Pipelining o Segmentación de Instrucciones

La segmentación de instrucciones es similar al uso de una cadena de montaje en una fábrica de manufacturación. En las cadenas de montaje, el producto pasa a través de varias etapas de producción antes de tener el producto terminado. Cada etapa o segmento de la cadena está especializada en un área específica de la línea de producción y lleva a cabo siempre la misma actividad. Esta tecnología es aplicada en el diseño de procesadores eficientes. A estos procesadores se les conoce como *pipeline processors*.

Un *pipeline processor* está compuesto por una lista de segmentos lineales y secuenciales en donde cada segmento lleva a cabo una tarea o un grupo de tareas computacionales. Puede ser representado gráficamente en dos dimensiones, en donde en el eje vertical encontramos los segmentos que componen el pipeline y en el segmento horizontal representamos el tiempo (Figura 1).

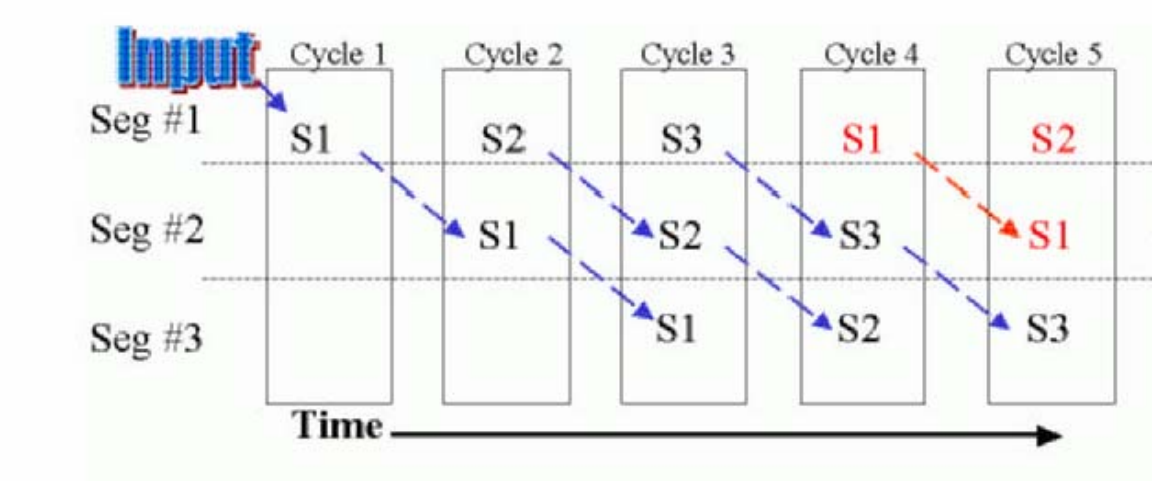


Figura 1. Diagrama notacional de un pipeline processor

Hay tres apectos importantes que deben ser considerados en *pipeline*. Lo primero que debemos observar es que el trabajo es dividido en piezas que más o menos ajustan dentro de los segmentos que componen el *pipeline*. Segundo, para que el *pipeline* trabaje de forma eficiente es necesario que las particiones de trabajo tomen aproximadamente la misma cantidad de tiempo. De no ser

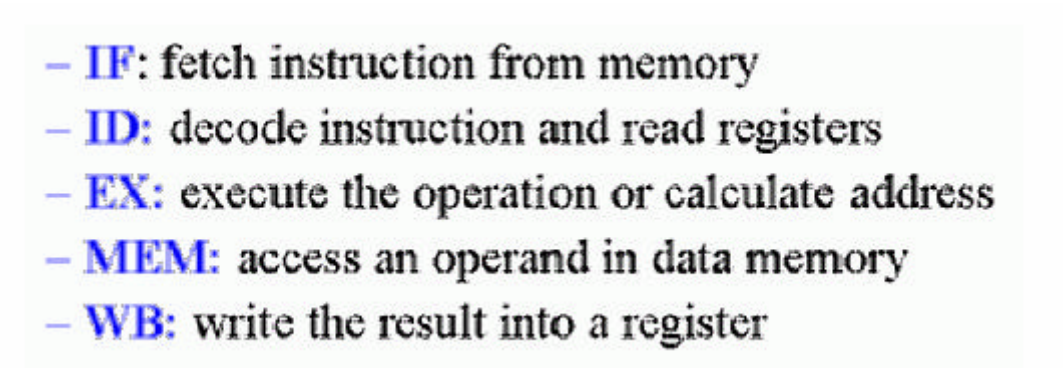
asi, el segmento que requiera más tiempo (T) hará que el *pipeline* se retrase y cada segmento requerirá T unidades de tiempo para completar su trabajo. Esto quiere decir que los segmentos rápidos estarán mucho tiempo ociosos. Tercero, para que el *pipeline* funcione adecuadamente, deben ocurrir pocas excepciones o *hazards* (riesgos) que puedan causar retardos o errores en el *pipeline*. En caso de ocurrir errores, la instrucción tiene que ser cargada nuevamente en el *pipeline* y se debe reiniciar la misma instrucción que ocasionó la excepción.

