ARCOM 2 – SEGMENTACIÓN

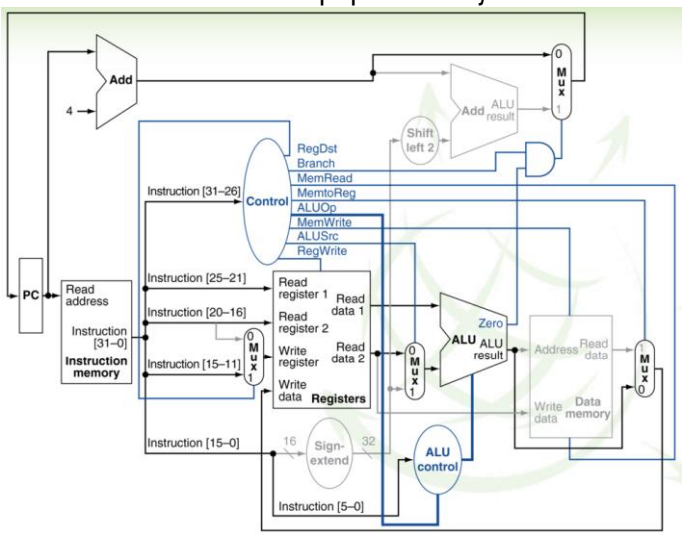
ARQUITECTURA MIPS

- MIPS es una arquitectura tipo RISC (instrucciones de tamaño y formatos uniformes) desarrollada para **aprovechar al máximo las técnicas de segmentación**.
- El ISA tiene **versiones de 32 y 64 bits**, además de **extensiones para aplicaciones específicas** (compresión de datos, instrucciones SIMD para coma flotante, etc.).
  - ↳ Estos 32 o 64 bits representan el ancho de la palabra, todas las instrucciones tienen esa longitud.

▷ Haremos un repaso del ISA de 32 bits como ejemplo de realizaciones uniciclo y multiciclo.

CAMINO DE DATOS

- El CAMINO DE DATOS es el hardware que procesa datos y direcciones en la CPU.



CICLO DE INSTRUCCIÓN

- El CICLO DE INSTRUCCIÓN son las etapas necesarias en un caso general para ejecutar cada instrucción.

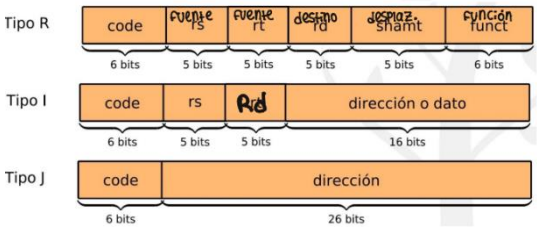
Etapas de ejecución:

1. Búsqueda de instrucciones (IF).
2. Decodificación y lectura de registros (ID).
3. Ejecución o cálculo de direcciones (EX).
4. Memoria de datos (MEM).
5. Escritura de registros (WB).

INSTRUCCIONES MIPS

TIPOS DE INSTRUCCIONES

- **Tipo I** ⟶ transferencia de datos, instrucciones con operandos inmediatos y saltos condicionales.
- **Tipo R** ⟶ operaciones aritméticas y lógicas.
- **Tipo J** ⟶ saltos incondicionales.



MODOS DE DIRECCIONAMIENTO

- Direccionamiento **directo** ⟶ el operando es una constante que aparece en un campo de la instrucción.
- Direccionamiento **registro** ⟶ el operando está en un registro.
- Direccionamiento **base con desplazamiento** ⟶ el operando está en una dirección de memoria obtenida como la suma de una constante (que está en la instrucción) y el contenido de un registro.
- Direccionamiento **relativo al PC** ⟶ el operando está en una dirección de memoria obtenida como la suma de una constante (que está en la instrucción) y el PC.
- Direccionamiento **pseudodirecto** ⟶ el operando está en una dirección de memoria obtenida como la concatenación de una constante (que está en la instrucción) y los MSB del PC.

Instrucción	Ejemplo	Significado	Comentarios
suma	`add $s1, $s2, $s3`	`$s1=$s2+$s3`	Tres operandos, detecta overflow
resta	`sub $s1, $s2, $s3`	`$s1=$s2-$s3`	Tres operandos, detecta overflow
suma inmediata	`addi $s1, $s2, 100`	`$s1=$s2+100`	Detecta overflow
suma sin signo.	`addu $s1, $s2, 100`	`$s1=$s2+$s3`	Tres operandos, no detecta overflow
resta sin signo.	`subu $s1, $s2, 100`	`$s1=$s2-$s3`	Tres operandos, no detecta overflow
multiplicación	`mul $s1, $s2`	`(Hi,Lo)=$s1×$s2`	Producto 64 bits en (Hi,Lo)
división	`div $s1, $s2`	`Lo=$s1 / $s2` `Hi=$s1 mod $s2`	Lo = división entera Hi = resto
Mover desde Hi	`mfhi $s1`	`$s1=Hi`	Recupera valor de Hi
Mover desde Lo	`mflo $s1`	`$s1=Lo`	Recupera valor de Lo

Instrucción	Ejemplo	Significado	Comentarios
and	`and $s1, $s2, $s3`	`$s1=$s2&+$s3`	Tres operandos
or	`or $s1, $s2, $s3`	`$s1=$s2|$s3`	Tres operandos
and inmediato	`andi $s1, $s2, 100`	`$s1=$s2&100`	Tres operandos
or inmediato	`ori $s1, $s2, 100`	`$s1=$s2|100`	Tres operandos
desplazamiento lógico izquierda	`sll $s1, $s2, 10`	`$s1=$s2<<10`	Despl. constante
desplazamiento lógico derecha	`srl $s1, $s2, 10`	`$s1=$s2>>10`	Despl. constante

Instrucción	Ejemplo	Significado	Comentarios
load word	`lw $s1, 100($s2)`	`$s1=Memory[$s2+100]`	Word de memoria a registro
store word	`sw $s1, 100($s2)`	`Memory[$s2+100]=$s1`	Word de registro a memoria
load byte	`lb $s1, 100($s2)`	`$s1=Memory[$s2+100]`	Byte de memoria a registro
store byte	`sb $s1, 100($s2)`	`Memory[$s2+100]=$s1`	Byte de registro a memoria
load upper immediate	`lui $s1, 100`	`$s1=100*2^16`	Carga sobre los 16 bits sup.

Instrucción	Ejemplo	Significado	Comentarios
branch on equal	`beq $s1, $s2, 25`	`if ($s1==$s2)` goto PC=PC+4+100	Relativo al PC
branch on not equal	`bne $s1, $s2, 25`	`if ($s1!=$s2)` goto PC=PC+4+100	Relativo al PC
set on less than	`slt $s1, $s2, $s3`	`if ($s2<$s3) $s1=1` `else $s1=0`	(instrucción aritmética)
set on less than imm.	`slti $s1, $s2, 100`	`if ($s2<100) $s1=1` `else $s1=0`	Complemento a dos
set on less than imm. unsigned	`sltiu $s1, $s2, 100`	`if ($s2<100) $s1=1` `else $s1=0`	Binario natural
jump	`j 2500`	PC=2500*4	Absoluto
jump register	`j $ra`	PC=$ra	Absoluto
jump and link	`jal 2500`	$ra=PC; PC=2500*4	Absoluto

IMPLEMENTACIÓN MONOCICLO

- En el MIPS, **cada instrucción se ejecuta en un ciclo de reloj** (CPI = 1). Entonces, es muy importante decidir una longitud de ciclo de reloj adecuada.
- Como hay instrucciones con **más etapas** de ejecución que otras, se tendrá que escoger una **longitud de ciclo** que permita que se ejecute la **más lenta**.
  ↳ Como consecuencia, el resto de instrucciones tendrán tiempos muertos durante su ciclo de ejecución.

Instrucción	Mem. Instr.	Lectura Reg.	ALU	Mem. Datos	Escritura Reg.	Total
`j dir`	X					1 etapa
`beq $t0, $t1, dir`	X	X	X			3 etapas
`add $t0, $t1, $t2`	X	X	X		X	4 etapas
`sw $t0, ($t1)`	X	X	X	X		4 etapas
`lw $t0, ($t1)`	X	X	X	X	X	5 etapas

- Opciones para **reducir el ciclo de reloj**:
  - Mejoras **tecnológicas de los circuitos** que permitan reducir el tiempo de ejecución de **cada etapa**.
  - Mejoras de la **organización del hardware** para que pueda ejecutar más de una instrucción **al mismo tiempo**.

