Arquitectura y Organización de Computadoras II

Pipelining

MSc. Ing. Ticiano J. Torres Peralta
Ing. Pablo Toledo

2023 – Segundo Cuatrimestre

Universidad Nacional de Tucumán
Facultad de Ciencias Exactas y Tecnología

Pipelining: Lo Básico

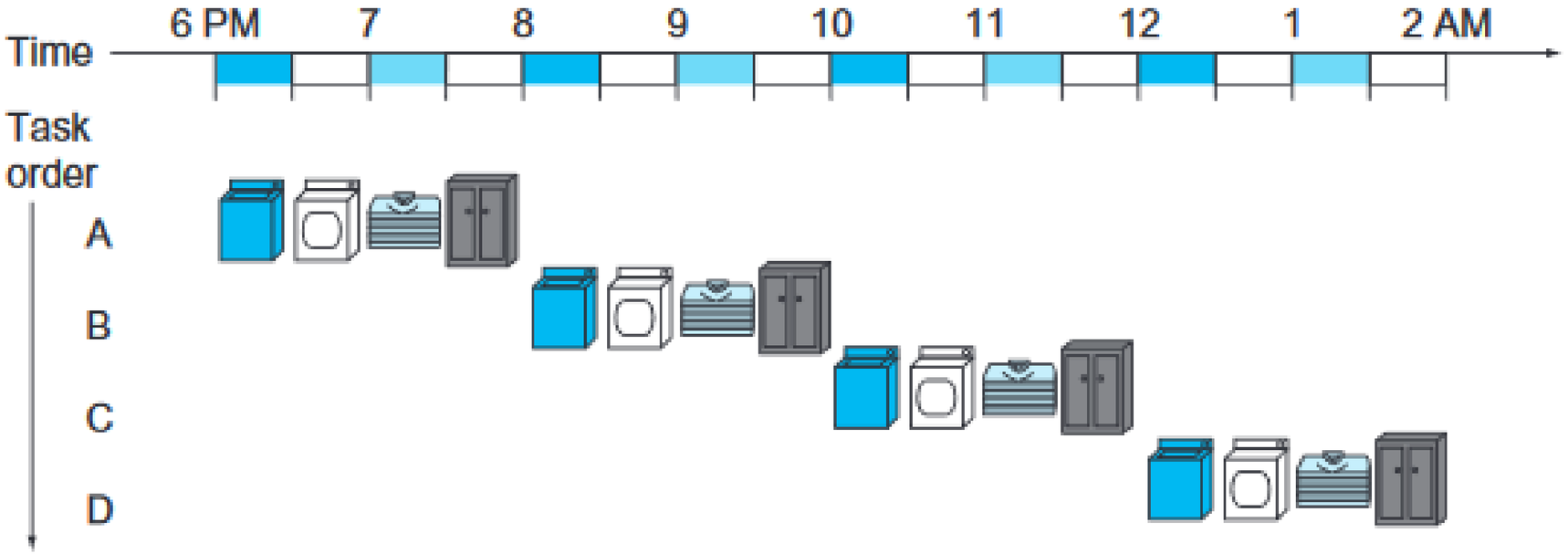
Pipelining (segmentación) es una implementación del paralelismo a nivel de instrucción y es casi universal. La forma mas fácil de entender de que se trata es con un ejemplo. Usando una analogía simple, supongamos que lavamos algo de ropa.

El enfoque sin pipelining sería:

1. Colocamos una carga de ropa sucia en la lavadora.
2. Cuando la lavadora termina su rutina, colocamos la carga húmeda en la secadora.
3. Cuando termina la secadora, colocamos la carga seca sobre una mesa y la doblamos.
4. Cuando terminamos de doblar, le pedimos a un amigo que guarde la ropa.

Y repita para más cargas.

