Plataformas de soporte computacional: arquitecturas avanzadas, sesión 2

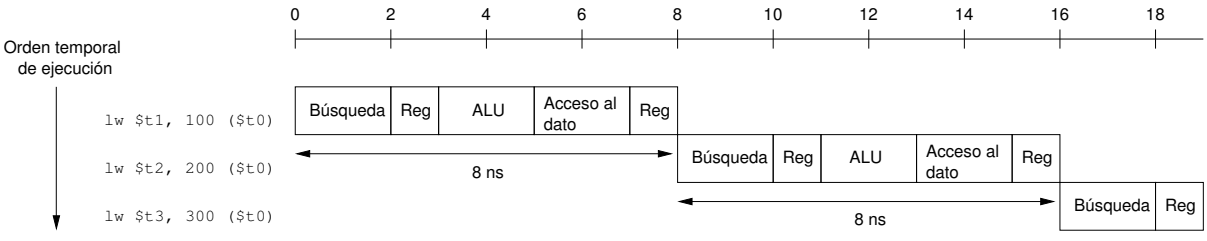
Diego R. Llanos, Belén Palop
Departamento de Informática
Universidad de Valladolid
{diego,b.palop}@infor.uva.es

Índice

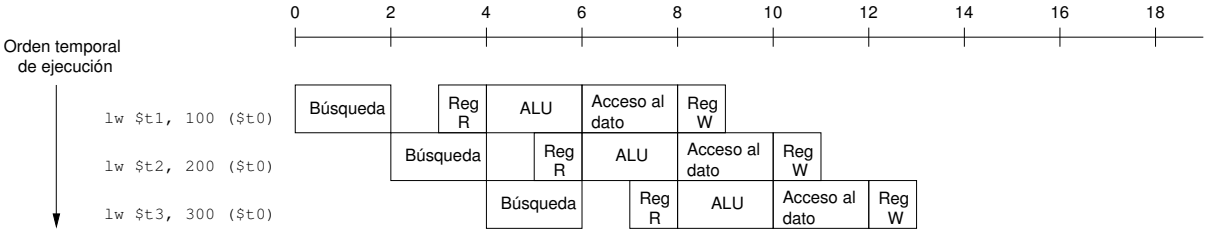
1. Segmentación

Fundamentos de la segmentación

- Tener una implementación multiciclo permite acelerar la ejecución de instrucciones, aprovechando que algunas instrucciones acaban antes que otras.
- Sin embargo, en cada etapa sólo está activa una unidad funcional.
- La segmentación busca aprovechar todas las etapas a la vez para ir progresando en la ejecución de varias instrucciones simultáneamente.



Ejecución multiciclo



Ejecución multiciclo segmentada

(La escritura del banco de registros se produce en la primera mitad del ciclo, y la lectura en la segunda)

Limitaciones de la segmentación

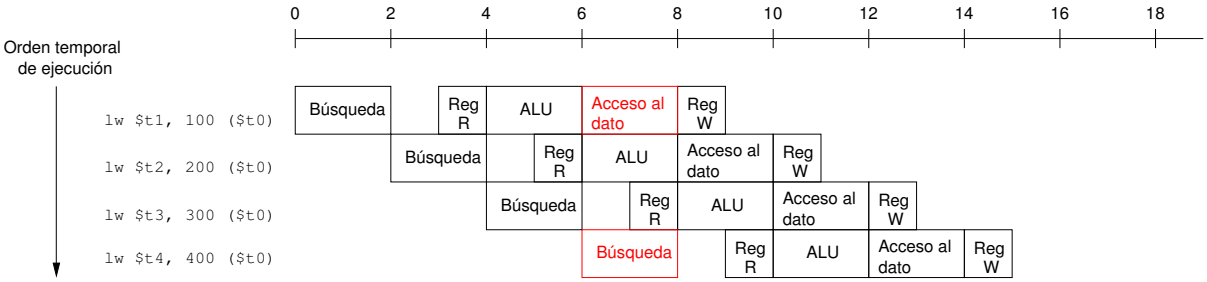
- La segmentación mejora la *productividad*, pero normalmente empeora el tiempo de ejecución de una instrucción individual. Motivos:
  - Las etapas pueden estar mal equilibradas.
  - La segmentación conlleva cierta carga adicional.
- Sin embargo, la productividad es una medida importantísima en informática.
- ¿Por qué los diseños RISC son tan apropiados para el uso de la segmentación?
  - Instrucciones de tamaño fijo: la etapa de búsqueda siempre dura lo mismo.
  - Pocos formatos de instrucciones: casi todas las instrucciones comienzan leyendo del banco de registros.
  - Los operandos sólo pueden ser constantes o registros: la ALU puede trabajar sin esperar a operandos traídos de memoria.
  - Los operandos están alineados en memoria: el acceso a un dato requiere una única transferencia.
- Sin embargo, las cosas no son tan simples...

