banco de registros**.



Hardware de interbloqueo

Necesitamos **añadir hardware de interbloqueo** del cauce, para preservar el patrón de ejecución correcto. En general, **el hardware de interbloqueo**, **detecta un riesgo y detiene el cauce** hasta que el riesgo se ha aclarado.

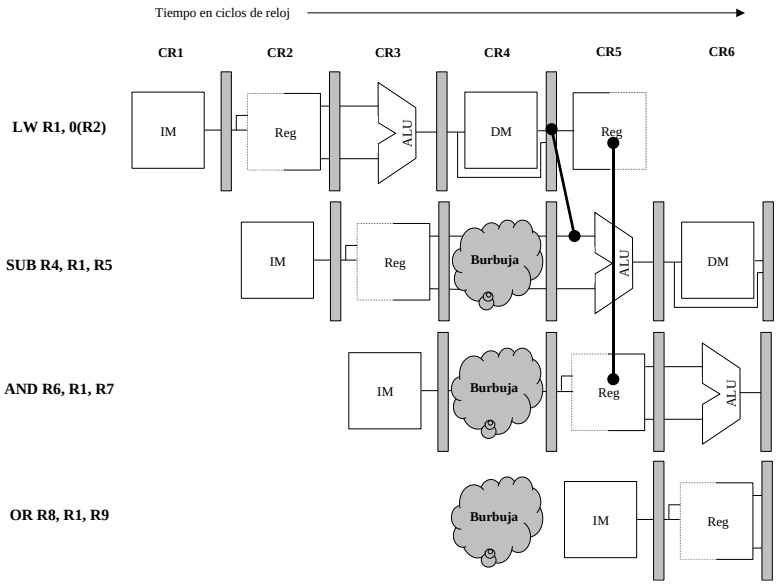
En este caso, **el interbloqueo detiene el cauce**, **empezando por la instrucción que quiere usar el dato hasta que la instrucción fuente los produce**.

Este interbloqueo del cauce introduce una detención o burbuja, como se hacía para los riesgos estructurales.

Observamos el cauce con la detención y el adelantamiento legal.

Debido a que la detención causa que el comienzo de la SUB se desplace un ciclo después, **el adelantamiento a la instrucción AND va ahora a través del banco de registros y no se necesita adelantamiento para la instrucción OR**.

No comienza ninguna instrucción **durante** el ciclo **4** (**y** nin**guna finaliza durante el ciclo 6**).



La figura muestra el cauce antes y después de la detención

LW	R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB	R4, R1, R5		IF	ID	EX	MEM	WB			
AND	R6, R1, R7			IF	ID	EX	MEM	WB		
OR	R8, R1, R9				IF	ID	EX	MEM	WB	

LW	R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB	R4, R1, R5		IF	ID	**Deten**	EX	MEM	WB		
AND	R6, R1, R7			IF	**Deten**	ID	EX	MEM	WB	
OR	R8, R1, R9				**Deten**	IF	ID	EX	MEM	WB

Ejemplo

Suponer que el 30% de las instrucciones son cargas y que la mitad de veces la instrucción que sigue a una instrucción de carga depende del resultado de la carga. Si este riesgo crea un retardo de un solo ciclo ¿Cuantas veces es más rápida la máquina ideal segmentada (con un CPI de 1) que no retarda la segmentación, si se compara con una segmentación más realista? Ignoramos cualquier otro tipo de detención.

$$CPI_{\text{instrucción siguiente a una carga}} = 1.5$$
$$CPI_{\text{efectivo}} = 0.7*1 + 0.3*1.5 = 1.15$$

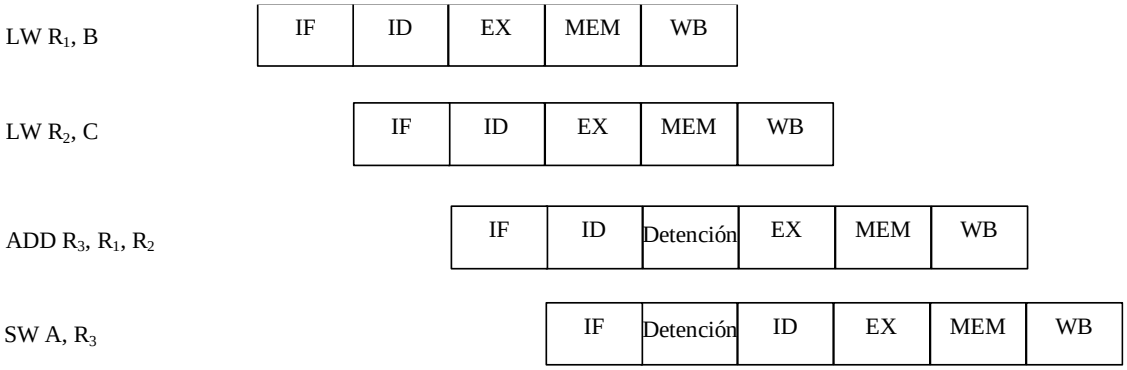
La máquina ideal es 1.15 veces más rápida.

Planificación del compilador para los riesgos de datos.

Muchos tipos de detenciones son muy frecuentes.

