Tema 3: Segmentación de Instrucciones

- **Objetivos**
- **Referencias**
- **Etapas de un cauce**
- **Parones**
  - **estructurales**
  - **por dependencia de datos**
  - **por dependencia de control**
- **Modificación de un cauce para operaciones multiciclo**
- **Ejemplo: el MIPS R4000**

Objetivos:

- **Conocer las características y el funcionamiento de un cauce de instrucciones simple**
- **Comprender las dificultades que surgen a la hora de segmentar la ejecución de instrucciones**
- **Conocer algunas de las soluciones que se aplican para resolver esas dificultades**

Referencias:

- Este tema está sacado de forma prácticamente íntegra del Hennessy-Patterson (la 2ª edición) (casi todas las figuras son de él)
- Para algunas transparencias se han usado como modelo las de Al Davis
- Como complemento se puede mirar el Kai Hwang

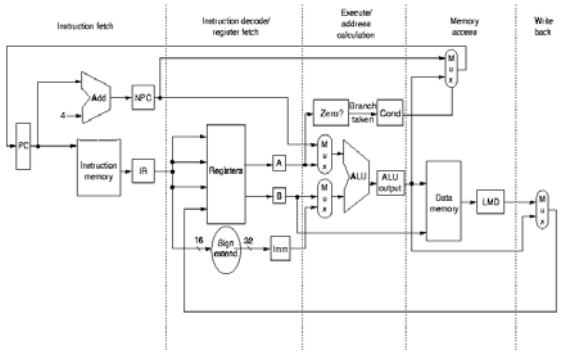
Etapas del cauce:

- **Búsqueda (IF)**
  - se accede a memoria a por la instrucción
  - se incrementa el CP
- **Decodificación / Búsqueda de operandos (ID)**
  - se decodifica la instrucción
  - se accede al banco de registros a por los registros operandos
  - se calcula el valor del operando inmediato con el signo extendido (por si hace falta más adelante)
- **Ejecución / Dirección efectiva (EX)**
  - si es una instrucción de proceso, se ejecuta en la ALU
  - si es un acceso a memoria, se calcula la dirección efectiva
  - si es un salto, se calcula el destino, y si se toma o no

Etapas del cauce (cont):

- **Acceso a memoria / terminación del salto (MEM)**
  - si es un acceso a memoria, se accede
  - si es un salto, se almacena el nuevo CP
- **Almacenamiento (WB)**
  - se almacena el resultado (si lo hay) en el banco de registros

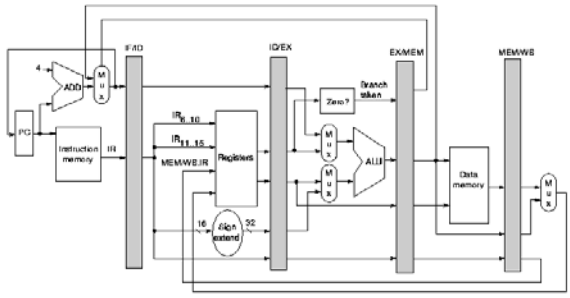
Máquina básica



Ejemplo de ejecución de instrucciones

Ciclo	1	2	3	4	5	6	7	8	9
Inst i	IF	ID	EX	MM	WB				
Inst i+1		IF	ID	EX	MM	WB			
Inst i+2			IF	ID	EX	MM	WB		
Inst i+3				IF	ID	EX	MM	WB	
Inst i+4					IF	ID	EX	MM	WB

Máquina más realista I



Ventajas de la segmentación:

- **Principalmente: el mejor rendimiento**
  - **–ganancia en rendimiento = Nº etapas del cauce (a reloj constante)**
  - **–pero: ¡cuidado! sólo es el máximo teórico**
- **Es completamente Hardware =>mejor que Software**
- **El modelo de programación no cambia**
  - **–pero: veremos que afecta al programador (sobre todo al compilador)**
- **PERO: hay que tener en cuenta que el control también se vuelve más complejo**

Parones: situaciones que impiden a la siguiente instrucción que se ejecute en el ciclo que le corresponde