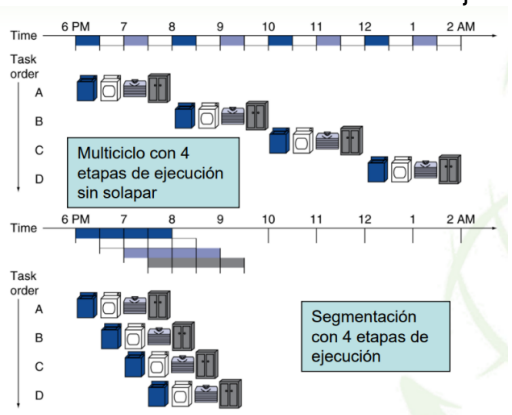
IMPLEMENTACIÓN MULTICICLO Y SEGMENTACIÓN

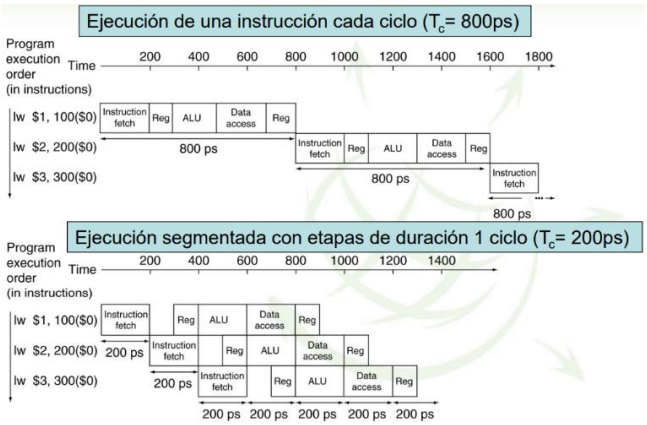
La implementación multiciclo y la segmentación son dos métodos de **mejora de la organización del hardware**:

- En las **implementaciones multiciclo** cada etapa de ejecución se ejecuta en un ciclo, de manera que en cada instante hay sólo una instrucción en ejecución.
  - Para conseguirla hay que **subdividir** la unidad de ejecución y colocar **registros entre etapas**.
  - ✓ Aprovecha mejor la **duración variable de las etapas** de ejecución.
- La **segmentación** se consigue a partir de una implementación multiciclo si se permite comenzar a ejecutar una instrucción mientras se ejecutan otras.
  - × **No** todas las instrucciones **terminan al mismo tiempo**.
  - × Cuando dos instrucciones necesitan usar la **misma unidad funcional** se pueden producir **conflictos**.
  - × Los **saltos** pueden provocar **paradas**.

SEGMENTACIÓN DEL CAUCE: CONCEPTOS BÁSICOS

- La **ejecución multiciclo con número de ciclos variable** acelera la ejecución de instrucciones permitiendo que unas acaben antes de otras.
- La **segmentación** permite lanzar a ejecución una instrucción antes de que acabe la anterior.
- El objetivo es aprovechar todas las etapas a la vez para ir progresando en la ejecución de diferentes instrucciones simultáneamente.
  - En cada instante habrá como **máximo una instrucción en cada etapa**.
  - **Idealmente** se inicia **una instrucción cada ciclo** de reloj.

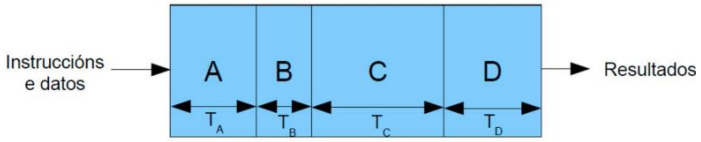




ETAPAS DE SEGMENTACIÓN DE CAUCE

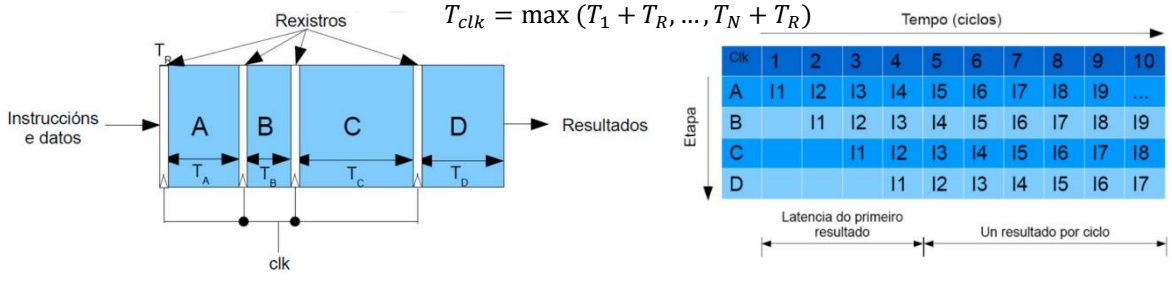
- **La complejidad del conjunto de instrucciones afecta directamente a la complejidad del pipeline**.
- Una unidad **convencional** de ejecución de N etapas tardará en ejecutar una instrucción la suma del tiempo que tarda cada una de las etapas.

$$T_{comb} = T_1 + \cdots + T_N$$



- Una unidad **segmentada** con N etapas tardará más en ejecutar cada instrucción por separado, pues tiene que cargar los registros entre cada etapa.
  ↳ El ciclo de reloj debe ser lo suficientemente largo para ejecutar la etapa más lenta y realizar su carga de registros.
  Por tanto, conviene que **todas las etapas** del pipeline tengan una **duración similar**.
- En cada ciclo de reloj se solapa la ejecución de diferentes instrucciones, de manera que, tras una **latencia inicial** igual al tiempo de ejecución de la primera instrucción, se obtiene **una instrucción por ciclo (idealmente)**.

$$T_{clk} = \max\left(T_1 + T_R, \ldots, T_N + T_R\right)$$



ACELERACIÓN EN LA SEGMENTACIÓN DE CAUCE

- Si todas las N etapas tienen la misma duración: $t_{entre\ inst\ segmentadas} = t_{entre\ inst\ no\ segmentadas}/N$.
  ↳ Si las etapas no están balanceadas la aceleración es menor.
- **Aceleración** que se obtiene con la segmentación: $SPEEDUP = t_{original}/t_{segmentación}$.
  - Esta aceleración se debe al **aumento de productividad**, pues aumenta el número de instrucciones finalizadas por unidad de tiempo.
  - La **latencia no disminuye**, pues aumenta ligeramente el tiempo que tarda en ejecutarse cada instrucción por separado.

➢ Procesador no segmentado
- Ciclo de reloj: 1 ns.
- 40% operaciones ALU → 4 ciclos cada una
- 20% operaciones de bifurcación → 4 ciclos cada una
- 40% operaciones de memoria → 5 ciclos cada una
- Sobrecoste de segmentación → 0.2 ns

➢ ¿Qué aceleración se obtendría con la segmentación?

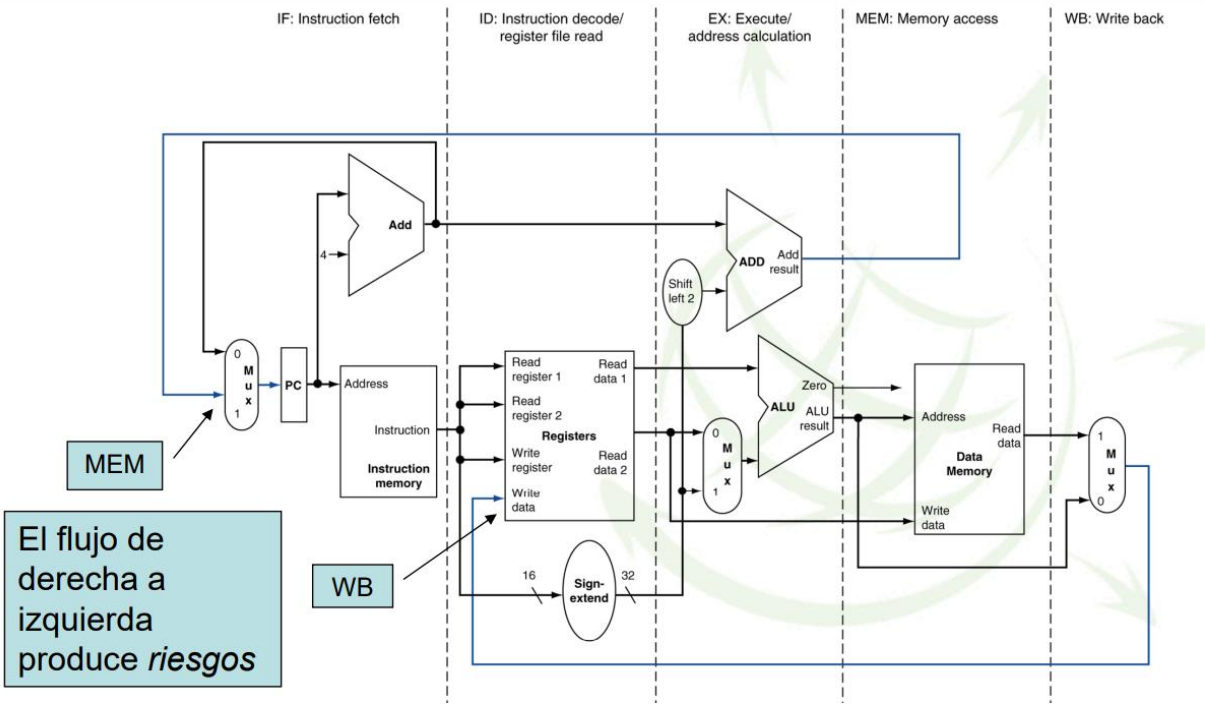
Comparamos el tempo medio por cada instrucción:

tCPU_*orig* = Tciclo*reloj* × CPI = 1ns/ciclo × (0.6 × 4 + 0.4 × 5) ciclos/instrucción = 4.4 ns/instrucción