El patrón general de generación de código **para una sentencia del tipo A=B+C produce una detención para la carga del valor del segundo dato C**.

El almacenamiento de A no provoca otra detención, ya que el resultado de la suma puede adelantarse a la memoria de datos para que sea usado por la store.



En lugar de permitir que el cauce se detenga, **el compilador puede intentar planificar la segmentación para evitar estas detenciones mediante la reorganización del código** para eliminar los riesgos.

Por **ejemplo**, **el compilador puede intentar evitar generar código con una instrucción load seguida por el uso inmediato del registro destino de la load**. Esta técnica se llama *planificación del cauce* o *planificación de las instrucciones.*

Ejemplo

Generar código de DLX que evite detenciones del cauce para la siguiente secuencia:

a = b + c
d = e + f

LW R_b, b
LW R_c, c
LW R_e, e
ADD R_a, R_b, R_c
LW R_f, f
SW a, R_a
SUB R_d, R_e, R_f
SW d R_d

Evitamos los interbloqueos:

LW R_c,c / ADD R_a, R_b, R_c
LW R_f,f / SUB R_d, R_e, R_f

Implementación de la detección de riesgos por dependencias de datos.

Emisión: Es el proceso de evolucionar desde la fase ID a la EX.

Para la segmentación **DLX** todos **los riesgos de datos pueden detectarse durante** la etapa **ID**.

En caso de riesgo se detiene la emisión de la instrucción. Una instrucción que ha realizado este paso se dice que se ha emitido.

De igual modo, **podemos determinar que adelantamiento será necesario durante ID** y fijar el control apropiado.

La detección temprana de interbloqueos en el cauce **reduce la complejidad del hardware**, **porque nunca tiene que suspender una instrucción que ha actualizado el estado de la máquina**, a no ser que se detenga la máquina por completo.

Situaciones detectables por el hardware de detección de riesgos al comparar los destinos y fuentes de instrucciones.

Esta tabla muestra que **la única comparación requerida es entre el destino y las fuentes de las dos instrucciones siguientes a la instrucción que escribió el destino**. En el caso de una detención, las dependencias del cauce serán como el tercer caso una vez continúa la ejecución.

Situación	Secuencia de código ejemplo	Acción

No dependencia	LW **R_1**, 45(R_2)
	ADD R_5, R_6, R_7
	SUB R_8, R_6, R_7
	OR R_9, R_6, R_7
		No hay riesgo. No existe dependencia sobre R_1 en las tres instrucciones siguientes.

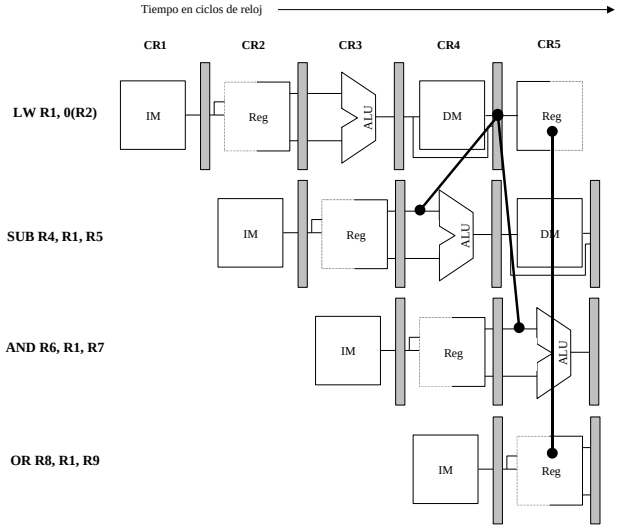
Dependencia que requiere detención	LW **R_1**, 45(R_2)
	ADD R_5, **R_1**, R_7
	SUB R_8, R_6, R_7
	OR R_9, R_6, R_7
		Los comparadores detectan el uso de R_1 en ADD y detienen ADD (y SUB y OR) antes que ADD comience EX

Dependencia superada por adelantamiento	LW **R_1**, 45(R_2)
	ADD R_5, R_6, R_7
	SUB R_8, **R_1**, R_7
	OR R_9, R_6, R_7
		Los comparadores detectan el uso de R_1 en SUB y adelantan el resultado de la carga a la ALU en el instante en que SUB comienza EX.

Dependencia con accesos en orden	LW **R_1**, 45(R_2)
	ADD R_5, R_6, R_7
	SUB R_8, R_6, R_7
	OR R_9, **R_1**, R_7
		No se requiere acción porque la lectura de R_1 por OR se presenta en la segunda mitad de la fase ID, mientras que la escritura del dato cargado se presentó en la primera mitad.

Implementación del interbloqueo load

Si hay un riesgo RAW siendo la instrucción fuente una load, la instrucción load estará en la etapa EX cuando una instrucción que necesita el dato cargado este en la etapa ID.



Podemos describir todas las situaciones de riesgo posibles con una pequeña tabla, que puede ser directamente trasladada a una implementación. La tabla detecta todos los interbloqueos load cuando la instrucción que usa el resultado de la load está en la etapa ID.