2. Riesgos (hazards)

Riesgos (hazards)

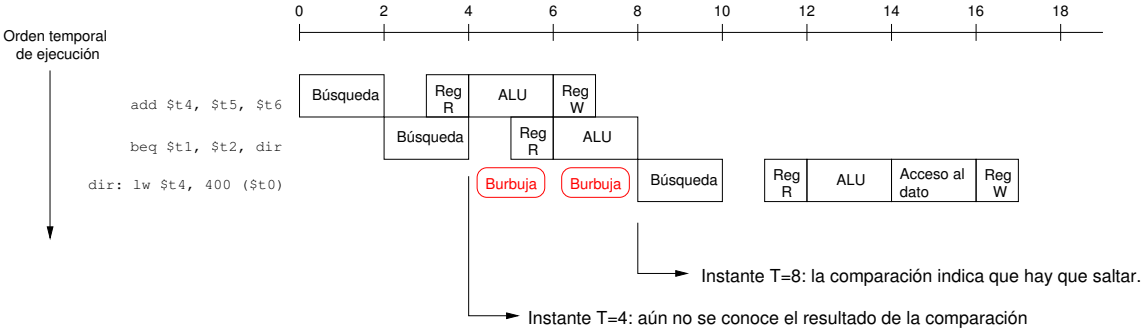
Son situaciones en las que la siguiente instrucción no se puede ejecutar en el siguiente ciclo. Varios tipos.

- *Riesgos estructurales:* cuando la circuitería no puede soportar una combinación determinada de instrucciones ejecutándose al mismo tiempo.
- Solución: duplicar las unidades funcionales (en nuestro ejemplo, memorias separadas para código y datos).



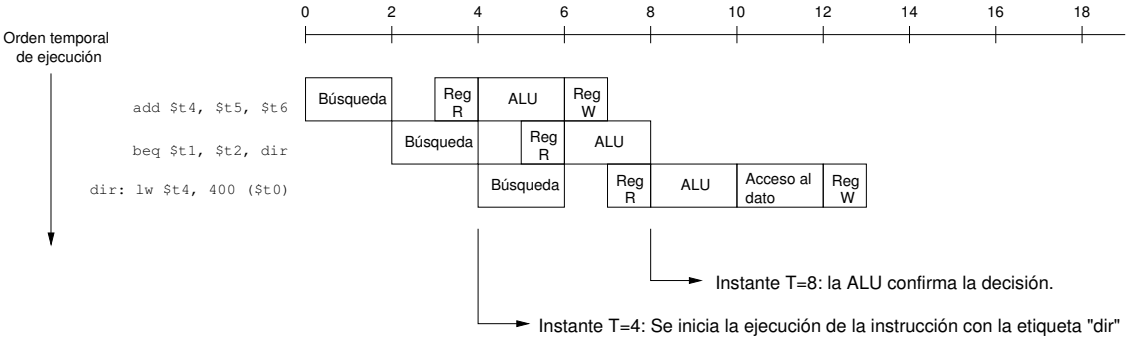
Riesgos de control: bloqueo

- *Riesgos de control:* Necesidad de tomar una decisión basándonos en el resultado de una instrucción que aún no se ha completado.
- Solución 1: Bloqueo *(stall)* del pipeline. Esperar hasta que el resultado esté disponible. Correcto pero lento.