Como una primera aproximación, consideremos una subdivisión del procesamiento de una instrucción en dos etapas o segmentos:

- Captación de la instrucción
- Ejecución de la instrucción

La primera etapa capta una instrucción y la almacena en un *buffer*. Cuando la segunda etapa está libre, la primera etapa le pasa la instrucción almacenada. Mientras la segunda etapa ejecuta la instrucción, la primera etapa usa algún ciclo de memoria no utilizado para captar y almacenar la siguiente instrucción. A esto se le conoce como prebúsqueda o precaptación de instrucción. Este proceso acelerá la ejecución de instrucciones. Si las dos etapas consideradas fueran de igual duración, el tiempo de ciclo de instrucción se reduciría a la mitad.

Para poder tener una aceleración mayor, el *pipeline* debe tener más etapas. Consideremos la siguiente división en 5 etapas que presenta la arquitectura MIPS:



Comparemos el tiempo promedio de ejecución en una implementación sin *pipeline* y con *pipeline*. Supongamos que se necesitan 2 nseg para acceso a memoria, 2 nseg para una operación en la Unidad Lógica Aritmétrica (ULA) y 1 nseg para leer o escribir registros. En la figura 2 se muestra el tiempo total para la ejecución de varios tipos de instrucciones tomando en cuenta el tiempo que demora cada uno de los componentes.

	IF	ID	ALU	MEM	WB	Total
lw	2 nseg	1 nseg	2 nseg	2 nseg	1 nseg	8 nseg
sw	2 nseg	1 nseg	2 nseg	2 nseg		7 nseg
add	2 nseg	1 nseg	2 nseg		1 nseg	6 nseg
branch	2 nseg	1 nseg	2 nseg			5 nseg

Figura 2. Tiempo total de ejecución de diversos tipos de instrucciones calculado a partir del tiempo de cada componente

En una implementación sin *pipeline*, cada instrucción toma exactamente 1 ciclo de reloj, por lo tanto el ciclo de reloj debe ser ajustado a la instrucción más lenta. En nuestro caso, el ciclo de reloj es ajustado a 8 nseg. En la figura 3 se muestra la secuencia de ejecución para dos instrucciones en un procesador sin *pipeline* y en la figura 4 se muestra el caso para un procesador con *pipeline.*