Pipelining: Lo Básico

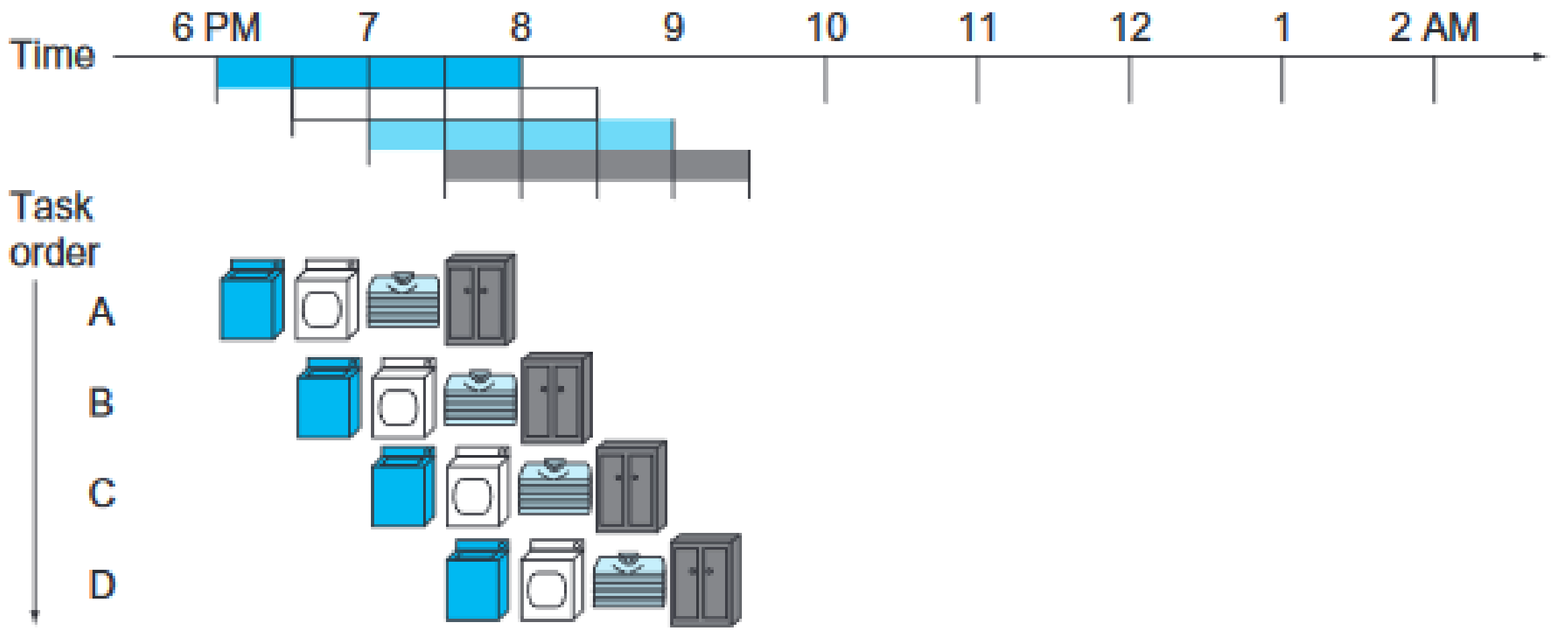


Pipelining: Lo Básico

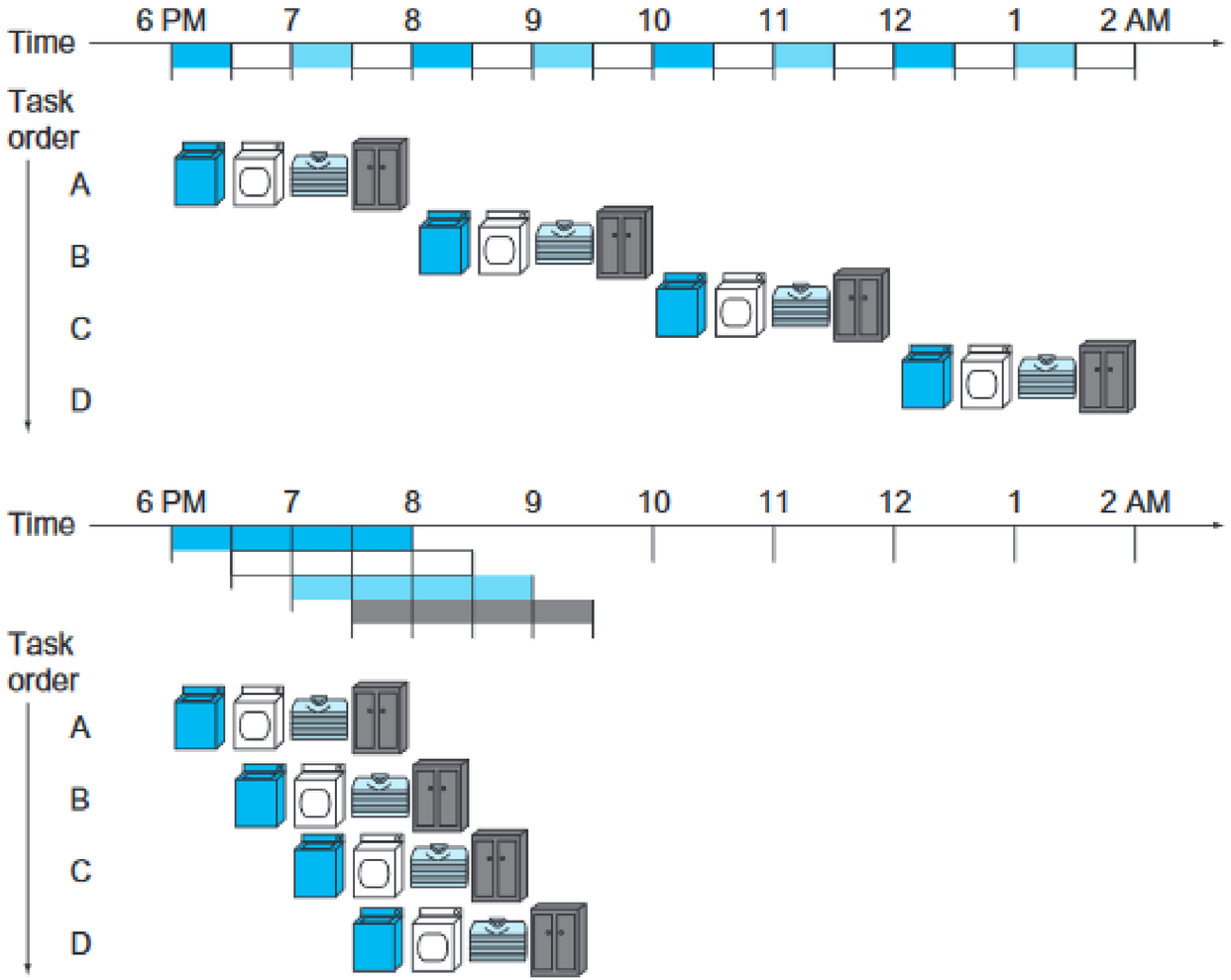
Un enfoque segmentado para esta tarea tomaría mucho menos tiempo. Tan pronto como la lavadora termina con la primera carga y se la coloca en la secadora, cargamos la lavadora con la segunda carga. Cuando la primera carga está seca, la colocamos sobre la mesa para doblarla, movemos la segunda carga a la secadora y metes la tercera carga a la lavadora. Luego, le pedimos a un amigo que guarde la carga doblada, comenzamos a doblar la segunda carga, la secadora tiene la tercera carga y hay una cuarta carga en la lavadora. En este punto todos las etapas en el proceso, comúnmente conocidos como pipeline stages (etapas de segmentación), funcionan al mismo tiempo. Siempre que tengamos recursos separados para cada etapa, podemos segmentar.

La siguiente figura muestra el proceso.

