Departamento de Arquitectura Instituto de Computación

Universidad de la República

Montevideo - Uruguay

Notas de Teórico

Pipeline

Arquitectura de Computadoras

(Versión 4.3b - 2020)

18 PIPELINE

18.1 Introducción

En este capítulo veremos una de las técnicas de diseño de procesadores que permite mejorar el rendimiento de los mismos medido en términos de cantidad de instrucciones ejecutadas por ciclo de reloj.

Como hemos visto los procesadores son, en definitiva, circuitos secuenciales sincrónicos, que trabajan con una cierta frecuencia de reloj. Tener mejor rendimiento, entendido como disminuir el tiempo de ejecución de los programas, ha sido históricamente la obsesión de los diseñadores de computadoras. En ese rendimiento hay muchos factores que juegan, siendo la capacidad de ejecución de instrucciones de parte del procesador un factor clave.

Una primera aproximación, obvia por tratarse de una máquina sincrónica, es aumentar la frecuencia del reloj: a mayor cantidad de MHz (ó GHz) del reloj del procesador, menor será su periodo y por tanto menor el tiempo de ejecución de las instrucciones que conforman un programa. Sin embargo esta aproximación no es tan sencilla. En cada estadio de la tecnología de fabricación de circuitos integrados está más o menos determinada la frecuencia máxima del reloj admitida por el circuito. Recordemos que los circuitos del tipo CMOS solo conducen corriente eléctrica, generando calor, en las transiciones; con lo que cuanto más alta la frecuencia mayor es el calor generado que debe disiparse para que la temperatura no suba hasta dañar el material (silicio) con el que están construidos los transistores. Esto impone un límite tecnológico a la frecuencia de trabajo.

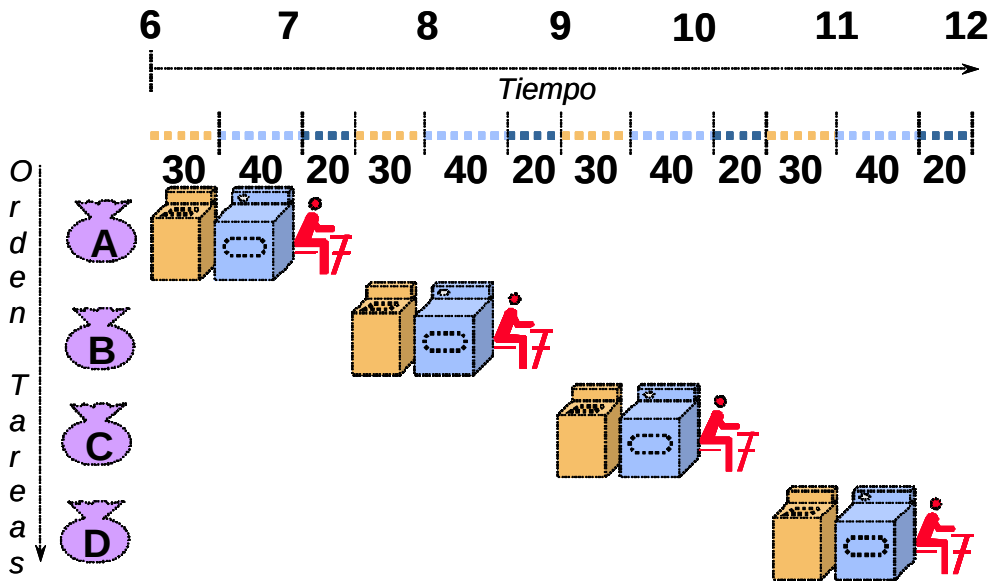
La técnica de pipeline apunta a mejorar el rendimiento sin necesidad de aumentar la frecuencia de trabajo, utilizando los mismos principios que Henry Ford introdujo en las formas de producción industrial a principios del siglo XX: la producción en cadena ó cadena (línea) de montaje.

18.2 Producción en Cadena

El principio de la producción en cadena es la división del trabajo total para realizar un objeto en un conjunto secuencial de tareas más simples, que pueden realizarse en paralelo. Esto no logra mejorar el tiempo que se requiere para la realización de un objeto en particular, pero mejora la capacidad de producción (la cantidad de objetos producidos por unidad de tiempo).

Ilustremos con un ejemplo de lavado de ropa en una lavandería. Supongamos que el proceso implica el lavado (en máquina), el secado (en otra máquina) y una tarea manual de doblado de la ropa.