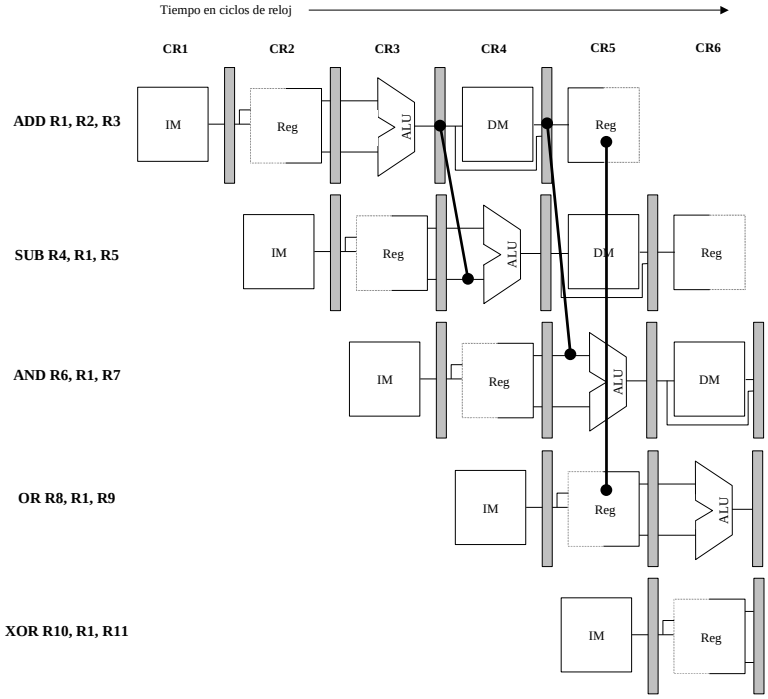
Campo código de operación de ID/EX (ID/EX.$IR_{0..5}$)	Campo código de operación de IF/ID (IF/ID.$IR_{0..5}$)	Comparación de campos de operandos
Load	ALU registro registro	ID/EX.$IR_{11..15}$==IF/ID.$IR_{6..10}$
Load	ALU registro registro	ID/EX.$IR_{11..15}$==IF/ID.$IR_{11..15}$
Load	Load, store, ALU inm, o salto	ID/EX.$IR_{11..15}$==IF/ID.$IR_{6..10}$

La lógica que detecta la necesidad de interbloqueos load durante la etapa ID de una instrucción requiere **tres comparaciones**.

Las líneas 1 y 2 de la tabla comprueban si el registro destino de la load es uno de los registros fuente para una operación registro registro en ID.

La línea 3 de la tabla determina si el registro destino de la load es una fuente para una dirección efectiva de una load o store, una ALU con inmediato o una comprobación de salto.

Implementación de la lógica de adelantamiento



La implementación de la lógica de adelantamiento es similar, aunque hay **más casos que considerar.**

La observación clave necesaria para implementar la lógica de adelantamiento es que **los registros intermedios contienen** tanto **los datos** que hay que adelantar **así como los campos de los registros fuente y destino**.

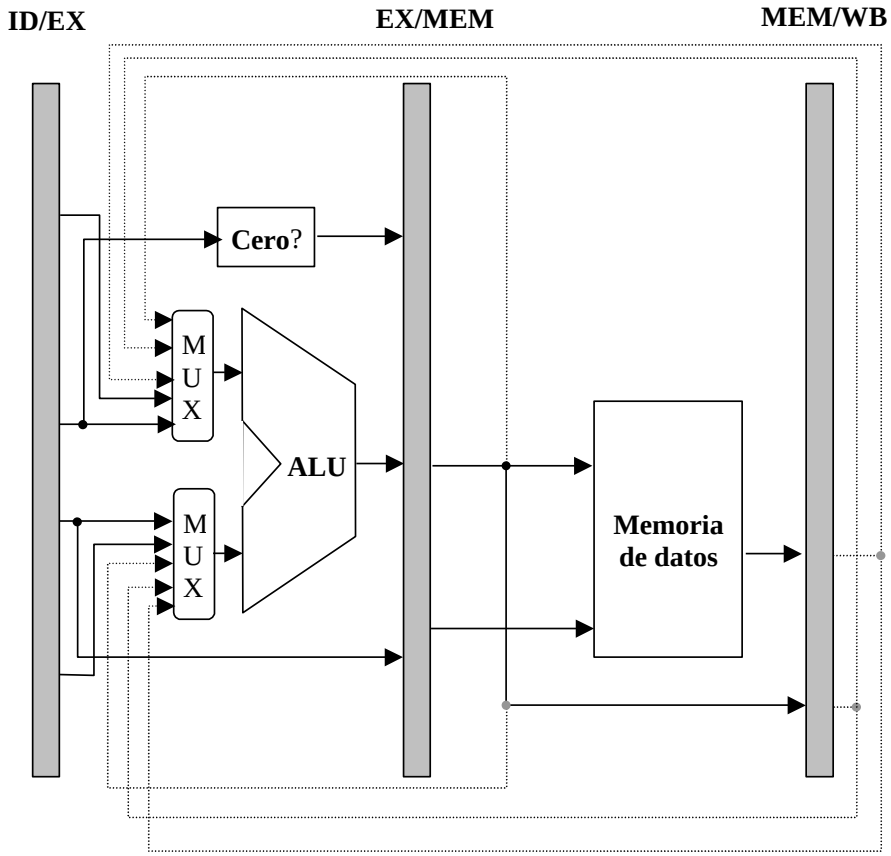
Todo adelantamiento, lógicamente, **ocurre**:

Desde la **salida de la ALU o** la **memoria de datos**

Hacia la **entrada** de la **ALU**, **la memoria de datos o la unidad de detección de ceros.**

Registro intermedio con la instrucción fuente	Código de operación de la instrucción fuente	Registro intermedio con la instrucción destino	Código de operación de la instrucción destino	Destino del resultado adelantado	Comparación (si igual entonces adelantar)
EX/MEM	ALU rer-reg	ID/EX	ALU rer-reg ALU inm Load, store Branch	Entrada alta de la ALU	EX/MEM.$IR_{16..20}$= ID/EX.$IR_{6..10}$
EX/MEM	ALU rer-reg	ID/EX	ALU rer-reg	Entrada baja de la ALU	EX/MEM.$IR_{16..20}$= ID/EX.$IR_{11..15}$
MEM/WB	ALU rer-reg	ID/EX	ALU rer-reg ALU inm Load, store Branch	Entrada alta de la ALU	MEM/WB.$IR_{16..20}$= ID/EX.$IR_{6..10}$
MEM/WB	ALU rer-reg	ID/EX	ALU rer-reg	Entrada baja de la ALU	MEM/WB.$IR_{16..20}$= ID/EX.$IR_{11..15}$
EX/MEM	ALU inm	ID/EX	ALU rer-reg ALU inm Load, store Branch	Entrada alta de la ALU	EX/MEM.$IR_{11..15}$= ID/EX.$IR_{6..10}$
EX/MEM	ALU inm	ID/EX	ALU rer-reg	Entrada baja de la ALU	EX/MEM.$IR_{11..15}$= ID/EX.$IR_{11..15}$
MEM/WB	ALU inm	ID/EX	ALU rer-reg ALU inm Load, store Branch	Entrada alta de la ALU	MEM/WB.$IR_{11..15}$= ID/EX.$IR_{6..10}$
MEM/WB	ALU inm	ID/EX	ALU rer-reg	Entrada baja de la ALU	MEM/WB.$IR_{11..15}$= ID/EX.$IR_{11..15}$
MEM/WB	Load	ID/EX	ALU rer-reg ALU inm Load, store Branch	Entrada alta de la ALU	MEM/WB.$IR_{11..15}$= ID/EX.$IR_{6..10}$
MEM/WB	Load	ID/EX	ALU rer-reg	Entrada baja de la ALU	MEM/WB.$IR_{11..15}$= ID/EX.$IR_{11..15}$

Adicionalmente a los comparadores y lógica combinacional necesaria para determinar cuando se necesita habilitar una ruta de adelantamiento, además necesitamos aumentar los multiplexores en las entradas de la ALU y añadir las conexiones desde los registros intermedios que se usan para adelantar los resultados. La figura muestra los segmentos relevantes del camino de datos segmentado con los multiplexores adicionales y las conexiones.



Adelantar los resultados a la ALU requiere la incorporación de tres entradas adicionales en cada multiplexor de la ALU y la incorporación de tres rutas a las nuevas entradas. Las rutas corresponden al adelantamiento de (1) la salida de la ALU al final de EX (2) la salida de la ALU al final de la etapa MEM y (3) la salida de la memoria al final de la etapa MEM.

Para DLX, el hardware de detección de riesgos y adelantamiento es razonablemente simple; veremos que estas cosas se complican cuando extendamos esta segmentación para tratar con punto flotante. Antes necesitamos tratar los saltos.