Pipelining: Lo Básico



Pipelining: Lo Básico



El ejemplo de lavar la ropa con y sin segmentación.

Pipelining

Una característica importante de la segmentación es que el tiempo que tarda una sola carga en terminar no se reduce (la latencia). El tiempo total para todas las cargas es más corto porque cada etapa trabaja en paralelo, por lo que se terminan mas cargas por hora (el ancho de banda). Por lo tanto, la segmentación mejora el rendimiento de nuestro sistema.

Ahora, si todas las etapas toman la misma cantidad de tiempo y hay suficiente trabajo por hacer, entonces la mejora debido a la segmentación es igual al número de etapas en la canalización. En el caso de nuestro ejemplo, esto sería una mejora de cuatro.

En el caso particular que muestra la figura en realidad es un valor de 2,3 porque sólo tenemos cuatro cargas que procesar. Observemos que el principio y el final del trabajo total no están completamente llenos. El inicio y finalización afecta el rendimiento cuando la cantidad de tareas no es grande en comparación con la cantidad de etapas.

Pipelining

Podemos convertir la discusión sobre la mejora que provee una segmentación en una fórmula. Si las etapas están perfectamente equilibradas, y con la suposición que siempre hay algo que procesar y cada etapa esta llena, entonces el tiempo entre instrucciones en un procesador con segmentación (y suponiendo otras condiciones ideales) es igual a

$$Time\ between\ instructions_{pipelined} = \frac{Time\ between\ instructions_{nonpipelined}}{Number\ of\ pipe\ stages}$$

Pipelining

Este principio se aplica a cualquier procesador con segmentación en cuanto a la ejecución de instrucciones. Una segmentación típica de instrucciones en un procesador MIPS es:

1. Se obtiene una instrucción desde la memoria.
2. Se lee los registros (operandos) mientras se decodifica la instrucción. La lectura y la decodificación ocurren en simultaneo. (Estamos hablando de instrucciones de longitud fija en una arquitectura GPR.)
3. Se ejecuta la operación o se calcular una dirección.
4. Se accede a un operando en la memoria de datos.
5. Se escribe el resultado en un registro.

Este pipeline típico tiene 5 etapas.

Pipelining

Analicemos un ejemplo para comprender mejor esta segmentación. Comparemos una implementación de ciclo único, donde todas las instrucciones toman un ciclo de reloj, con una implementación con segmentación.

Recordemos que en el modelo de ciclo único, todas las instrucciones deben finalizar en un solo ciclo de reloj, por lo que el ciclo de reloj debe ampliarse para dar cabida a la instrucción más lenta.

Ahora, supongamos que para con segmentación, el tiempo de operación para las principales unidades funcionales para este ejemplo es lo siguiente:

- 200 ps para acceso a la memoria
- 200 ps para operación de ALU
- 100 ps para lectura o escritura de archivos de registro.

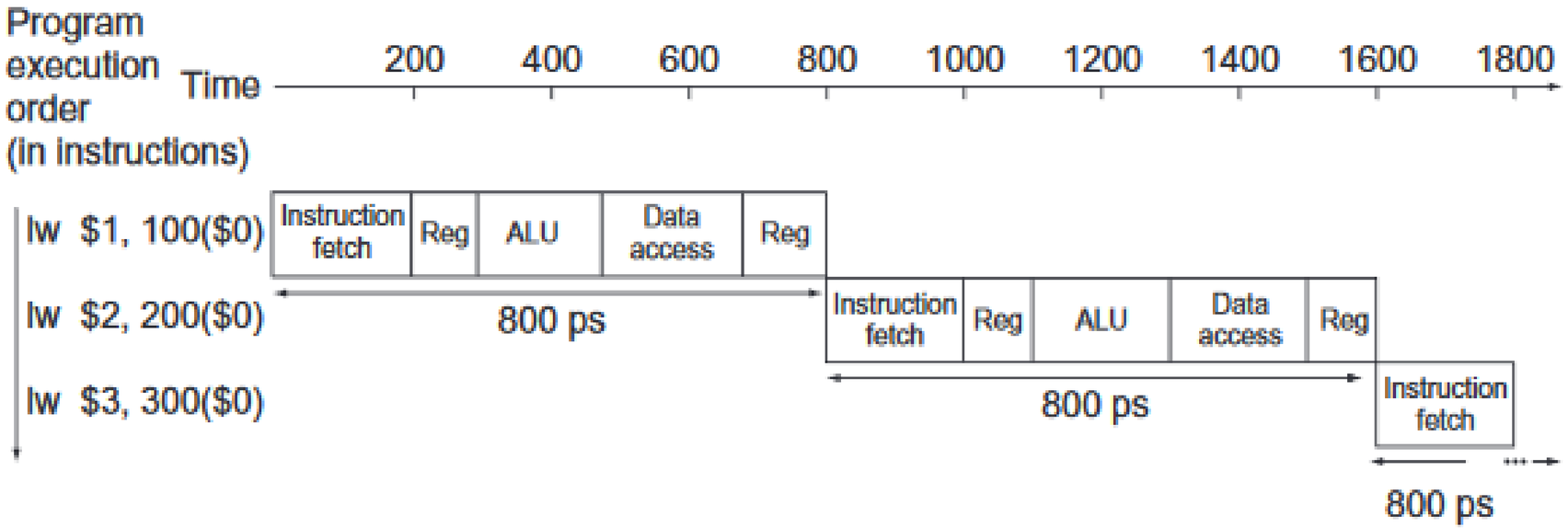
En la siguiente figura podemos ver que el ciclo del reloj no puede ser inferior a 800 ps, el tiempo necesario para Load word (lw), la instrucción más lenta.