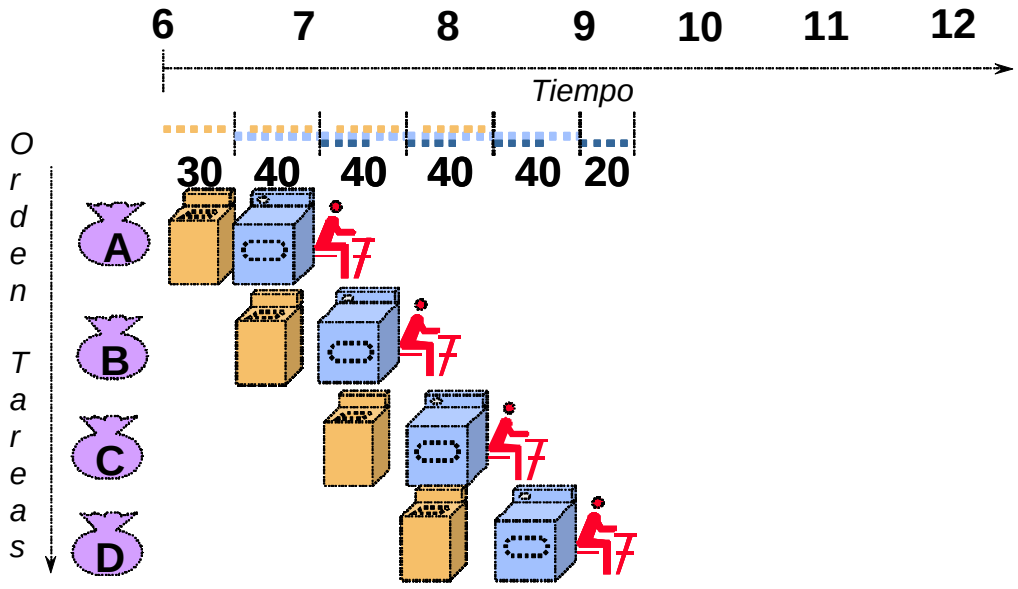
Supongamos que el lavado demora 30 minutos, el secado 40 minutos y el doblado 20. Si tenemos 4 cargas de ropa A, B, C y D para realizar el ciclo completo de lavado/secado/doblado vamos a demorar 90 minutos (1 hora y media). Para completar las 4 cargas de ropa vamos a requerir, en principio, 360 minutos (6 horas) si las hacemos una tras otra, como se ve en el siguiente diagrama:

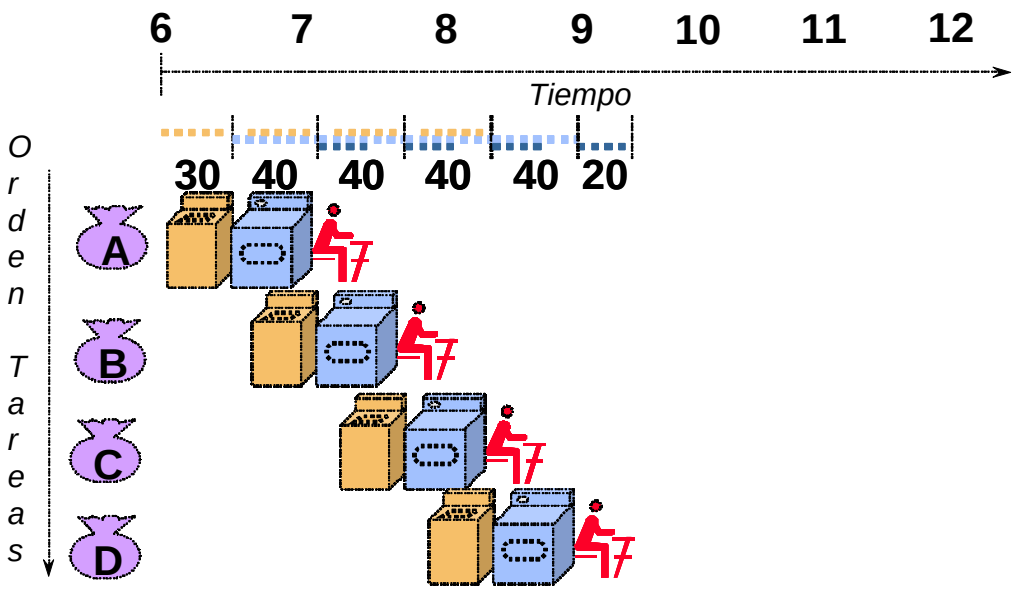


Si aplicamos la técnica de la producción en cadena, lo que podemos hacer es: lavamos la carga de ropa “A”, cuando se termina de lavar ponemos a secar la carga de ropa “A” y al mismo tiempo ponemos a lavar la carga de ropa “B”. Cuando se termina de secar la carga “A” podemos comenzar a doblar la carga “A” y simultáneamente poner a secar la carga “B”.

Notar que como el tiempo de lavado es menor que el de secado, el lavado de la carga “B” termina antes del secado de “A”. En ese momento podríamos esperar que termine el secado de “A” y pasar la carga “B” de la lavadora a la secadora y allí comenzar a lavar la carga “C” mientras se seca la carga “B” y se dobla la carga “A”. La alternativa es no comenzar a lavar la “B” hasta 10 minutos después de comenzado el secado de “A”. De esta manera el secado de “A” terminaría junto al lavado de “B”. Lo mismo haríamos con el lavado de “C” respecto del secado de “B”.