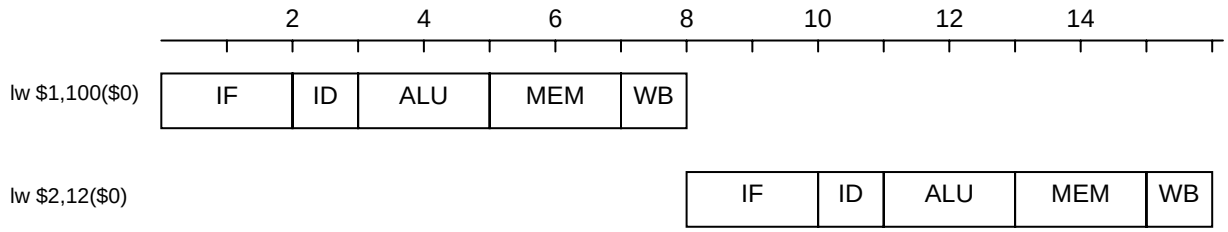


Figura 3. Tiempo total de ejecución en un procesador sin pipeline

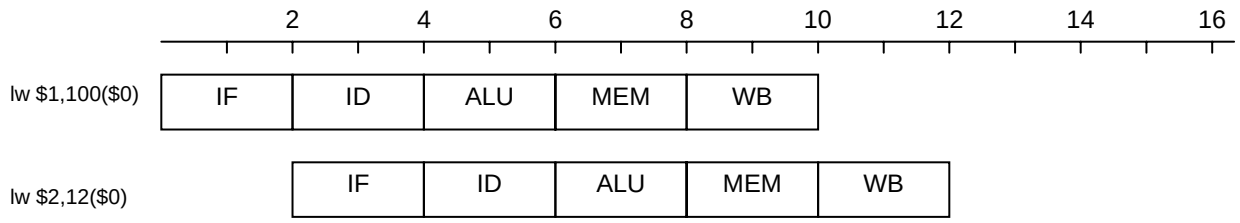


Figura 4. Tiempo total de ejecución en un procesador con pipeline

Para implementar eficientemente las instrucciones MIPS en un procesador *pipeline*, se debe asegurar que las instrucciones sean de la misma longitud y por lo tanto se facilitan las etapas de IF e ID. Además esta arquitectura posee pocos formatos de instrucción que son consistentes entre si que permiten una decodificación rápida de las instrucciones pues los campos de registros en los formatos siempre ocupan la misma posición dentro del formato. Adicionalmente, la decodificación de las instrucciones y la lectura de los contenidos de los registros ocurren al mismo tiempo.

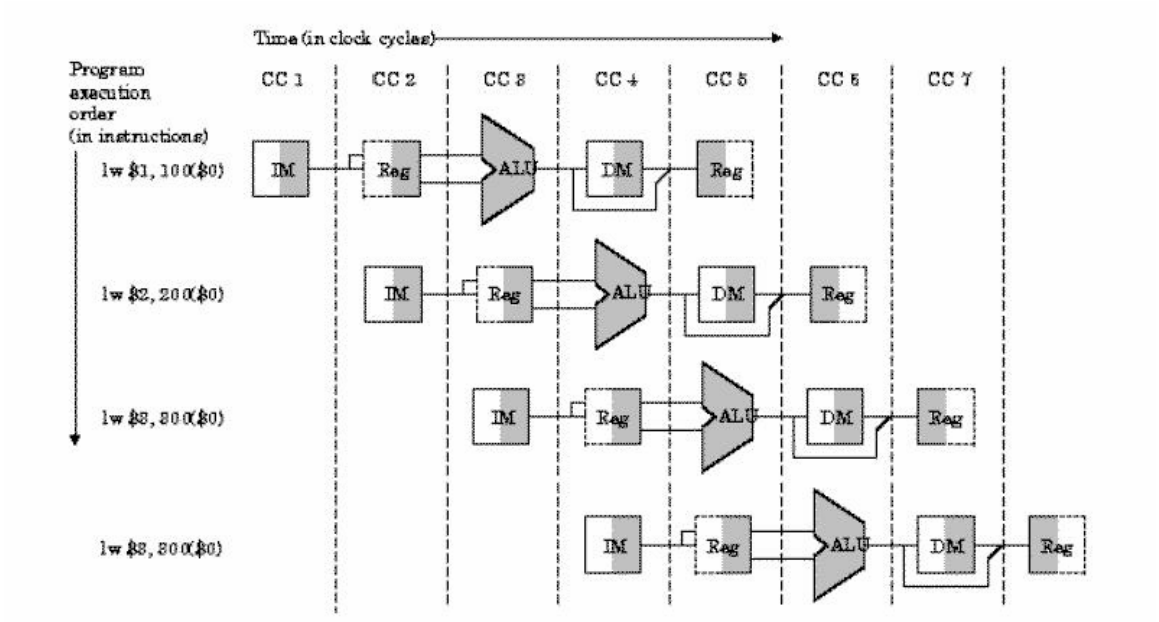


Figura 5. Diagrama espacial del pipeline

Pipelining incrementa la productividad del CPU, es decir, el número de instrucciones completadas por unidad de tiempo, pero no reduce el tiempo de ejecución de una instrucción individual. De hecho, usualmente se incrementa el tiempo de ejecución de cada instrucción debido al *overhead* producido por el control del *pipeline*. El incremento en la productividad de las instrucciones significa que un programa corre más rápido y tiene menor tiempo total de ejecución.

Para poder implementar un procesador con *pipeline* se debe determinar que sucede en cada ciclo de reloj y estar seguros que en el solapamiento de las instrucciones no tengamos competencia por ciertos recursos. Por ejemplo, si se dispone de una única ALU entonces no se podrá calcular la dirección efectiva de un operando y al mismo tiempo llevar a cabo una operación de resta. Cada etapa del *pipeline* está activa en cada ciclo de reloj. Esto requiere que todas las operaciones en una etapa se completen en un ciclo de reloj.