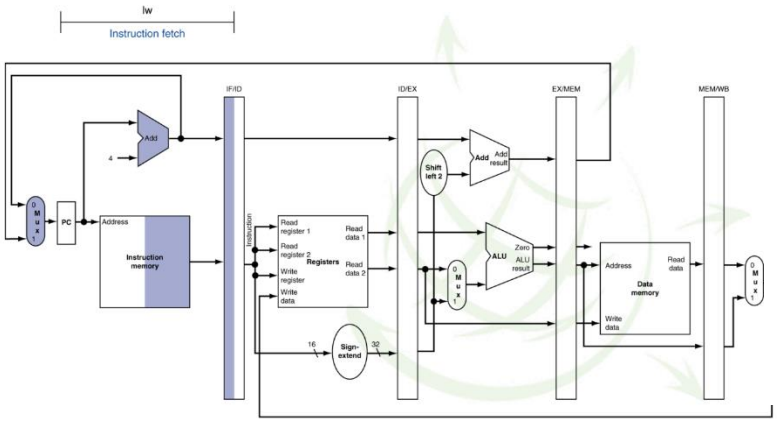
tCPU_*nuevo* = (1 + 0.2) ns/ciclo x (1 ciclo/instrucción) = 1.2 ns/instrucción

S = tCPU_*orig* / tCPU_*nuevo* = 4.4/ 1.2 = 3.7

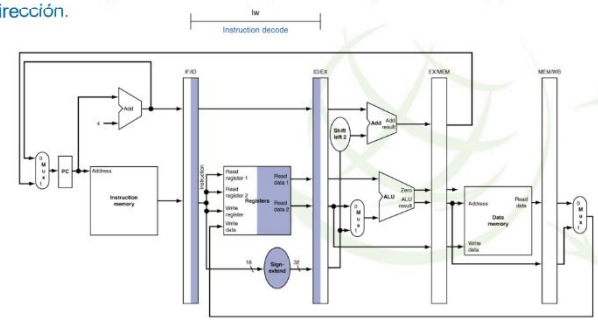
CAMINO DE DATOS SEGMENTADO EN MIPS



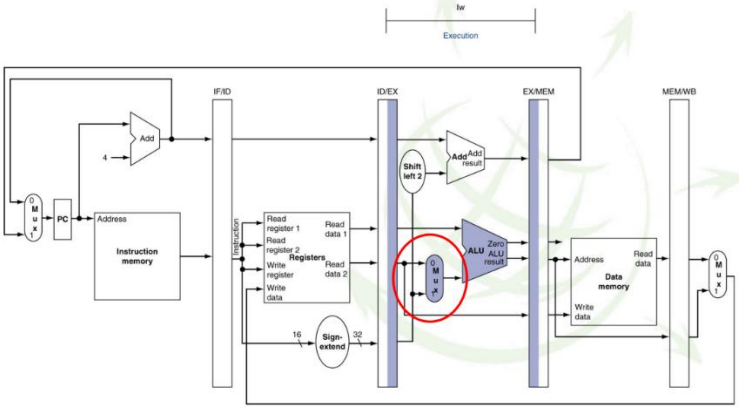
Veamos un ejemplo para las instrucciones load y store.



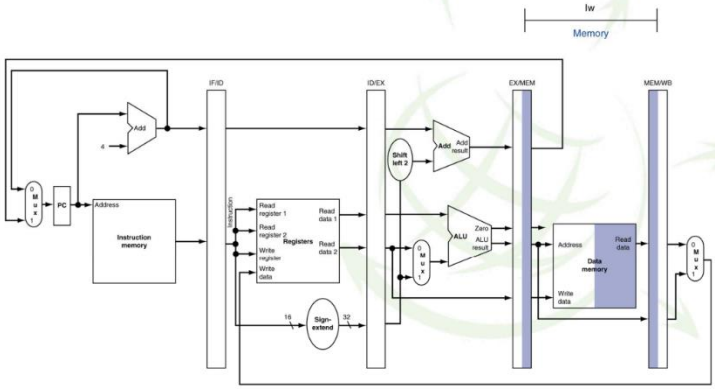
- Decodificación de la instrucción:
  - **Para Store, lectura de dos registros**. El inmediato es el desplazamiento para calcular la dirección de almacenamiento y se le extiende el signo para operar en la ALU.
  - **Para Load, se hace lectura de un registro** (base de la dirección) y se extiende en signo el inmediato, que es el desplazamiento para calcular la dirección.



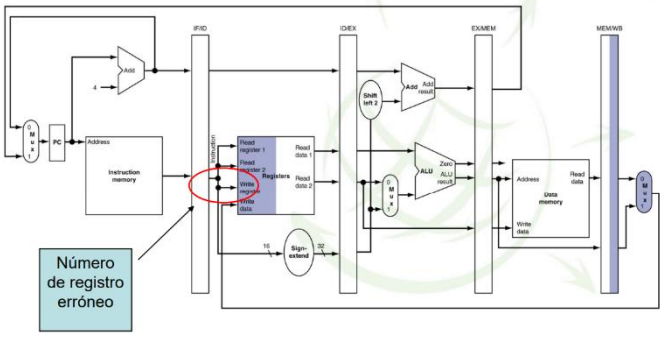
- El multiplexor selecciona el inmediato con el signo extendido. La ALU calcula la dirección, sumando base y desplazamiento.



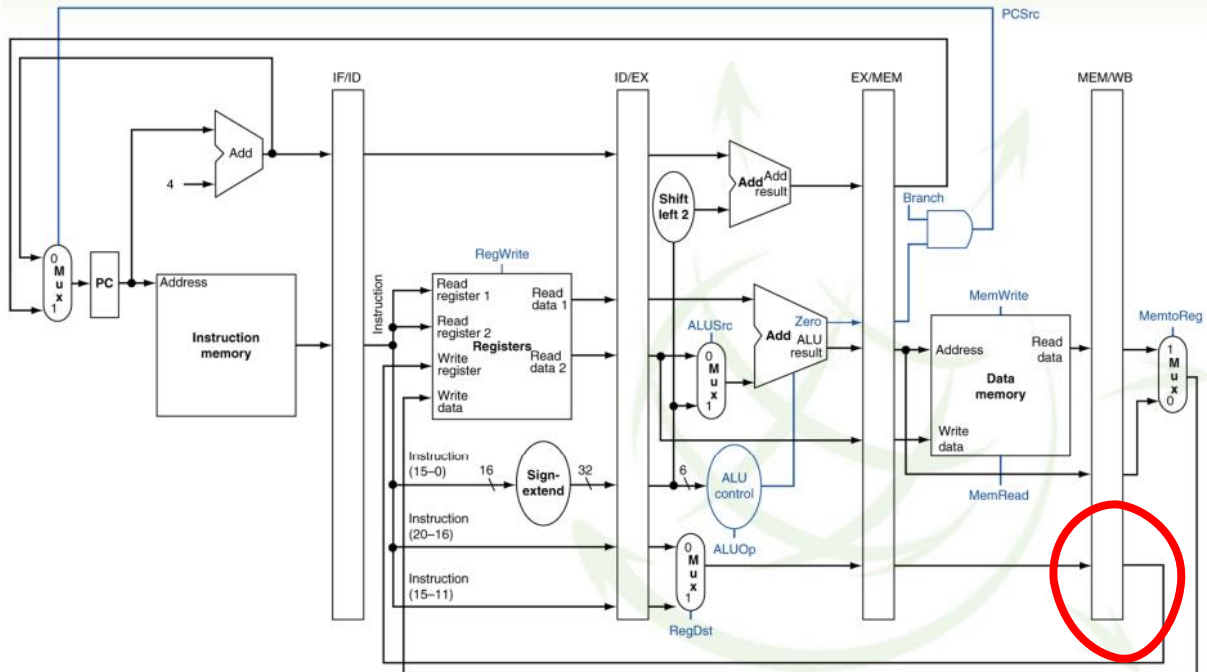
- Se realiza la lectura de la memoria (caché de datos) usando la dirección calculada en la ALU.



- El multiplexo selecciona la salida de memoria y escribe el valor en el banco de registros completando la instrucción Load.
- Pero… con el camino de datos que teníamos podemos afectar a la ejecución de otra instrucción que esté en etapa ID.



- Para evitar esto, se añade **hardware adicional** además del necesario para permitir la ejecución multiciclo.

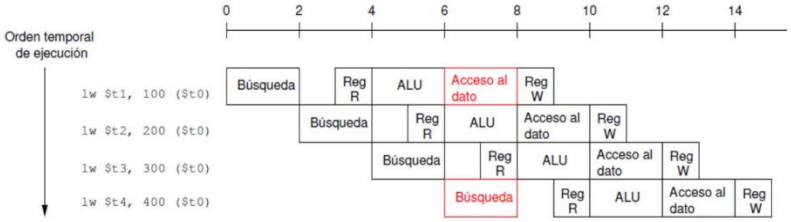


RIESGOS EN LA EJECUCIÓN

- El **CPI ideal** de un procesador segmentado es **1**, pero existen situaciones que **impiden que se inicie una instrucción en cada ciclo**.
  - Los ciclos del pipeline en los que no se computa se denominan BURBUJAS DEL PIPELINE y en ellos se introduce en el pipeline un CÓDIGO DE NO OPERACIÓN (NOP).
  - Los RIESGOS son situaciones que limitan la posibilidad de que dos instrucciones se ejecuten simultáneamente.
    - De **estructura** ⟶ el hardware no permite la ejecución de determinadas instrucciones a la vez.
    - De **datos** ⟶ se tiene que esperar a que una instrucción anterior lea o escriba un dato.
    - De **control** ⟶ no se sabe determinar la instrucción correcta que se tiene que ejecutar después de un salto.