Estas dos alternativas están representadas en los siguientes diagramas:





Notar que ambas alternativas dan como resultado el mismo tiempo total: 30 + 40 + 40 + 40 + 40 + 20 = 210 minutos (3,5 horas). Es decir con la “lavandería en pipeline” logramos mejorar de 6 horas a 3,5 horas. Dicho de otra manera, pasamos de un promedio de una carga de ropa cada 90 minutos a un promedio de una carga cada 52,5 minutos!!.

Algunas anotaciones importantes:

- cada carga demora lo mismo que antes (90 minutos) en completar el ciclo. Incluso las “B”, “C” y “D” pueden demorar más (100 minutos) si usamos la alternativa de comenzar el lavado apenas se pueda y esperar luego que se libere la secadora (primera alternativa de “pipeline”).
- las cargas se entregan cada 40 minutos: es decir que en régimen la capacidad de producción (lavado/secado/doblado) es de 1 carga / 40 minutos. Nos dio 52,5 porque hay una penalización hasta llegar al régimen (debe “llenarse” de tareas la cadena de montaje).
- si el tiempo de cada tarea fuera igual entre sí (en este caso 30 minutos), no habría esperas entre tareas y el funcionamiento de la línea sería óptimo. En este caso en régimen obtendríamos 1 carga / 30 minutos.
- cuando esto no se cumple, la tarea que demora más es la que determina la capacidad de producción.
- cuando se cumple, la capacidad de producción en régimen es: n / duración total, siendo “n” la cantidad de etapas del pippeline (o de “estaciones” en la terminología de la línea de montaje).
- lo anterior significa que aumentando la cantidad de etapas mejoramos el rendimiento. Sin embargo hay un límite al crecimiento de “n”: dado que las tareas deben durar lo mismo y ser independientes entre sí, a medida que intento aumentar la cantidad de tareas en que subdivido el trabajo, cada vez es más difícil lograr esa condición.

Otro elemento que ha estado implícito en toda esta descripción es que las tareas involucradas en la línea de montaje deben ser independientes entre sí y no pueden tener otro vínculo que el de fin a comienzo. Supongamos por un momento que en la lavandería hay un único enchufe para alimentar eléctricamente las máquinas (o, más realistamente, que hay una llave eléctrica limitadora que, para proteger la instalación eléctrica, impide poner a funcionar ambas máquinas en forma simultánea por el alto consumo de cada una). En este caso no hay paralelismo posible entre el lavado y el secado, aunque sí entre

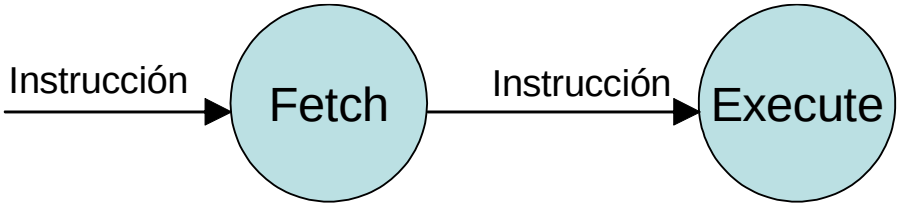
cualquiera de ellas y el doblado. Esto hace que las tareas de lavado y secado funcionen como una sola y por tanto el rendimiento desciende a 1 carga / 70 minutos (70 minutos es la duración combinada, serialmente, de ambas tareas).

18.3 Pipelines

18.3.1 Pipeline Simple

Un ejemplo de pipeline simple es el que fue implementado en los procesadores Intel 8086. Divide el ciclo de instrucción en dos etapas: Fetch y Execute.

Un diagrama de su funcionamiento es:



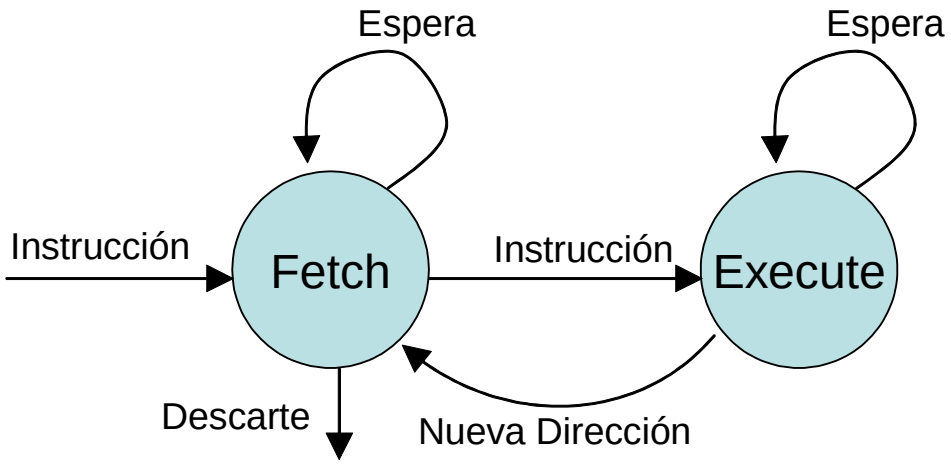
Este diagrama sin embargo no explicita algunas características de esta división en sub-tareas que impactan sobre el funcionamiento del pipeline:

- En los procesadores Intel las instrucciones son de largo variable, por lo que la etapa de “Fetch” puede involucrar más de un acceso a memoria. Es decir no tiene una duración fija.
- Por una razón similar (se trata de una arquitectura CISC) la etapa de “Execute” tendrá una duración que dependerá significativamente del tipo de instrucción y de los tipos de operando involucrados.