Hay 3 tipos:

Estructurales
provocados por conflictos por los recursos

Por dependencia de datos (parones por datos)
ocurren cuando dos instrucciones se comunican por medio de un dato (ej.: una lo produce, y la otra lo usa)

Por dependencia de control (parones por control)
ocurren cuando la ejecución de una instrucción depende de cómo se ejecute otra (ej.: un salto, y los dos posibles caminos)

Supongamos un parón en la instrucción i + 2

Ciclo	1	2	3	4	5	6	7	8	9	10
Inst i	IF	ID	EX	MM	WB					
Inst i+1		IF	ID	EX	MM	WB				
Inst i+2			(parón)	IF	ID	EX	MM	WB		
Inst i+3					IF	ID	EX	MM	WB	
Inst i+4						IF	ID	EX	MM	WB

Efecto: el rendimiento disminuye del máximo teórico

(ganancia de rendimiento, a reloj constante)

Máximo teórico: número de etapas del cauce

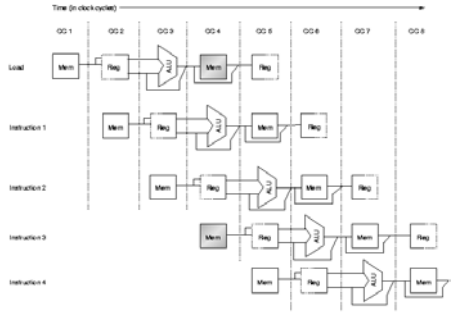
Con parones: (CPI no seg / CPI seg) =

= (CPI no seg / (CPI ideal + Nº parones por instrucción)) =

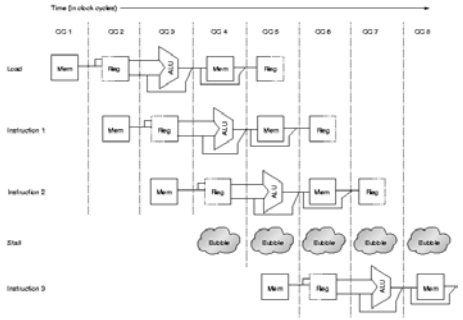
= (CPI no seg / (1 + Nº parones por instrucción)) =

= (Nº etapas del cauce / (1 +Nº parones por instrucción))

Ejemplo: una instrucción de carga seguida de otras varias. La máquina tiene un único puerto de memoria



Efecto: el *load* y la instrucción 3 tienen un conflicto: compiten por el puerto de memoria. El parón en el cauce:



- **Posibles limitaciones en los recursos**

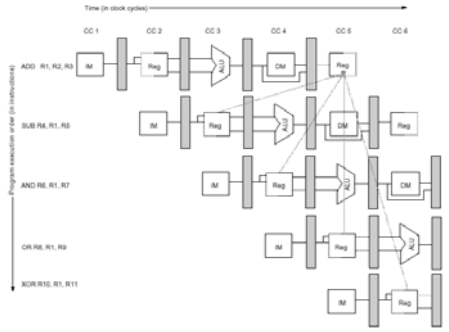
(=> parones estructurales)

- **1 puerto de memoria en lugar de 2**
  (=> fetch - acceso mem)
- **1 ALU en lugar de 2**
  (=> incremento PC - instrucción de proceso)
- **banco de registros con pocos puertos**
  (=> lectura - escritura)

- **Solución a los parones estructurales:**
  añadir más recursos

PERO (¡lo de siempre!) : hay que encontrar un equilibrio entre rendimiento y coste

Ejemplo de código con dependencia de datos



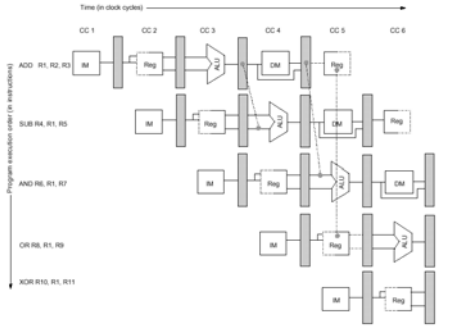
Solución: adelantamiento de resultados (bypassing, forwarding)