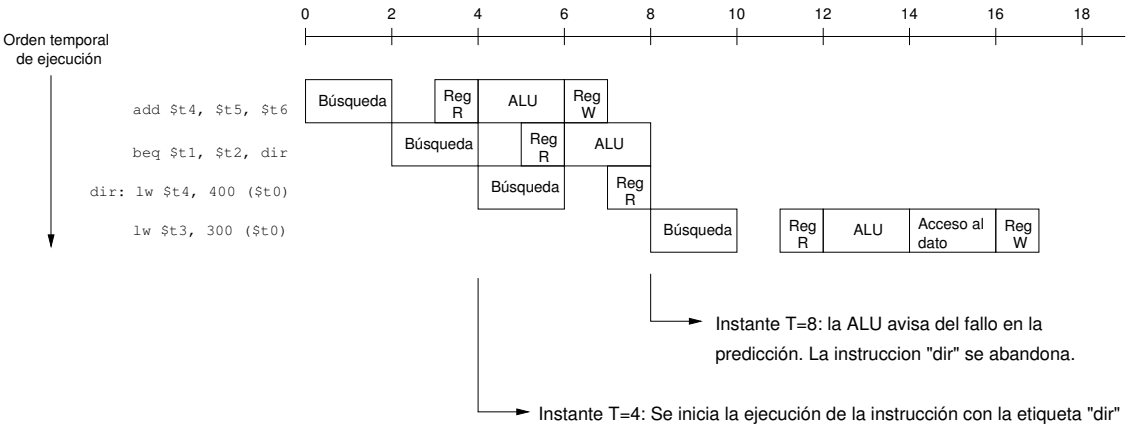


Riesgos de control: predicción de saltos

- Solución 2: Predicción de saltos. Intenta adivinar el resultado de la comparación. Si no acierta, hay que abandonar/deshacer la instrucción puesta en marcha.
- Ejemplo de predicción correcta:



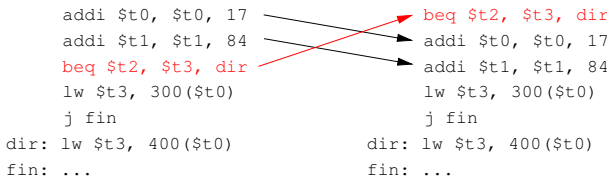
- Ejemplo de predicción errónea: en este caso la instrucción errónea no provoca ningún cambio que sea necesario deshacer.



- Posibles heurísticas: los saltos hacia atrás que cierran un bucle normalmente se toman.
- Predictores dinámicos: tasa de acierto del 90 %.

Riesgos de control: decisión retardada

- Solución 3: decisión retardada. El compilador mueve el salto dos instrucciones hacia arriba, siempre que las dos instrucciones no guarden relación con el salto: aprovecha el tiempo hasta poder decidir si se salta.



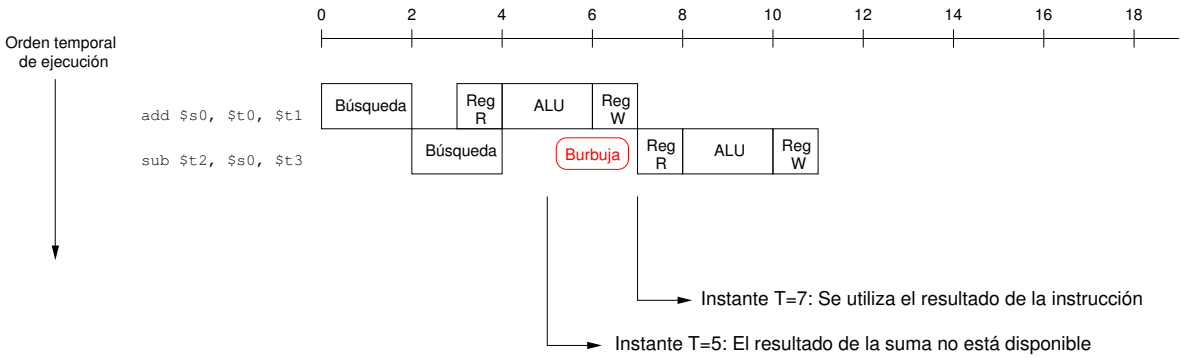
- Con lógica adicional puede reducirse la burbuja a un único ciclo: en ese caso, bastaría con una única instrucción intercalada.
- Si no hay ninguna instrucción que pueda ponerse, el compilador introduce la instrucción `nop`. Esto sucede el 50 % de las veces.
- Solución adoptada en la arquitectura MIPS: hay que tenerlo en cuenta si se programa directamente en código máquina.