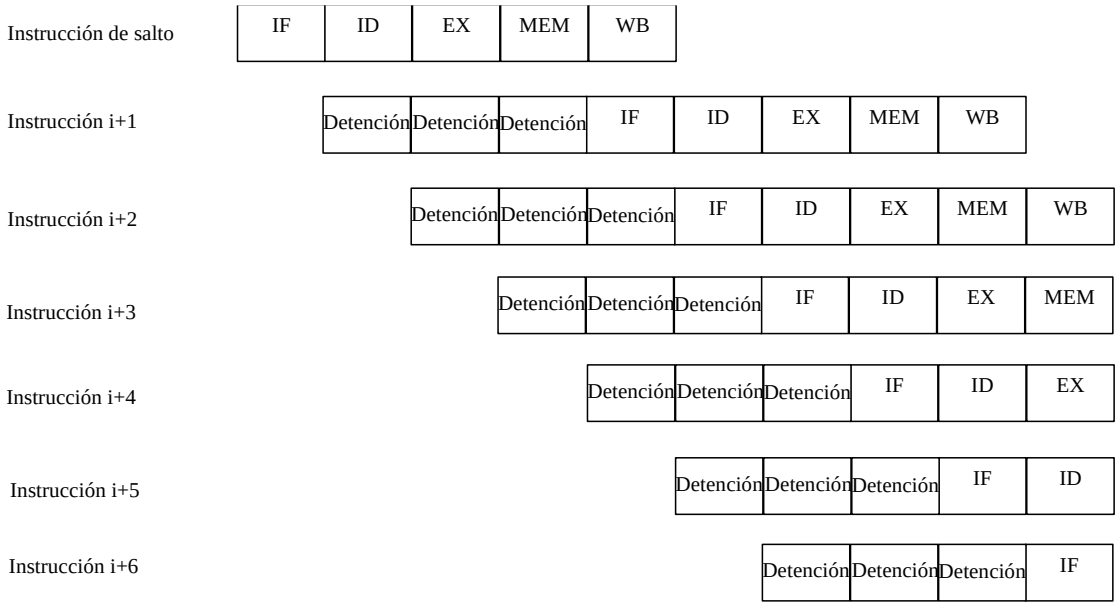
1.2.6. Riesgos de control

Los **riesgos de control** pueden provocar **mayor pérdida** de rendimiento **que** los riesgos por **dependencia de datos**.

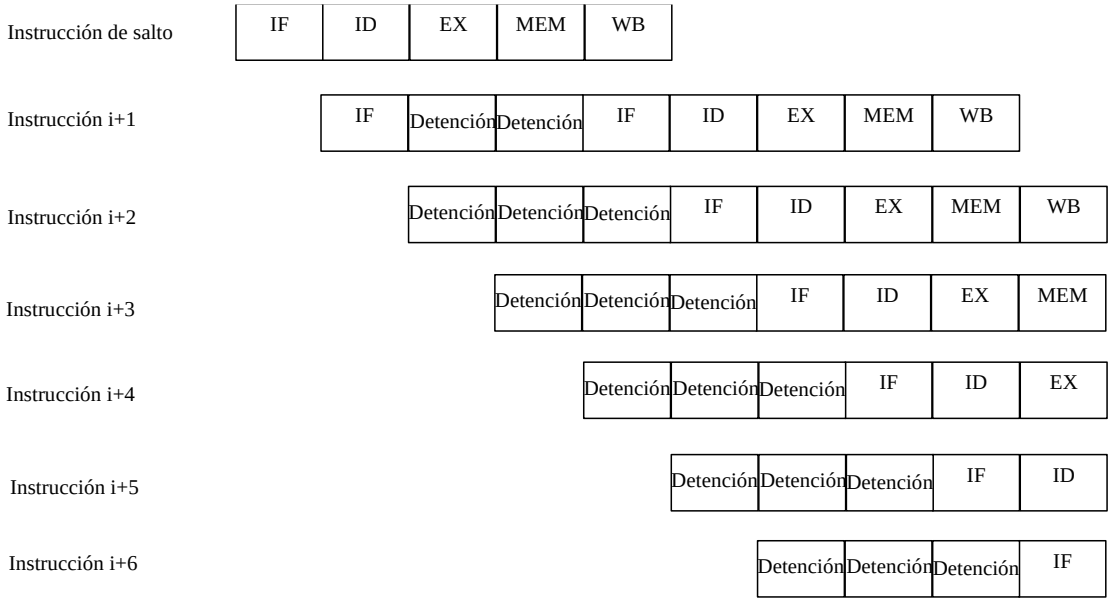
Cuando se ejecuta un salto pueden ocurrir **dos cosas**, en función de la evaluación de la condición de salto (realizada en fase EX):

> **El salto es efectivo:** El PC cambia su valor por una dirección nueva calculada por la ALU.
>
> **ALUoutput ← PC + ($(IRl_{16})^{16}$##$IRl_{16..31}$);**
>
> El cambio del PC no se realiza hasta el final de MEM, después de completar el cálculo de la dirección y la comparación.
>
> **El salto no es efectivo:** El PC cambia su valor por PC+4.

El **método más simple** de tratar con saltos es **detener el cauce tan pronto** como **detectemos** el **salto** antes de alcanzar la etapa MEM, que determina el nuevo PC.



La detención no ocurre hasta después de la etapa **ID** (no queremos parar el cauce hasta que sepamos que la instrucción es un salto).



El ciclo IF de la instrucción siguiente al salto debe repetirse en cuanto conocemos el resultado del salto. Por eso, **el primer ciclo IF es esencialmente una detención**, porque nunca realiza trabajo útil.

Esta detención puede implementarse fijando el registro IF/ID a cero en los tres ciclos. Debemos resaltar que **si el salto no es efectivo**, entonces **la repetición de la etapa IF es innecesaria** ya que **la instrucción correcta fue cargada**.

Un salto causa una **detención de tres ciclos** en el cauce DLX: **Un ciclo** corresponde la **repetición de IF** y **dos ciclos** están **inactivos**.

Vamos a desarrollar varios esquemas para sacar partido a este hecho, pero en primer lugar **vamos a examinar como podríamos reducir la penalización del salto** en el peor caso.

Ejemplo

Supongamos una frecuencia de salto de un 30% y un CPI ideal de 1, ¿Qué relación de velocidad encontramos entre la máquina segmentada ideal y la máquina con detenciones por saltos?.