RIESGOS ESTRUCTURALES

- Los RIESGOS ESTRUCTURALES se producen cuando los recursos hardware no son capaces de satisfacer la demanda de uso.
  - ▷ Por ejemplo, que sólo exista una unidad de división en punto flotante o una única memoria compartida para datos e instrucciones.
- La solución es **duplicar las unidades funcionales**



➢ Se consideran dos diseños alternativos y por defecto se considera un CPI de 1:
- A: diseño sin riesgos estructurales
  - T_A (período de la señal de reloj)= 1ns
- B: diseño con riesgos estructurales que suponen 1 ciclo perdido
  - T_B= 0.9ns
  - 30% de instrucciones de acceso a datos con riesgos estructurales

➢ ¿Alternativa más rápida comparando el tiempo por instrucción?

t_{inst}(A) = CPI x T_A = 1 x 1ns = 1ns

t_{inst}(B) = CPI x T_B = (0.7x 1 + 0.3 x (1+1)) x 0.9ns = 1.17ns

RIESGOS DE DATOS: BLOQUEO Y FORWARDING

- Los RIESGOS DE DATOS se producen cuando la ejecución de una instrucción depende de un dato que aún no está disponible.

Tipos de riesgos de datos:

- RAW (*Read After Write*) o DEPENDENCIA VERDADERA ⟶ una instrucción intenta leer un dato antes de que otra anterior lo escriba.
  - Las instrucciones no pueden ejecutarse en paralelo ni solaparse completamente, provoca una **parada**.
  - Se pueden detectar por **hardware** o por **compilador**.

```
i :    add $1, $2, $3
i +1: sub $4, $1, $3
```

La instrucción sub necesita el valor de $1 producido por la instrucción add => dependencia RAW en $1

- WAR (*Write After Read*) o ANTIDEPENDENCIA ⟶ una instrucción intenta modificar un dato antes de que otra anterior lo lea.
  - No puede ocurrir en un MIPS con pipeline de 5 etapas ya que ID es la etapa 2 y WB la 5, **no** provoca una **parada**.

```
i:    sub $4, $1, $3
i +1: add $1, $2, $3
i +2: mul $6, $1, $7
```

Habría riesgo si suponemos un pipeline en que la instrucción add modifica el valor de $1 antes de que la instrucción sub lo lea

- WAW (*Write After Write*) o FALSA DEPENDENCIA o DEPENDENCIA DE SALIDA ⟶ una instrucción intenta escribir un dato antes de que otra anterior lo escriba.
  - No puede ocurrir en un MIPS con pipeline de 5 etapas, pues estas siempre se ejecutan en el mismo orden, **no** provoca una **parada**.

```
i:    sub $1, $4, $3
i +1: add $1, $2, $3
i +2: mul $6, $1, $7
```

Si la instrucción add modifica el valor de $1 antes de que la instrucción sub lo modifique

SOLUCIONES A LOS RIESGOS DE DATOS

- WAR o WAW ⟶ renombrado de registros.
  - Renombrado **estático** ⟶ por el compilador.
  - Renombrado **dinámico** ⟶ por el hardware.
- RAW:
  - **Bloquear** la instrucción hasta que se realice la escritura necesaria.
  - El compilador **reordena el código** para mitigar el riesgo ⟶ introduce una secuencia de instrucciones independientes entre las conflictivas.
    - ↳ Si no hay instrucciones independientes inserta instrucciones NOP.
  - **Envío adelantado** (*forwarding, bypassing*, anticipación).

➢ Se considera un pipeline como el del MIPS, pero con los siguientes retardos: **una carga de datos necesita 3 ciclos de acceso a memoria, un almacenamiento necesita 5 y la carga de cada instrucción 4 ciclos excepto la primera instrucción, que requiere 1 ciclo**. Cuando hay conflicto se retrasa la etapa ID, es decir, se detectan en la etapa IF.

➢ En el código de más abajo las dependencias de datos RAW detectadas:
- $r2: I1 -> I2
- $r3: I2 -> I3

- Si no me dicen lo contrario, cuando hay dependencia RAW se debe esperar al ciclo WB de la instrucción fuente para iniciar el ciclo de decodificación y lectura de operandos (ID) de la instrucción destino. El mismo registro puede ser escrito durante la primera mitad de un ciclo de reloj y leído durante la segunda mitad.

Instrucción																
I1: lw $r2, 0($r0)	IF	ID	EX	M	M	M	WB									
I2: addi $r3, $r2, 20		IF1	IF2	IF3	IF4	–	ID	EX	M	WB						
I3: sw $r3, 0($r1)							IF1	IF2	IF3	IF4	ID	EX	M	M	M	M
I4: addi $r0, $r0, 4											IF1	IF2	IF3	IF4	ID	EX
I5: addi $r1, $r1, 4															IF1	IF2

Ejemplo: C code for `A = B + E; C = B + F;` Reordenando evitamos algunas de las paradas (derecha). No estamos considerando adelantamiento en ninguno de los casosy suponemos la unidad segmentada del MIPS.

