UNIDAD 5: Mejora del rendimiento con la segmentación.

5.1 Un resumen de segmentación

La *segmentación (pipelining)* es una técnica moderna para implementar procesadores, mediante la cual se traslapa la ejecución de diferentes instrucciones en el mismo ciclo de reloj, cada instrucción en una etapa o segmento diferente.

Para introducir los conceptos de segmentación, se iniciará con una analogía con una lavandería. En una lavandería, para una carga de ropa (*x* cantidad de kilos), las tareas a realizar pueden dividirse en las siguientes etapas:

1. Colocar la carga de ropa sucia en la lavadora.
2. Sacar la ropa de la lavadora para pasarla a la secadora.
3. Doblar la ropa seca.
4. Acomodar la ropa limpia y seca en su lugar correspondiente.

Si para una carga de ropa cada tarea requiere de media hora para su ejecución, al trabajar sobre cuatro cargas de ropa: A, B, C y D, se requerirán de 8 horas para concluir con el lavado de las 4 cargas. Como se muestra en la figura siguiente:

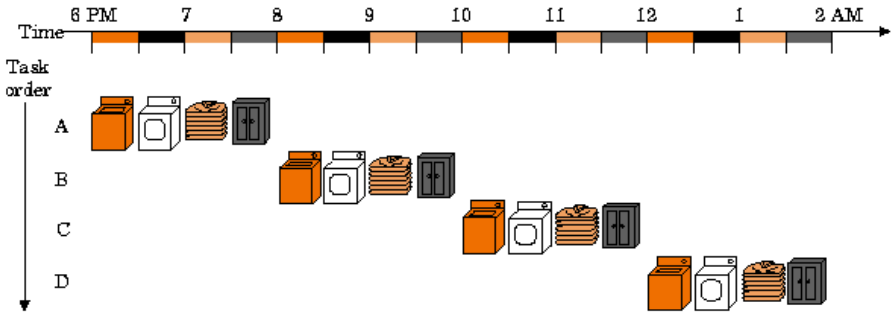

Fig. 5. 1 Lavado de 4 cargas sin segmentación.

Ahora, si se cuenta con los recursos suficientes pueden traslaparse algunas tareas, de manera que después de que la carga A se extraiga de la lavadora y pase a la secadora, la carga B pueda introducirse en la lavadora. Al mismo tiempo se tendría trabajando a la lavadora y a la secadora con dos cargas diferentes.

En la media hora siguiente, puede doblarse la ropa que corresponde a la carga A, pasar la carga B a la secadora e introducir la carga C en la lavadora. De esta manera se traslaparían diferentes cargas de ropa durante el proceso de lavado y para el lavado de 4 cargas se requeriría solo de 3.5 horas, esto se muestra en la figura 2. Lo cual es la aplicación de *segmentación* en el proceso de lavado. Y cada uno de los pasos en los que se dividió el proceso se conoce como *una etapa de la segmentación* o *un segmento.*

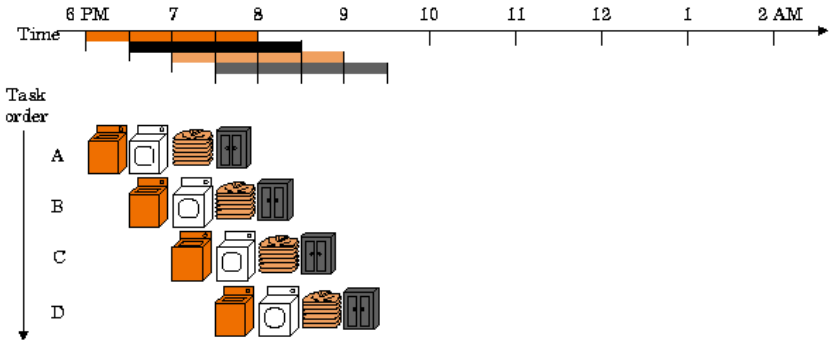

Fig. 5. 2 Lavado de 4 cargas con segmentación.

Puede notarse en la figura 5.2 que el tiempo invertido para una carga de ropa sigue siendo de dos horas, lo que se disminuyó fue el tiempo invertido para las cuatro cargas. En otras palabras, *la segmentación no disminuye el tiempo de ejecución pero si aumenta la productividad*, y de esta manera se aumenta el rendimiento.

Un proceso similar se aplicará a la ejecución de las instrucciones. La ejecución de cada instrucción se dividirá en diferentes etapas, conectando cada una a la siguiente, para formar una especie de cauce - las instrucciones entrarán por un extremo, se procesarán a través de las etapas y saldrán por el otro extremo.

La productividad de la segmentación está determinada por la frecuencia con que una instrucción salga del cauce. Como las etapas están conectadas entre sí, todas las etapas deben estar listas para proceder al mismo tiempo. El tiempo requerido para que una instrucción avance un paso es de *un ciclo de reloj*, la duración del ciclo está determinada por el tiempo que necesita la etapa más lenta (porque todas las etapas progresan a la vez).

Las etapas en las que se puede dividir la ejecución de las instrucciones MIPS son:

1. Atrapar las instrucciones de la memoria.
2. Leer los registro mientras se decodifica la instrucción.
3. Ejecutar la instrucción o calcular una dirección.
4. Realiza un acceso a la memoria de datos.
5. Escribir el resultado en un registro.

A excepción de las instrucciones de carga, las demás instrucciones no requerirían de todas las etapas de segmentación. Por ejemplo, para 3 instrucciones de carga en una implementación no segmentada (similar a una implementación multiciclos), la ejecución se comportará como:

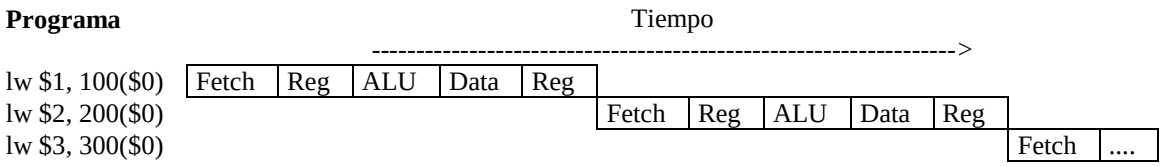

Fig. 5. 3 Tres instrucciones de carga sin segmentación.

Hasta que concluye la primera instrucción de carga se continúa con la segunda; y cuando concluya la segunda se continuará con la tercera. Con la segmentación, se puede traslapar el acceso a registros de la primera carga con la captura de la segunda; la operación de la ALU de la primera carga con el acceso a registros de la segunda y con la captura de la tercera, y así sucesivamente. De manera que la ejecución de las instrucciones tendría el comportamiento mostrado en la figura 5.4, que evidentemente requiere menos tiempo.

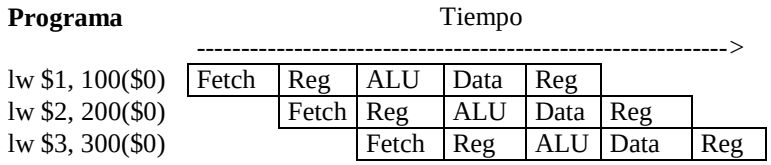

Fig. 5.4 Tres instrucciones de carga con segmentación.

Para implementaciones segmentadas, el objetivo del diseñador es equilibrar la duración de las diferentes etapas. Si las etapas están perfectamente equilibradas, entonces el tiempo por instrucción de la máquina segmentada - suponiendo condiciones ideales es igual a:

$$\frac{\text{Tiempo por instrucción en la máquina no segmentada}}{\text{Número de etapas de la segmentación}}$$

Bajo estas condiciones, la mejora de velocidad debida a la segmentación es igual al número de etapas. Sin embargo, habitualmente las etapas no están perfectamente equilibradas y la segmentación involucra algún gasto en recursos de hardware.

La segmentación es una técnica de implementación, que explota el paralelismo entre las instrucciones de un flujo secuencial. En este capítulo se desarrolla una implementación segmentada del subconjunto estudiado en el capítulo anterior. Cabe aclarar que no todos los repertorios son adecuados para la segmentación, en MIPS es posible por que:

- Todas las instrucciones son del mismo tamaño.
- Se tiene pocos formatos de instrucción, con los campos de registros fuentes ubicados en el mismo lugar.
- Los operandos de memoria solo aparecen en cargas o almacenamientos.
- Los operandos están alineados en memoria.

Riesgos en la segmentación

Son situaciones en las que la ejecución de la siguiente instrucción no puede avanzar en el siguiente ciclo de reloj, por diferentes circunstancias, hay 3 tipos de riesgos:

- **Riesgos de Estructura:** Significa que el hardware no puede soportar la combinación de instrucciones que se quiere ejecutar en el mismo ciclo.

  En la lavandería, por ejemplo, si la lavadora y la secadora forman parte del mismo equipo, no será posible trabajar estas dos etapas con dos cargas diferentes.

En un procesador, si tiene una sola memoria, para datos y código, no será posible escribir o leer un dato, mientras se atrapa una instrucción. Para evitar estos riesgos, se debe definir correctamente al camino de datos.

- **Riesgos por dependencias de datos:** Una instrucción depende del resultado de una instrucción previa que aún está en la segmentación.

  En la sección 5.4 se revisa a detalle este tipo de estos riesgos, en la sección 5.5 se muestra una técnica para resolverlos, conocida como *anticipación*. Y en la sección 5.6 se describe otra técnica, conocida como *detenciones*, la cual será necesaria en aquellos casos donde la *anticipación* no sea suficiente para resolverlos.

- **Riesgos de Control:** Surgen de la necesidad de hacer una decisión basada en los resultados de una instrucción, mientras otras se están ejecutando.

  Por ejemplo, en un brinco condicional, mientras se determina si el brinco se hará o no, otras instrucciones ingresarán al procesador. Si se determina que el brinco no se llevará a cabo, esas instrucciones prosiguen su ejecución. Pero si el brinco se realiza, habrá que eliminar las instrucciones que ya ingresaron al procesador.

En las siguientes dos secciones se plantea el diseño del camino de datos y control para una implementación segmentada, en estas dos secciones aún sin considerar las situaciones de riesgo, éstas se considerar a partir de la sección 5.4.

5.2 Un camino de datos segmentado

Hasta el momento se han revisado dos implementaciones: una implementación de un sólo ciclo y una implementación multiciclos. En la implementación de un solo ciclo se duplicaban algunas unidades funcionales, puesto que se esperaba que la instrucción culmine en el siguiente flanco de reloj. En la implementación multiciclos fue posible reutilizar el hardware en diferentes ciclos de reloj.

La implementación de un sólo ciclo es la más adecuada como base para aplicar la segmentación, ya que se espera que todo el hardware trabaje al mismo tiempo, pero con diferentes instrucciones. En la figura 5.5 se muestra una aproximación de como podría hacerse la segmentación, las cinco etapas requeridas por cada instrucción son:

1. IF: (*Instruction fetch)* Captura de la instrucción.
2. ID: (*Instruction decode*) Decodificación de la instrucción y lectura de registros.
3. EX: (*Execution*) Ejecución o cálculo de una dirección.
4. MEM: (*Memory Access*) Acceso a memoria.
5. WB: (*Write Back*) Retro escritura.

El flujo de las instrucciones es de izquierda a derecha a través de las cinco etapas hasta completar su ejecución. En el ejemplo de la lavandería, la ropa recorría una línea de cuatro etapas sin regresar hacia etapas anteriores.

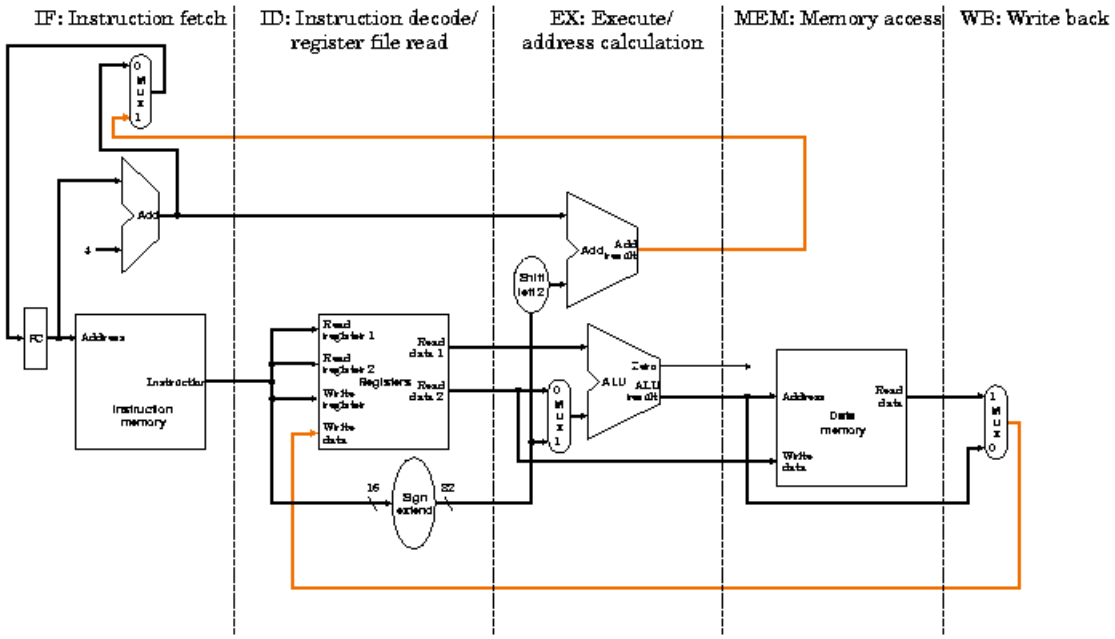

Fig. 5.5 El camino de datos de un solo ciclo es el más adecuado para segmentarse.