Riesgos de control: Mover el salto hacia atrás

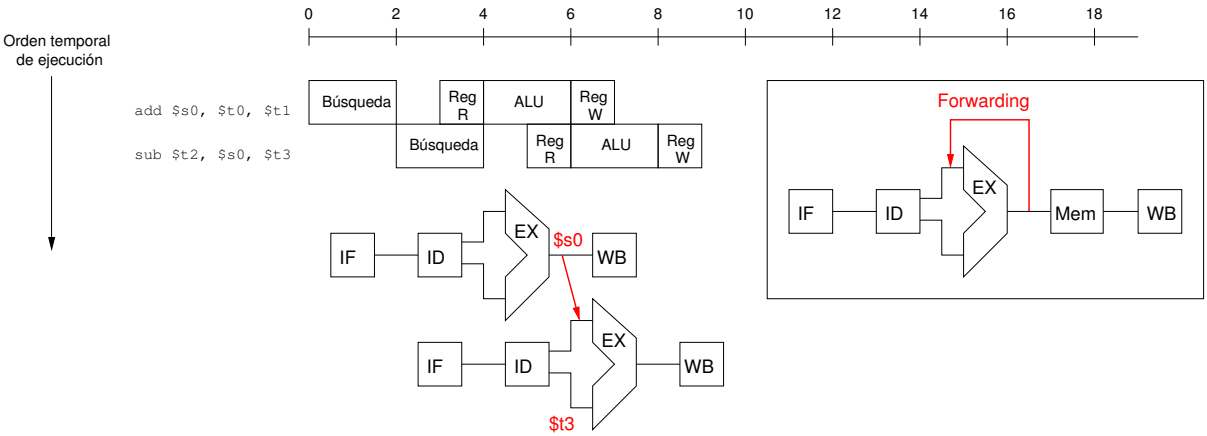
- Solución 4: Pasa por mover el salto hacia atrás en el *pipeline*.
- Esto se consigue añadiendo lógica para disponer del resultado de la instrucción de salto al final de la fase de lectura de registros.
- Esta solución reduce el coste de predecir incorrectamente un salto.

Riesgos de datos

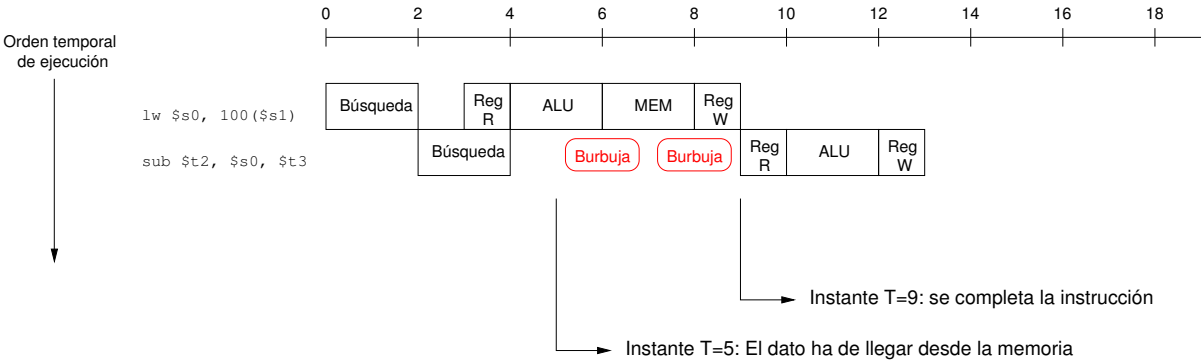
- *Riesgos de datos:* Cuando la ejecución de una instrucción depende de un dato que aún no está disponible.



- Solución 1: *forwarding* o *bypassing*. Se trata de cortocircuitar el pipeline, copiando los datos de la salida de la ALU a su entrada.



- El *forwarding* no soluciona el riesgo de datos si el dato ha de llegar desde la memoria:



- El *forwarding* puede reducir las burbujas de dos a una, pero el bloqueo es inevitable.

3. Implicaciones para el compilador (y el programador)

Implicaciones para el compilador (y el programador)

1. La gestión de los riesgos puede llevarse a cabo por hardware (responsabilidad del microprocesador) o por software (responsabilidad del compilador).
2. Independientemente de quién se ocupe, el compilador no puede traducir código ignorando el funcionamiento del *pipeline*, porque desaprovechará oportunidades para mejorar el rendimiento.
3. En lo que respecta al programador, el hecho de que los compiladores modernos y las arquitecturas reordenen instrucciones puede afectar al funcionamiento de un programa paralelo.