$CPI_{ideal} = 1$
$CPI_{maquina\ con\ detenciones\ por\ saltos} = 0.3*4+0.7*1= 1.2 + 0.7 = 1.9$

Relación de rendimiento = $\frac{1.9}{1}$

Reducción del número de ciclos de detención de salto

La **relación de rendimiento entre** la **máquina ideal** y la **máquina con detenciones por saltos** es de **aproximadamente el doble de rendimiento en la ideal**.

Por lo tanto, **la reducción del número de ciclos de detención** sea un **factor crítico**.

El número de ciclos de reloj en una detención por salto puede reducirse mediante dos pasos:

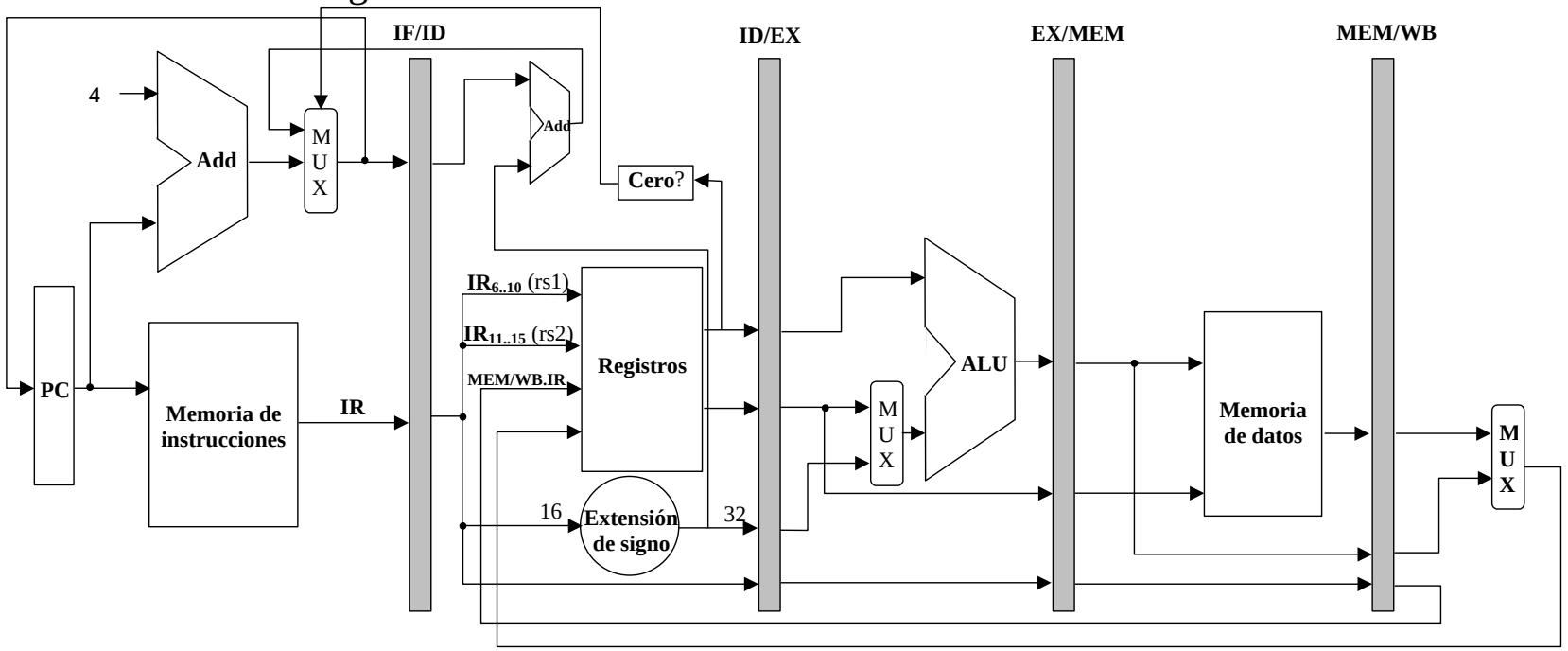
1. **Encontrar si el salto es efectivo** o no efectivo **antes** en el cauce
2. **Calcular antes el PC efectivo** (por lo tanto la dirección destino del salto).

Para optimizar el comportamiento del salto, **deben realizarse ambas cosas** –no sirve de nada conocer el destino del salto sin saber sin saber si este es efectivo. Ambos pasos deben realizarse lo antes posible en el cauce.