Sin embargo hay dos excepciones a este flujo de izquierda a derecha:

- En la etapa WB se realiza la escritura del resultado en el archivo de registros, el cual está ubicado en la segunda etapa del camino de los datos.
- La selección del siguiente valor del PC, seleccionado entre el PC incrementado en cuatro y la dirección destino de un brinco toma lugar en la etapa MEM.

El flujo de datos de derecha a izquierda no afecta a la instrucción actual, posiblemente tenga influencia sobre las instrucciones siguientes, que ya están en alguna etapa de la segmentación. La primera flecha de derecha a izquierda puede considerarse como un riesgo por dependencia de datos y la segunda como un riesgo de control.

Una forma de mostrar lo que ocurre en la ejecución segmentada es pretender que cada instrucción tiene su propio camino de datos, y colocar estos caminos de datos sobre el eje del tiempo para mostrar la relación entre ellos.

En la figura 5.6 se muestra la ejecución de las instrucciones que se ilustró en la figura 5.4 desplegando sus caminos de datos sobre una línea de tiempo común (los caminos de datos son versiones estilizadas de la figura 5.5).

Se observa que al no almacenar en algún lugar la información generada entre unidades funcionales, para tres instrucciones se necesitarán tres caminos de datos. Cuando se realizó la implementación multiciclos, se agregaron registros temporales para mantener algunos datos durante la ejecución de una instrucción. De forma similar, será necesario agregar registros para mantener los datos entre las diferentes etapas de segmentación. Al agregarle cuatro registros a la figura 5.5 se obtiene la figura 5.7, donde los nombres de los registros se toman de las dos etapas que conectan.

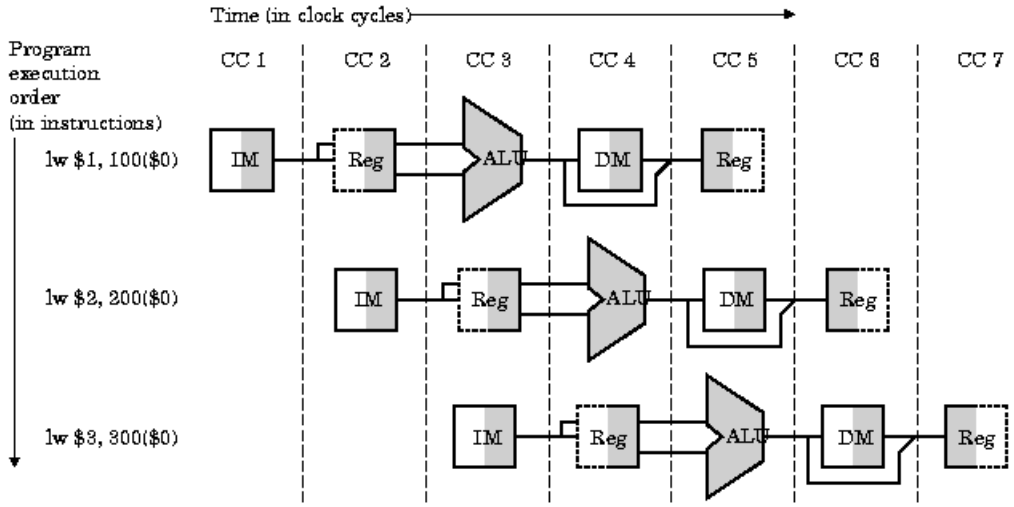

Fig. 5.6 Tres instrucciones de carga, cada una con su camino de datos.

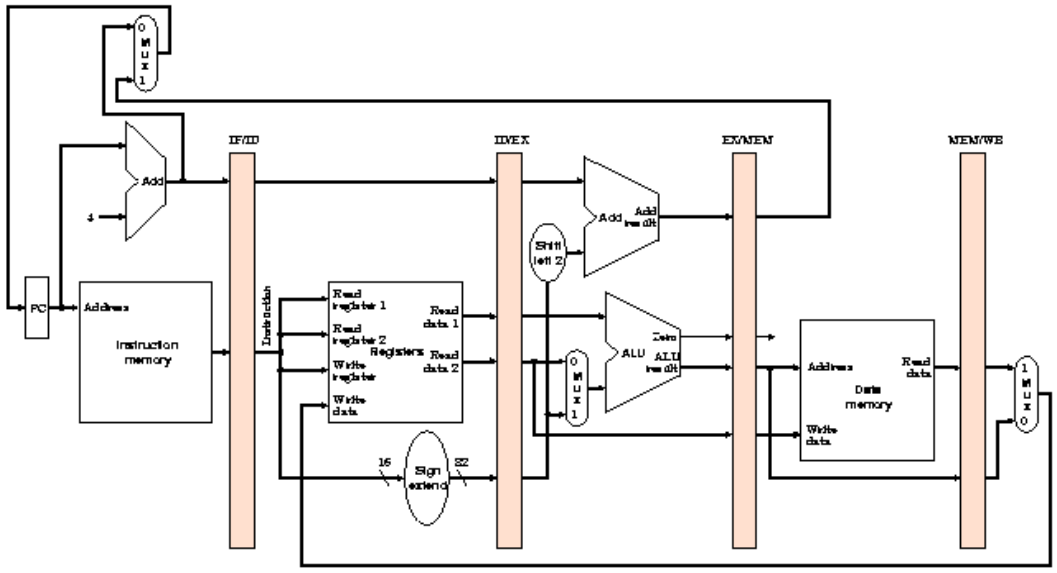

Fig. 5.7 Una versión segmentada del camino de los datos.

Se revisará el camino de datos de la figura 5.7 con una instrucción de carga (*lw*), por que es la única que utilizará las cinco etapas de segmentación. Se analizará etapa por etapa:

1. *Captura de la Instrucción*. En esta etapa la instrucción LW utilizará las unidades funcionales resaltadas en la figura 5.8, de la memoria de código sólo se resalta la mitad derecha puesto que su acceso es sólo para lectura. Al mismo tiempo, en esta etapa se calculará el valor de PC + 4 y se escribirá en el PC para introducir a la siguiente instrucción a la segmentación (aún no se consideran los riesgos). En el registro IF/ID se escribirá la instrucción, y el valor de PC + 4. Cabe aclarar que en esta etapa el hardware desconoce el tipo de instrucción.

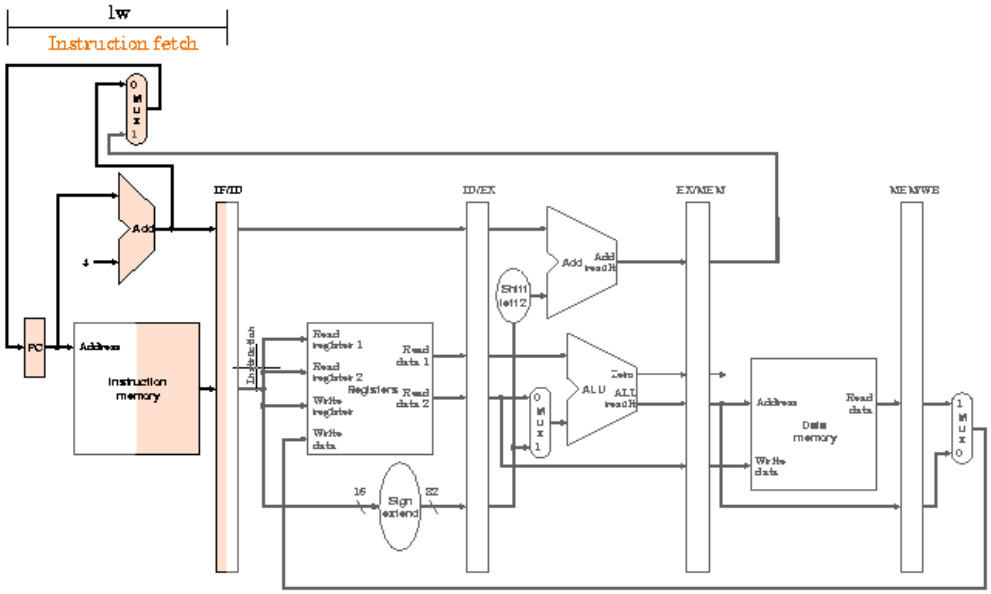

Fig. 5.7 Una instrucción de carga en la etapa de captura.

2. *Decodificación de la instrucción y lectura del archivo de registro*. Para esta etapa los datos que se evalúan se toman del registro IF/ID y los resultados al final de la misma se escribirán en el registro ID/EX. Los aspectos de interés particular para la instrucción LW son: la lectura del registro 1 y la extensión del signo de la constante para que utilice 32 bits. También se lee el registro 2, pero este no es importante para la instrucción bajo consideración. En la figura 5.8 se resaltan las unidades funcionales afectadas por la instrucción LW en la etapa 2.

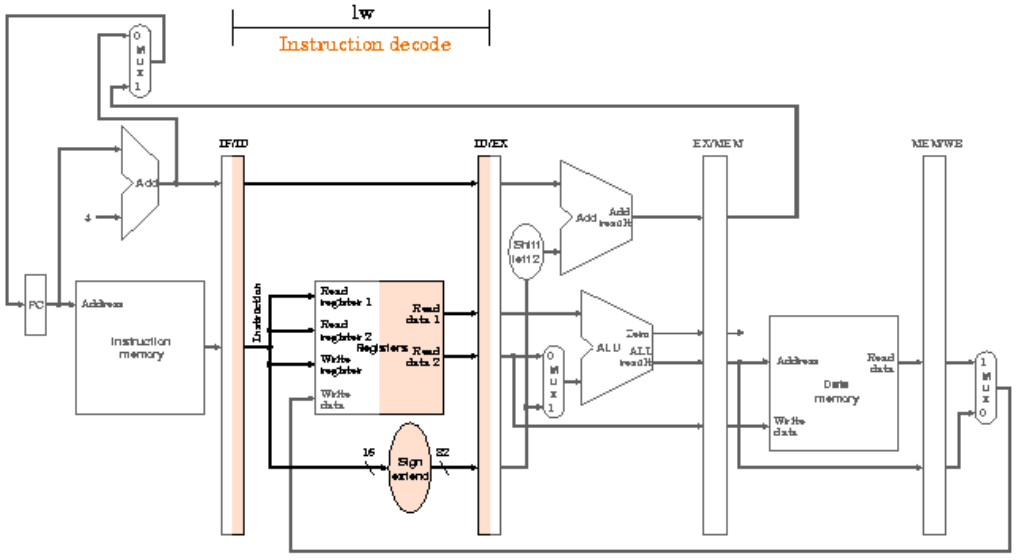

Fig. 5.8 La instrucción de carga en la etapa de decodificación y lectura de registros.

3. *Ejecución o Cálculo de una dirección*. Los datos se tomarán del registro ID/EX y los resultados se escribirán en el registro EX/MEM. En la figura 5.9 se muestra como lo importante para la instrucción LW es el cálculo de la dirección del dato que se leerá en

memoria. En esta etapa ya se tiene identificado el tipo de instrucción por lo que el control deberá colocar como segundo operando de la ALU a la constante extendida en signo. El primer operando de la ALU corresponde con el registro base.

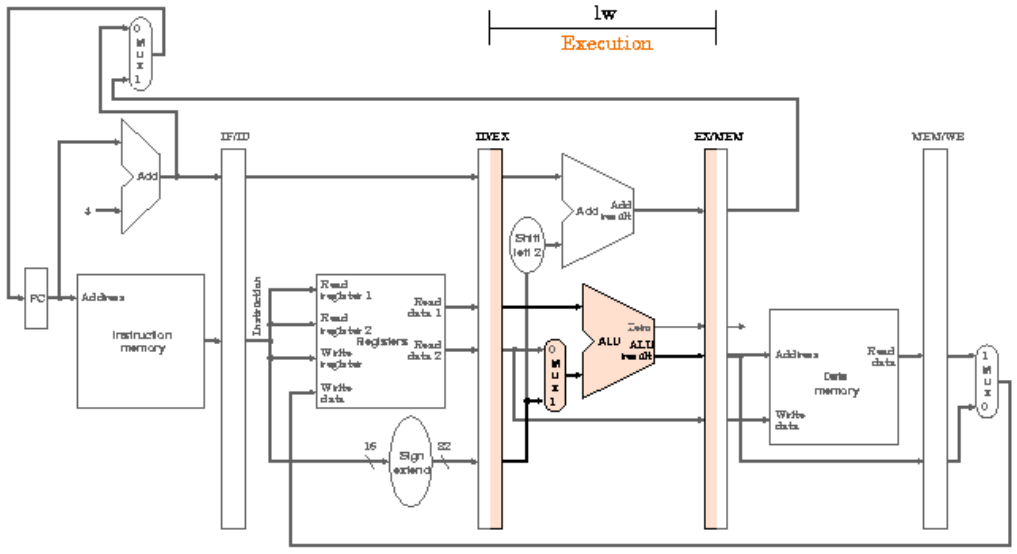

Fig. 5.9 La instrucción de carga en la etapa de ejecución, aquí se calcula la dirección a accesar.

4. *Acceso a memoria.* En esta etapa se leerá un dato de memoria, la dirección a leer se toma del registro EX/MEM y el dato leído se escribirá en el registro MEM/WB, en la figura 5.10 se resalta sólo la mitad derecha de la memoria porque sólo se está haciendo una lectura.