Por riesgos de datos RAW

Parada
parada
parada
parada

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

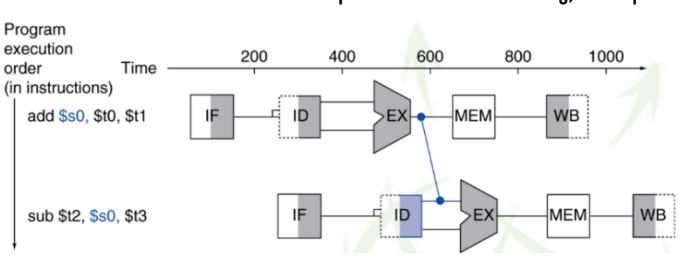
Código original: 23 ciclos

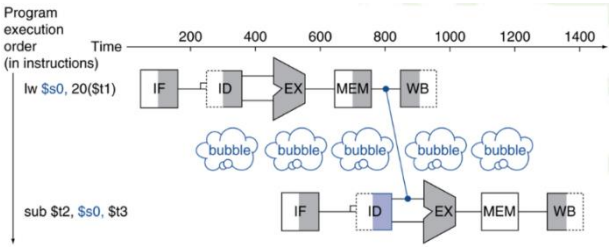
```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

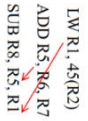
Código reordenado: 19 ciclos

FORWARDING

- El FORWARDING consiste en enviar resultados de las últimas etapas del pipeline a las primeras para evitar esperas.
  - ↳ Por tanto, requiere **conexiones adicionales.**
  - – **Forwarding** → el hardware adelanta el dato que hace falta nada más se calcula, sin esperar al WB.
  - – **Bloqueo + Forwarding** → se detiene por hardware la ejecución de instrucciones hasta que el dato necesario esté disponible.
    - ↳ Si no se puede realizar forwarding, el bloqueo duraría un ciclo más.

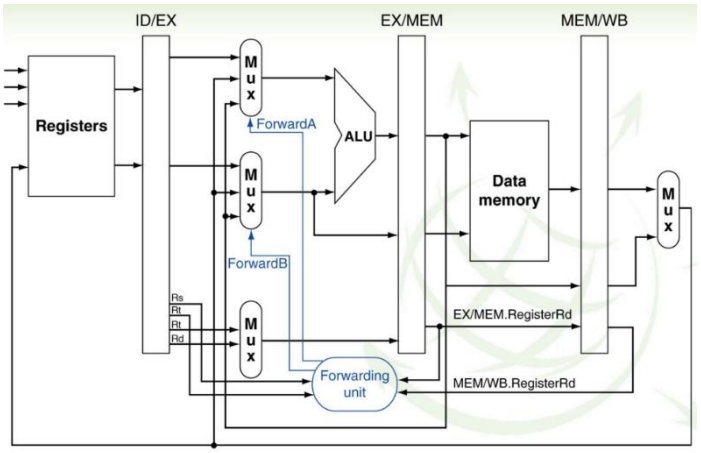


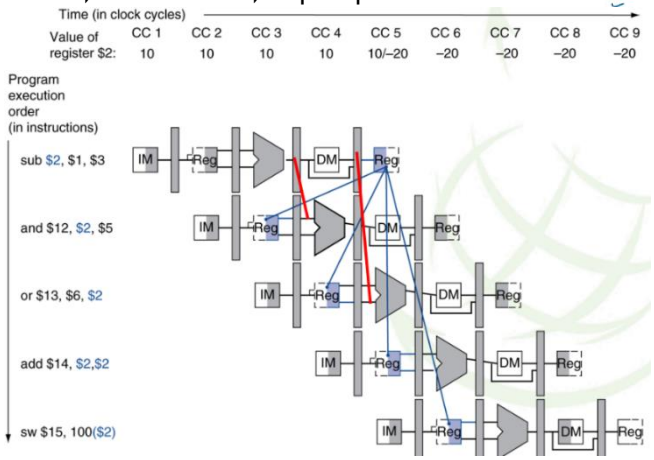




Gracias al *forwarding* SUB no sufre penalización en ciclos de espera:
Los operandos se pasan desde la etapa EX de ADD y desde la etapa MEM de la instrucción LW

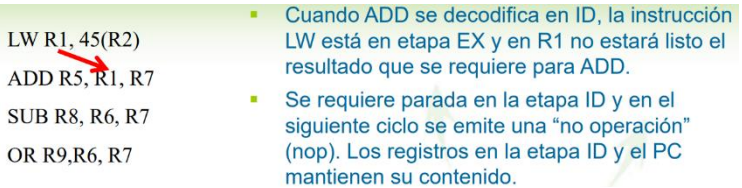
- Permite obtener las **entradas de la ALU** desde **cualquier registro de segmentación**, no solo del ID o EX, así que requiere modificar la ALU.

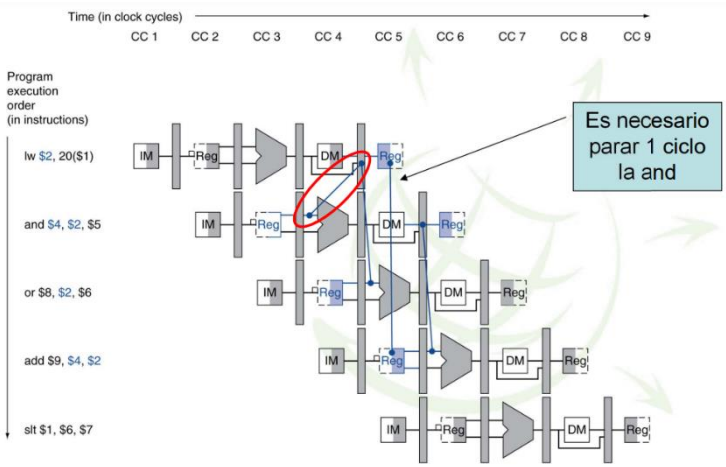


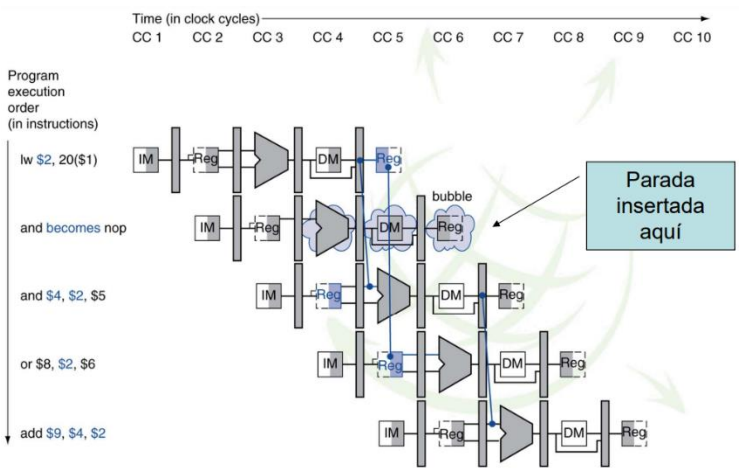


- **Requerimientos** para realizar forwarding:
  - **Detectar** si uno de los datos se puede traer desde otra etapa del pipeline analizando los códigos de las instrucciones que están en la etapa ID y que van a necesitar en la EX resultados de otras instrucciones.
  - Generar **señales de control** en los multiplexores que controlan la entrada a la ALU para seleccionar la procedencia del operando.

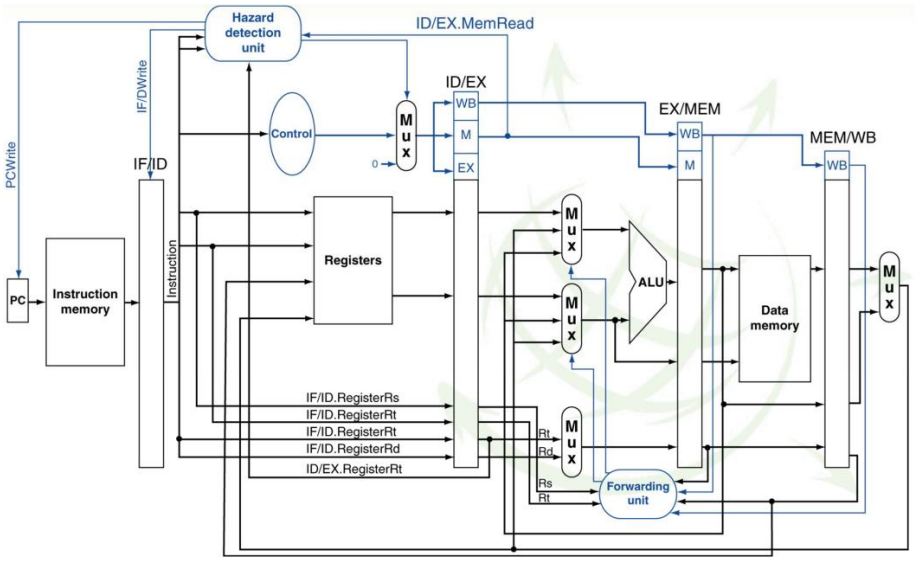
- El CONTROL DE INTERBLOQUEO DE INSTRUCCIONES es un proceso que consiste en detectar cuándo una instrucción **no puede avanzar por dependencia** en operandos con otra, **aunque se aplique adelantamiento**, provocando que se **pare la emisión de instrucciones** hasta que se solucione el bloqueo.
  - ↳ La **duración** de la detección depende del **tipo** de instrucciones involucradas.
  - Se detecta en la etapa **ID** pues en ella se averigua qué registros son operandos de entrada.
  - Se activa una parada si:
    1. En la etapa ID hay una instrucción tipo ALU, salto o de almacenamiento.
    2. En la etapa EX se está ejecutando una carga.
    3. El registro destino de la carga es operando de entrada de la instrucción que está en etapa ID.







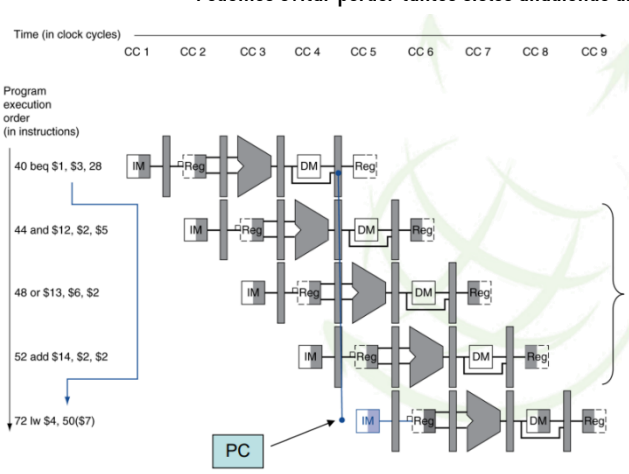
CAMINO DE DATOS CON DETECCIÓN DE **INTERBLOQUEO** Y DE **ADELANTAMIENTO**



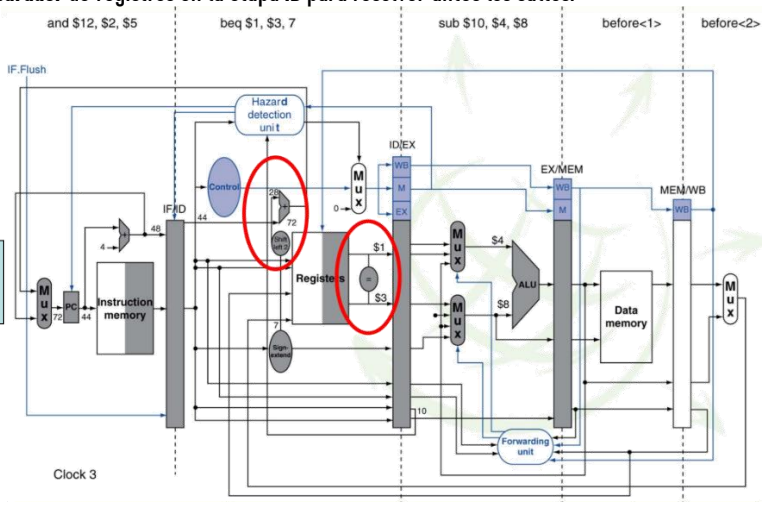
RIESGOS DE CONTROL: TRATAMIENTO DE SALTOS

- Los RIESGOS DE CONTROL se producen cuando se debe tomar una decisión basada en los resultados de una instrucción mientras otras se están ejecutando.

- En el pipeline del MIPS normalmente los saltos condicionales se resuelven en la etapa EX.
  Para **optimizar** su ejecución es necesario **comparar los registros** en una etapa anterior, normalmente la **ID**.
  - Si el salto se decide en la etapa MEM y continuamos como si no se fuera a saltar, se perderían 3 ciclos si al final se saltase.
  - ↳ Podemos evitar perder tantos ciclos añadiendo un **sumador** y **comparador** de registros en la etapa **ID** para resolver antes los saltos.



Eliminar estas instrucciones del pipeline



- Llamaremos SALTO TOMADO a la situación en la que se modifica el PC y SALTO NO TOMADO a aquella en la que no se modifica.

SOLUCIONES A LOS RIESGOS DE CONTROL

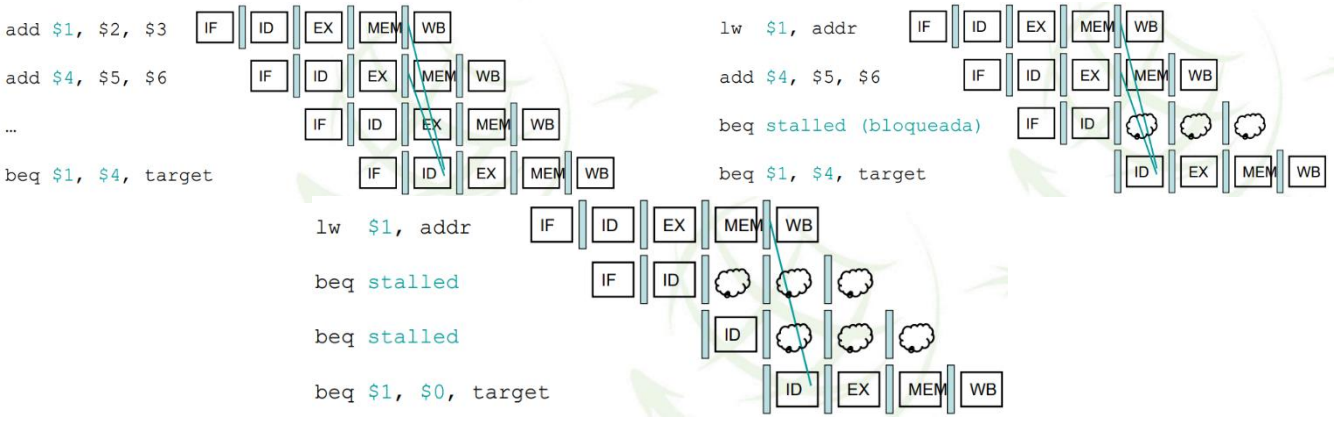
1. **Detener** el cauce hasta que se tome la decisión.
2. Aplicar una técnica de **salto retardado**.
3. Aplicar una técnica de **predicción de salto**.

SOLUCIÓN 1: DETENCIÓN DEL CAUCE

- Se detiene la carga de la siguiente o siguientes instrucciones hasta que el resultado se compute.

La cantidad de ciclos de bloqueo depende de qué es destino el/los registros de comparación de salto:

- Si es destino de una instrucción **ALU** que se ejecuta **2 o 3 ciclos atrás** ⟶ se pierden **0 ciclos** (se puede resolver con forwarding).
- Si es destino de una instrucción **ALU** que se ejecuta **1 ciclo atrás** o de una de **carga** que se ejecuta **2 ciclos atrás** ⟶ se pierde **1 ciclo**.
- Si es destino de una instrucción de **carga** que se ejecuta **1 ciclo atrás** ⟶ se pierden **2 ciclos**.



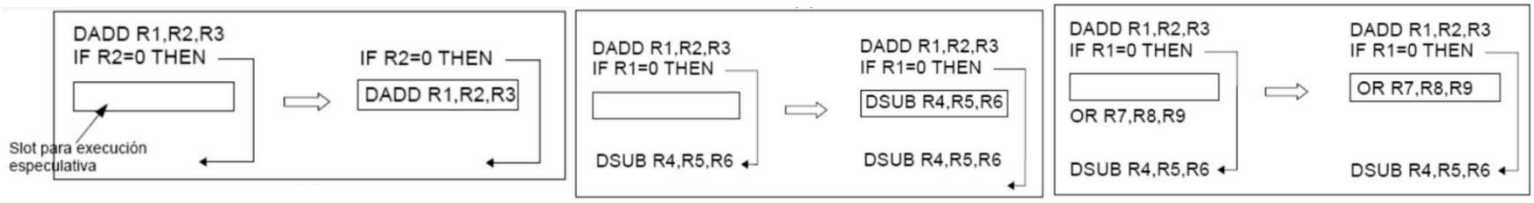
× Si el **pipeline es profundo** el **coste** de la parada sería demasiado **alto**, por lo que predecir el resultado puede ser una mejor opción.
  - En general, el pipeline del **MIPS** predice los saltos como **no tomados** siempre, así que carga las siguientes instrucciones después del salto y las va ejecutando. Si al final la predicción resultó ser **incorrecta** se producirán **paradas**.

SOLUCIÓN 2: DECISIÓN RETARDADA

○ El compilador mueve varias instrucciones a las **ranuras** que quedan bajo el salto (en el caso del MIPS, 2 como máximo), siempre que estas instrucciones **no guarden relación con el salto**. Así **aprovecha** el tiempo necesario para decidir si se realiza el salto.
  ↳ Si no hay instrucciones que puedan moverse, el compilador introduce **paradas** bajo el salto (lo cual ocurre la mitad de las veces).