Pipelining

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

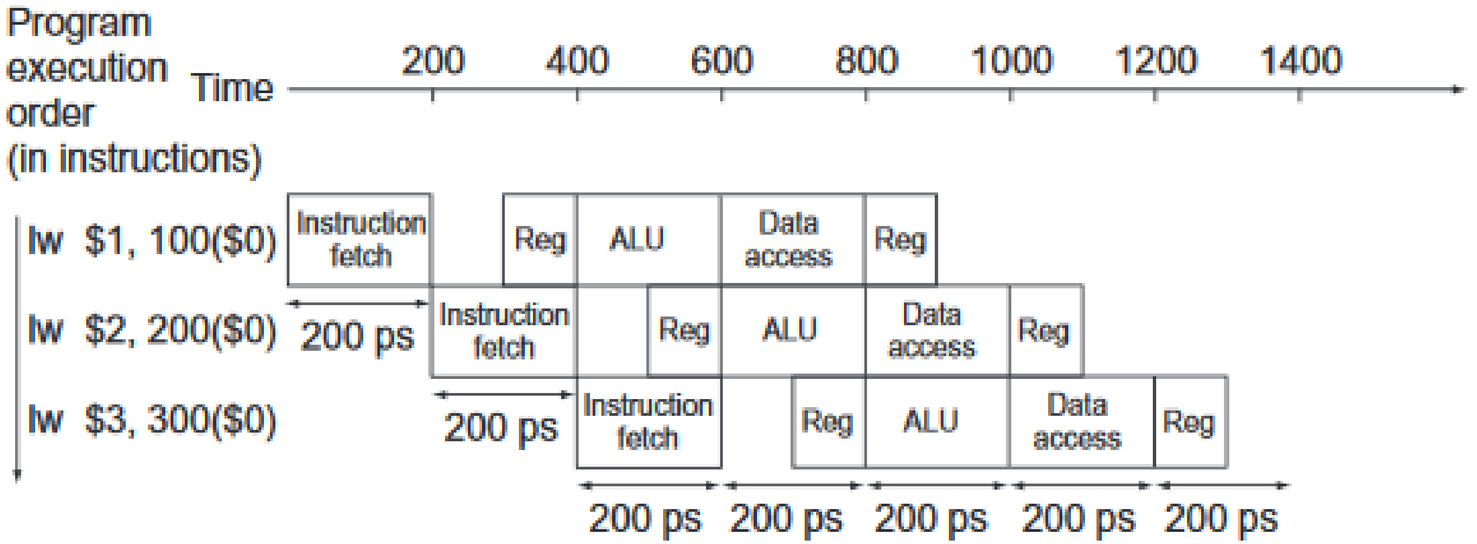
Pipelining

Esto significa que para una ejecución de tres instrucciones lw consecutivas, el tiempo entre la primera y la cuarta instrucción en un modelo de ciclo único es de 3 x 800 ps o 2400 ps, como se ve en la siguiente figura.



Pipelining

Para el modelo con segmentación, cada etapa es un ciclo de reloj, por lo que el ciclo de reloj debe ser lo suficientemente largo para acomodar la etapa más lenta. Esto significa que debe ser de 200 ps, aunque algunas etapas solo toman solo 100 ps. En la figura, podemos ver que con la canalización obtenemos una mejora de cuatro en el rendimiento; el tiempo entre la primera y la cuarta instrucción es 3 x 200 o 600 ps.



Pipelining

En condiciones ideales y con una gran cantidad de instrucciones, la mejora que provee la segmentación es aproximadamente igual al número de etapas de el pipeline. La fórmula sugiere que un pipeline de cinco etapas debería ofrecer una mejora de casi cinco veces con respecto al tiempo sin segmentación de 800 ps, ósea un ciclo de reloj de 160 ps. El ejemplo muestra, sin embargo, que las etapas pueden estar imperfectamente equilibradas. En la práctica, el tiempo por instrucción excederá el mínimo teórico y la mejora real será menor que el número de etapas del pipeline.

Además, incluso nuestra afirmación de una mejora por cuatro para nuestro ejemplo no se refleja en el tiempo total de ejecución, que es 1400 ps frente a 2400 ps. Esto se debe a que la cantidad de instrucciones ejecutadas es poca. Supongamos que ejecutamos 1.000.003 de instrucciones de carga. Tendríamos:

$$\frac{800{,}002{,}400\ ps}{200{,}001{,}400\ ps} \simeq \frac{800\ ps}{200\ ps} \simeq 4$$

Pipelining: Pautas de Diseño

El conjunto de instrucciones (ISA) MIPS fue diseñado para la ejecución con segmentación. En primer lugar, todas las instrucciones MIPS tienen la misma longitud. Esta restricción hace que sea mucho más fácil buscar instrucciones en la primera etapa del pipeline y decodificarlas en la segunda. MIPS tiene sólo unos pocos formatos de instrucción, y las direcciones de los registros (operandos) se ubican siempre en el mismo lugar en cada instrucción. Esta simetría significa que la segunda etapa puede leer los registros al mismo tiempo que se decodifican las instrucciones. Si esta simetría no estuviera presente, la segunda etapa tendría que dividirse en dos partes.