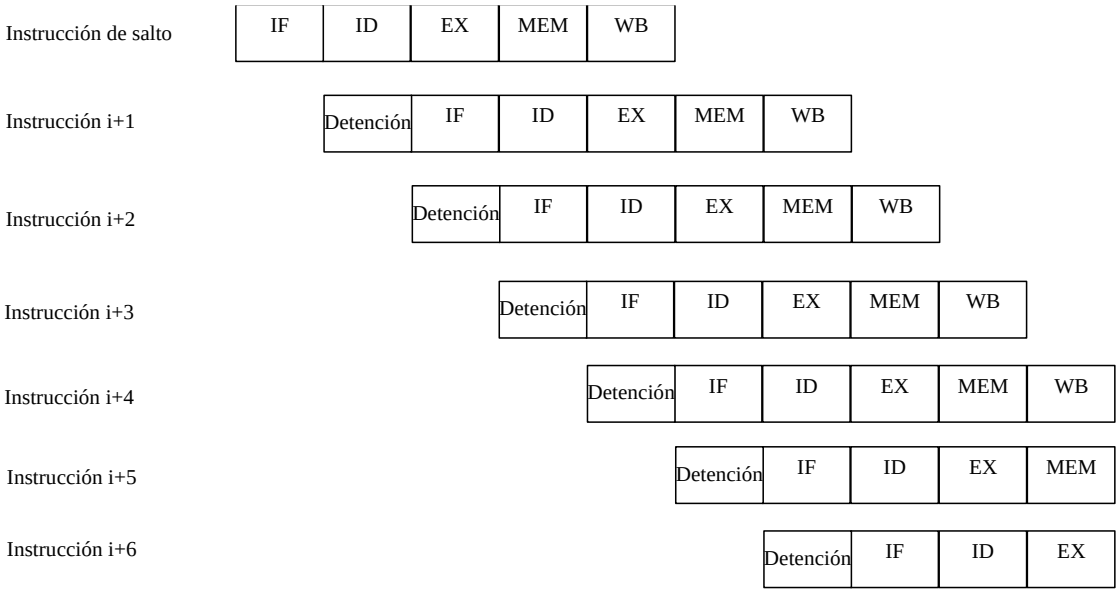
En DLX **los saltos (BEQZ y BNEZ) requieren comprobar** si **un registro** es **igual a cero**.

Es posible completar esta decisión al final del ciclo ID moviendo el test cero a este ciclo.

Para aprovechar este adelantamiento de la decisión de salto, ambos **PCs (efectivo y no efectivo) deben calcularse previamente**.
Computar el destino del salto durante ID requiere un sumador adicional porque la ALU, no esta disponible hasta EXE. La figura muestra el camino de datos revisado.



En estas condiciones sólo es necesario **un ciclo de detención** en los saltos.



Estructura revisada de la segmentación, se muestra el uso de un sumador separado para calcular la dirección destino del salto:

Etapa	**Cualquier instrucción**		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC, PC ← (if (Regs[IF/ID.IR$_{6..10}$] op 0) {IF/ID.NPC+(IF/ID.IR$_{16}$)^{16} ## IF/ID.IR$_{16..31}$} else {PC+4});		
ID	ID/EX.A ← Regs [IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs [IF/ID.IR$_{11..15}$]; ID/EX.IR ← IF/ID.IR ID/EX.Inm ← (IF/ID.IR$_{16}$)^{16} ## IF/ID.IR$_{16..31}$		
	Instrucción ALU	**Instrucción load o store**	**Instrucción de salto**
EX			
MEM			
WB			

Tanto el cálculo de la dirección de salto como la evaluación de la condición se realiza para todas las instrucciones.

Sólo se sustituye el PC en caso de evaluación positiva de la condición y de que la instrucción sea de salto. Por lo tanto la decodificación de la instrucción deberá ser previa a la carga del PC con el valor correspondiente.

En algunas máquinas, los riesgos por saltos son incluso más caros en ciclos de reloj que en nuestro ejemplo, **puesto que el tiempo para evaluar la condición de salto y computar el destino puede ser incluso mayor**.

El retardo por salto, a no ser que sea tratado, se transforma en una penalización por salto.

Muchas máquinas antiguas que implementan **conjuntos** de **instrucciones** más **complejos tienen retardos por saltos de cuatro ciclos de reloj o más y las máquinas segmentadas de mayor profundidad a menudo** tienen **penalizaciones** por salto de **seis o siete ciclos de reloj**.

Muchas **máquinas VAX** tienen **retardos** de salto de **cuatro ciclos de reloj como mínimo**.

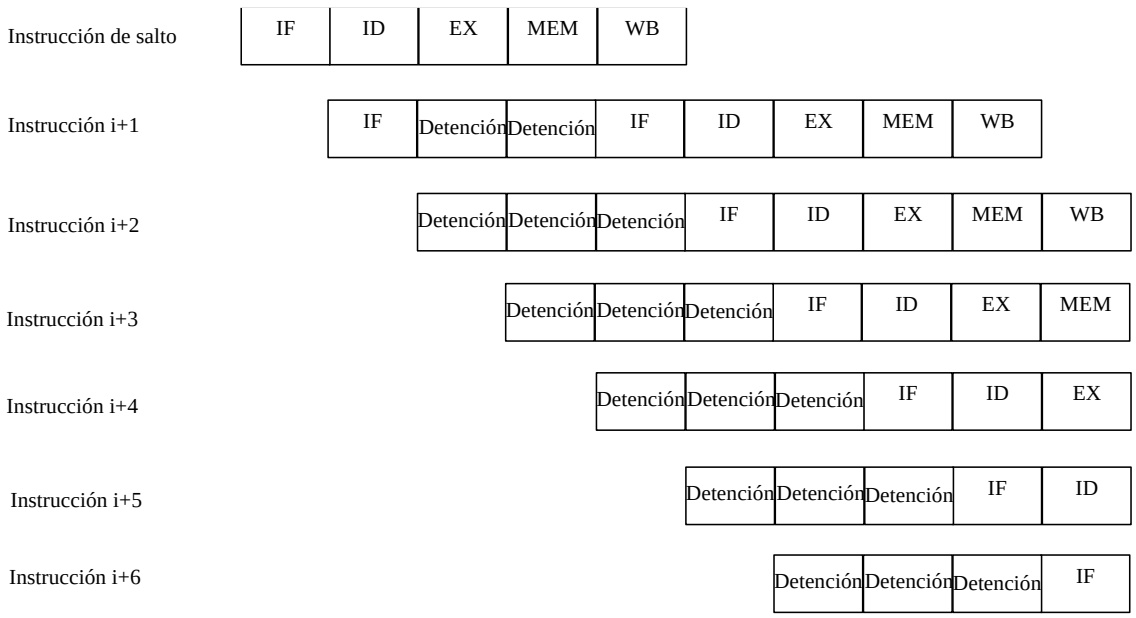
En general, cuanto **más profundo es el cauce**, **peor** es la **penalización** en ciclos de retardo **para los saltos**.