Riesgos en el *pipeline*

Los procesadores con *pipeline* presentan una serie de problemas conocidos como *hazards*, y que pueden ser de tres tipos:

- Riesgos Estructurales: ocurren cuando diversas instrucciones presentan conflictos cuando tratan de acceder a la misma pieza de hardware. Este tipo de problema puede ser aliviado teniendo hardware redundante que evitan estas colisiones. También se pueden agregar ciertas paradas (*stall*) en el *pipeline* o aplicar reordenamiento de instrucciones para evitar este tipo de riesgo.
- Riesgos de Datos: ocurren cuando una instrucción depende del resultado de una instrucción previa que aún está en el *pipeline* y cuyo resultado aún no ha sido calculado. La solución más fácil es introducir paradas en la secuencia de ejecución pero esto reduce la eficiencia del *pipeline*.
- Riesgos de Control: son resultado de las instrucciones de salto que necesitan tomar una decisión basada en un resultado de una instrucción mientras se están ejecutando otras.

Riesgos Estructurales o *Structural Hazards*

Ocurre cuando no se dispone de suficiente hardware para soportar ciertos cómputos dentro de un segmento particular del *pipeline*. Por ejemplo en la figura 6 se puede observar que en el ciclo de reloj 5 (CC5) se debe llevar a cabo una escritura en el archivo de registros y una lectura del archivo de registros. Esto puede resolverse duplicando el archivo de registros pero puede llevarnos a problemas de inconsistencia.

Otra alternativa es modificar el hardware existente para que pueda soportar operaciones de lectura y escritura concurrentes. Se puede modificar el archivo de registros de forma tal que la escritura de registros se lleva a cabo en la primera mitad del ciclo de reloj y la lectura de registros se efectúa en la segunda mitad del ciclo de reloj.

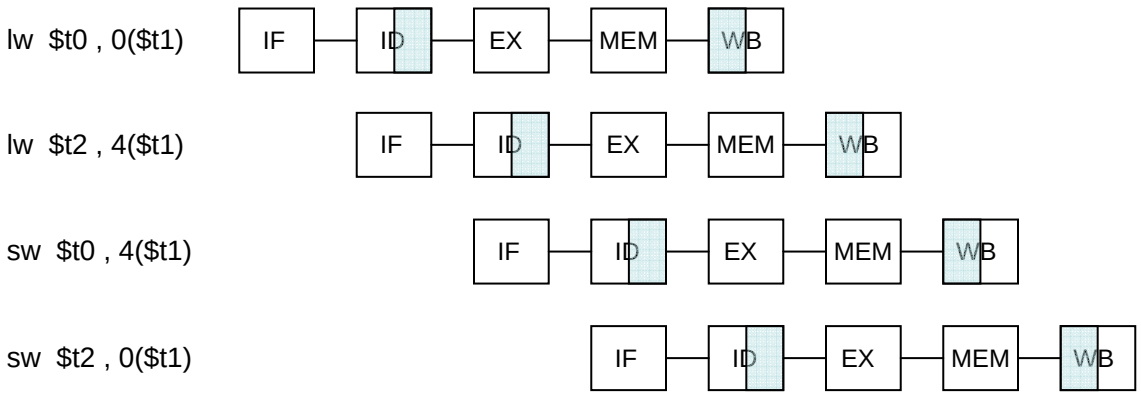


Figura 6. Riesgo estructural en el acceso al archivo de registros

Otro riesgo estructural puede ocurrir durante una instrucción de salto, si no disponemos de dos ALU en el segmento EX. Para este tipo de instrucción es necesario calcular la dirección de salto pues en la instrucción se dispone del desplazamiento necesario para llegar al destino y también es necesario calcular la diferencia entre los operandos fuentes para determinar si se cumple o no la condición.

Riesgos de Datos o *Data Hazards*

Un *data hazard* ocurre cuando la instrucción requiere del resultado de una instrucción previa y el resultado aún no ha sido calculado y escrito en el archivo de registros. En la figura 7 se presenta un ejemplo de riesgo de datos.

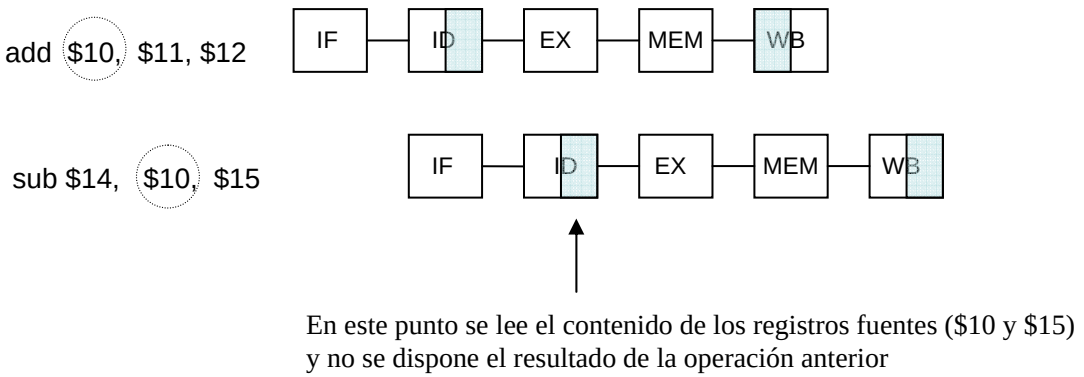


Figura 7. Riesgo de Datos en el registro $10

Hay diversas formas para solventar este problema:

- *Forwarding*: se adiciona circuitos especiales en el *pipeline* de forma que el valor deseado es transmitido/adelantado al segmento del *pipeline* que lo necesita (Figura 8).

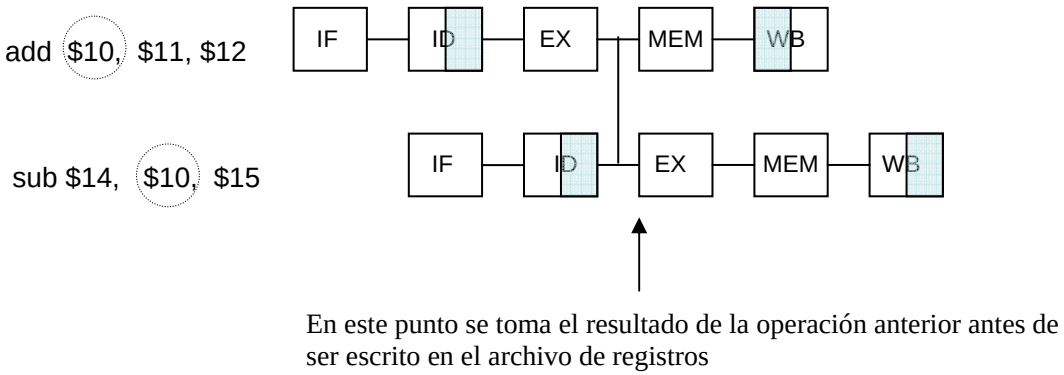


Figura 8. Forwarding para evitar riesgo de datos en el registro $10

- Inserción de paradas (*Stall insertion*): es posible insertar una o más paradas en el *pipeline* (no-op) que retardan la ejecución de la instrucción actual hasta que el dato requerido sea escrito en el archivo de registros (Figura 9).

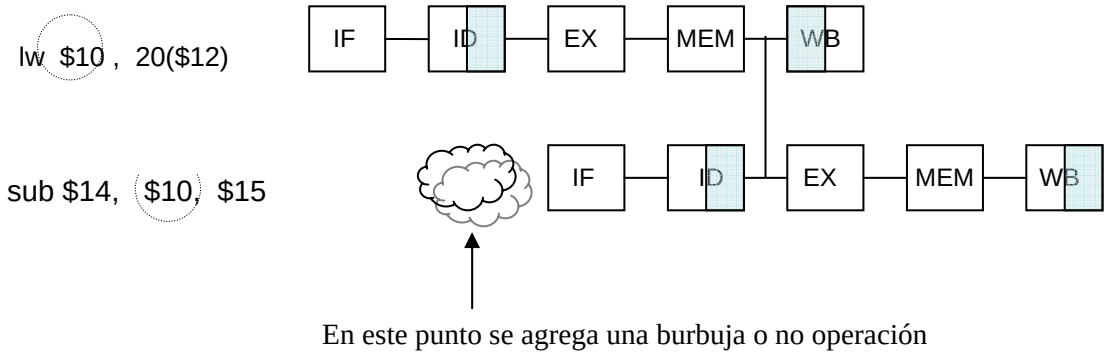


Figura 9. Inserción de no-op para evitar riesgo de datos en el registro $10

- Reordenamiento de código: en este caso, el compilador reordena instrucciones en el código fuente o el ensamblador reodena el código objeto de forma que se colocan una o más instrucciones entre las instrucciones que presentan dependencia de datos. Para esto se necesitan compiladores o ensambladores inteligentes.