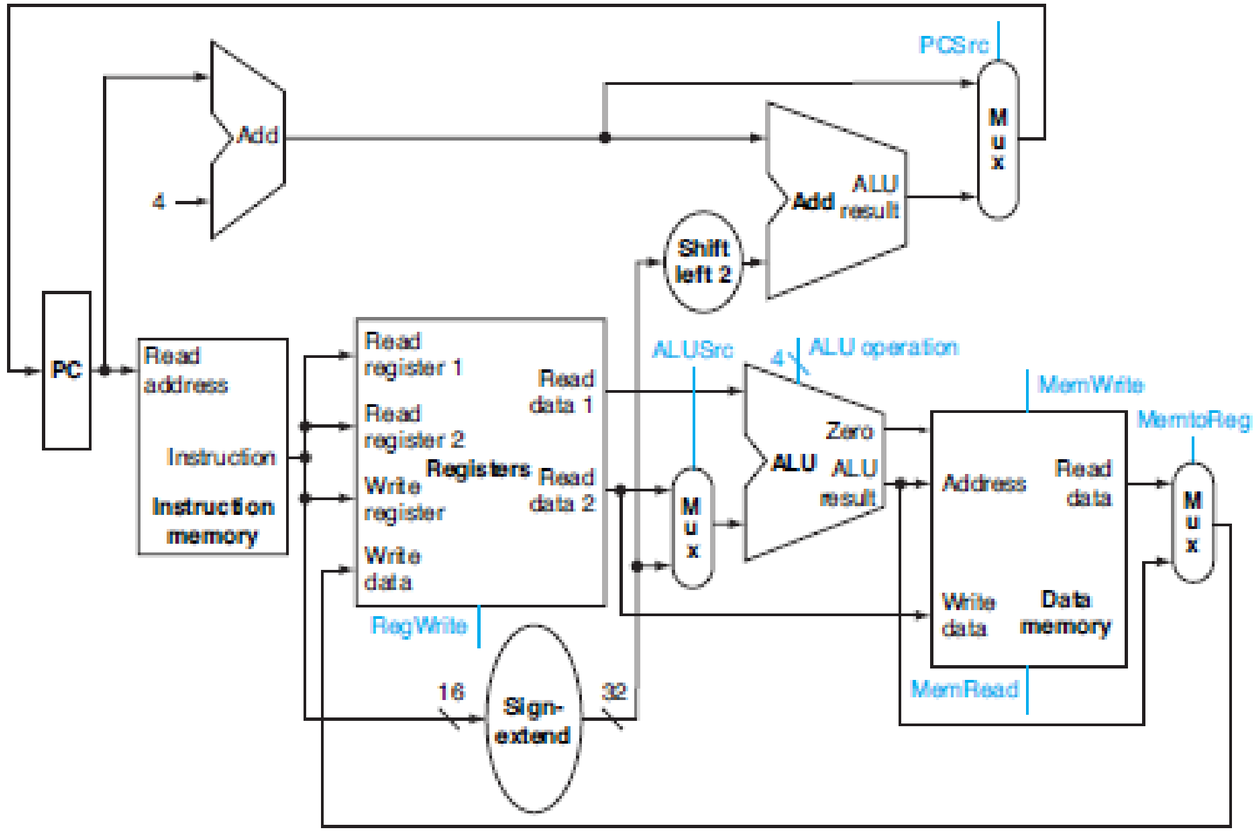
En contraste, para ISAs como x86, donde una instruction varía entre 1 a 15 bytes, la segmentación es considerablemente más desafiante. Las implementaciones recientes de x86 en realidad traducen las instrucciones x86 en operaciones simples que parecen instrucciones MIPS y segmentan las operaciones simples en lugar de las instrucciones x86 nativas.

Pipelining: Pautas de Diseño

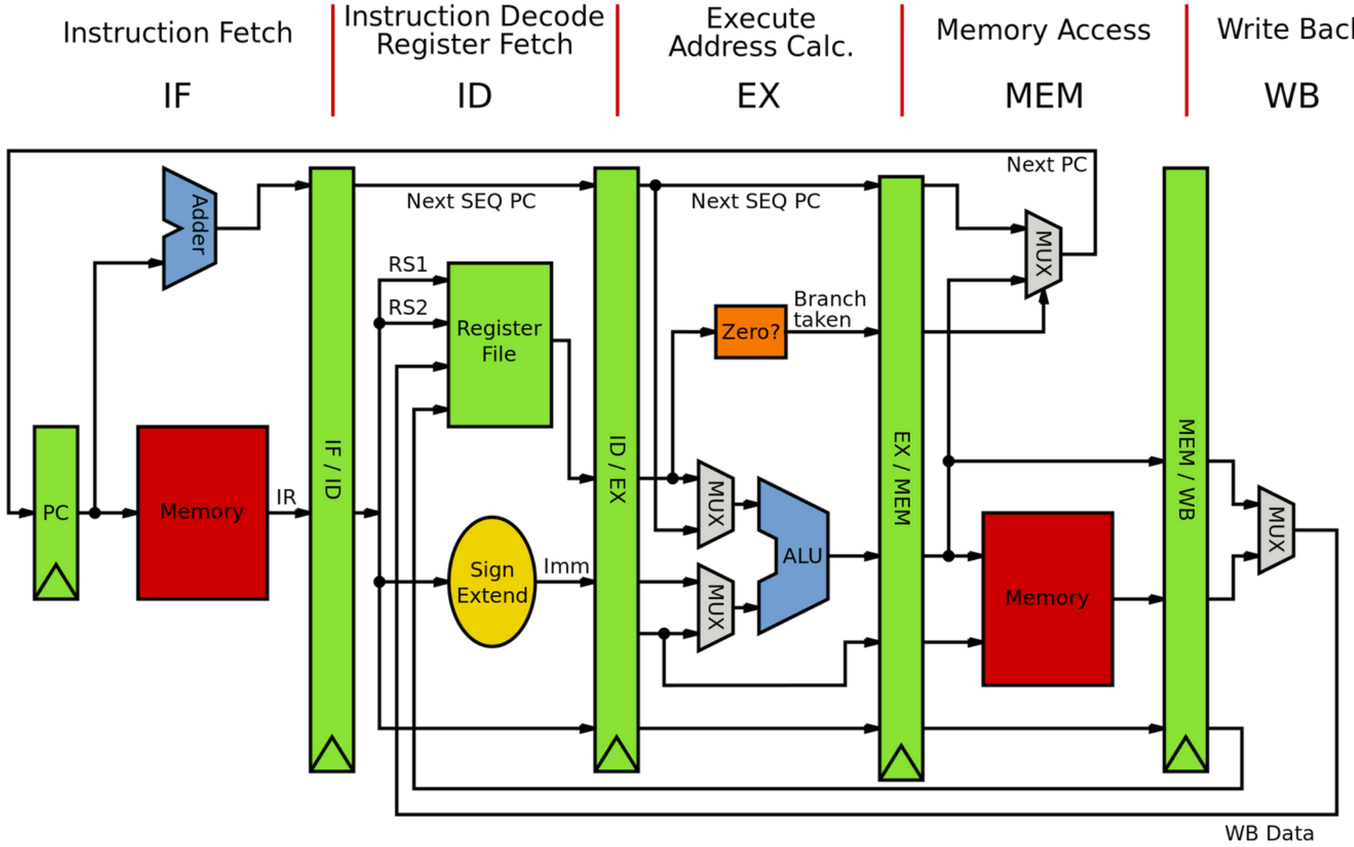
En segundo lugar, los operandos que acceden a memoria sólo aparecen en cargas y almacenes (arquitectura LOAD-STORE). Esta restricción significa que podemos usar la etapa de ejecución para calcular la dirección efectiva de memoria y luego acceder a la memoria en la siguiente etapa.