Reducción de las penalizaciones de saltos en la segmentación.

Estudiaremos **cuatro métodos de reducción** de penalizaciones debidas a saltos.

Congelación de la segmentación:

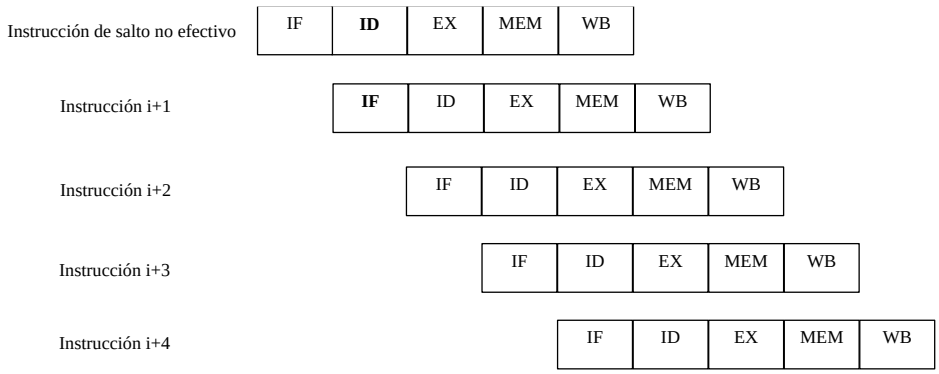
El esquema más sencillo es detener todas las instrucciones después del salto, hasta conocer el destino correcto. La sencillez del esquema es su principal atractivo.



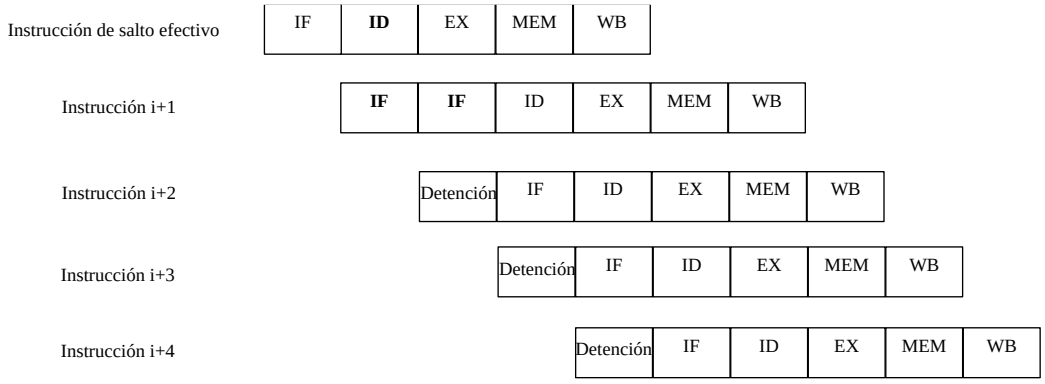
Predecir el salto como no efectivo

En este esquema permitimos que el hardware continúe como si el salto no se ejecutase. La complejidad radica en no cambiar el estado de la máquina hasta que no se conozca el resultado del salto. Esto supone el conocimiento de cuando una instrucción cambia el estado y como deshacer el cambio.
Se implementa continuando la búsqueda de instrucciones como si no ocurriese nada extraordinario. Si el salto es efectivo detenemos la segmentación, recomenzamos las búsquedas y deshacemos los cambios de estado (una posibilidad simple es limpiar la segmentación).

Cuando el salto no es efectivo: Se determina en ID, simplemente continuamos



Cuando el salto es efectivo durante ID, reanudamos la búsqueda en el destino del salto. Esto hace que todas las instrucciones que siguen al salto se detengan un ciclo de reloj.



Predecir el salto como efectivo

Una vez decodificado el salto y calculada la dirección destino, suponemos que el salto se va a realizar y comenzamos la búsqueda y ejecución en el destino.
En DLX no se conoce la dirección destino antes de conocer la evaluación del salto, por lo tanto esta estrategia no es útil. En otras máquinas con códigos de condición más potentes (más lentas) el destino del salto se conoce antes que la evaluación del salto, es en esta situación donde el esquema tiene sentido.