Ambas situaciones provocarán situaciones de espera en una ú otra etapa: o bien se hizo el fetch de la próxima instrucción, pero aún no terminó de ejecutarse la anterior, o bien la etapa de execute completó la ejecución de una instrucción, pero la etapa de “Fetch” todavía está leyendo de memoria una instrucción muy larga.

También existe una situación de interrelación: las instrucciones de salto pueden provocar que se deba hacer el “fetch” de un lugar distinto al siguiente en la memoria (que es lo que asume la etapa de “Fetch”). En este caso se debe descartar la instrucción leída en la operación de fetch completada ó en curso e iniciar un nuevo fetch a la nueva dirección (la determinada por el salto).

Poniendo estos elementos en forma explícita el diagrama quedaría:



18.3.2 Pipeline de 5 etapas

Como vimos en un capítulo anterior, una posible descomposición del ciclo de instrucción incluye 5 etapas:

Fetch: la instrucción es leída de memoria y se incrementa el PC (Program Counter)

Decode: la instrucción es decodificada y las correspondientes señales de control activadas.

Read: los operandos en registros o inmediatos son apropiadamente movidos a registros de trabajo de la ALU. Los operandos en memoria son leídos y traídos a los registros auxiliares de la ALU.

Execute: se ejecuta la operación. Si es un salto condicional relativo se evalúa la condición y se calcula la dirección destino.

Write: se transfiere el resultado de la ejecución desde el registro auxiliar hacia el registro destino ó la memoria.

Si asumimos que todas las etapas se ejecutan en el mismo tiempo (ej: un ciclo de reloj), el diagrama de tiempos del pipeline sería:

	tiempo									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción 1	F	D	R	E	W					
Instrucción 2		F	D	R	E	W				
Instrucción 3			F	D	R	E	W			
Instrucción 4				F	D	R	E	W		
Instrucción 5					F	D	R	E	W	
Instrucción 6						F	D	R	E	W

Aun suponiendo que el *Fetch* se haga en un solo ciclo de reloj (lo que no es cierto si se tratara de una arquitectura CISC que tiene largo variable de instrucción), hay algunos elementos que pueden afectar el funcionamiento óptimo del pipeline. El más evidente es que existe el mismo problema con las instrucciones de salto que vimos para el caso anterior. Ya sea que identifiquemos el salto en la etapa de *Decode* (si es incondicional) ó en la etapa de *Execute* (si es condicional), las instrucciones siguientes que ya se hayan leído se deberán descartar.

Este no es el único problema como veremos más adelante.

18.3.3 Pipeline MIPS de 5 etapas

La arquitectura MIPS (microprocesador RISC desarrollado originalmente en la Universidad de Stanford) tenía, en su diseño original, un pipeline de 5 etapas:

Fetch: la instrucción es leída de memoria y se incrementa el PC (Program Counter)

Decode / Register Fetch: la instrucción es decodificada y las correspondientes señales de control activadas. Los operandos en registros o inmediatos son apropiadamente movidos a registros de trabajo de la ALU. En caso de salto condicional, la condición es verificada y la dirección a saltar calculada.

Execute: se ejecuta la operación. Si la instrucción es de memoria (load/store) se calcula la dirección de memoria a acceder.