4. Más allá de la segmentación

Más allá de la segmentación

- *Supersegmentación*: Hacer *pipelines* más largos. Buscan reducir el ciclo de reloj. Una vez que el pipeline se llena, salen más instrucciones por unidad de tiempo.
- *Técnicas superescalares*: Replican algunas unidades funcionales para ejecutar más instrucciones por unidad de tiempo.
  - Inconveniente: es difícil mantenerlas todas ocupadas a la vez.
- *Planificación dinámica del pipeline*: Si el *pipeline* se bloquea (por ejemplo debido a un fallo cache), examinar las instrucciones siguientes a ver si alguna puede ir avanzando.

- Control de la segmentación mucho más complicado.
- Las instrucciones pueden acabar en orden o en desorden: esto afecta a la gestión de interrupciones, y a los saltos mal predichos (llegamos a la *ejecución especulativa*).

5. Propuesta de trabajos para la asignatura PSC

Propuesta de trabajos para la asignatura PSC

1. *Planificación dinámica: el algoritmo de Tomasulo.* Historia, funcionamiento general, ejemplo. Fuente básica: sección 3.2 del "Computer Architecture: A Quantitative Approach", de Hennessy y Patterson, tercera ed.
2. *Predicción de saltos dinámica: opciones de diseño.* Fuente básica: sección 3.4 del "CA:AQA".

Los trabajos realizados deberán incluir:

- Un entregable de unas 5 páginas escrito en LaTeX ("article,10pt,twocolumn") con el desarrollo del tema y la bibliografía utilizada.
- Una presentación de unos 30 minutos (25 exposición + 5 para preguntas).
- Las presentaciones se harán el jueves 7 de enero de 2010.

6. Memorias cache

Memorias cache (repaso meteórico)

- Definición: "memoria situada entre la memoria principal y el procesador que almacena los datos e instrucciones más utilizados por el procesador".
- Conceptos clave:
  - *"Más utilizados"*: las cachés almacenan un subconjunto de la información presente en la memoria.
  - *Bloque cache* (*cache line*): Unidad de intercambio de información entre las cachés y la memoria.
  - *Tasas de acierto y de fallo*: Porcentaje de veces en las que el dato está en la caché (o que no está).
  - *Niveles de caché*: Nada impide poner varios niveles. Esto lleva a una "jerarquía de memoria" (aunque cuanto más nos alejemos del procesador, menos útil es la caché, porque las tasas de acierto se multiplican entre sí).

Mecanismos de correspondencia caché

- Tres tipos de correspondencia caché:
  - *Correspondencia directa:* A cada bloque de la memoria le corresponde exactamente un "marco de bloque" caché.
  - *Correspondencia asociativa:* A cada bloque de la memoria le corresponde *cualquier* "marco de bloque" caché.
  - *Correspondencia asociativa por conjuntos:* División de la caché en conjuntos. El mecanismo de correspondencia es directo al conjunto y asociativo dentro del conjunto.
- La correspondencia directa es más rápida, pero puede dar lugar a "fallos de conflicto".
- La correspondencia asociativa es demasiado lenta, evita los fallos de conflicto pero puede dar lugar a "fallos de capacidad".
- En la práctica: Caché L1 directa; caché L2 asociativa por conjuntos.

Accesos a memoria

- Los bloques cache almacenan un conjunto de palabras consecutivas de memoria → si se produce un fallo, la lectura de la siguiente palabra generará un acierto (al estar en ese bloque).
- Para minimizar los fallos caché, es fundamental acceder a los datos tal y como están almacenados en la memoria.
- Ejemplo en C (donde las matrices se guardan por filas):

Acceso por filas	Acceso por columnas

`#include<stdio.h>`
`#define T 10000`
`float a[T][T], b[T][T], c[T][T];`
`main(){`
`  int i,j;`
`  for (i=0; i<T; i++)`
`  for (j=0; j<T; j++)`
`  c[i][j]=a[i][j]+b[i][j];}`
	`#include<stdio.h>`
	`#define T 10000`
	`float a[T][T], b[T][T], c[T][T];`
	`main(){`
	`  int i,j;`
	`  for (j=0; j<T; j++)`
	`  for (i=0; i<T; i++)`
	`  c[i][j]=a[i][j]+b[i][j];}`

Tiempo: 13,78 segundos	**Tiempo: 59,38 segundos**

- En Fortran, las matrices se guardan por columnas (!)