•La instrucción 1, que calcula el nuevo valor de R1, lo tiene listo al final del ciclo 3

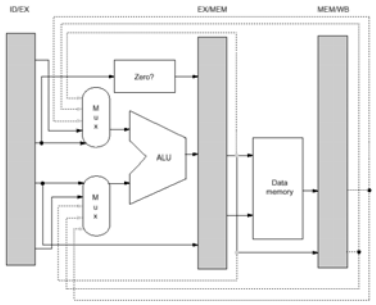
•La instrucción 2, que lo lee, en realidad necesita que esté listo al principio del ciclo 4

•Se puede modificar la ALU con una realimentación, para que el nuevo valor de R1 vaya de la salida a la entrada (además, sigue siendo enviado al banco de registros en la fase WB

La ejecución queda ahora:



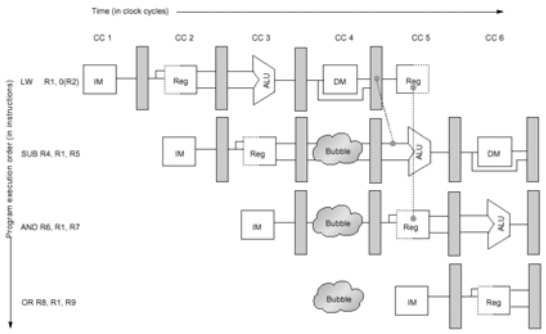
Máquina más realista II



Observaciones al adelantamiento de resultados:

- **Se deben habilitar caminos de realimentación de todas a todas las unidades, p.ej.: de la ALU a memoria, de la ALU a la ALU, etc**
  las dependencias se dan entre instrucciones de todos los tipos
- **Hay que tener en cuenta que el control se complica aún más:**
  hay que decidir cuándo se realimenta un resultado, controlar la realimentación, etc
- **Aún así, puede no ser suficiente**
  por ejemplo, si una instrucción de memoria genera un dato para una de ALU:

```
lw r1,0(r2)
sub r4,r1,r5
```

Clasificación de las dependencias de datos

- **Lectura después de Escritura (LDE, RAW)**
  – la que hemos visto hasta ahora: una instrucción genera un dato que lee otra posterior
- **Escritura después de Escritura (EDE, WAW)**
  – una instrucción escribe un dato después que otra posterior
  – en una máquina segmentada simple sólo se da si se deja que las instrucciones se adelanten unas a otras
- **Escritura después de Lectura (EDL, WAR)**
  – una instrucción modifica un valor antes de que otra anterior que lo tiene que leer, lo lea
  – tampoco se puede dar en nuestro cauce

Ejemplos:

LDE:
add **r1**,r2,r3
add r4,**r1**,4

EDE:
lw **r1**,0(r2)
add **r1**,r2,r3

(si modificamos el cauce: un acceso a memoria son dos ciclos, y las instrucciones se pueden adelantar)

EDL:
sw **r1**,0(r2)
add **r1**,r3,r4

(también hacen falta modificaciones: r1 se lee al ir a escribir (2 ciclos), y las instrucciones se pueden adelantar)

Aprovechamiento del cauce : reordenación de instrucciones

Se trata de calcular:

```
a = b + c
d = e - f
g = a + d
```

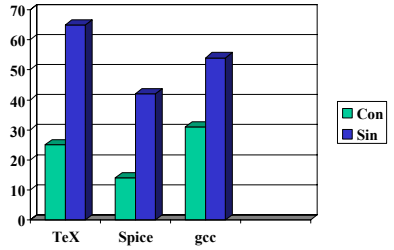
Código inicial

```
lw rb,b
lw rc,c
add ra,rb,rc
lw re,e
lw rf,f
sub rd,re,rf
add rg,ra,rd
sw rg,g
```

Código reordenado

```
lw rb,b
lw rc,c
lw re,e
lw rd,f
add ra,rb,rc
sub rd,re,rf
add rg,ra,rd
sw rg,g
```