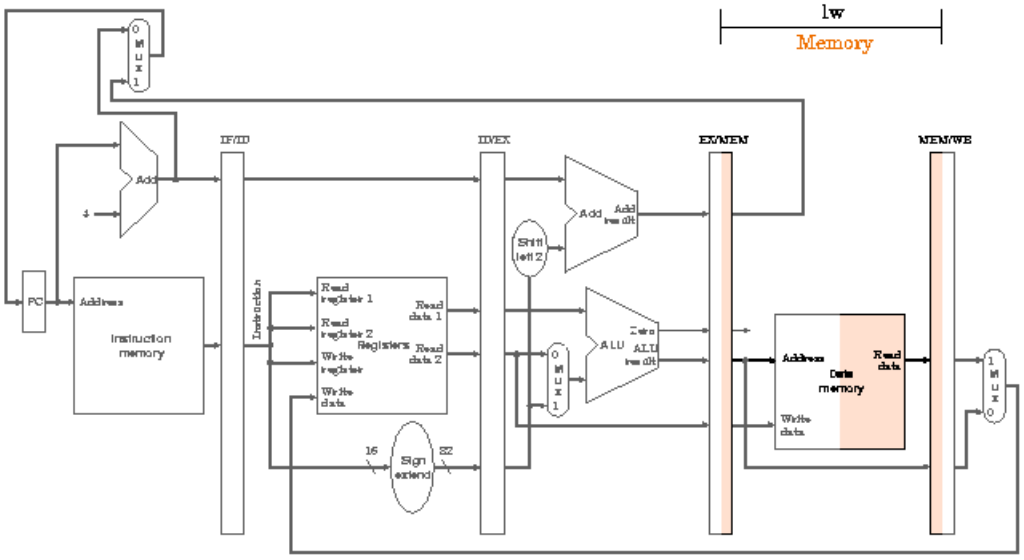

Fig. 5.10 La instrucción de carga en el acceso a memoria para la lectura del dato.

5. *Retro escritura*. El último paso de la instrucción LW consiste en la escritura del dato que está en el registro MEM/WB en el registro correspondiente, el nombre de *retro escritura* es porque aunque se está ejecutando el paso 5, el archivo de registros está

ubicado en la etapa 2. Sin embargo, esto no afecta en la ejecución, puesto que el archivo de registros puede leerse y escribirse en el mismo ciclo de reloj; y en el caso de esta implementación segmentada, la lectura y escritura se harán por diferentes instrucciones. En la figura 5.11 se muestra la ejecución de este paso con la instrucción LW.

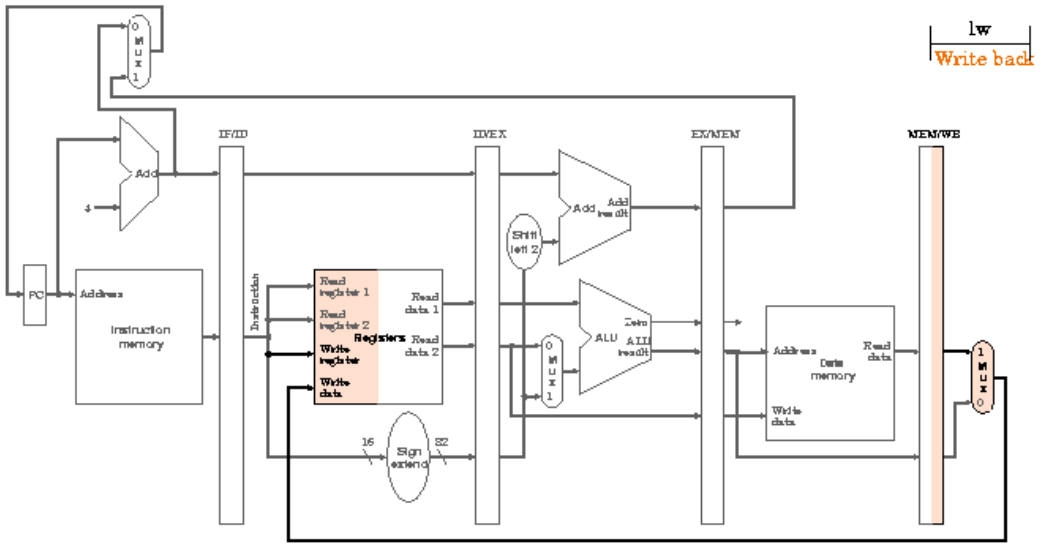

Fig. 5.11 Culminación de la instrucción de carga, se escribe el dato en un registro.

Sin embargo, si se revisa a detalle la figura 5.11 se encontrará una incongruencia, el dato a escribir es correcto porque de la etapa 5 se regresa a la etapa 2, pero el número de registro en el que se escribirá el dato puede ser incorrecto, porque se está tomando de la instrucción que en ese momento esté en la etapa 2. En la medida en que la instrucción LW avanzó en la segmentación, también debería haber avanzado el registro destino y no se hizo así, la solución se presenta en la figura 5.12.

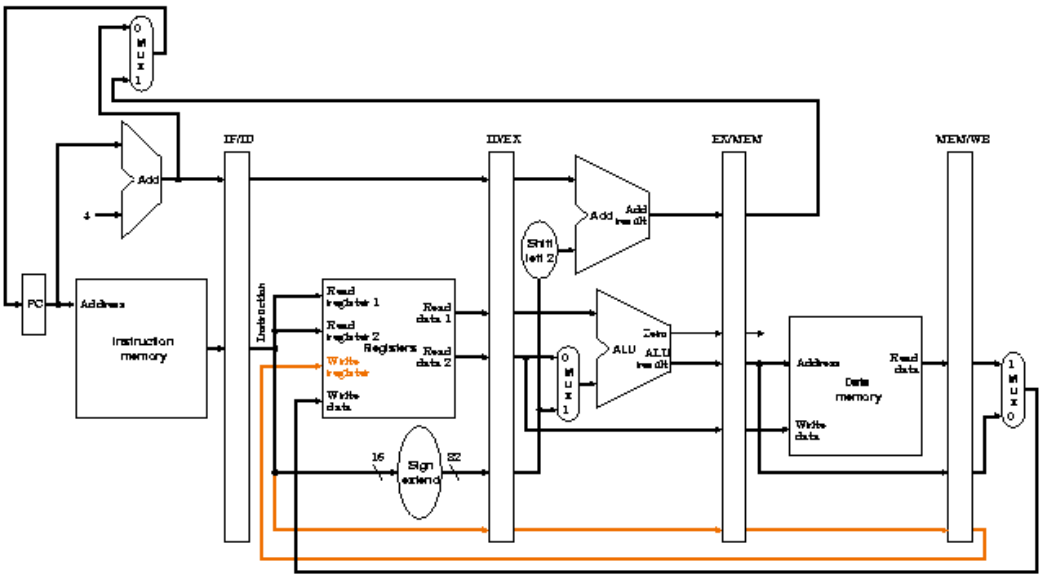

Fig. 5.12 El registro destino avanza conforma avanza la instrucción en la segmentación.

Como un resumen, en la figura 5.13 se muestra la porción del camino da datos que utiliza la instrucción LW. En la memoria de instrucciones y en la memoria de datos sólo se sombrea la mitad de la derecha, porque su acceso es sólo para lectura. El archivo de registros se sombrea completo, aunque debe recordarse que la lectura se hace en la etapa 2 y la escritura en la etapa 5.

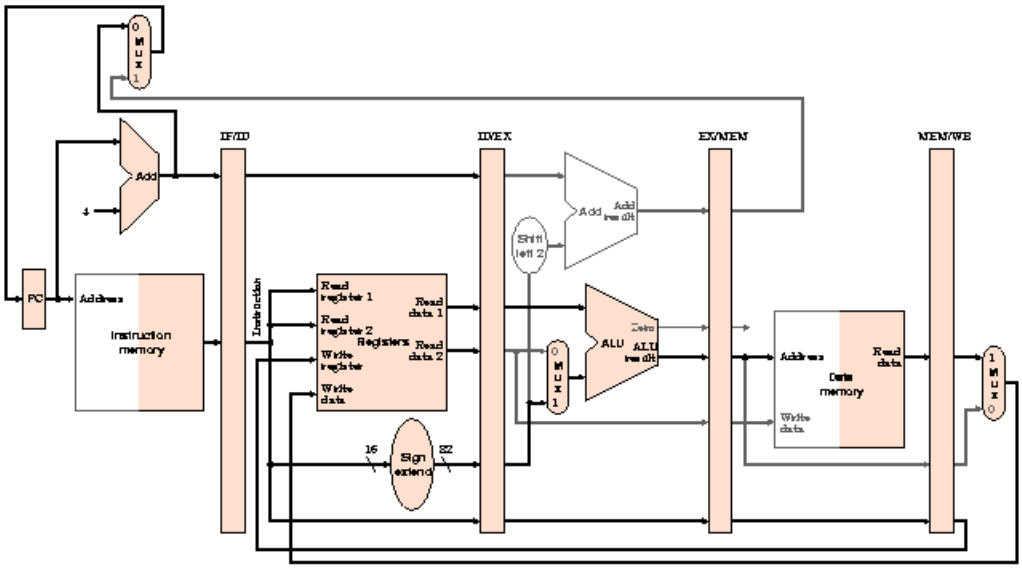

Fig. 5.13 Porción del camino de datos que usa una instrucción LW.

Representación gráfica de la segmentación.

La segmentación puede ser difícil de entender, dado que diferentes instrucciones se están ejecutando simultáneamente en un sólo camino de datos en cada ciclo de reloj.

Se tienen diferentes formas de representarla gráficamente, éstas se revisarán considerando la siguiente secuencia de instrucciones:

```
lw      $10, 20 ($1)
sub     $11, $2, $3
```

Una ***representación ciclo-a-ciclo*** es aquella que emplea un diagrama del camino de datos para cada ciclo de reloj, ilustrando todos los detalles sobre lo que está ocurriendo en el hardware.

Para la secuencia bajo revisión se requiere de 6 ciclos de reloj, desde que la instrucción LW entra a la etapa 1, hasta que la instrucción SUB sale de la etapa 5. Por lo tanto, bajo este esquema de representación son necesarias 6 figuras, éstas corresponden con las figuras de la 5.14 a la 5.19. Cabe aclarar que solo se están considerando dos instrucciones, pero podría ser que cuando la instrucción LW entró a la segmentación, ya existían otras cuatro instrucciones en las otras etapas. Y después de la instrucción SUB pudiera ser que nuevas instrucciones ingresen a la segmentación.

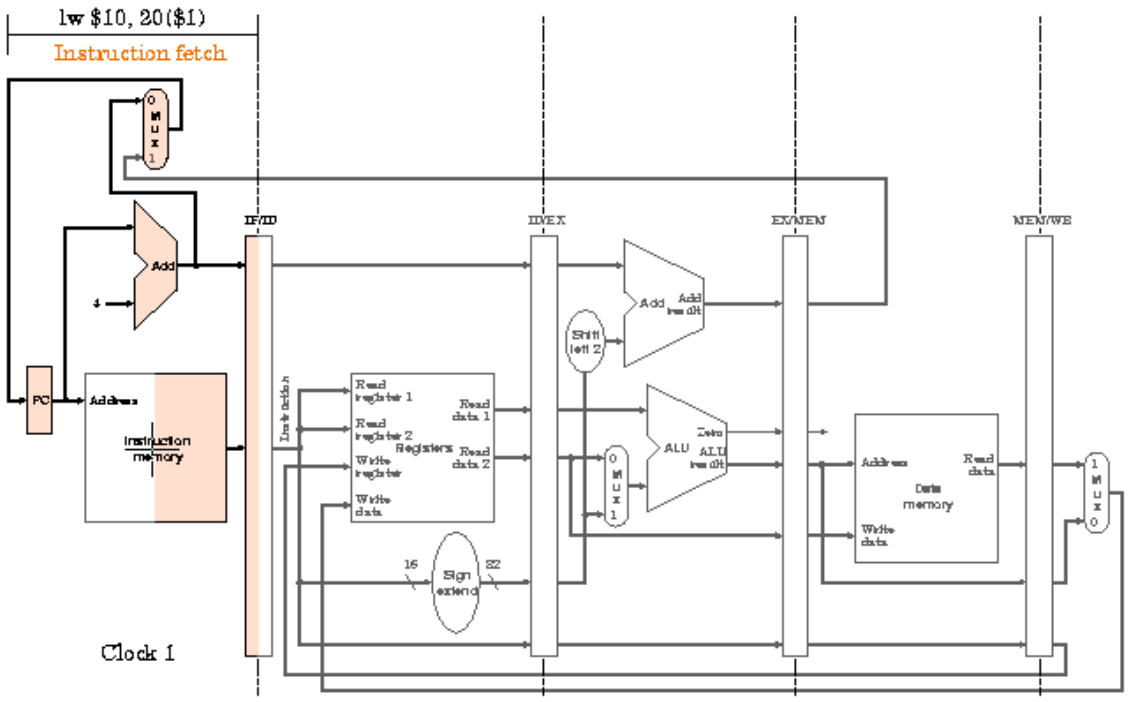

Fig. 5.14 Ciclo de reloj 1, la instrucción LW entra a la segmentación.