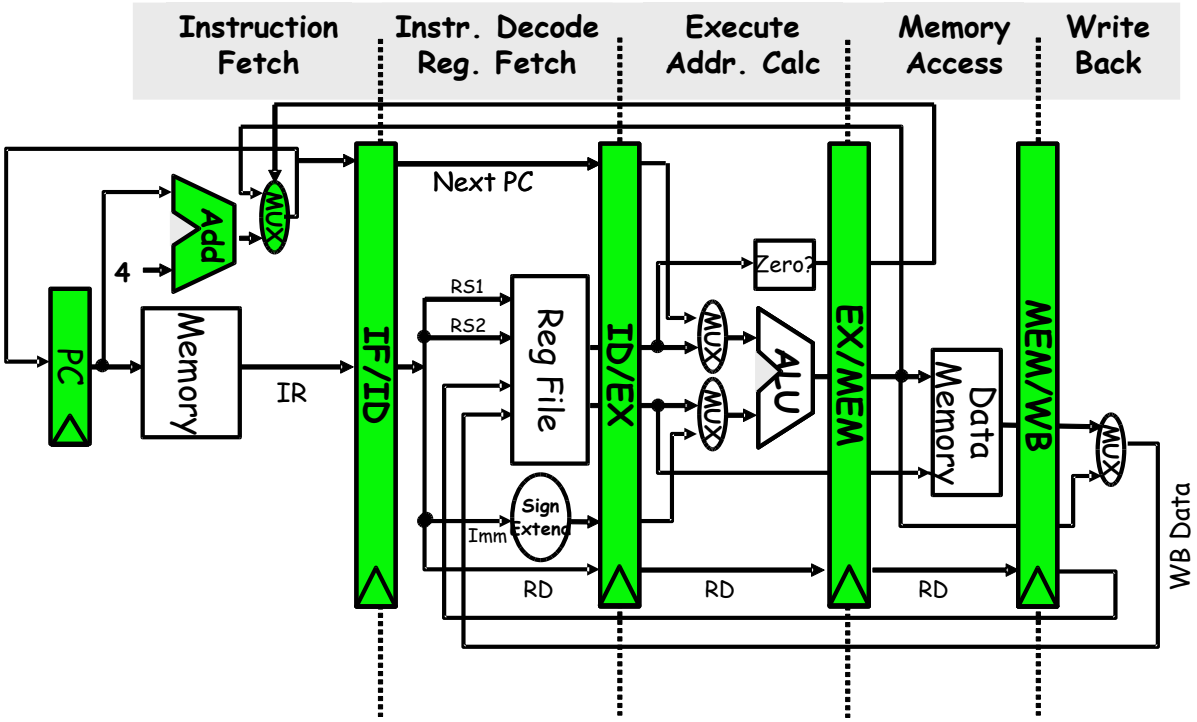
Memory Read/Write: si la operación es de memoria se lee o escribe la misma.

Write Back: se transfiere el resultado de la ejecución desde el registro auxiliar hacia el registro destino.

Los diseñadores de MIPS eligieron que todas las instrucciones recorran todas las etapas aún cuando no las requieran. Esto incluso para el caso de la etapa de acceso a

memoria, que solo está presente en las instrucciones LOAD y STORE del set de instrucciones. Esto es porque optaron por un diseño del pipeline radical: no tiene mecanismos de hardware para resolver ese tipo de situaciones, donde eventualmente una etapa del proceso no se ejecute. De hecho la sigla MIPS corresponde a Microprocessor without Interlocks Pipeline Stages (Microprocesador sin Dependencias en las Etapas de Pipeline).

El siguiente es un diagrama de las etapas del pipeline de MIPS y su vinculación con los recursos de hardware que utiliza en cada una de ellas:



18.3.4 Pipeline SPARC de 4 etapas

La arquitectura SPARC (microprocesador RISC basado en un desarrollo originado de la Universidad de Berkeley) posee, en su diseño original, un pipeline de 4 etapas

Fetch: la instrucción es leída de memoria y se incrementa el PC (Program Counter)

Execute: la instrucción es decodificada y se ejecuta la operación correspondiente. Si la instrucción es de memoria (load/store) se calcula la dirección de memoria a acceder, si es un salto relativo se computa la dirección destino.

Memory: si la operación es de memoria se lee o escribe la misma.

Store: se transfiere el resultado de la ejecución desde el registro auxiliar hacia el registro destino.

18.4 Aceleración del Pipeline

Podemos calcular el aumento de la velocidad de procesamiento de instrucciones que se logra con la técnica de pipeline de la siguiente manera.

Supongamos un pipeline de ***k*** etapas, cada una de duración ***T***. Para ejecutar ***n*** instrucciones tendríamos los siguientes tiempos:

Sin pipeline: $T_{tot} = nkT$

Con pipeline: $T_{tot} = [n + k - 1]T$

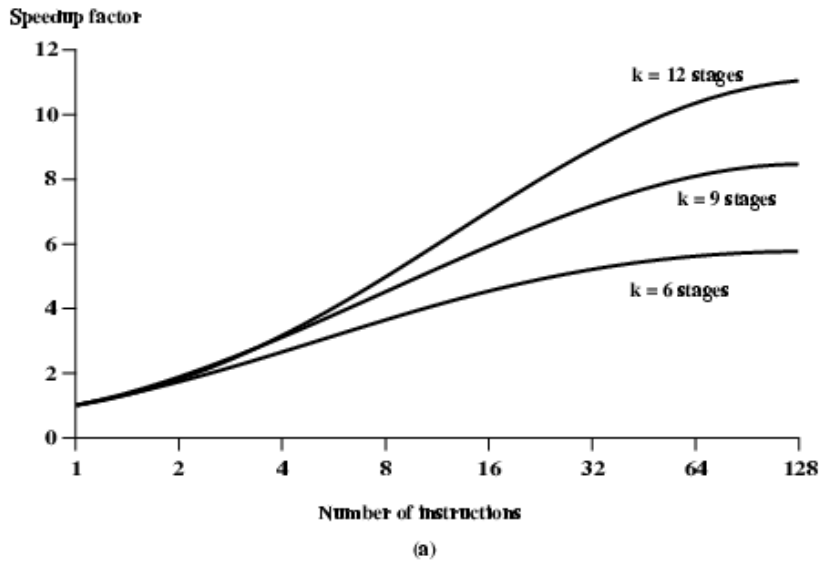
El primer tiempo es sencillo de justificar: si las instrucciones se ejecutan una tras otra, cada una demora ***kT*** (ya que cada etapa demora ***T*** y hay ***k*** etapas) y por tanto ***n*** instrucciones demoran lo expresado en la fórmula.