Por ultimo, los operandos deben estar alineados en la memoria. Por lo tanto, no debemos preocuparnos de que la transferencia de un objeto requiera dos accesos a la memoria. Ósea, los datos solicitados se pueden transferir de la memoria al procesador en una sola etapa.

Pipelining: Pautas de Diseño



Pipelining: Pautas de Diseño



Pipelining: Riesgos

Hay tres posibles riesgos (**hazards**) que pueden afectar el funcionamiento del pipeline:

- Riesgo estructural (**Structural Hazard**)
- Riesgo de dato (**Data Hazard**)
- Riesgo de control (**Control Hazard**)

Pipelining: Riesgos

Riesgo estructural:

Los riesgos estructurales ocurren cuando el hardware no puede soportar la combinación de instrucciones que queremos ejecutar en el mismo ciclo de reloj. En nuestro ejemplo de lavandería, se produciría un riesgo estructural si la misma persona es responsable de doblar y guardar la ropa.

Dado que el conjunto de instrucciones MIPS fue diseñado para ser segmentado, es fácil para los diseñadores evitar riesgos estructurales en su implementación. Supongamos, sin embargo, que tuviéramos una sola memoria (con un solo canal de acceso) en lugar de dos memorias (o una con varios canales de acceso en paralelo). En este caso, en el mismo ciclo de reloj una instrucción buscaría acceder a datos de la memoria mientras que otra instrucción buscaría una instrucción de esa misma memoria. Sin dos memorias (o la posibilidad de accederla en paralelo), nuestro pipeline podría tener un peligro estructural.

Pipelining: Riesgos

Riesgo de dato:

Los riesgos de datos ocurren cuando el proceso debe detenerse porque un paso de una instrucción debe esperar a que se complete otra. Con el ejemplo del lavarropas, supongamos que se encuentra un calcetín en la etapa de doblar la ropa al que le falta un par. Lo que podrías hacer es detener todo el proceso mientras revisas las cargas para encontrar su par.

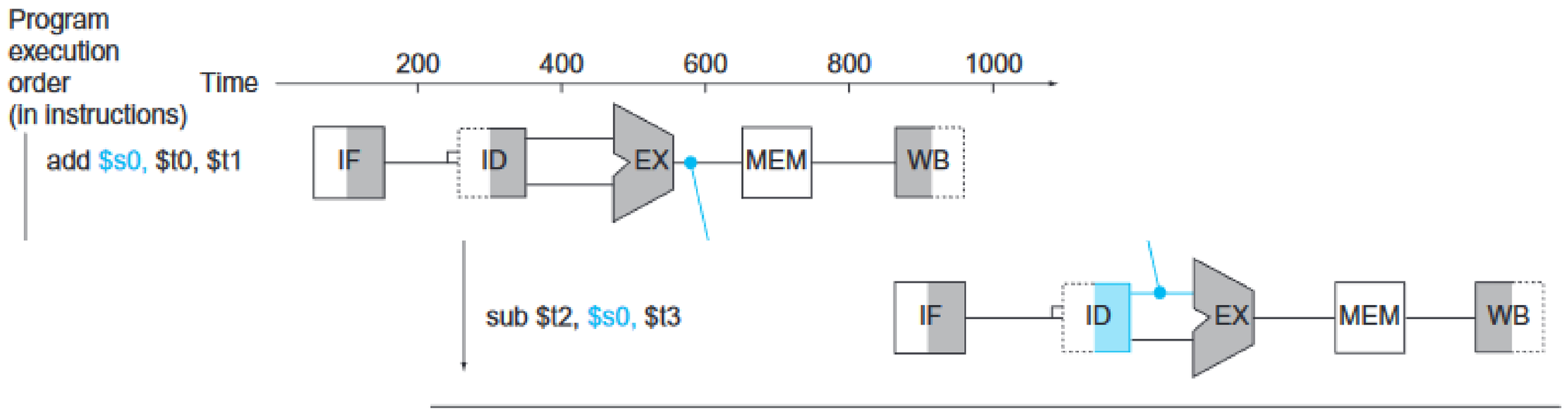
En la computadora, un riesgo de datos surge cuando los datos de una instrucción depende de otra anterior que aún está en proceso. Supongamos que:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Sin intervención, este riesgo podría paralizar gravemente el proceso. En este caso, la instrucción add no escribe su resultado hasta la ultima (quinta) etapa, lo que significa que tendríamos que desperdiciar tres ciclos de reloj.

Pipelining: Riesgos

Sin intervención, este riesgo podría paralizar gravemente el proceso. En este caso del ejemplo, la instrucción add no escribe su resultado hasta la ultima (quinta) etapa, lo que significa que tendríamos que desperdiciar tres ciclos de reloj.