```
addi $t0, $t0, 17          beq $t2, $t3, dir
addi $t1, $t1, 84          addi $t0, $t0, 17
beq $t2, $t3, dir          addi $t1, $t1, 84
lw $t3, 300($t0)           lw $t3, 300($t0)
j fin                      j fin
dir: lw $t3, 400($t0)      dir: lw $t3, 400($t0)
fin: ...                   fin: ...
```

- Hay varias maneras de implementar la decisión retardada con 1 ranura en **función de la información disponible durante la compilación**:
  – Mover la **instrucción anterior al salto** a la ranura.
  – Si no se puede usar la solución anterior, copiar la **instrucción de destino de salto** a la ranura.
  – Mover la **instrucción de después del salto** a la ranura.
    ▸ La instrucción que se coloque en la ranura de salto **no puede cambiar el estado del sistema** (es decir, no puede realizar escrituras ni en registros ni en memoria) antes de que se decida si se salta o no.

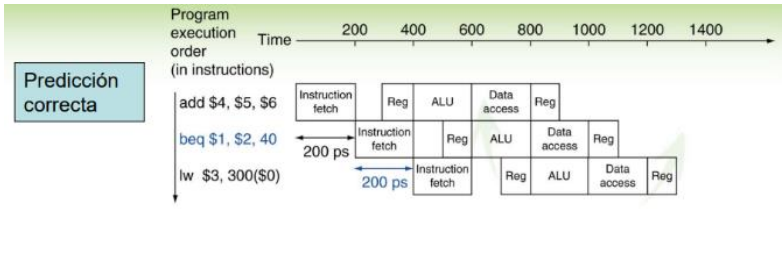


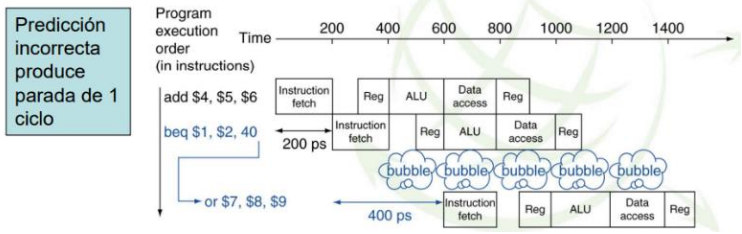
SOLUCIÓN 3: PREDICCIÓN DE SALTO

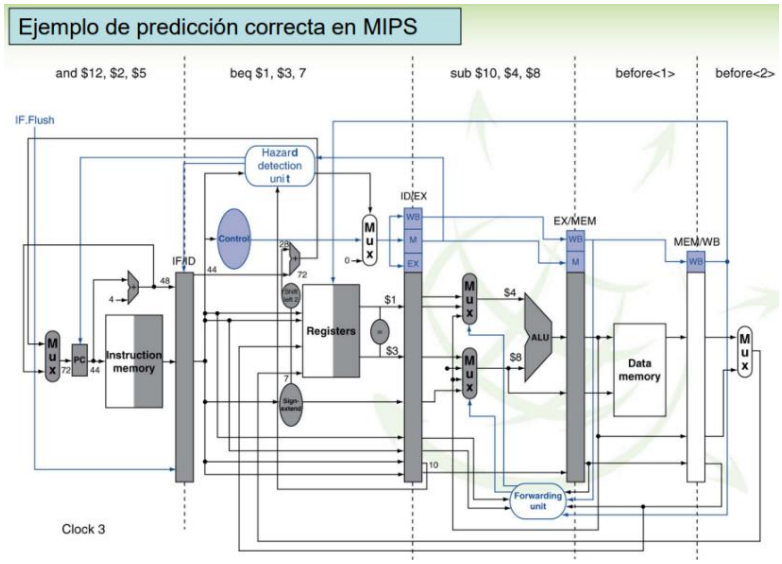
– Predicción de salto ESTÁTICA → predicción fija basada en el comportamiento típico del salto.
  ▷ En un salto asociado a un lazo o a un *if*, si es hacia atrás se predecirá que se toma, pues seguramente será para cerrar un lazo. Si es hacia delante se predice que no se toma.
– Predicción de salto DINÁMICA → mide el comportamiento real del salto mediante el hardware y va tomando una decisión adaptativa.
  ▪ Asume que en el futuro se seguirá la misma tendencia:
  Cuando falla en la predicción hay que **parar** mientras se carga la nueva instrucción y **actualizar** la historia.

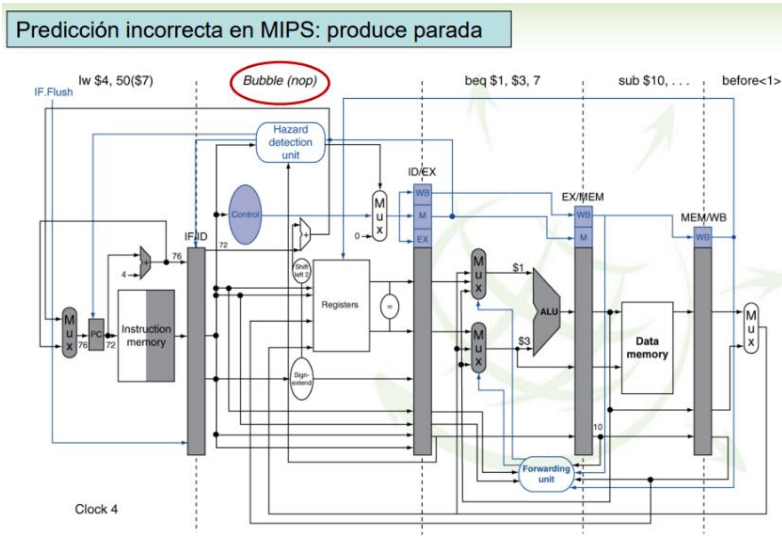
PREDICCIÓN DE SALTO FIJA

- En el MIPS de 5 etapas la predicción de **salto tomado** supone **1 ciclo de pérdida**, por lo que es **mejor** la predicción de **salto no tomado.**
  ▸ En otros pipelines esto puede cambiar.









PREDICCIÓN DE SALTO **DINÁMICA**

- En pipelines más **profundos** que el MIPS y en sistemas **superescalares**, la penalización por saltos es mayor, ya que hay varios pipelines implicados en la ejecución. Por tanto, se usan soluciones más complejas.

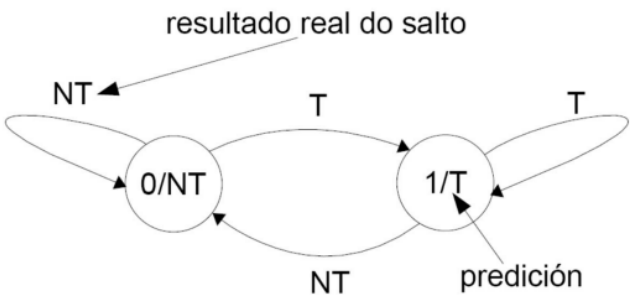
- Para las predicciones de salto dinámicas se necesita:
  - Un búfer de predicción de salto (una tabla de historia) en el que se almacene el resultado del salto: tomado (T) o no tomado (NT).
  - Indexado mediante las direcciones de instrucciones de salto recientes.
- Para ejecutar un salto:
  1. Chequear el búfer de predicción, pues se espera un resultado igual.
  2. Iniciar la búsqueda de la siguiente instrucción o la de destino de salto.
  3. Si la predicción es errónea hay que vaciar el pipeline y cambiar la predicción.

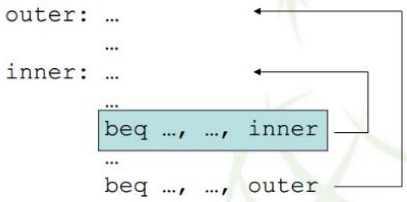
- Incluso usando un predictor hay que **calcular la dirección de salto**. Esto tiene una **penalización de 1 ciclo** en el MIPS cuando se predice **T**.
  - Una posible mejora es tener un BÚFER DE DESTINO DE SALTO que opere como **caché** de direcciones de destino y se **indexe mediante el PC** en la etapa IF.
    - ↳ Así, si se acierta y la instrucción es un salto que se predijo T, se puede buscar la instrucción de destino del salto inmediatamente.

PREDICTOR DE **1 BIT**

- Se almacena 1 bit indicando la predicción para la siguiente ocurrencia de la misma instrucción de salto:
  - Si la predicción es correcta ⟶ el bit no cambia.
  - Si la predicción no es correcta ⟶ el bit se complementa.
- ✓ Filtra **cambios puntuales**.
- ✗ No es capaz de **adaptarse** al comportamiento del programa.
- ✗ Con un único predictor de 1 bit los saltos de lazos internos se predicen erróneamente 2 veces:
  - ↳ Una posible mejora es hacer una **predicción separada para cada instrucción de salto** o usar un **predictor más complejo**.

- Predición 1 1 1 1 1 0 1 1 1
- Real T T T T 0 T T T T

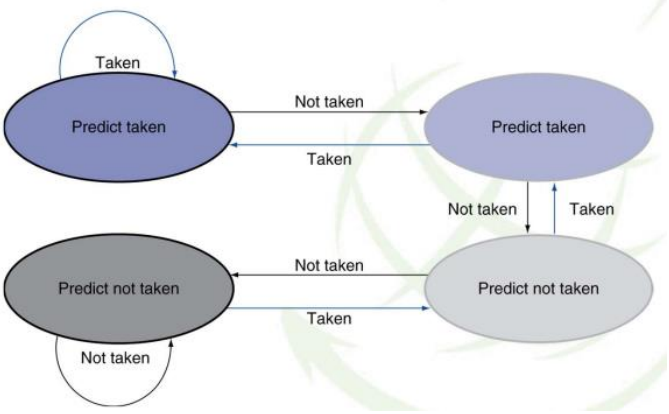


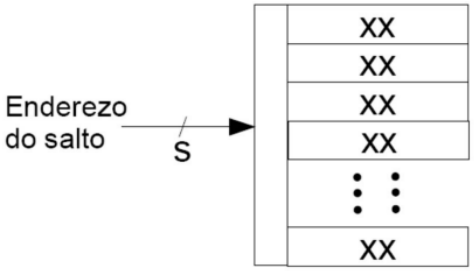


Predicción errónea como tomado en la última iteración del lazo más interno.
Entonces la próxima vez produce predicción errónea como no tomado en la primera iteración del lazo más externo.

PREDICTOR DE **2 BITS**

- Se almacenan 2 bits indicando la predicción para la siguiente ocurrencia de la misma instrucción de salto.
  - Los bits sólo cambian después de 2 predicciones erróneas.
- Hay diferentes implementaciones, pero todas se basan en considerar **2 estados** de predicción "**fuerte**" y otros **2** de predicción "**débil**".
- Usa una **caché** para almacenar las predicciones:
  - Se indexa como una caché ordinaria: la parte menos significativa de la dirección de la instrucción de salto (la etiqueta) se usa para extraer el estado de la predicción.
  - La probabilidad de que dos saltos tengan en común estos bits es muy baja, pero no imposible.