Para el caso del pipeline, si recordamos los diagramas de tiempo vistos, vemos que el tiempo total es el correspondiente a ***n*** veces el de la etapa de *fetch* y luego debe completarse la ultima instrucción, lo que requiere ***k-1*** etapas, de allí la expresión de la fórmula.

Por tanto podemos definir la "aceleración" del pipeline como la relación entre ambos tiempos de ejecución:

Aceleración: $S = nk / [n + k - 1]$

El siguiente gráfico muestra como mejora el rendimiento a medida que aumentamos el número de instrucciones ejecutadas, para distintos diseños de pipeline:



Notemos que si ***n >> k***, entonces ***S = k***, lo que coincide con un resultado ya encontrado: la ganancia de rendimiento coincide con el número de etapas. Como ya se adelantó la técnica de agregar cada vez mas etapas para lograr mayores rendimientos tiene la limitante de que a medida que ***k*** crece se hace más difícil mantener la independencia de las etapas y la similitud en la duración de las mismas, principios básicos del funcionamiento del pipeline y de la expresión hallada para la aceleración.

18.5 Obstáculos / Peligros (Hazards)

Como dijimos para que un pipeline pueda funcionar de manera óptima, no deben existir dependencias entre las etapas del mismo. En la práctica esto no es posible. El ejemplo más simple es el del salto (condicional o no), ya que invalida el *fecth* de la próxima instrucción ya realizado (y eventualmente otras etapas que también haya recorrido la próxima instrucción y, eventualmente las siguientes, dependiendo del momento en que se determina el salto).

Hay otros posibles problemas y situaciones que generan problemas para el funcionamiento óptimo del pipeline. En general estas situaciones se denominan en inglés *hazards*, término que puede ser traducido como *obstáculo* ó *peligro*.

Estos hazards se clasifican en tres grandes grupos:

Hazards Estructurales: hay un conflicto de hardware entre dos o más etapas del pipeline para alguna combinación de instrucciones.
Hazards de Datos: existen dependencias de datos entre instrucciones. Por ejemplo la ejecución de una instrucción depende del resultado de otra previa, que aún está en el pipeline.
Hazards de Control: causados por instrucciones de salto u otras modificaciones del registro IP (Instruction Pointer ó PC = Program Counter).

18.5.1 Hazards Estructurales

Un ejemplo de este tipo de *hazard* se da en el hardware de acceso a memoria entre la etapa de *fetch* y alguna que lea ó escriba en memoria:

	tiempo									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción 1	F	D	R	E	W					
Instrucción 2		F	D	R	E	W				
Instrucción 3			F	D	R	E	W			
Instrucción 4				F	D	R	E	W		
Instrucción 5					F	D	R	E	W	
Instrucción 6						F	D	R	E	W

En este ejemplo (señalado con azul) si la instrucción 1 tiene efectivamente etapa de *Read* (o sea posee un operando en memoria) se produce un conflicto entre dicha etapa y la etapa de *Fetch* de la instrucción 3 por el uso del bus de comunicación con la memoria: la instrucción 1 necesita leer un operando de la memoria y a la vez se necesita leer la instrucción 3. Esta situación podría resolverse mediante una arquitectura *Harvard*, en la cual tenemos memorias separadas para datos e instrucciones.

Otro ejemplo (señalado con rojo) podría darse si pensamos que la etapa de *Fetch* necesita calcular la dirección de la próxima instrucción (sumando 4, por ejemplo en el caso de RISC). Si para esta operación intentara utilizar la ALU generaría un conflicto con la etapa de *Execute* de otra instrucción (ej: instrucciones 1 y 4 del diagrama de tiempo). Esto se puede solucionar agregando un sumador independiente de la ALU para el cálculo de la dirección de la próxima instrucción.

Una solución general al problema de la falta de un recurso de hardware es esperar. La espera no es una solución óptima pero es la más sencilla, en el sentido que no agrega hardware. Una forma de esperar es agregar, artificialmente, operaciones del tipo “NOP” (No OPeration), que no requieren de recursos. Esta técnica también se denomina: burbuja de pipeline.

Es habitual encontrar una combinación de soluciones: se agrega hardware para los casos más comunes (ej: sumador en la etapa de *Fetch*) y se implementan esperas para los casos menos comunes.