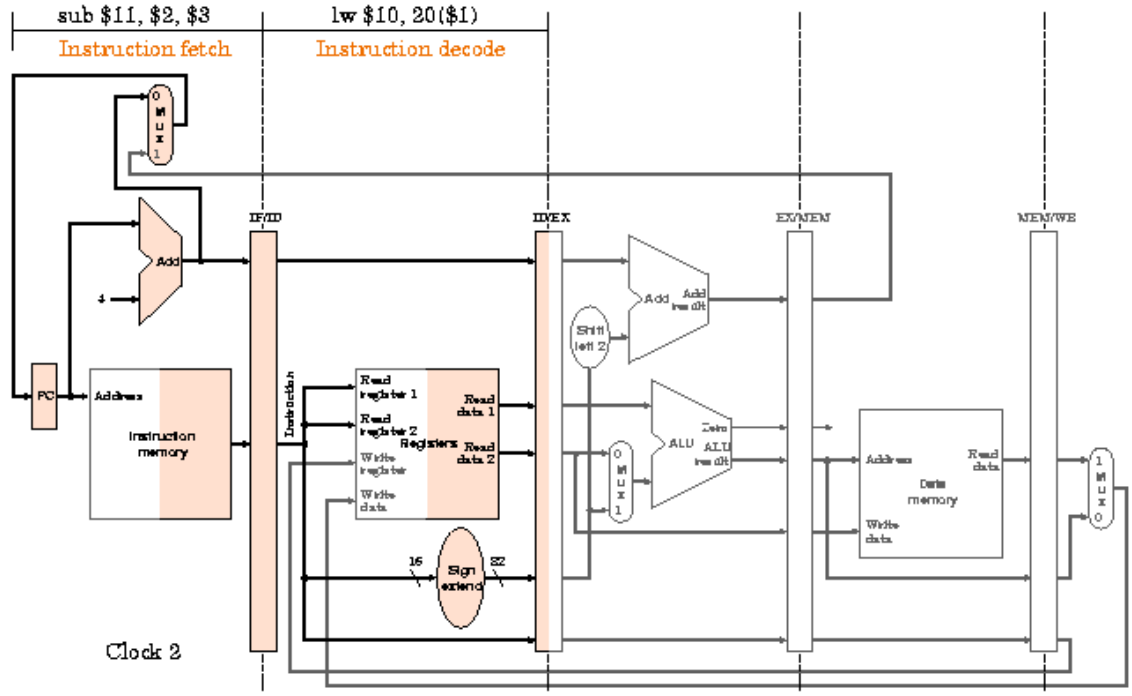

Fig. 5.15 Ciclo de reloj 2, la instrucción LW avanza a la etapa de decodificación y la instrucción SUB entra a la segmentación.

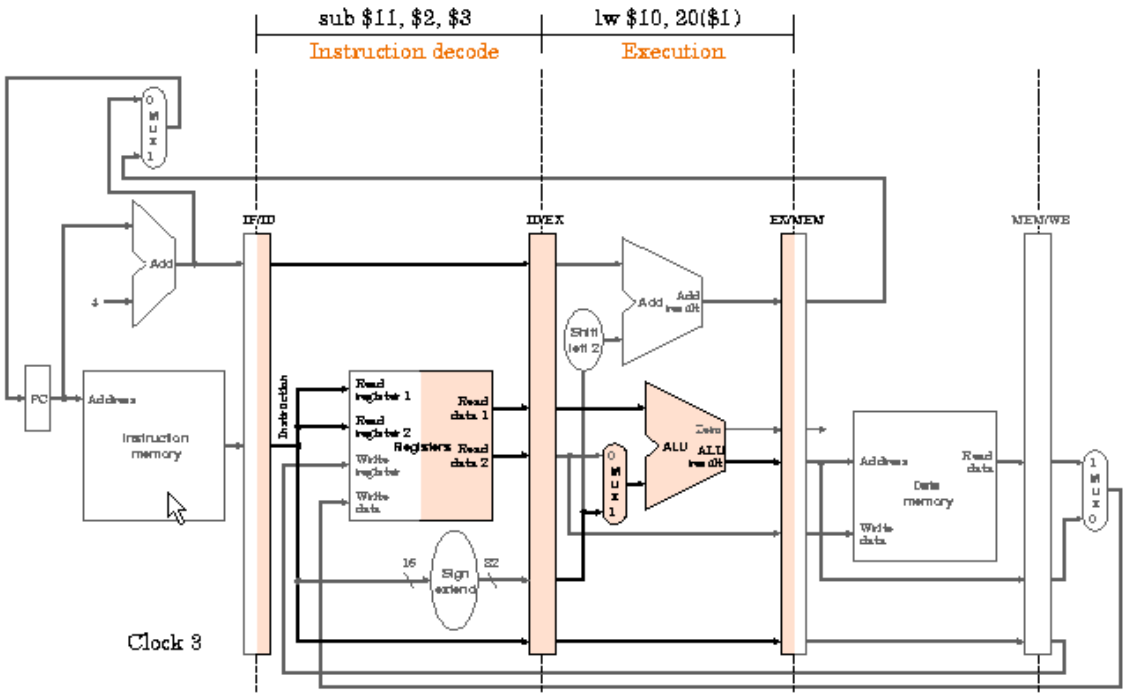

Fig. 5.16 Ciclo de reloj 3, la instrucción LW calcula la dirección de acceso a memoria y la instrucción BEQ avanza a la etapa de decodificación.

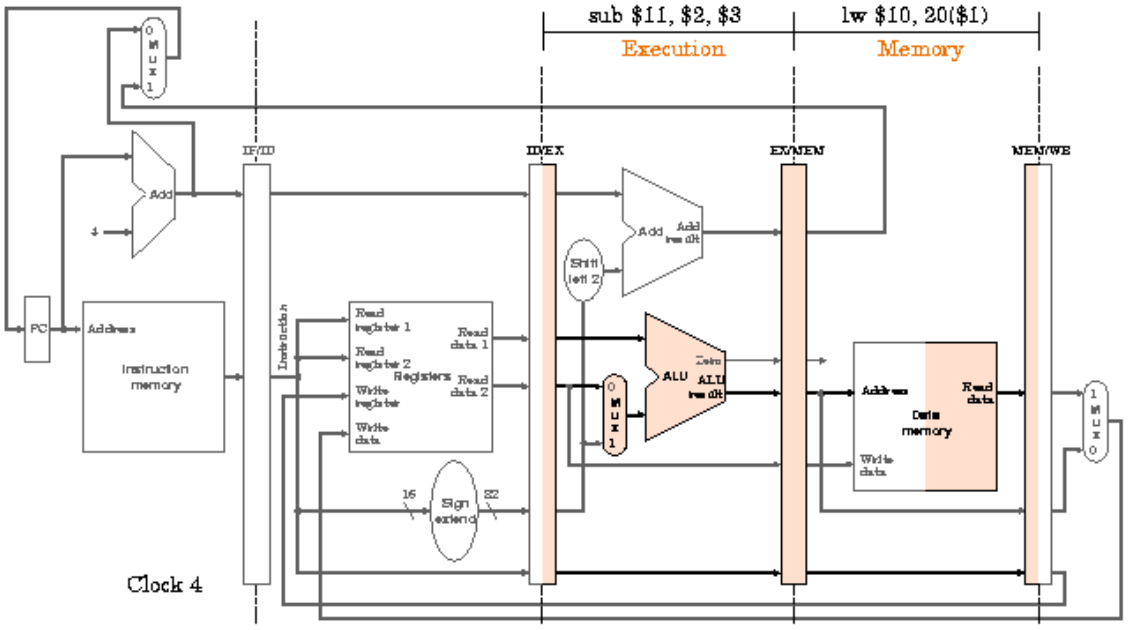

Fig. 5.17 Ciclo de reloj 4, la instrucción LW hace una lectura de memoria y la instrucción BEQ realiza la resta en la etapa de ejecución.

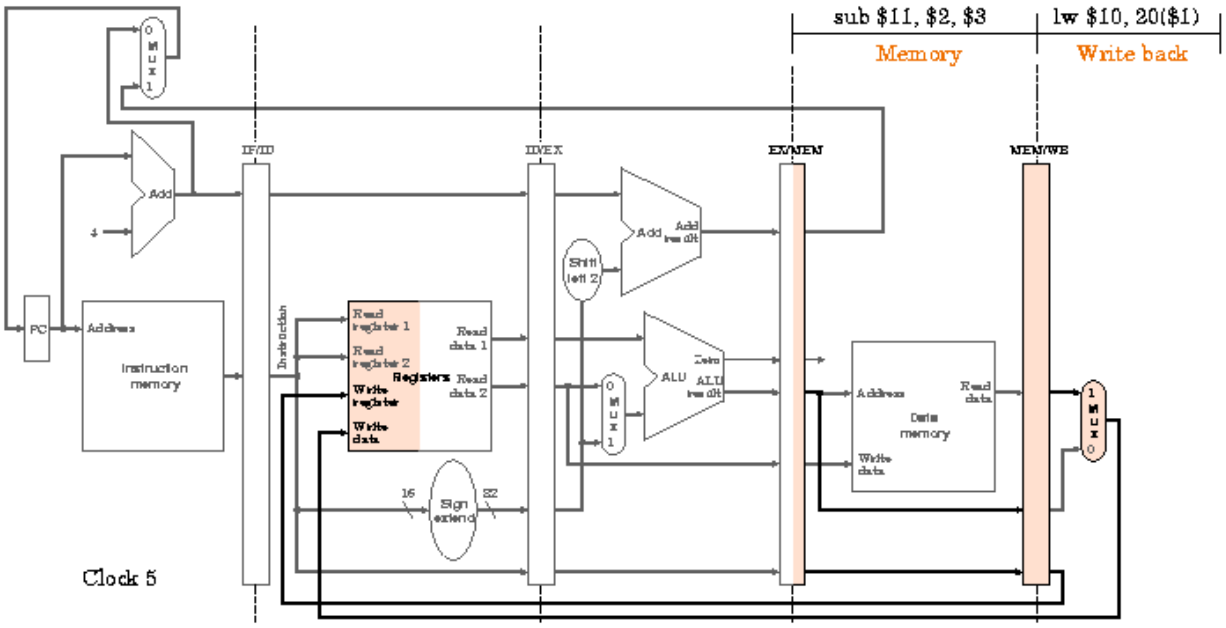

Fig. 5.18 Ciclo de reloj 5, la instrucción LW escribe el dato leído en el registro correspondiente y la instrucción BEQ llega a la etapa de memoria en la que pasará desapercibida.

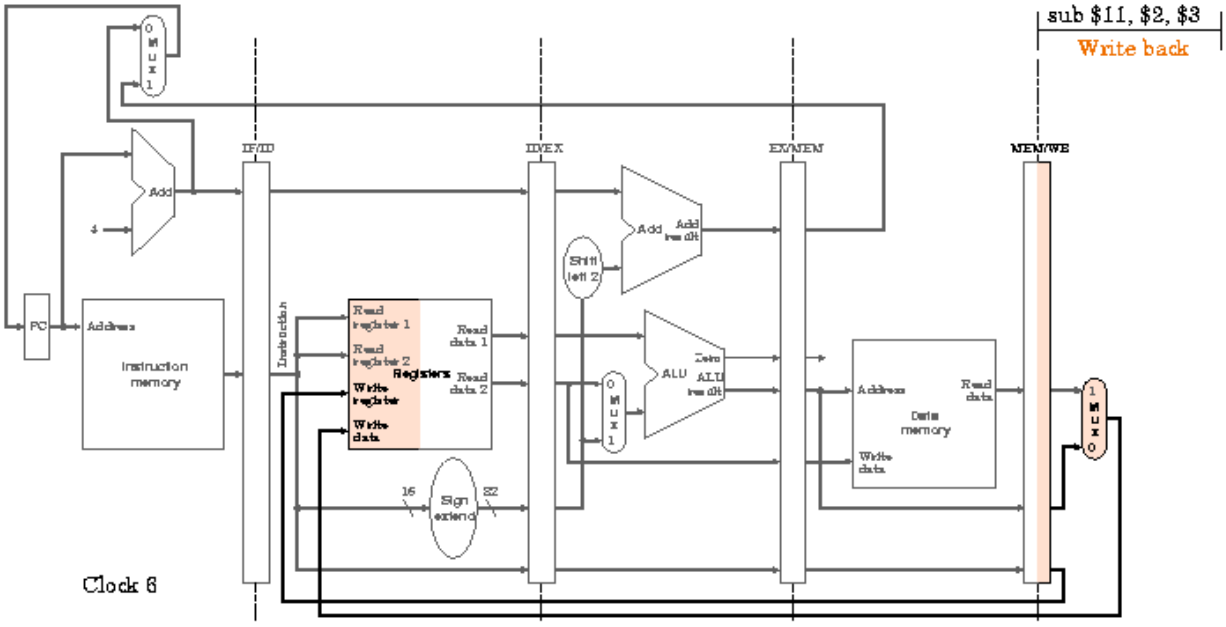

Fig. 5.19 Ciclo de reloj 6, la instrucción LW ya salió de la segmentación y la instrucción BEQ escribe el resultado de la resta en el registro correspondiente

La ***representación ciclo-a-ciclo*** es demasiado detallada, si se pretende revisar una secuencia significativa de instrucciones, no resultará conveniente.

Otra forma de representar la segmentación es mediante una ***representación abreviada,*** ésta es una representación simplificada en la que la información se presenta en un plano, en el eje horizontal se muestran los diferentes ciclos de reloj y en el eje vertical se ilustra la secuencia de instrucciones bajo ejecución. La ***representación abreviada*** es útil porque en cada ciclo se identifican las instrucciones que se están ejecutando y en que etapa se encuentra cada una de ellas, para la secuencia de 2 instrucciones bajo estudio se tiene la figura 5.20.

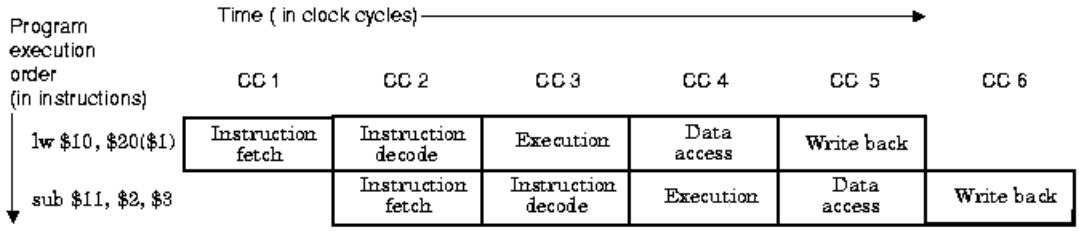

Fig. 5.20 Representación abreviada de una ejecución segmentada.

Sin embargo, la ***representación abreviada*** no muestra los recursos que la instrucción está usando, aparenta que cualquier instrucción utiliza todos los recursos en cada una de las etapas de la segmentación.