  - Cuando se resuelve el salto se almacena el nuevo estado en la misma posición.
- ✓ Filtra **cambios esporádicos**.
- ✓ Evita el 50% de los fallos que se tendrían al usar un predictor de 1 bit para secuencias de saltos tipo T NT T NT T NT….

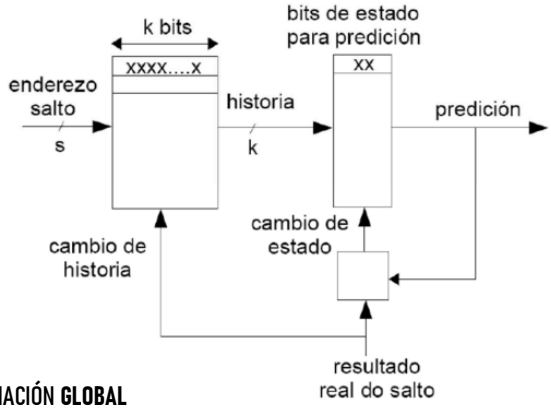




PREDICTOR BASADO EN INFORMACIÓN **LOCAL**

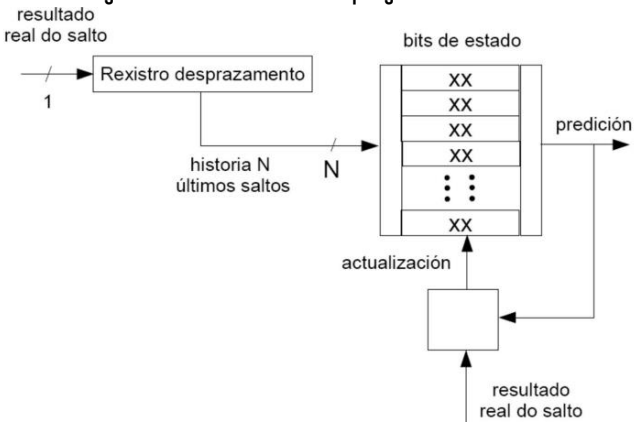
- Tiene en cuenta la historia del comportamiento **de un mismo salto** para tomar la decisión.
  1. Se almacena el resultado de cada salto correspondiente a sus últimas k ocurrencias.
  2. Con estos bits se accede a los bits de estado que informan de la predicción.
  3. Una vez conocido el resultado real del salto, se actualiza el contenido de la memoria de historia y el filtro de decisión.

✓ Usa **dos filtros de decisión**, y así evita el 100% de los fallos que sucederían al usar un predictor de 1 bit para secuencias tipo T NT T NT…



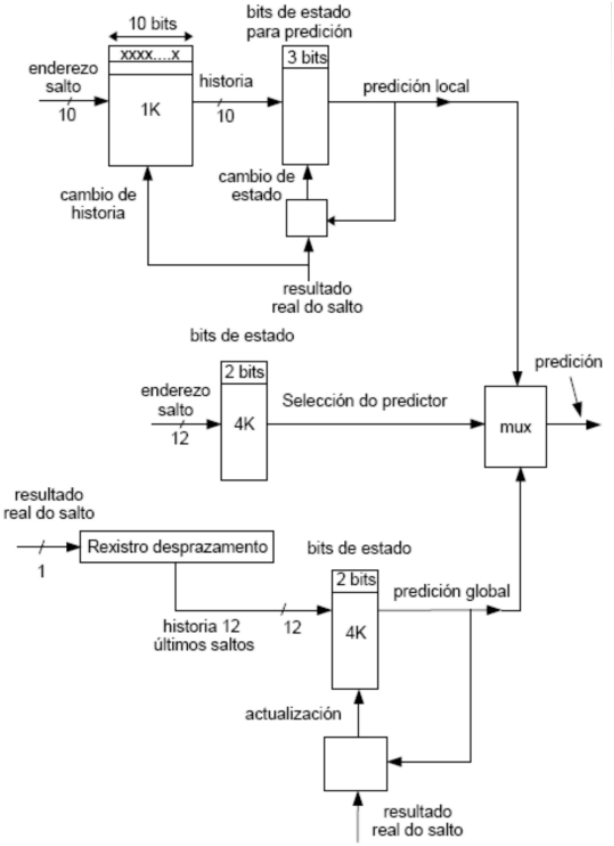
PREDICTOR BASADO EN INFORMACIÓN **GLOBAL**

- Tiene en cuenta la historia del comportamiento de **las últimas N instrucciones de salto** para tomar una decisión.
  - Se usa un **registro de desplazamiento** para almacenar el **resultado de los saltos**.
  - El contenido del registro indexa una **memoria** que guarda los **bits de estado** correspondientes al filtro de decisión.



PREDICTOR **COMBINADO**

- Combina un predictor **global** y otro **local**.
- La **tasa de acierto** de los saltos en procesadores reales depende del **tipo de programa** y suele estar entre 85% y 99%.



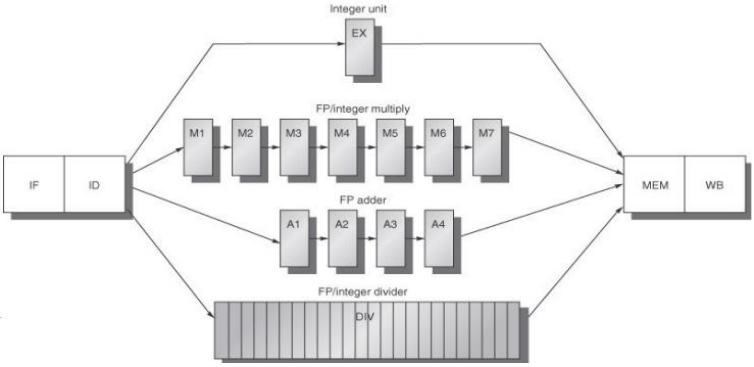
Ejemplo:

- P. global con memoria de 4K entradas almacenando 2 bits de estado por entrada
- P. local con memoria de 1K entradas. Con la historia del salto se indexa una memoria de 1K entradas con 3 bits de estado por cada entrada.
- Selección adaptativa de tipo de predictor dependiendo del valor de 2 bits de estado
- Tamaño total de 29Kbits de almacenamiento

EXTENSIÓN A OPERACIONES MULTICICLO

- Las instrucciones en punto flotante tienen la **misma segmentación** que las enteras, pero con las siguientes modificaciones:
  - La etapa EX tiene **diferente latencia** dependiendo de la **instrucción**.
  - Existen **diversas unidades** funcionales para **cada tipo de operación**.

- Se añade un **banco de registros** separado para operaciones en **punto flotante**.
- Se añade una **unidad de multiplicación** segmentada de **7 etapas** en **enteros** y **punto flotante**.
- Se añade una **unidad de suma** de **4 etapas** en punto flotante.
- Se añade una **unidad de división no segmentada** que requiere **varios ciclos** reutilizando la **misma etapa**.
  ↳ En esta unidad no se puede ejecutar una instrucción por ciclo.



- Chequeo de **riesgos estructurales** ⟶ hay que detectarlos en la unidad de DIV y en la etapa WB. Para detectarlos:
  - Parar el paso de la instrucción a EX si es una división y hay otra división en ejecución.
  - Parar la emisión si se detecta que van a llegar a WB más instrucciones que puertas de acceso de escritura al banco de registros.
- Chequeo de **riesgos por dependencias RAW** ⟶ similar al caso de pipeline simple con una única etapa de ejecución, pero con **más casos** posibles.
- Chequeo de **riesgos por dependencias WAW** ⟶ son **poco comunes**. Para detectarlos:
  - Determinar si hay una instrucción en una etapa posterior a ID que tiene como destino el mismo registro que la instrucción en ID, pero que haría la escritura del registro un número de ciclos posterior.
    Si sucede, parar el paso a EX de la instrucción que está en una etapa posterior a ID.

EXCEPCIONES

- Las EXCEPCIONES se producen por diferentes **eventos en la CPU** (código de operación indefinido, overflows, syscalls, ...) y requieren que el programa se ejecute de nuevo **desde el punto en que se produjo** la excepción.
- Para poder restablecer el estado es necesario que se haya **guardado** el contenido de los **registros** y de la **memoria**.
  - Las excepciones que permiten recuperar un estado anterior por completo se llaman EXCEPCIONES PRECISAS.
  - Es complicado gestionarlas sin sacrificar el rendimiento, ya que reestablecer el estado del sistema significa almacenar información y perder ciclos.

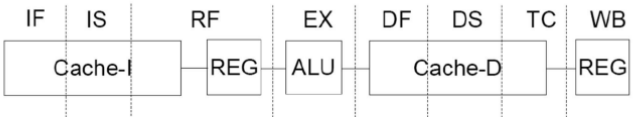
  Posibles soluciones:
  - Incorporar un **banco de registros adicional** para guardar los valores producidos por **cada instrucción**.
  - Utilizar una **rutina software** que permita **repetir la ejecución** de instrucciones desde la que produjo la excepción.
  - **Parar la emisión de instrucciones** si en la etapa EX se detecta **riesgo de excepción**.
- Si suceden múltiples excepciones anidadas, se gestiona **la más antigua primero**.