18.5.2 Hazards de Datos

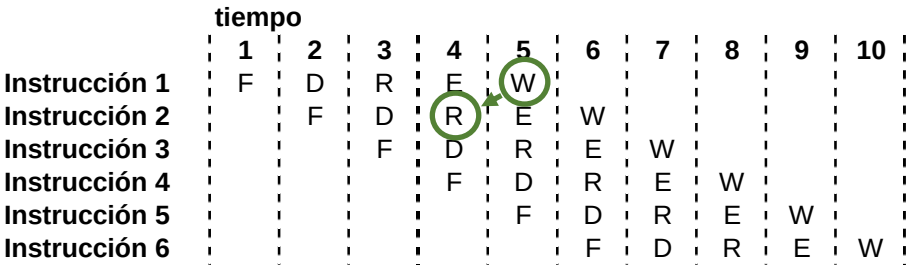
En este caso el problema aparece porque hay datos en común entre instrucciones que están en el pipeline, ya sea una instrucción siguiente que utiliza un resultado aún no calculado como operando ó altera un operando antes que sea leído como operando.

Se distinguen 3 tipos de hazards de datos:

RAW (Read After Write): ocurre cuando una instrucción posterior intenta obtener un operando que está siendo calculado por una instrucción anterior, que aún está en el pipeline y aún no llegó a la etapa donde escribe el resultado en el registro destino (W ó WB). Este tipo de obstáculo se denomina “dependencia”.
Veamos un ejemplo (usaremos instrucciones de 3 operandos, el primero es destino y los otros dos son origen):

```
add r1, r2, r3
sub r4, r1, r3
```

en este caso la segunda instrucción (sub) necesita como operando origen el contenido de r1, pero si suponemos un pipeline de 5 etapas (F-D-R-E-W) el r1 debe volcarse al registro auxiliar en la etapa R de la segunda instrucción, pero su valor recién se guardará en r1 al realizarse la etapa W de la primer instrucción, lo que ocurre mas adelante como vemos en el siguiente diagrama de tiempos:



WAR (Write After Read): ocurre cuando una instrucción posterior escribe un registro que actúa como operando de una instrucción anterior antes que ésta pueda leerlo. Este tipo de hazard de datos, que se denomina “anti-dependencia” no es posible en los pipelines que hemos visto (porque en todos la etapa de *Write* está luego de la de *Read*). Sin embargo es posible que sucedieran en diseños más avanzados (del tipo “out of order execution”), los que obviamente dispondrán de mecanismos para evitarlos.

WAR (Write After Write): ocurre cuando una instrucción anterior escribe un registro que también actúa como resultado de una instrucción posterior después de ésta. Este tipo de hazard de datos, que se denomina “dependencia de salida” tampoco es posible en los pipelines que hemos visto (porque en todos la etapa de *Write* está luego de la de *Read*). También como en el caso anterior es posible que sucedieran en diseños más avanzados (del tipo “out of order execution”), los que obviamente dispondrán de mecanismos para evitarlos.

Para el caso del RAW las formas de resolver el problema pueden ser:
- esperar (introducir una burbuja en el pipeline)
- recurrir a técnicas avanzadas como la ejecución fuera de orden (out of order execution)

- hacer ***register forwarding***. Esta técnica consiste en propagar hacia las etapas tempranas del pipeline los resultados apenas estén disponibles, sin necesidad de esperar a que culmine la etapa de *Write*. En este caso la etapa de *Read* podrá leer el operando desde el registro indicado ó desde la salida de la ALU directamente, en función de la lógica de control que implemente esta funcionalidad.

18.5.3 Hazards de Control

Los obstáculos de este tipo lo constituyen los ya comentados problemas que se generan al ejecutarse saltos, ya sea incondicionales o condicionales. La etapa de *Fetch* asume, por defecto, que la próxima instrucción a ejecutarse es la que está a continuación en memoria, lo que nunca es correcto si la instrucción es un salto incondicional ó no es siempre correcto si se trata de un salto condicional.

Las técnicas para resolver la penalización provocada por este tipo de hazards incluyen: prefetch del destino, multiples flujos (streams), salto demorado y predicción del salto.

18.5.3.1 Prefetch del destino

Esta técnica consiste en realizar simultáneamente el fetch de la próxima instrucción en memoria y de la instrucción apuntada por la dirección destino del salto. Cuando el salto se efectiviza la instrucción que no corresponde a la secuencia lógica se descarta.

Esta técnica fue utilizada inicialmente en la IBM 360.

18.5.3.2 Múltiples flujos

Este caso va un poco más allá del anterior y duplica las etapas del pipeline que están ubicadas hasta la que decide el salto. De esta manera se puede avanzar en la ejecución de ambos flujos de instrucciones, por lo que al momento de tomar el salto no hay ninguna penalización, ya que no es necesario “vaciar” el pipeline (en realidad se vacía uno de los dos: el que estaba ejecutando el flujo que finalmente no resultó elegido por el salto).

Esta técnica fue utilizada inicialmente en la IBM 370. Tiene el inconveniente de propender a generar hazards estructurales.

18.5.3.3 Salto demorado

El salto demorado es una técnica muy utilizada en los procesadores RISC. La idea es simple: siempre se ejecuta la instrucción que está luego del salto (se dice que la instrucción luego del salto está ubicada en el “delayed slot”). Es responsabilidad del compilador (ó del programador de bajo nivel) acomodar allí una instrucción útil. Si no hay forma de hacerlo se deberá colocar un NOP y se recomienda nunca colocar dos saltos seguidos (en algunos casos si se hace se produce una “excepción”).