Un punto medio entre la ***representación ciclo-a-ciclo*** y la ***representación abreviada*** se obtiene con la ***representación de múltiples-ciclos***, en la cual, además de proporcionar la información presentada en la figura 5.20, muestra los recursos que utiliza una instrucción determinada en cada etapa. En la figura 5.21 se muestra la ***representación de múltiples-ciclos*** para las 2 instrucciones bajo estudio.

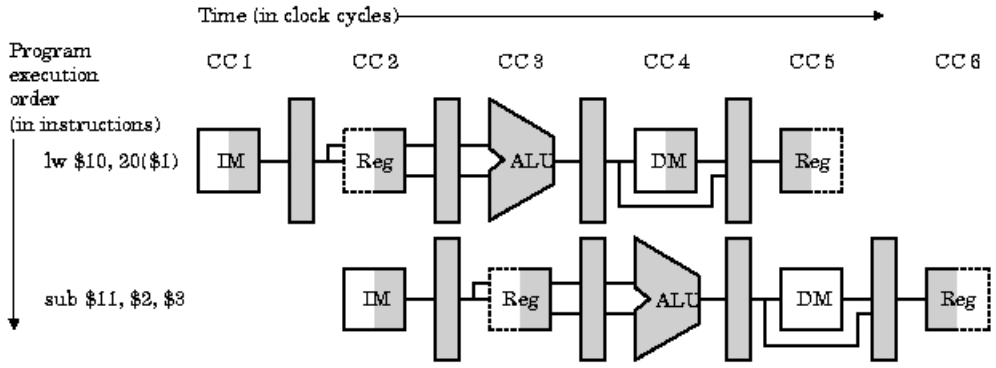

Fig. 5.21 Representación de múltiples-ciclos de una ejecución segmentada.

En la etapa 2 aparece el archivo de registros y se repite en la etapa 5, sin embargo, en la etapa 2 la parte izquierda está con línea punteada, por que en esa etapa no hay acceso a la parte de escritura. En la etapa 5 es lo contrario, solo hay acceso a la escritura, no así a la lectura de registros. Entonces, la línea punteada de alguna manera indica que el archivo de registro se ha dividido para usarse en dos etapas diferentes.

Es diferente a la etapa 4 de la instrucción LW, en esa etapa la memoria no está con línea punteada por que la memoria físicamente está disponible para usarse en lecturas o escrituras. Pero en el caso de la instrucción LW se sombrea la mitad derecha, por que solo se harán lecturas. La instrucción SUB no tiene acceso a la memoria, debido a ello, aunque ésta se representa en el diagrama, no debe ser sombreada.

5.3 Un control segmentado

Primero se deben colocar las señales de control requeridas por cada unidad funcional o multiplexor del camino de datos. Puesto que se partió de una implementación de un sólo ciclo, de ese diseño se toman las señales de control para agregarlas a la implementación segmentada. El resultado se presenta en la figura 5.22.

Puede notarse que los saltos condicionales se determinan en la etapa de acceso a memoria, esto provoca algunas situaciones difíciles de tratar, por que si el salto se va a realizar, habrá que eliminar de alguna manera a las instrucciones que han ingresado a la segmentación y que están en las etapas anteriores. De momento se supondrá que los brincos no se realizan, en otra sección se analizará como resolver tales situaciones (*riesgos de control*).

Para especificar el control segmentado, es necesario especificar un conjunto de valores en cada una de las etapas de la segmentación. Puesto que cada línea de control está asociada con una componente activa en sólo una etapa de la segmentación, es posible dividir las señales de control en cinco grupos, de acuerdo a las etapas de segmentación:

1. *Captura de la Instrucción*. Las señales de control para la lectura de memoria de código y escritura del PC están siempre acertadas, por que cambiarán su valor en cada ciclo de reloj, por lo que no hay líneas de control especiales.

2. *Decodificación de la instrucción y lectura del archivo de registro*. Sucede una situación similar a la etapa anterior, no hay líneas de control para un ajuste opcional.

3. *Ejecución o Cálculo de una dirección*. En esta etapa, dependiendo del tipo de instrucción debe seleccionarse:
   - El segundo operando de la ALU (ALUSrc).
   - La operación que realizará la ALU (ALUOp).
   - El registro destino (RegDst).

4. *Acceso a memoria.* Aquí se determina si:
   - Se trata de un brinco (branch).
   - Se escribirá en memoria (MemWrite).
   - Se leerá de memoria (MemRead).

5. *Retro escritura*. Son dos señales que se deben definir:
   - La que determina si se escribirá el resultado de la ALU o el dato de memoria (MemtoReg).
   - La que habilita la escritura en el archivo de registros (RegWrite).

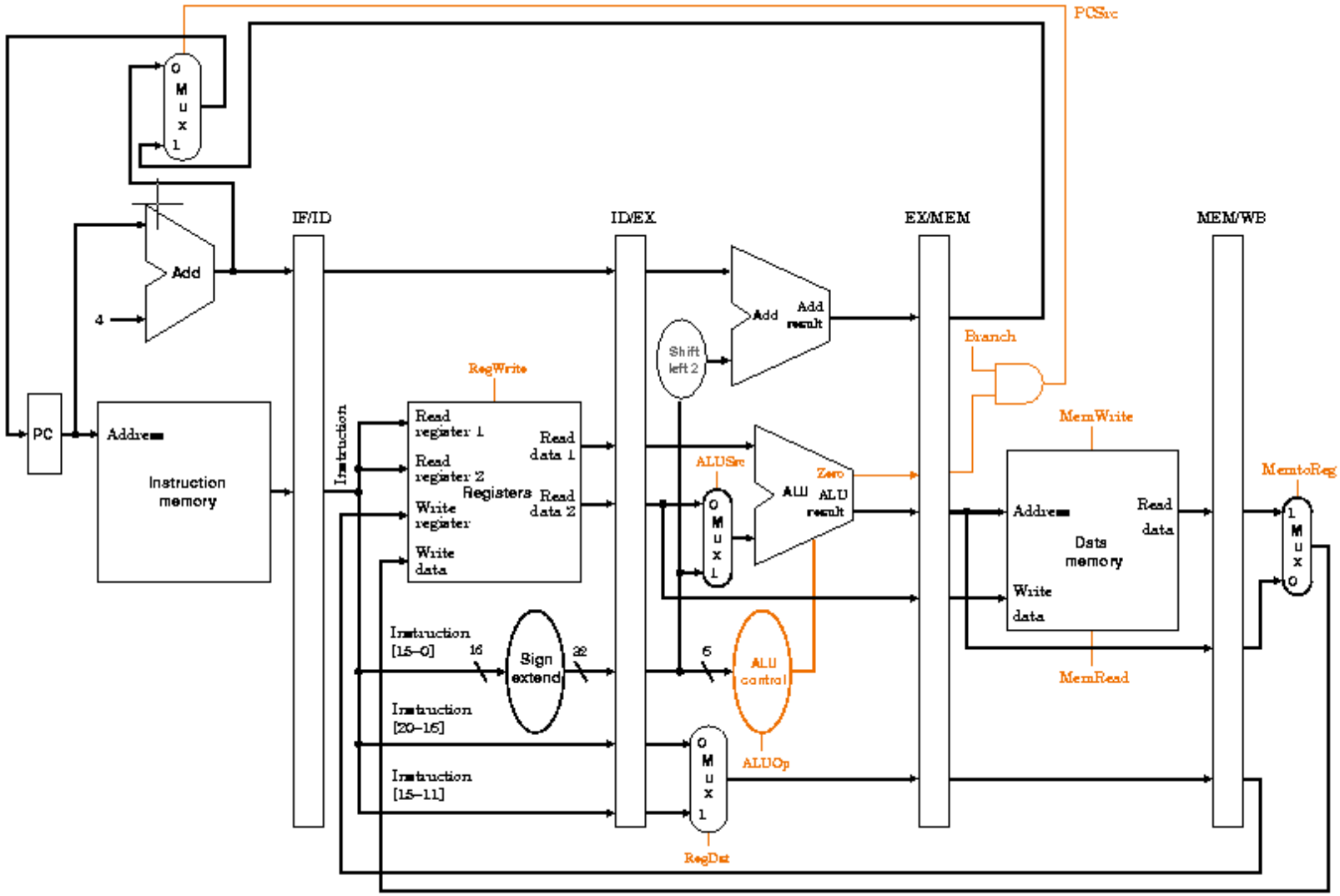

Fig. 5.22 El camino de datos segmentado con las señales de control especificadas.

Cabe aclarar que nuevamente se usará el control de la ALU desarrollado para la implementación de un solo ciclo, por lo que las señales ALUOp determinarán si la operación a realizar será una suma, una resta o si dependerá del campo de función.

Valor de ALUOp	Operación deseada en la ALU
00	Suma
01	Resta
10	Depende del campo de función

Tabla 5.1 Comportamiento de la ALU con respecto a ALUOp.

El efecto de acertar o desacertar las señales de control se mostró en la tabla 4.4, por conveniencia se repite como tabla 5.2, la única diferencia entre ambas tablas es que en la 4.4 aún se consideraba a la señal PCSrc. Sin embargo, para que el control genere esta señal necesita conocer el valor de la bandera *zero*. Para omitir esa entrada extra al control, se utilizó una compuerta AND de dos entradas, de manera que ahora el control genera la señal Branch cuando detecta un brinco (primer entrada de la AND) y la realización del brinco depende del valor de la bandera *zero* (segunda entrada a la AND).

Nombre de la señal	Efecto cuando es desacertada	Efecto cuando es acertada
RegDst	El número del registro destino para la escritura viene del campo rt (20-16)	El número del registro destino para la escritura viene del campo rd (15-11)
RegWrite	Ninguno	Se escribirá un dato en el archivo de registros
ALUSrc	El segundo operando de la ALU es el segundo dato leído en el archivo de registros	El segundo operando de la ALU son los 16 bits de desplazamiento tomados de la instrucción y extendidos en signo
Branch	El PC es remplazado por PC + 4	El PC será remplazado por la suma de una dirección calculada para un brinco si se generó la bandera zero
MemRead	Ninguno	Se hace la lectura de la memoria de datos
MemWrite	Ninguno	Se hace la escritura en la memoria de datos
MemtoReg	El valor del dato que se escribirá en el archivo de registros viene de la ALU	El valor del dato que se escribirá en el archivo de registros viene de la memoria de datos

Tabla 5.2 El efecto de cada una de las señales de control.

La consideración importante en el diseño del control, es que las señales deben de tener el valor correcto en la etapa correcta. Apoyados en la descripción anterior y en las tablas 5.1 y 5.2, se construye la tabla 5.3, en la que se indican las señales involucradas en cada una de las etapas y su valor para cada tipo de instrucción.

	Etapa de Ejecución/ Cálculo de dirección				Etapa de acceso a memoria			Etapa de retro escritura	
Instrucción	Reg Dst	ALUOp1	ALUOp0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
Tipo R	1	1	0	0	0	0	0	1	0
LW	0	0	0	1	0	1	0	1	1
SW	X	0	0	1	0	0	1	0	x
BEQ	X	0	1	0	1	0	0	0	x

Tabla 5.3 Los valores de las líneas de control distribuidos en tres grupos que corresponden a las tres últimas etapas de la segmentación.

Los valores de las señales son los mismos que los que se requerían en una implementación de un solo ciclo, la diferencia en esta implementación es que ahora esos valores se requieren en diferentes etapas.

En la etapa 1 se lee la memoria de instrucciones para atrapar la instrucción a ejecutar en el registro IF/ID. En la etapa 2 se toma del registro IF/ID al opcode, que determina la instrucción bajo ejecución. Los 6 bits del opcode son las entradas al circuito de control, el cual es el mismo que el de una implementación de un sólo ciclo (combinacional), de manera que las señales de control toman sus valores en forma inmediata, sin embargo estos no se requieren en la etapa 2, por lo que deben escribirse en el registro ID/EX para que viajen junto con la instrucción en las diferentes etapas de la segmentación.

En la etapa 3, en el registro ID/EX no solo se encuentran los datos, también están todas las señales de control, en esta etapa se usarán algunas de ellas (de acuerdo a la tabla 5.3) y el resto se escribe en el registro EX/MEM para que avancen en la segmentación. En la etapa 4 ocurre algo similar, se toman las señales de control a utilizarse del registro EX/MEM y las restantes se escriben en el registro MEM/WB. En la etapa 5, del registro MEM/WB se toman las últimas señales de control del registro MEM/WB y se aplican sobre los datos correspondientes.

Por ello, los registros que separan a las cinco etapas de segmentación deben extenderse en tamaño para que incluyan a las señales de control. En la figura 5.23 se muestra el hardware correspondiente al control, su generación y su desplazamiento por las diferentes etapas.

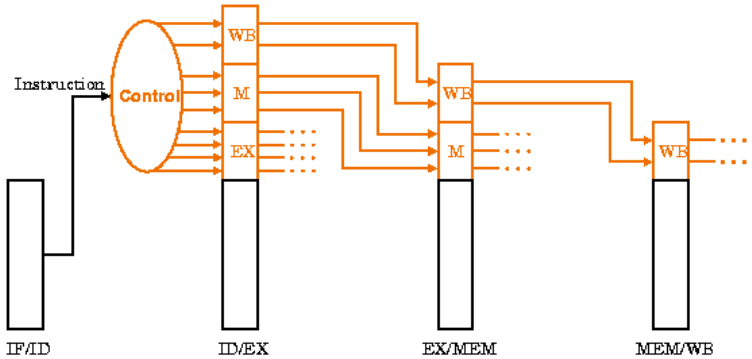

Fig. 5.22 El control acondicionado para la segmentación.