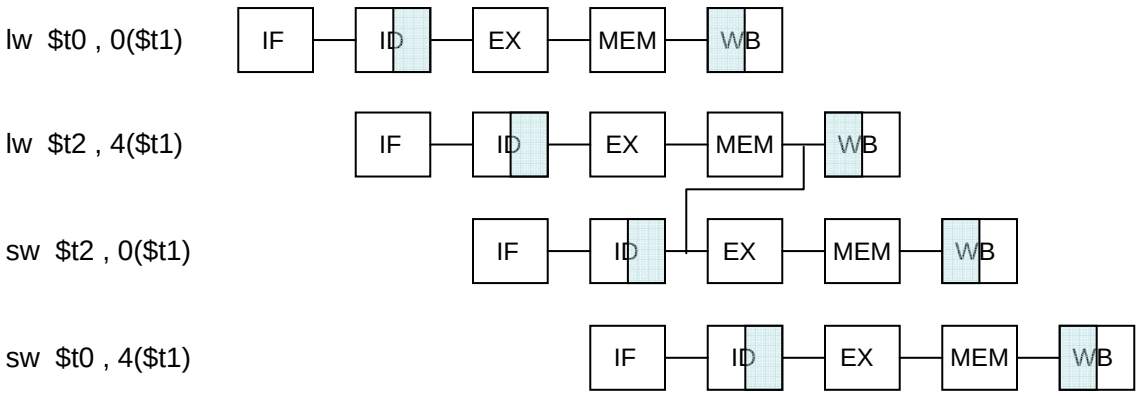


Figura 10. Ejemplo de riesgo de datos

En la tercera instrucción no se dispone del resultado de la segunda y no se puede aplicar adelantamiento del resultado (Figura 10) . Una alternativa es que se incluya una parada o burbuja en el *pipeline* (Figura 11).

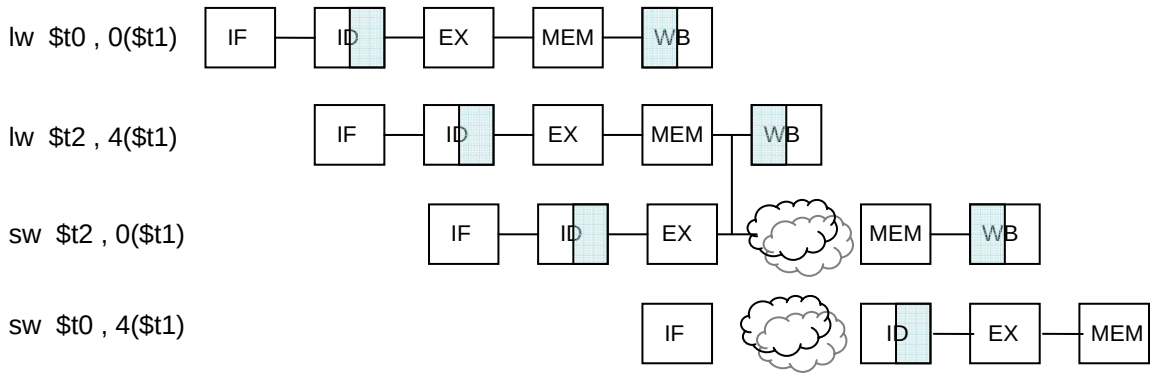


Figura 11. Ejemplo de riesgo de datos con inserción de no-op

Otra forma más eficiente es cambiando el orden de ciertas instrucciones. En este caso si intercambiamos la instrucción 3 y 4 resolvemos el problema de dependencia de datos (Figura 12).