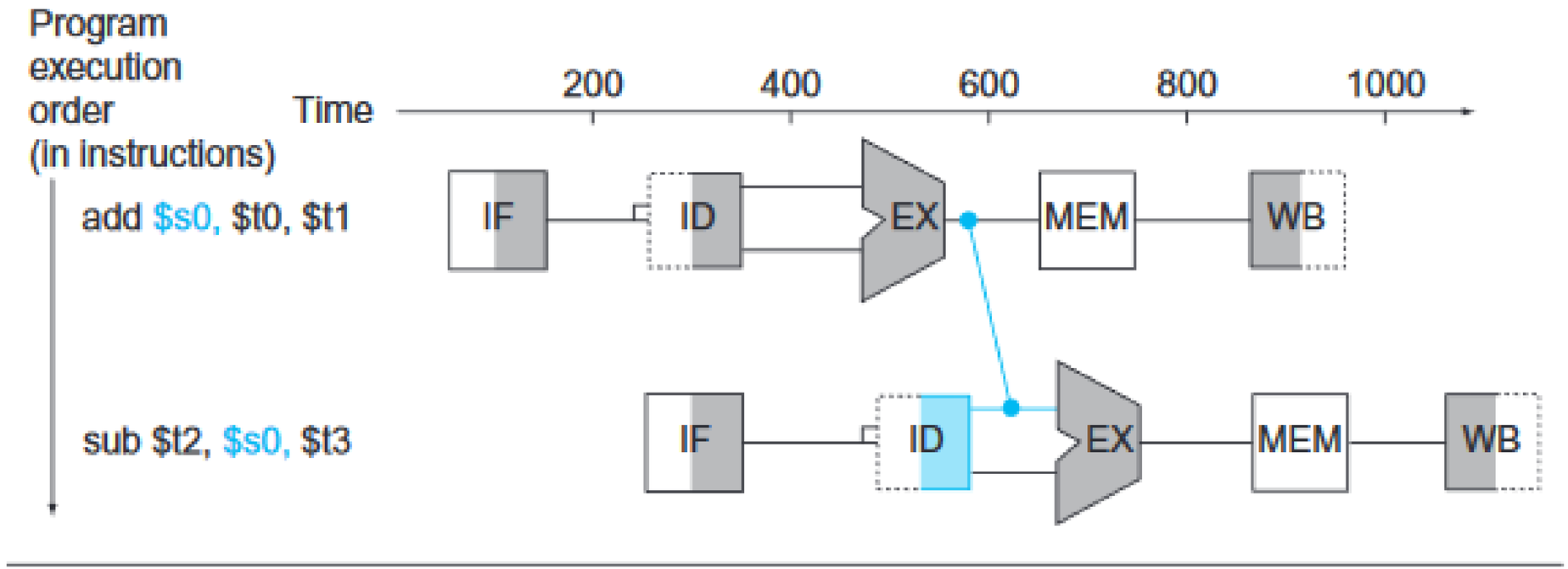
Pipelining: Riesgos

Podríamos tratar de confiar en los compiladores para eliminar esos peligros, pero los resultados serían insatisfactorios. Estas dependencias ocurren con demasiada frecuencia y la penalización en la demora es demasiado grande como para esperar que el compilador nos rescate.

La solución principal a este problema se basa en la observación de que no es necesario esperar a que se complete la instrucción antes de intentar intervenir y resolver el riesgo. Siguiendo el ejemplo anterior, tan pronto como la ALU crea el resultado para el add, podemos proporcionarla como entrada para la resta. Agregando hardware adicional para recuperar temprano el elemento faltante de los recursos internos se denomina **forwarding** o **bypassing**.

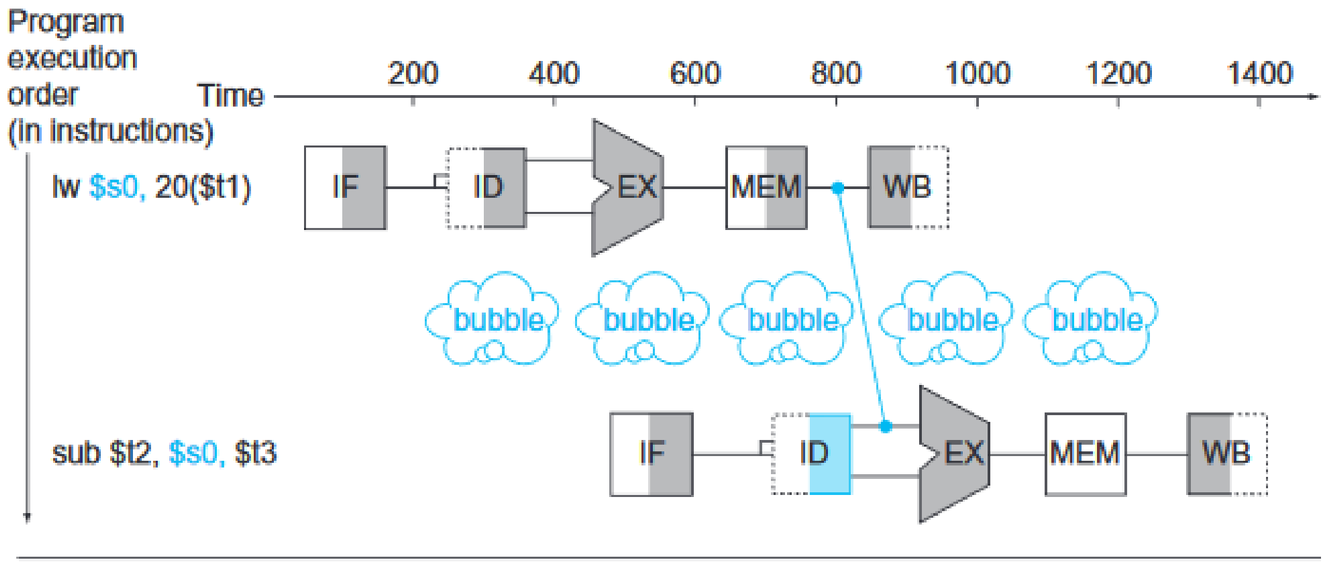
Esta técnica sólo se puede utilizar cuando la ruta de reenvío es válida, es decir, sólo si la etapa de destino es posterior a la etapa de origen.