En la figura 5.23 se muestra el camino de datos y el control para una implementación segmentada, las señales de control se muestran como un bus que se divide en la etapa en la que se aplicará.

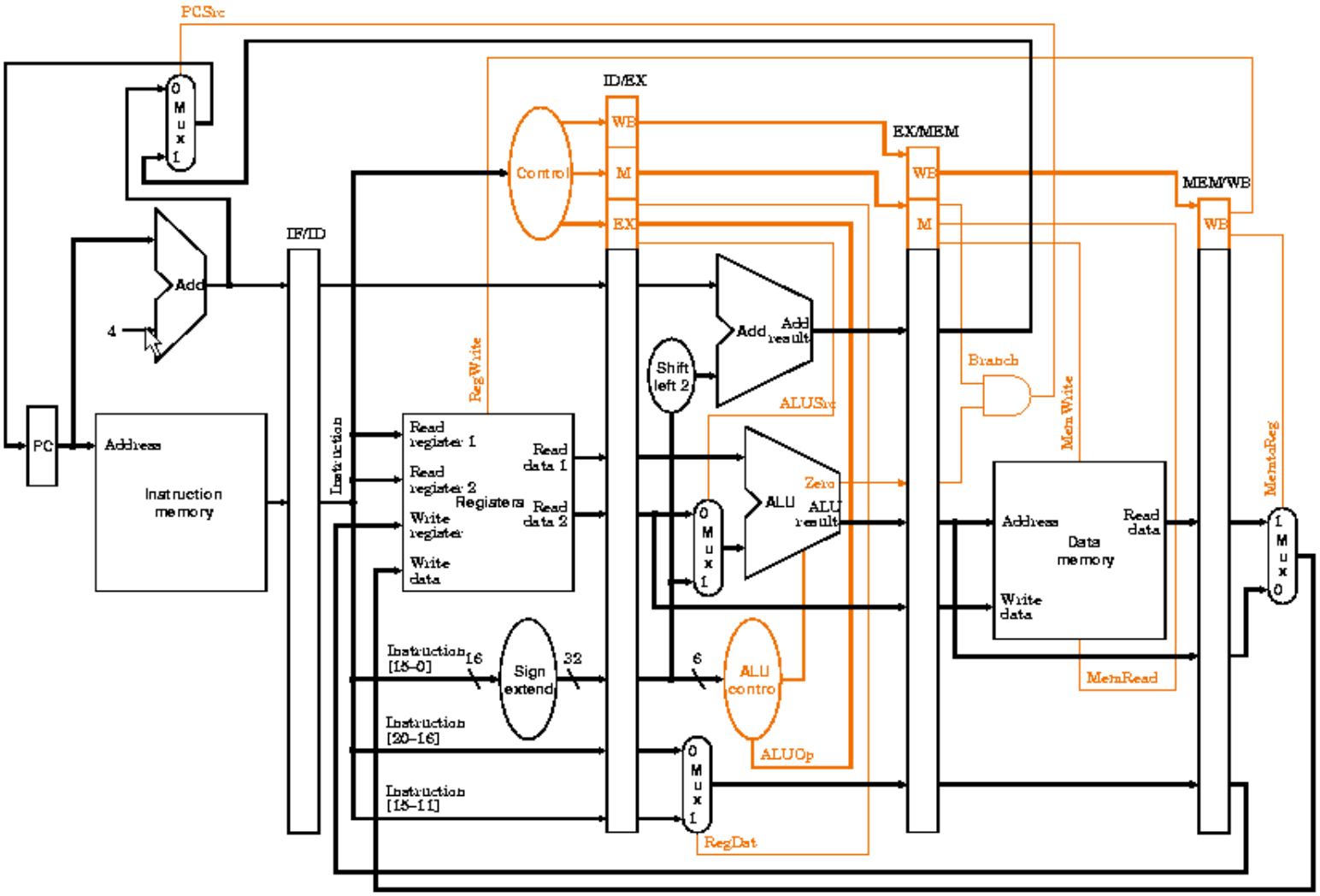

Fig. 5.23 Una implementación segmentada: Camino de datos y control

Ejemplo: Ejecución segmentada.

Evaluar la ejecución de las cinco instrucciones siguientes a través de la segmentación:

```
lw      $10, 20 ($1)
sub     $11, $2, $3
and     $12, $4, $5
or      $13, $6, $7
add     $14, $8, $9
```

Respuesta:

Se requiere de 5 ciclos de reloj para concluir con la instrucción lw, sin embargo, una vez que lw termine su ejecución, las siguientes cuatro instrucciones terminarán en los siguientes cuatro ciclos, una instrucción por ciclo. Por lo que en total se requiere de 9 ciclos para la ejecución de las cinco instrucciones.

En las siguientes nueve figuras, de la 5.24 a la 5.32 se muestra el comportamiento del camino de los datos y el valor de las señales de control en cada ciclo de reloj, ésta es la ***representación ciclo-a-ciclo***. Las etiquetas *before<i>*, se agregan para indicar que antes de la ejecución del código bajo consideración había otras instrucciones en la segmentación. Y las etiquetas *after<i>* indican el ingreso de otras instrucciones.

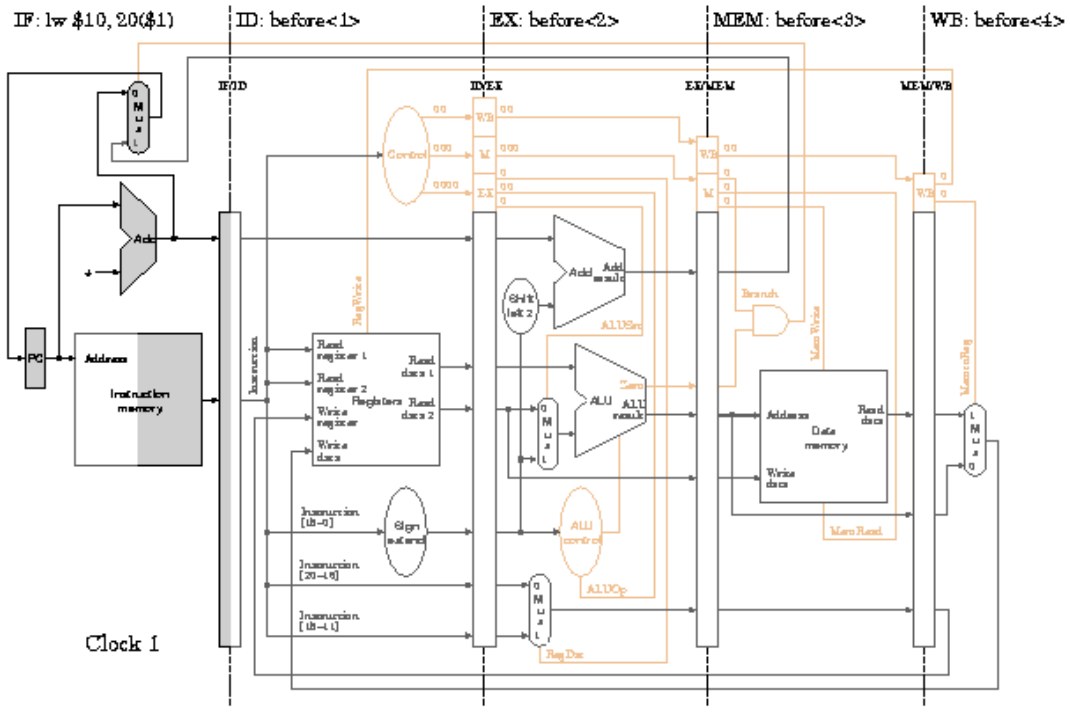

Fig. 5.24 Ciclo de reloj 1 de la secuencia bajo ejecución.

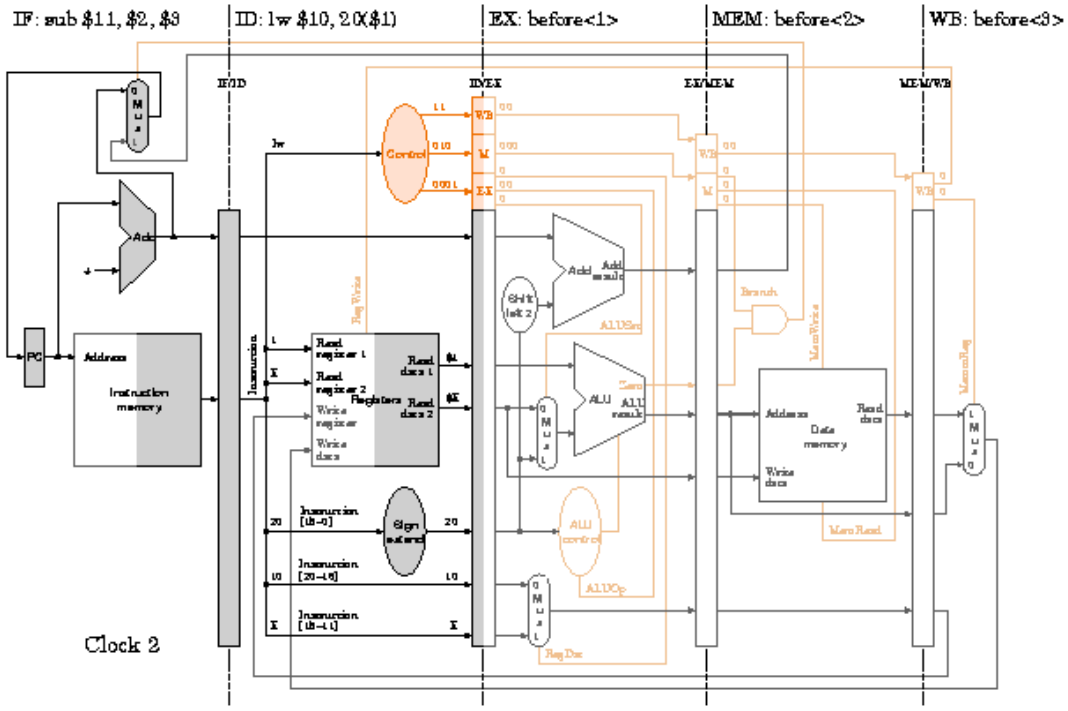

Fig. 5.25 Ciclo de reloj 2 de la secuencia bajo ejecución.

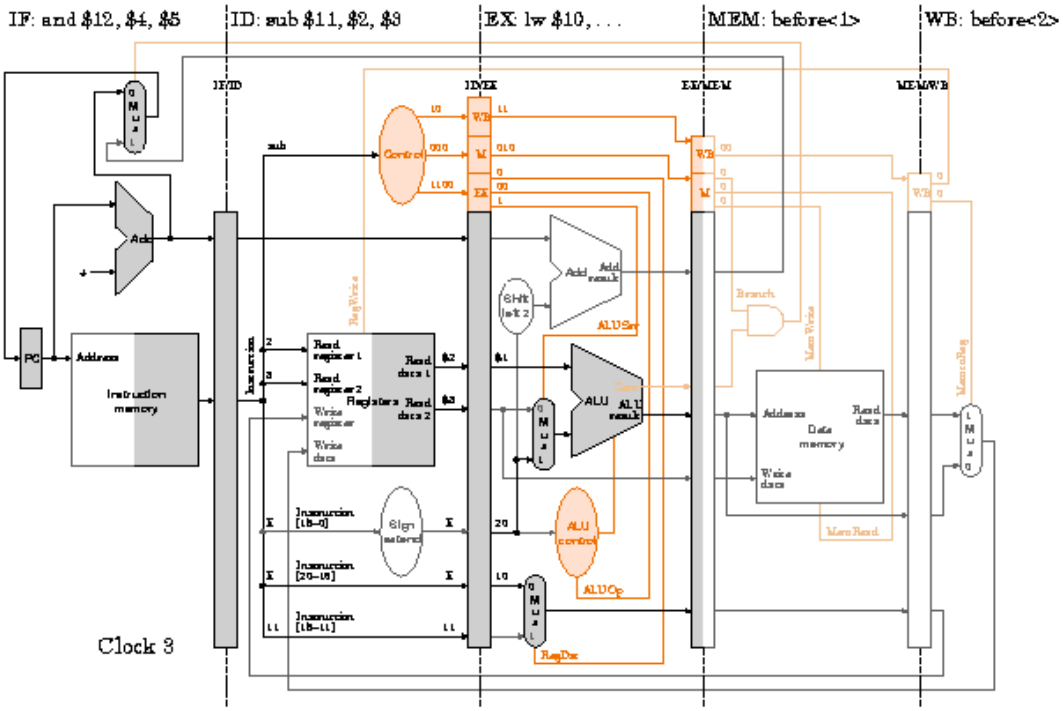

Fig. 5.26 Ciclo de reloj 3 de la secuencia bajo ejecución.