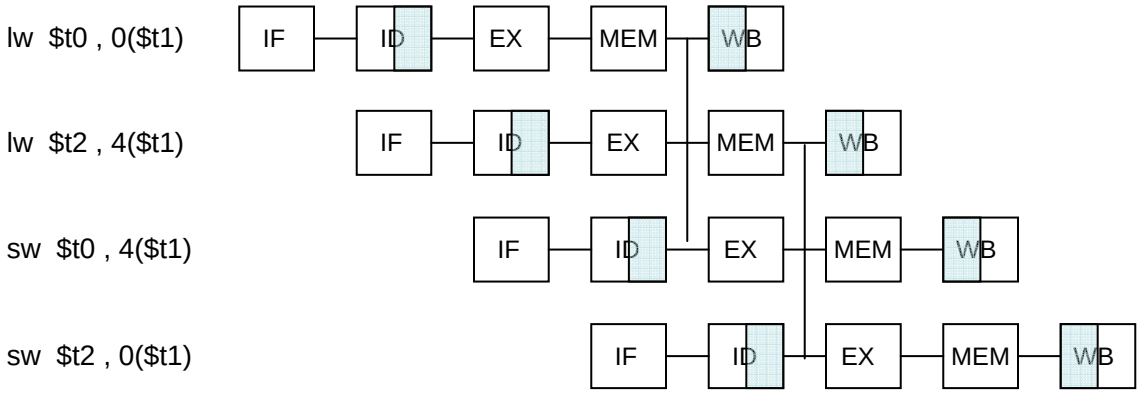


Figura 12. Reordenamiento de instrucciones para evitar un riesgo de datos

Riesgos de Control o Control Hazards

Un riesgo de control se produce cuando es necesario llevar a cabo una decisión basada en el resultado de una instrucción mientras se están ejecutando otras instrucciones. Cuando se ejecuta un salto no se conoce de antemano cuál será la siguiente instrucción que deberá ser ejecutada. Si la condición del salto falla entonces se debe ejecutar la instrucción inmediata, si la condición del salto se cumple se debe actualizar el PC con la dirección de la siguiente instrucción que debe ejecutarse.

Una estrategia para atacar este tipo de problema es asumir que la condición del salto no se cumplirá y por lo tanto la ejecución continuará con la instrucción que se encuentra inmediatamente después de la instrucción de salto. Si la condición del salto se cumplió entonces se deberá desechar del *pipeline* las instrucciones que fueron captadas y la ejecución continua con la instrucción que se encuentra en la dirección de salto (Figura 13).

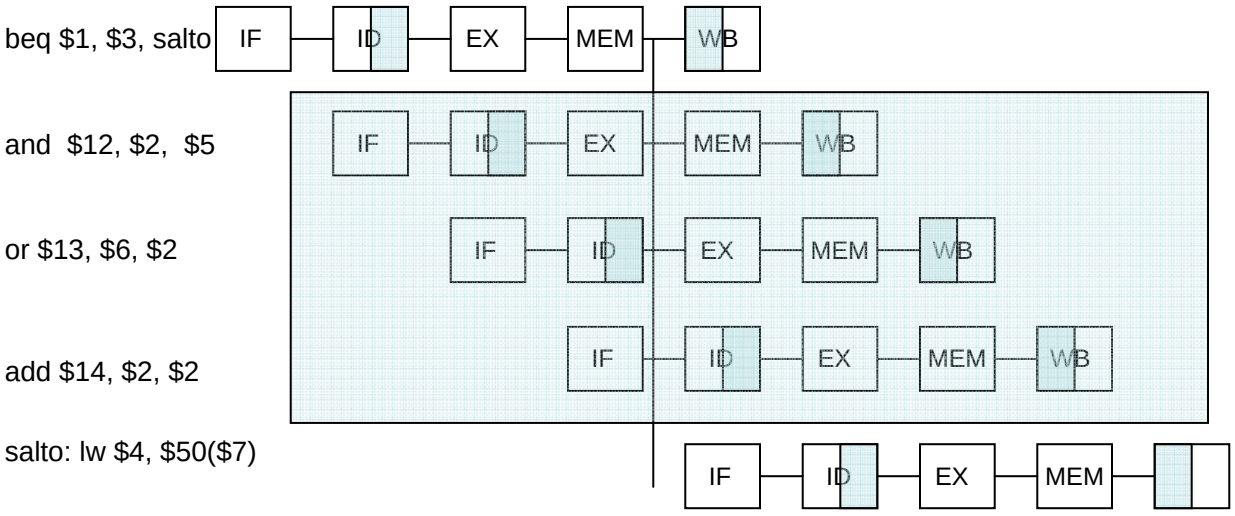


Figura 13. Riesgo de Control con decisión del salto en MEM

En el ejemplo de la figura 13, si la condición del salto se cumple, se deben desechar del *pipeline* las tres instrucciones que entraron en el *pipeline* pues se había asumido que dicha condición no se iba a cumplir.

Una forma para mejorar el rendimiento de los saltos es reduciendo el costo de no haber tomado el salto. La decisión del salto se toma en el segmento MEM del *pipeline*. En este segmento se tiene un sumador que le adiciona al PC el desplazamiento para llegar a la instrucción destino. Si esta decisión se puede mover a una etapa o segmento anterior entonces sólo una instrucción tendrá que ser sacada del *pipeline* cuando se tome una decisión incorrecta en el caso de los saltos. Muchas implementaciones MIPS mueven la ejecución del *branch* a la etapa ID. Para ello mueven el sumador que usa el *branch* para la etapa ID (Figura 14).