Eficacia de la reordenación:



Porcentaje de las cargas que provocan un parón

Hasta ahora: el CP se actualiza en la fase de búsqueda

Pero: si en el cauce entra un salto, hay dos instrucciones que pueden ser la siguiente que hay que buscar:

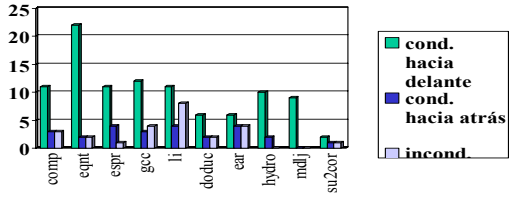
la que está a continuación del salto
la que es el destino del salto

PROBLEMA: necesitamos saber ya cuál es la siguiente instrucción que vamos a meter en el cauce, pero:

si es un salto condicional, aún no se sabe cuál de las dos es (se sabe en la fase EX)
en cualquier caso, el contador de programa incrementado se almacena en la fase (WB)

Esto sólo sería importante si el porcentaje de saltos tomados (o no) que aparecen en el código es significativo

Porcentaje de instrucciones que son saltos



En promedio, para enteros:

13% de instrucciones son saltos condicionales hacia delante
3% de instrucciones son saltos condicionales hacia atrás
4% son saltos incondicionales

Todo esto depende de las optimizaciones que use el compilador, pero como orientación sirve.

Resumiendo: 1 de cada 5 instrucciones es un salto

Conclusión: sí hay que tenerlos en cuenta

¿Qué se puede hacer?

Plan A: en cuanto veamos que una instrucción es un salto, paramos el cauce, hasta que sepamos a dónde salta

Sabemos si es un salto en la fase ID;
Conocemos el destino en MEM (usando realimentación)
Total: 3 ciclos de parón

Si el porcentaje de saltos es del 30%, y el CPI ideal es 1, la máquina tiene una ganancia de aprox. el 50% del máximo teórico

Conclusión: No interesa

Plan B:

1º: intentamos averiguar si el salto se toma o no en una fase anterior

2º: vamos calculando el destino lo antes posible, por si acaso hace falta

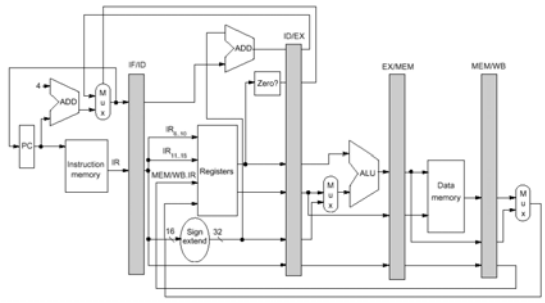
Por ejemplo, en la fase de decodificación (ambas)

Implicaciones:

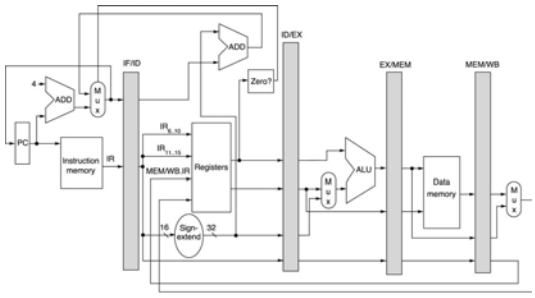
el parón es menor (1 ciclo)

el coste es mayor (1 sumador)

Máquina más realista III



Máquina más realista IV



Formas de tratar la presencia de un salto en el flujo de instrucciones:

no hacer nada

es el esquema más sencillo (y barato)
también es el que peor rendimiento da

suponer que el salto no se va a tomar

se siguen buscando instrucciones a continuación del salto
en caso de que se tome, se invalidan

suponer que el salto se va a tomar

se calcula el destino, y se empieza a buscar por él
en nuestra máquina no es viable: se conocen el sentido y el destino a la vez => no es suposición, es certeza

retardar la ejecución del salto

ejecutar la instrucción siguiente, tanto si el salto se toma, como si no

Suponiendo que el salto no se toma

si en realidad no se toma:

i (salto)	IF	ID	EX	MM	WB		
i+1		IF	ID	EX	MM	WB	
i+2			IF	ID	EX	MM	WB
i+3				IF	ID	EX	MM	WB

si en realidad sí se toma:

i (salto)	IF	ID	EX	MM	WB		
i+1		IF	**nada**	**nada**	**nada**	**nada**	
destino			IF	ID	EX	MM	WB
d+1				IF	ID	EX	MM	WB

•Si el salto no se toma, el cauce no se toca.