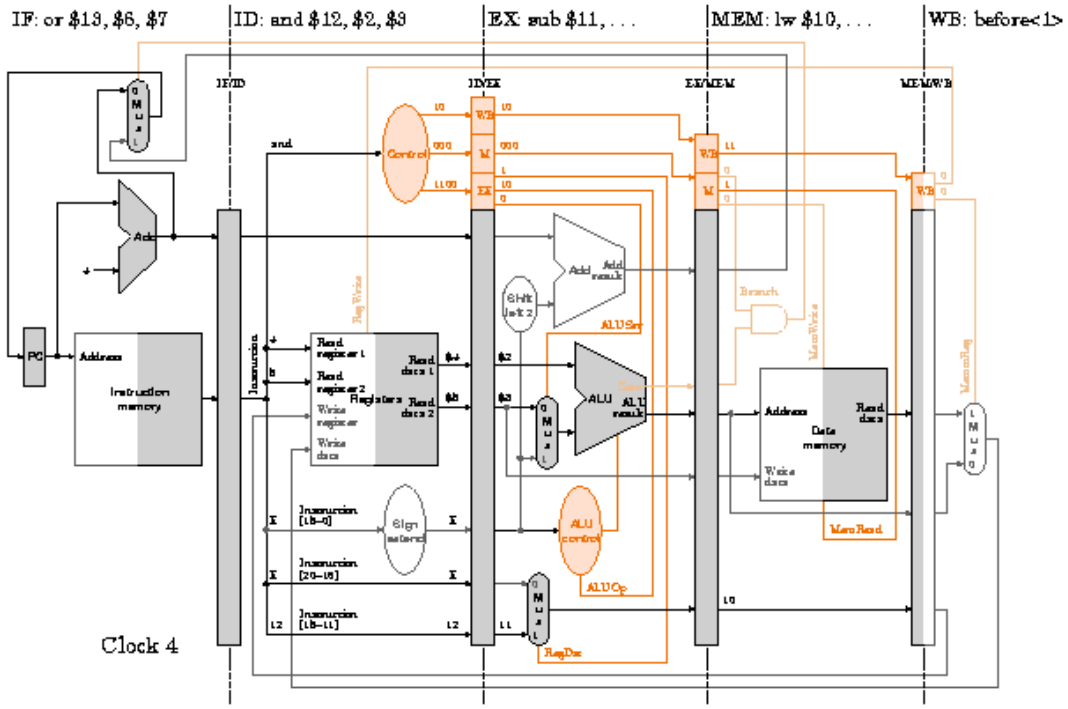

Fig. 5.27 Ciclo de reloj 4 de la secuencia bajo ejecución.

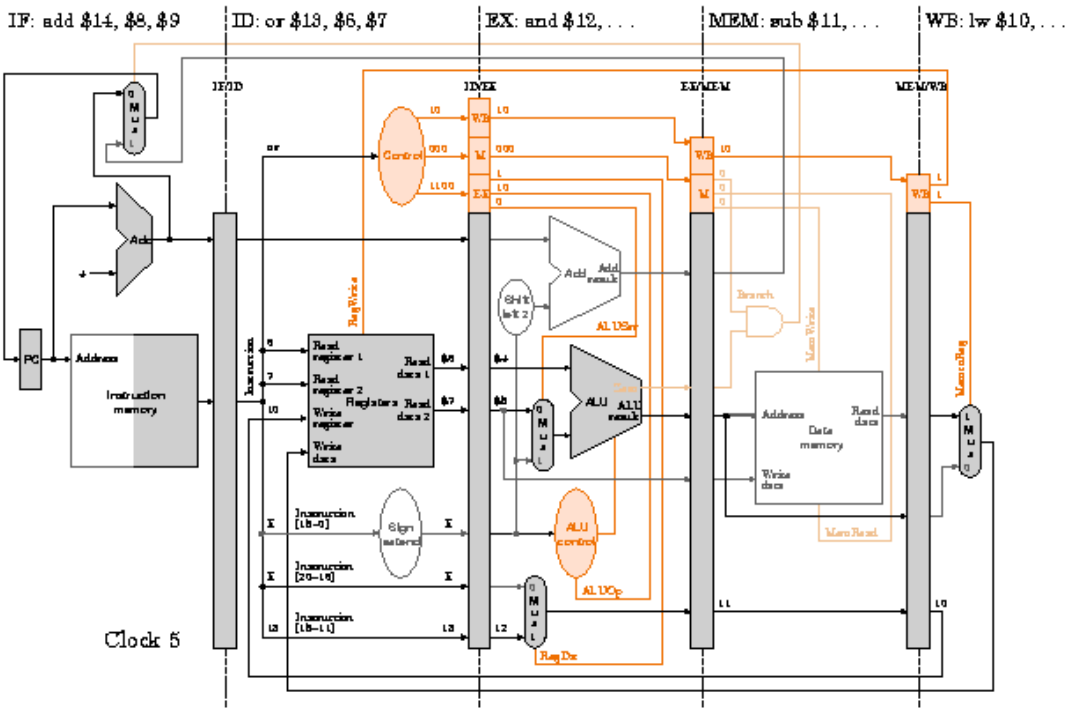

Fig. 5.28 Ciclo de reloj 5 de la secuencia bajo ejecución.

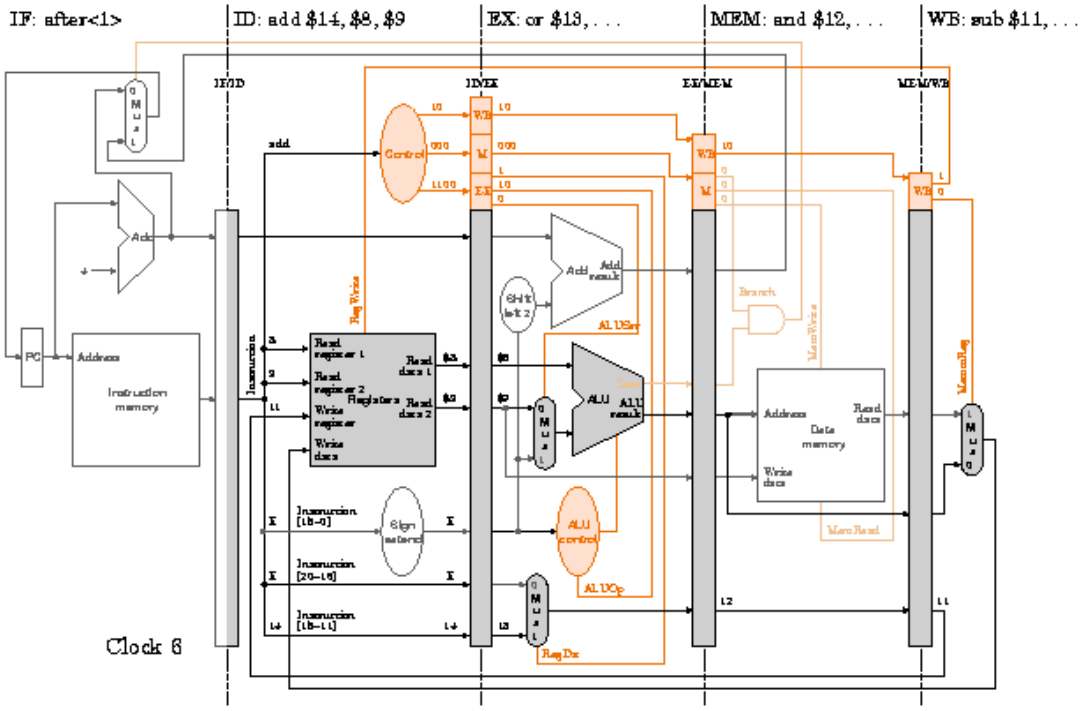

Fig. 5.29 Ciclo de reloj 6 de la secuencia bajo ejecución.

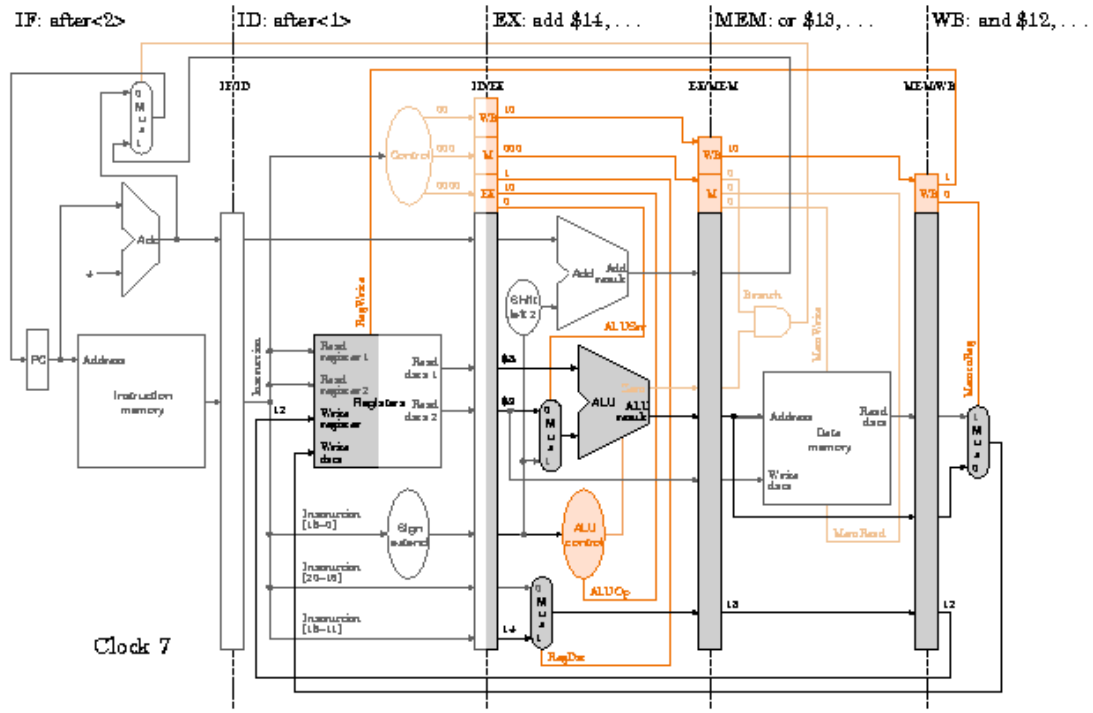

Fig. 5.30 Ciclo de reloj 7 de la secuencia bajo ejecución.

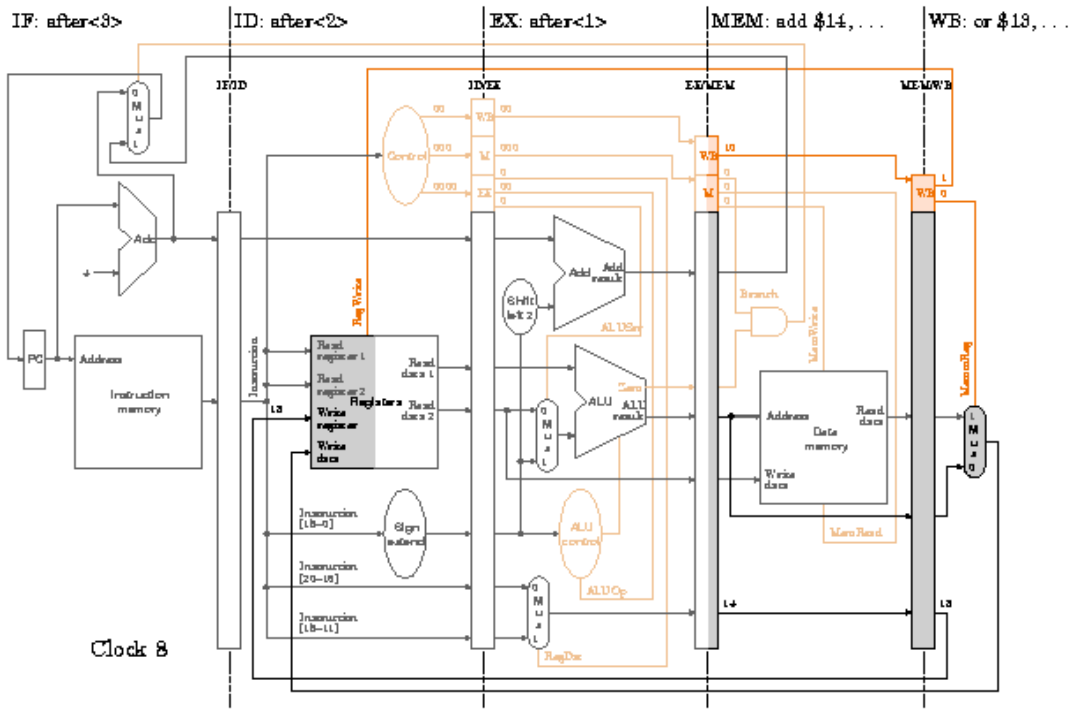

Fig. 5.31 Ciclo de reloj 8 de la secuencia bajo ejecución.

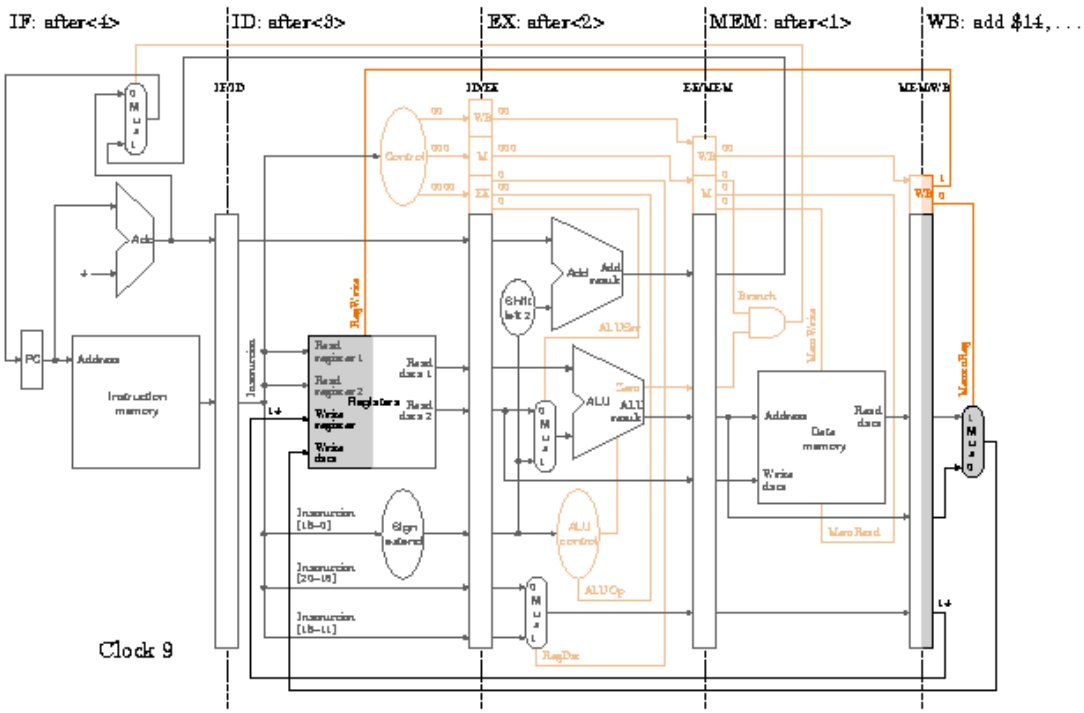

Fig. 5.32 Ciclo de reloj 9 de la secuencia bajo ejecución.