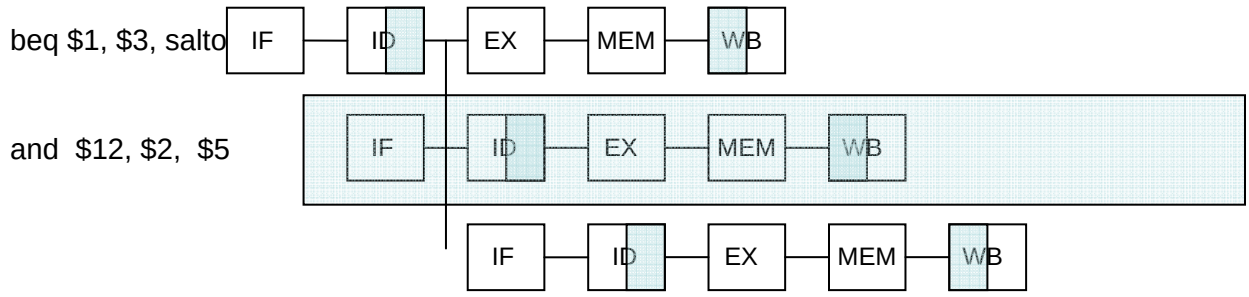


Figura 14. Riesgo de Control con decisión del salto en la etapa ID

Ejercicio.

Muestre qué sucede en el *pipeline* cuando la evaluación de la condición en el salto indica que debe tomarse el mismo. Asuma que el *pipeline* está optimizado para tomar la decisión en la etapa ID y que además simpre asume que las condiciones de los saltos no se cumplen.

```
      sub $10, $4, $8
      beq $1, $3,  X
      and $12, $2, $5
      or $13, $2, $6
      add $14, $4, $2
      slt $15, $6, $7

X:    lw $4, 60($7)
```

En la figura 15 podemos observar el comportamiento del *pipeline* para el ejercicio. En el ciclo de reloj 4, la etapa IF capta la instrucción localizada en la dirección de memoria X y se introducen burbujas en el *pipeline* para desechar la instrucción que fue captada antes de conocer el resultado de la condición del salto.

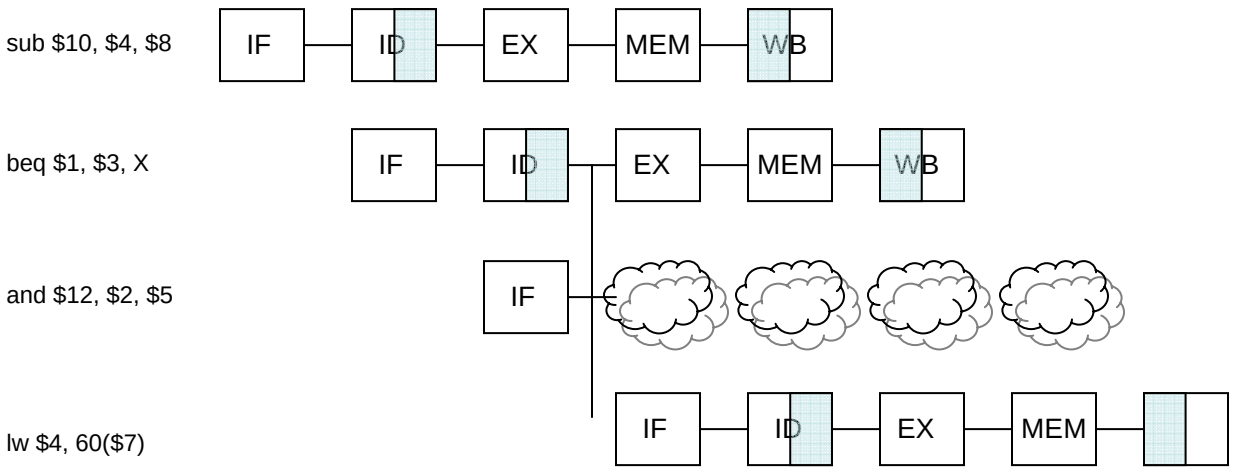


Figura 15. Ejemplo

Otra solución para atacar los riesgos de control es conocido como *delayed-branch.* En este caso los compiladores y ensambladores tratan de colocar una instrucción que siempre será ejecutada después del *branch*. Lo que buscan es que la instrucción que se ejecute después del *branch* sea válida y útil y no tenga que ser desechada después de ser captada.

Otra forma de riesgos de control incluyen a las excepciones. Supongamos una instrucción aritmética que produce un desbordamiento u *overflow*. Necesitamos transferir el control a la rutina de servicio de la excepción inmediatamente después de la instrucción porque no queremos que el resultado inválido contamine otros registros o localidades de memoria. En este caso, también se debe limpiar el pipeline y empezar a buscar instrucciones a partir de la nueva dirección (rutina de servicio de interrupciones).