```
DIV.D F0,F2,F4      (División en doble precisión)
ADD.D F10,F10,F8    (Suma en doble precisión)
SUB.D F12,F12,F14   (Resta en doble precisión)
```

- Estas instrucciones son independientes y se pueden ejecutar a la vez.
- La división requiere varios ciclos de EX y acabará después de la suma y la resta.
- Podría producirse una excepción en la ejecución de la división (por ejemplo, un *overflow*) después de que hayan terminado suma y resta, cambiando así el estado del punto de excepción.
- Para resolver la excepción se podría aplicar cualquiera de las 3 técnicas mencionadas en la página anterior.

CASO PRÁCTICO: MIPS R4000

- 8 etapas de pipeline
- Las instrucciones se leen de memoria caché y los datos se escriben a memoria caché -> se segmentan los accesos a caché en varias etapas.
- **RF:** se chequea la etiqueta de la dirección de línea caché para comprobar que la lectura se corresponde con la línea correcta. También se hacen la decodificación de la instrucción, lectura de registros y detección de conflictos **como en la etapa ID del pipeline del MIPS** de 5 etapas.
- **EX**: **Se evalúa la condición de los saltos condicionales y se calcula la dirección de dichos saltos** además de hacer las operaciones propias de una ALU.
- **TC**: chequeo de la etiqueta de la dirección de la línea caché para determinar si la lectura/escritura fue un acierto.



- Diferencias con el pipeline MIPS de 5 etapas:
  - Más caminos posibles de adelantamiento, por ejemplo, desde DS, TC y WB a EX.
  - Más latencia para las instrucciones dependientes de un Load.
  - Más latencia en la resolución de saltos.
- **Ejemplo:** se produce adelantamiento de LD a DADD.
  - Si DADD y DSUB no dependiesen de LD, se produciría adelantamiento de LD a OR.

LD R1	IF	IS	RF	EX	DF	DS	TC	WB
DADD R2,R1		IF	IS	RF	Stall	Stall	EX	DF
DSUB R3,R1			IF	IS	Stall	Stall	RF	EX
OR R4,R1				IF	Stall	Stall	IS	RF

- **Caso de salto tomado (fallo de predicción):**
  - Si la instrucción del slot (ranura de retardo) no es coherente deberá anularse antes de que cambie el estado del sistema.
  - Al ser predicción de salto no tomado cuando se resuelve el salto en la etapa EX se anularán las dos instrucciones emitidas IS+2 e IS+3 produciendo pérdida de 2 ciclos.

BR	IF	IS	RF	EX	DF	DS	TC	WB
Slot		IF	IS	RF	EX	DF	DS	TC
IS+2			IF	IS	NOP	NOP	NOP	NOP
IS+3				IF	NOP	NOP	NOP	NOP
Destino Salto					IF	IS	RF	EX

CREENCIAS ERRÓNEAS

- La segmentación es **fácil** → la idea básica es simple, pero la dificultad está en los detalles.
- La segmentación es **independiente de la tecnología**:
  - **Más transistores** hacen que sean posibles más técnicas complejas, por ejemplo, para detectar saltos, lo cual reduce el **tiempo de ejecución**.
  - El **diseño del ISA** relacionado con segmentación necesita tener en cuenta las **tendencias tecnológicas**.

DIFICULTADES

- Un diseño de ISA poco cuidadoso puede hacer la segmentación más complicada.

Ejemplos:

- **Conjuntos de instrucciones complejos** (VAX, IA-32) → sobrecoste considerable para hacer que la segmentación funcione.
- **Modos de direccionamiento complejos**.
- **Saltos retardados** → los pipelines avanzados tienen slots de retardo grandes.

CONCLUSIONES

- El hardware del **camino de datos** y la **lógica de control** se complican con la segmentación:
  - Caminos de adelantamiento.
  - Lógica para detección de conflictos de datos.
  - Lógica para predicción de saltos.
- La segmentación **mejora la productividad**, pero **no el tiempo de ejecución de cada instrucción** (latencia).
- **Incrementar la longitud del cauce** segmentado (el número de etapas de segmentación) **no tiene por qué incrementar el rendimiento**:
  - Los conflictos de datos hacen que incrementando la longitud se incremente el tiempo que se invierte en cada instrucción.
  - Los conflictos de control provocan que un incremento en la longitud ralentice los saltos.
  - La sobrecarga por los registros de segmentación puede limitar el aumento de la frecuencia de reloj.