•Si se toma, se anula la ejecución de la instrucción siguiente al salto, es decir, se borran los latches entre fases.

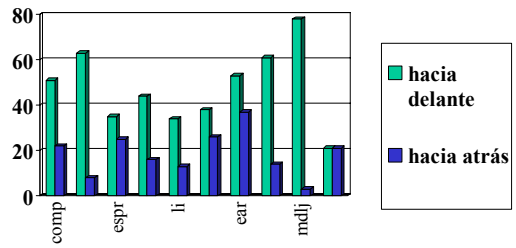
Rendimiento:

Si el salto en realidad no se toma: penalización 0
Si el salto sí se toma: penalización 1 ciclo

Con respecto a la opción de no hacer nada, en el caso peor el rendimiento es el mismo

Sería interesante saber qué porcentaje de saltos no se toma

Porcentaje de saltos tomados sobre el total de ejecutados



En promedio:

- el 62% de los saltos ejecutados en programas de enteros se toman
- en programas de coma flotante, el porcentaje sube al 70%

Por lo tanto:

- sería más interesante la opción de suponer que el salto se va a tomar (es más probable)

PERO:

- para buscar la instrucción destino del salto, es necesario calcularla
- en nuestra máquina, este valor se obtiene en la fase ID
- para entonces, también sabemos si el salto se toma o no =>ya no es necesario suponer
- en otras máquinas puede ser útil

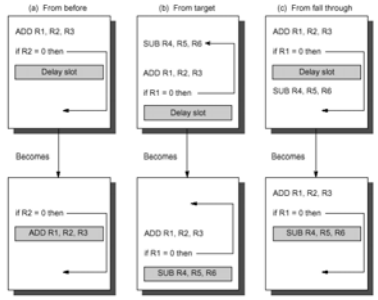
Cuarta opción: ejecución retardada del salto

Cuando se llega a un salto, la siguiente instrucción se ejecuta, tanto si el salto se toma como si no ("delay slot")

Implicaciones:

- a la hora de generar código es necesario tener esto en cuenta => cambio en la compilación / programación
- el número de instrucciones situadas a continuación de un salto que se ejecutan puede variar con la arquitectura
- a veces no es posible utilizar el "*delay slot*" para hacer trabajo útil => nop's

Posibilidades para rellenar el "delay-slot"



Requisistos para aprovechar el "delay - slot"

Si las instrucciones se toman de antes del salto:

- no debe haber dependencia entre ellas y el salto
- siempre mejora el rendimiento

Si las instrucciones se toman del destino del salto:

- no debe afectar al programa el que se ejecuten si el salto no se toma
- puede haber ocasiones en las que el código no se copie, sino que se duplique (al destino se llega desde varios puntos)
- mejora el rendimiento si el salto se toma

Si las instrucciones se toman a continuación del salto:

- no debe afectar al programa el que se ejecuten si el salto se toma
- mejora el rendimiento si el salto no se toma

PROBLEMA:

Puede ser difícil encontrar instrucciones a continuación del salto, o del destino, que no afecten al programa si se ejecutan cuando el salto se toma o no, respectivamente.