Pipelining: Riesgos



Pipelining: Riesgos

Forwarding funciona bien pero no puede evitar todas las paradas en el pipeline. Por ejemplo, supongamos que la primera instrucción fue una carga de $s0 en lugar de un add. En este caso, los datos deseados estarían disponibles sólo después de la cuarta etapa, lo cual es demasiado tarde para la entrada de la tercera etapa de sub. Entonces, incluso con Forwarding, tendríamos que detener una etapa por un riesgo de datos. Este tipo de situación se denomina **load-use data hazard**.



Pipelining: Riesgos

La figura anterior muestra un importante concepto llamado parada de tubería (**pipeline stall**), comúnmente conocida como burbuja (**bubble**). Tratar con estos no es simple, pero se puede manejar mediante detección de hardware o software que reordena el código para tratar de evitar una parada en una situación load-use. Por ejemplo:

Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from `$t0`:

```
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)
```

Pipelining: Riesgos

```
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)
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

También es importante tener en cuenta que cada instrucción escribe como máximo un resultado y lo hace al final de la última etapa. Por lo tanto, también es difícil hacer forwarding cuando se necesita hacer con varios resultados a la vez.

Pipelining: Riesgos

Riesgo de Control:

El tercer riesgo surge en la necesidad de tomar una decisión basada en un resultado de una instrucción mientras otras instrucciones se ejecutan. Supongamos que tenemos que limpiar los uniformes de un equipo de fútbol. Dado lo sucia que está la ropa, debemos determinar si la temperatura del detergente y del agua es la correcta. Esto significa que debemos esperar hasta después de la segunda etapa para examinar un uniforme seco y ver si necesitamos cambiar la configuración.

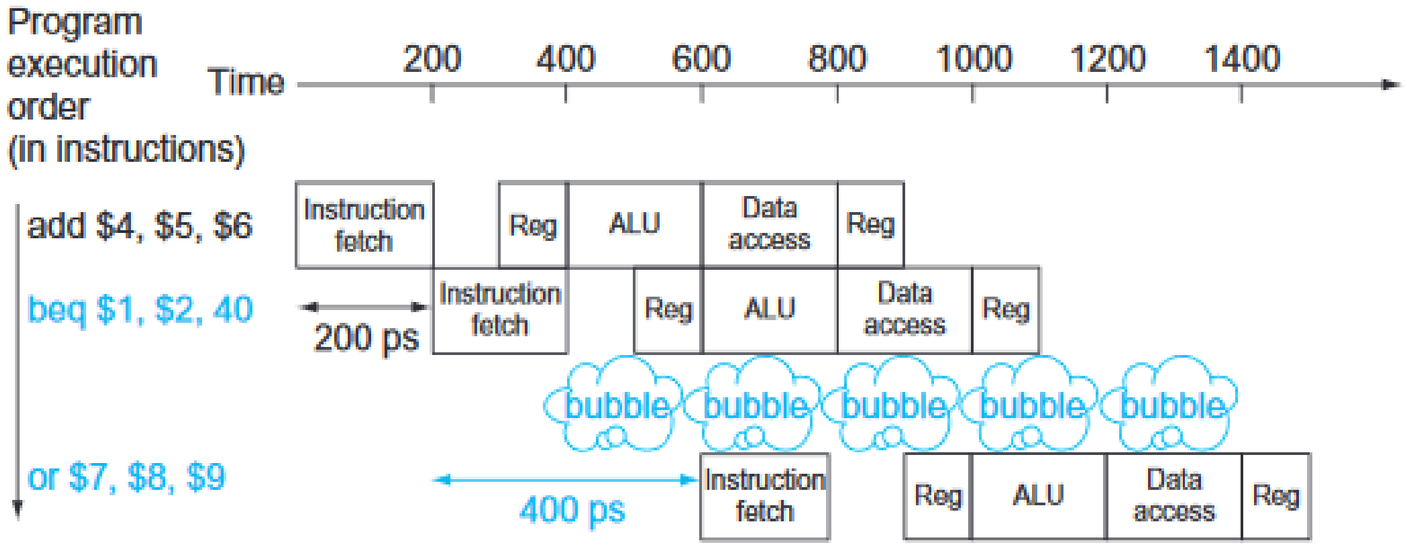
La tarea de decisión equivalente en una computadora es la instrucción de ramificación.

Pipelining: Riesgos

Hay dos soluciones a este problema:

1 . Stall: Simplemente opere secuencialmente hasta que se resuelva la ramificación.
Esta opción funciona, pero es lenta.

Aun suponiendo que podemos agregar hardware adicional para poder probar los registros y calcular la dirección de la ramificación durante la segunda etapa, igual tendríamos una burbuja.



Pipelining: Riesgos

Si no podemos resolver la ramificación en la segunda etapa, como suele ser el caso de pipelines más largos, veríamos golpe aún mayor en el rendimiento. Este costo es demasiado alto para que lo utilicen la mayoría de las computadoras y es la motivación para una segunda solución:

2. Predecir: si está bastante seguro de saber para donde ramifica, simplemente anticipe y ejecute la siguiente instrucción de acuerdo con la predicción.

Un enfoque simple es predecir que se cancelarán todas las ramas. Cuando tienes razón, el proceso no se ralentiza. Sólo cuando se toman ramas se detiene el oleoducto.

Pipelining: Riesgos

Un enfoque simple es predecir que nunca se ramifica. Cuando tienes razón, el proceso no se ralentiza. Sólo cuando se toman ramas se tiene que stall el pipeline.