La idea se complementa con el hecho que los procesadores que utilizan este recurso deben resolver las instrucciones de salto en la segunda etapa del pipeline. De esta manera no incurren en penalización por salto, ya que no se requiere vaciar el pipeline. Esto es cierto para las primeras generaciones de procesadores RISC (ej: los primeros diseños de

MIPS y SPARC).

18.5.3.4 Predicción del salto

Como no siempre es posible alocar una instrucción útil en el “delayed slot” se termina teniendo una penalización de todos modos, lo que también ocurre cuando tenemos pipelines más complejos que resuelven los saltos en etapas mas avanzadas del mismo. Por esto es que se recurre a otro tipo de técnicas (que pueden combinarse con el salto demorado), como ser la predicción del salto, que aplica a los saltos condicionales, naturalmente.

La técnica consiste en determinar, en base a un cierto algoritmo o criterio, si el salto se va a tomar o no. Esto permite seguir haciendo el fetch de las instrucciones que con cierta probabilidad serán las que el flujo lógico del programa ejecute. Si la predicción es acertada no habrá penalización ya que no será necesario vaciar ninguna etapa del pipeline. Por tanto en este caso la eficiencia del pipeline está condicionada por la eficiencia del mecanismo de predicción del salto condicional.

Hay distintos mecanismos utilizados en la predicción:

Asumir que nunca se ejecuta: este criterio es como si se considerara el salto como una instrucción cualquiera. Siempre se carga la próxima instrucción en memoria.

Asumir que siempre se ejecuta: este criterio continúa la carga de instrucciones en la dirección destino del salto. Estadísticamente los saltos condicionales se toman en más del 50% de los casos.

Predicción basada en la condición del salto: parte de la constatación que determinados tipos de condición del salto (que están codificados en el código de operación de la instrucción de salto) tienen mayor tendencia a realizar el salto y otros a no realizarlo. Con esta técnica se consigue más del 75% de aciertos.

Conmutar Taken / No Taken: utiliza la historia reciente de los saltos, mediante una máquina de estados que “recuerda” si el último salto fue tomado o no. La predicción debe fallar dos veces seguida para cambiarla. El problema de la técnica es que predice el comportamiento de un salto en función de lo que hizo uno anterior que no tiene por qué ser el mismo. Por eso es que es apropiado para bucles (loops), pero no en general.

Tabla de Historia de Saltos (BHT = Branch History Table): es un perfeccionamiento del mecanismo anterior, donde se almacena en una tabla la información de un cierto conjunto de saltos recientemente evaluados. Entonces permite realizar el algoritmo anterior sobre el comportamiento del mismo salto, sin que sea interferido por otros saltos. Esto mejora la eficiencia de la predicción.

18.6 Ejecución Fuera de Orden (Out of Order Execution)

Una de las propuestas más radicales para atender toda la problemática de los distintos tipos de “hazards” que ocurren en los pipeline es la posibilidad de realizar la ejecución de las instrucciones fuera del orden establecido por su ubicación en memoria pero, obviamente, respetando su orden lógico para mantener la integridad de los resultados.

Para lograr esto los procesadores que utilizan esta técnica manejan un conjunto de instrucciones que ya fueron leídas desde la memoria hacia una especie de “estacionamiento, y las van catalogando como “prontas para ejecutar” o no en función que sus dependencias de datos y control hayan sido satisfechas. Las instrucciones “prontas para ejecutar” avanzan en el pipeline a la etapa de ejecución y a medida que se van generando nuevos resultados, se van actualizando los estados de las instrucciones que aún

esperan. De esta manera nuevas instrucciones quedan prontas para ejecutar y así sigue el proceso.

El orden lógico de ejecución es el que determina, en definitiva, la existencia de restricciones de datos o control (los hazards) y por tanto es tenido en cuenta al momento de determinar la condición de “pronta para ejecutar” de una instrucción.

Por ejemplo en el trozo de programa:

```
add r1, r2, r3
add r5, r2, r4
sub r4, r1, r3
```

las dos primeras instrucciones pueden ejecutarse en cualquier orden que el resultado del programa no va a ser alterado, pero la instrucción 3 (tres) debe ejecutarse luego de la 1 (uno) porque depende de ella. Suponiendo que las tres instrucciones están en un momento en el “estacionamiento” y no hay instrucciones pendientes que guarden resultados en r2 ó r3, las dos primeras instrucciones recibirán la categoría de “prontas para ejecutar”, pero no la recibirá la tercera. Cuando la primer instrucción complete la ejecución, la condición de la tercer instrucción pasará a “pronta para ejecutar”.

Un sistema de este tipo, si bien tiene una lógica de control muy compleja, permite altas tasas de ocupación de las etapas del pipeline y, en particular, de la ALU. Es decir logra una gran eficiencia del pipeline.