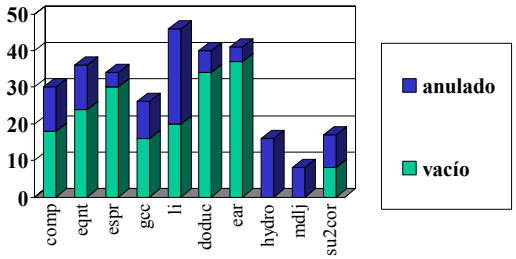
SOLUCIÓN:

Saltos con anulación (o cancelación):

se predice si se van a tomar o no (por el compilador)
se rellenan sus "delay-slots" con instrucciones según corresponda
si el salto se comporta como se espera => todo bien
si no => se anula la ejecución de los "delay-slots"

Con esto se relajan los requisitos sobre las instrucciones que van en los "delay-slots"

Ejecución de los "delay-slots", en porcentaje sobre el total de los saltos condicionales



Observaciones sobre el uso de "delay-slots":

- en promedio, el 30% se desperdician (35% en enteros, 25% en coma flotante)
- o lo que es igual, hasta un 70% se aprovechan
- sin embargo, en nuestra máquina, los saltos sólo tienen 1 "delay-slot"; en otras, con cauces más largos, pueden tener más
- esto puede hacer difícil el rellenarlos con trabajo útil
- lo mismo ocurre con procesadores superescalares (ya los veremos en detalle: ejecutan varias instrucciones a la vez)

Resumiendo, los "delay-slots" tienen bastante utilidad en cauces sencillos, pero en cauces más complejos su efectividad es poca

En otras palabras: ya no se usan

PROBLEMA: Unas instrucciones tardan más en ejecutarse que otras

Ejemplo: una multiplicación y un desplazamiento lógico

Si el ciclo de reloj se ajusta a la más lenta, con las rápidas (y con las demás etapas del cauce) se está perdiendo tiempo

Si el ciclo de reloj se ajusta a la más rápida, a las lentas no les da con un ciclo

SOLUCIÓN: Se segmenta también la fase de ejecución

Según esto, nuestra máquina consta de:

- hardware para ejecutar las fases que son iguales en todas las instrucciones (IF, ID, MEM y WB)
- además, una serie de unidades funcionales para ejecutar cada tipo de instrucción
  - operaciones comunes de enteros
  - multiplicación de enteros / coma flotante
  - suma de coma flotante
  - división enteros / coma flotante
- estas unidades de ejecución van a estar a su vez segmentadas

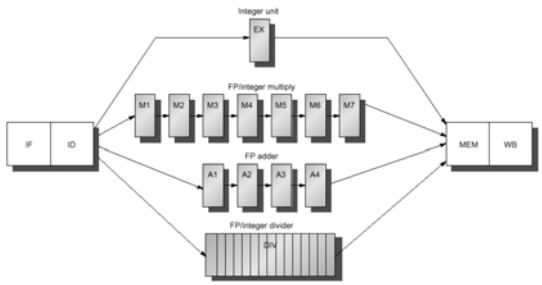
El flujo de instrucciones a través del cauce es ahora:

- las instrucciones pasan por las fases IF e ID
- a continuación, cada instrucción pasa a la unidad funcional que la ejecute
- las instrucciones terminan en su unidad funcional , y recorren las fases MEM y WB

Es necesario modificar la máquina:

- añadiendo latches entre las fases de las unidades funcionales
- posibilitando el flujo desde ID hasta cualquier unidad funcional
- por supuesto, el control se complica (pero no demasiado)

El nuevo cauce



Ejemplo de ocupación del nuevo cauce

MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MM	WB		
LD			IF	ID	*EX*	**MM**	WB				
SD				IF	ID	*EX*	*MM*	WB			

- Se supone que las instrucciones no tienen dependencias entre sí
- Las fases en *cursiva* indican cuándo se necesitan los datos
- Las fases en **negrita** indican cuándo se generan los resultados

Complicaciones que aparecen en el nuevo modelo:

- **las posibilidades de dependencia (=>parones) aumentan**
  - –algunas unidades funcionales están segementadas y pueden tener varias instrucciones en ejecución
  - –además, varias unidades funcionales pueden estar activas simultáneamente
- **y los parones además serán más largos puesto que el cauce es más largo (dependiendo de la unidad funcional)**
- **las instrucciones no tienen por qué terminar en orden**
  - –cuidado con las dependencias EDE
  - –puede haber varias instrucciones en la fase WB a la vez (conflicto estructural)
- **sin embargo, las dependencias EDL siguen sin poderse dar**

Ejemplo:

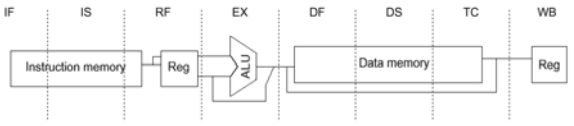
CICLO	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
ld f4,0(r2)	IF	ID	EX	MM	WB											
multd f0,f4,f6		IF	ID	**P**	M1	M2	M3	M4	M5	M6	M7	MM	WB			
add f2,f0,f8			IF	**P**	ID	**P**	**P**	**P**	**P**	**P**	**P**	A1	A2	A3	A4	MM
sd f2,0(r2)				**P**	IF	**P**	**P**	**P**	**P**	**P**	**P**	ID	EX	**P**	**P**	**P**

- **Hay dependencias RAW de cada instrucción con la anterior => parones**
- **La instrucción 4 necesita el resultado de la 3, pero para su fase MEM, por lo que puede pasar a la fase EX en el ciclo 13**
- **Hay dependencias estructurales: en el ciclo 16, la instrucción 4 no puede pasar a la fase MM porque la instrucción 3 está ahí.**

El MIPS R4000

- **Representativo de arquitecturas que aparecieron a partir de los 80**
- **Es una máquina de 64 bits**
- **Implementa el repertorio MIPS-3**
- **Tiene un cauce con gran número de etapas (supersegmentación)**
- **Esto le permitía utilizar una frecuencia alta (100-200 MHz)**
- **Utilizado por NEC, Nintendo, Silicon Graphics, Sony...**

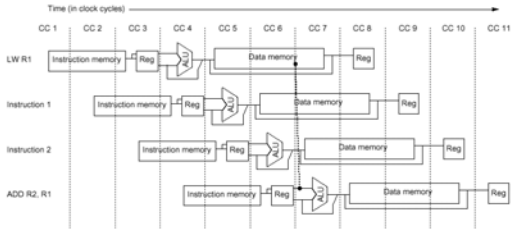
El cauce del R4000



IF: fetch, fase 1
IS: fetch, fase 2
RF: Decod, búsqueda de operandos, comprobación de dependencias

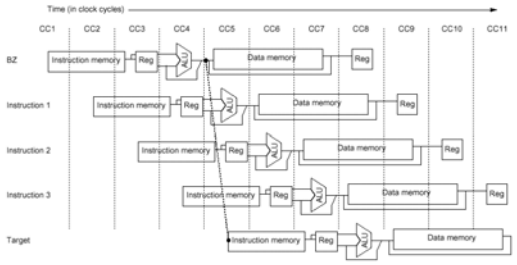
EX: ejecución, cálculo de dir. efectiva, cálculo/comprobación de salto
DF: acceso a mem, fase 1
DS: acceso a mem, fase 2
TC: comprobación de acierto de caché
WB: almacenamiento

Ejecución de un 'load' en el R4000



El retardo de una carga son dos ciclos (el posible parón, si la siguiente instrucción necesita el dato)

Ejecución de un salto en el R4000



Cuál es el destino del salto, y si se toma, se sabe al fin de la fase EX, por lo tanto, la penalización son 3 ciclos

Los saltos en el R4000

- **El R4000 permite el aprovechamiento de un 'delay-slot' (aunque haya 3)**
- **Para los otros dos, se predice que el salto no se va a tomar**
  - –ejecuta 2 instrucciones de la continuación del salto
  - –si el salto se toma, se anula su ejecución
- **Dispone de instrucciones de 'salto probablemente tomado'**
  - –equivalente a nuestro salto con anulación

Las operaciones de coma flotante

- **Tiene 3 unidades funcionales:**
  - **división**
  - **multiplicación**
  - **suma**
- **La unidad de suma se usa también al final de la multiplicación y la división**
- **La duración de las operaciones va de 2 ciclos para una negación a 112 ciclos para una raíz cuadrada**

Causas de parones en el R4000