Tarea 11

1. Determinar el tamaño de cada uno de los cuatro registros que separan las diferentes etapas de la segmentación (justifique su respuesta).

2. Mostrar la *representación abreviada*, para la secuencia de cinco instrucciones considerada en el ejemplo anterior.

3. Mostrar la *representación de múltiples-ciclos* de una ejecución segmentada, para la secuencia de cinco instrucciones considerada en el ejemplo anterior (Recordar que solo se sombrean las unidades funcionales que están siendo utilizadas).

4. Considerando la ejecución del siguiente código:

```
add     $1, $2, $3
add     $4, $5, $6
add     $7, $8, $9
add     $10, $11, $12
add     $13, $14, $15
add     $16, $17, $18
```

En los ciclos de reloj 5 y 7 ¿Cuáles registros se están leyendo y cuales se están escribiendo? (De los registros de propósito general - $1 a $31-).

5.4 Riesgos por dependencia de datos

Hasta el momento se han considerado secuencias de código en las que no hay dependencias de datos, en esos casos la implementación mostrada en la figura 5.23 trabaja correctamente.

Los **Riesgos por dependencias de datos** ocurren cuando la ejecución de una instrucción depende del resultado de una instrucción previa que aún está en la segmentación. Por ejemplo, si se ejecutan las instrucciones:

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

De acuerdo a las etapas en las que se dividió la ejecución de instrucciones, el resultado de la suma se escribirá al final de la quinta etapa en el registro $s0 y la resta hace su lectura de registros en la segunda etapa. Suponiendo que cada etapa tarda 2 ns, el resultado de la suma se escribe a los 10 ns, y la resta esperaba leer el registro a los 4 ns, como se muestra en la figura 5.33, la resta leerá el valor anterior de $s0.

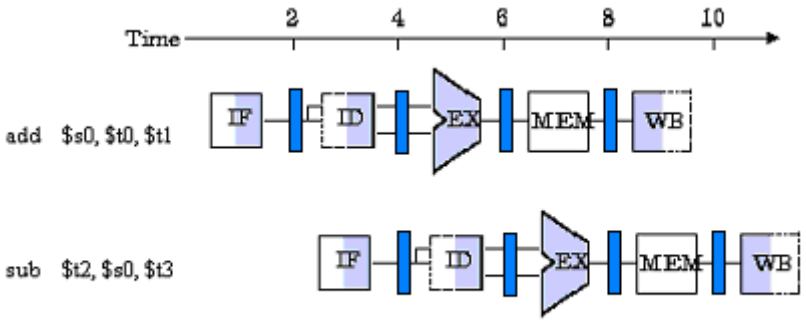

Fig. 5. 33 Riesgos por dependencias de datos.

Una posible solución es insertar tres instrucciones NOP (no operación) entre estas dos instrucciones para que cuando la resta requiera el valor de $s0, éste ya lo tenga escrito correctamente. Esta solución se muestra en la figura 5.34, el efecto de las instrucciones NOP se muestra sin sombrear las unidades funcionales de las etapas 2, 3, 4 y 5, por que solo se hace la lectura de la instrucción.

En la figura puede observarse que la instrucción de suma escribe el resultado a los 10 ns, y la resta lo utiliza sin ningún problema en el ciclo comprendido entre 10 ns y 12 ns.

Sin embargo, existen dos desventajas en esta solución: La primera es que el compilador debe detectar los riesgos y cada vez que los encuentre, debe resolverlos calculando el número de NOPs adecuados para insertarlos y la segunda es que la ejecución se vuelve demasiado lenta, no se obtienen los beneficios que se esperarían de la segmentación.

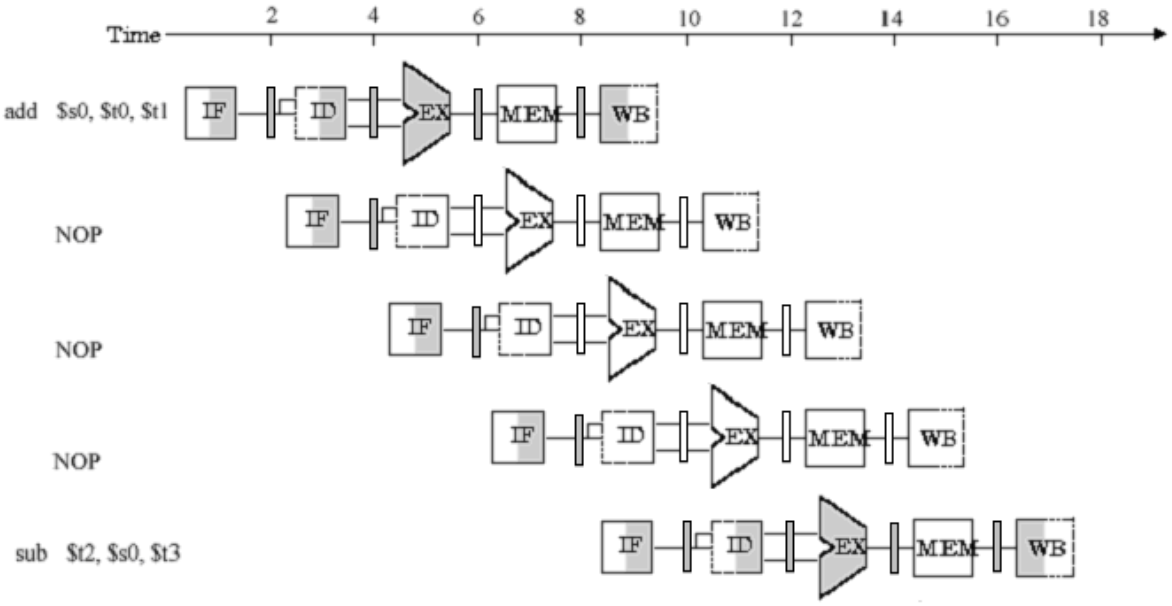

Fig. 5. 34 La inserción de instrucciones NOP resuelve los riesgos por dependencias de datos.

Buscando como mejorar esta solución se plantea la siguiente pregunta, si la ALU ya tiene el resultado al final de la etapa 3, ¿Por qué esperar a que este resultado sea escrito en el

registro $s0? ¿Sería posible tomarlo inmediatamente después de que lo generó la ALU?, en la figura 5.35 se muestra que si es posible, porque la resta requiere el valor de $s0 en el siguiente ciclo de reloj.

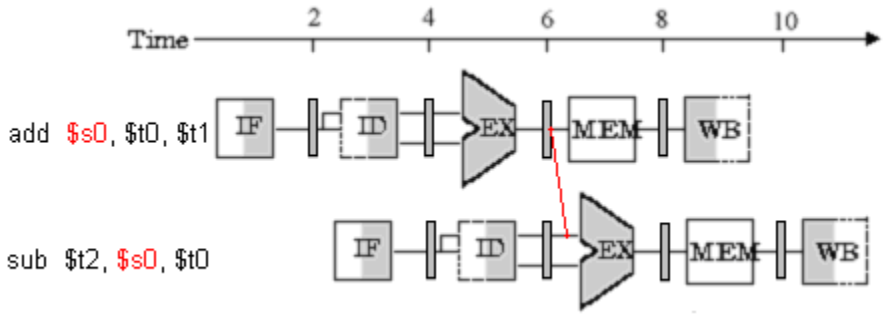

Fig. 5. 35 Se muestra que es posible tomar el resultado antes de que se escriba en el registro $s0

Al esquema mostrado en la figura 5.35 se le conoce como *anticipación.* La anticipación es una técnica para resolver los riesgos por dependencias de datos que consiste en tomar el dato de uno de los operandos de la ALU antes de que éste sea escrito en el registro destino. Tiene como ventaja que los riesgos se resuelven a nivel de hardware, por lo que el compilador no debe insertar instrucciones NOP y por lo tanto, no provoca retrasos en la ejecución de un programa.

Debe tomarse en consideración que el camino de datos es único y que los resultados de una etapa se escriben en alguno de los registros encargados de separar etapas. De manera que la línea roja indica que de ahí se tomará la información y se regresará a una etapa anterior para que se utilice por la instrucción siguiente.

Ahora, que ocurre cuando se intenta ejecutar la siguiente secuencia:

```
lw      $s0, 20 ( $t1 )
sub     $t2, $s0, $t3
```

En este caso, la carga obtendrá el dato de memoria hasta el final de la etapa 4, por lo que aún con la anticipación la resta no puede disponer del valor correcto para $s0, lo necesita un ciclo de reloj antes de que la carga lo obtenga. Una solución sería insertar una instrucción NOP entre estas dos instrucciones, sin embargo, como se comentó anteriormente, esta solución no es adecuada por que involucra la modificación del compilador.

Se requiere que la resta se *detenga* un ciclo de reloj para que luego pueda usarse la anticipación. Las *detenciones* son necesarias en aquellos casos en que la *anticipación* no es suficiente. Si la resta se *detiene* por un ciclo de reloj, en su lugar debe insertarse un conjunto de señales inofensivas, a este conjunto se le conoce como una *burbuja*. Una *burbuja* se inserta entre las dos instrucciones en conflicto y no afecta la ejecución del programa.

En la figura 5.36 se muestra como después de insertar una *burbuja* ya es posible anticipar el resultado esperado en $s0 para usarse en la instrucción siguiente. Las burbujas se insertan

al nivel de hardware, por lo que son transparentes al compilador. Es un hecho que las burbujas producen un retraso de un ciclo de reloj, pero sólo se insertarán cuando el dato a cargar en un registro será utilizado como operando de la ALU en la siguiente instrucción.

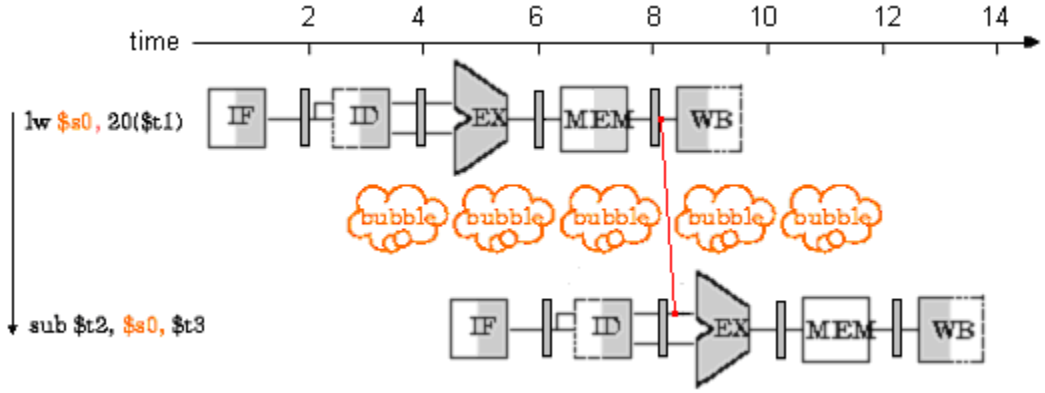

Fig. 5. 36 Algunas veces son necesarias las detenciones.

La anticipación se revisa en la sección 5.5 y las detenciones en la sección 5.6.

5.5 Anticipación

La anticipación tomará el dato de uno de los operandos de la ALU, antes de que éste sea escrito en el registro destino. Se considera la siguiente secuencia de código para obtener los posibles casos:

```
sub   $2, $1, $3      # El registro $2 es escrito por sub
and   $12, $2, $5     # El 1er operando ($2) depende de sub
or    $13, $6, $2     # El 2º operando ($2) depende de sub
add   $14, $2, $2     # Los 2 operandos dependen de sub
sw    $15, 100( $2 )  # El registro base ($2) depende de sub
```

En la figura 5.37 se muestra la representación de múltiples ciclos de la ejecución de esta secuencia. Considerando como ciclo de reloj 1 cuando la instrucción SUB entra a la segmentación, se tiene que la instrucción SUB genera con la ALU el valor que se escribirá en $2 y lo escribe en el registro EX/MEN al final del ciclo de reloj 3.

En el ciclo de reloj 4, la instrucción AND puede tomar esta información del registro EX/MEN y regresarla a la etapa 2 para que sea su primer operando de la ALU, al final de este ciclo, la instrucción SUB escribirá el valor para $2 en el registro MEM/WB.

Al comienzo del ciclo de reloj 5, la OR toma el valor de $2 del registro MEM/WB y lo regresa a la etapa 3 para que sea su segundo operando de la ALU. Durante ese ciclo se espera que se haga la escritura del registro $2 para que su valor sea usado por la instrucción ADD durante el mismo ciclo.

Esto es posible si los registros que separan las etapas de la segmentación y el archivo de registros activan su escritura en flancos diferentes, así, durante el flanco de subida se puede

escribir el registro MEM/WB con la información y las señales de control correspondientes, durante la primera mitad del ciclo estas señales se establecen, de manera que cuando ocurre el ciclo de bajada, el archivo de registros escribirá la información correcta (la instrucción SUB escribirá el valor de $2) y durante la segunda mitad del ciclo, se puede disponer de los datos actualizados para su lectura (la instrucción ADD leerá el nuevo valor de $2).

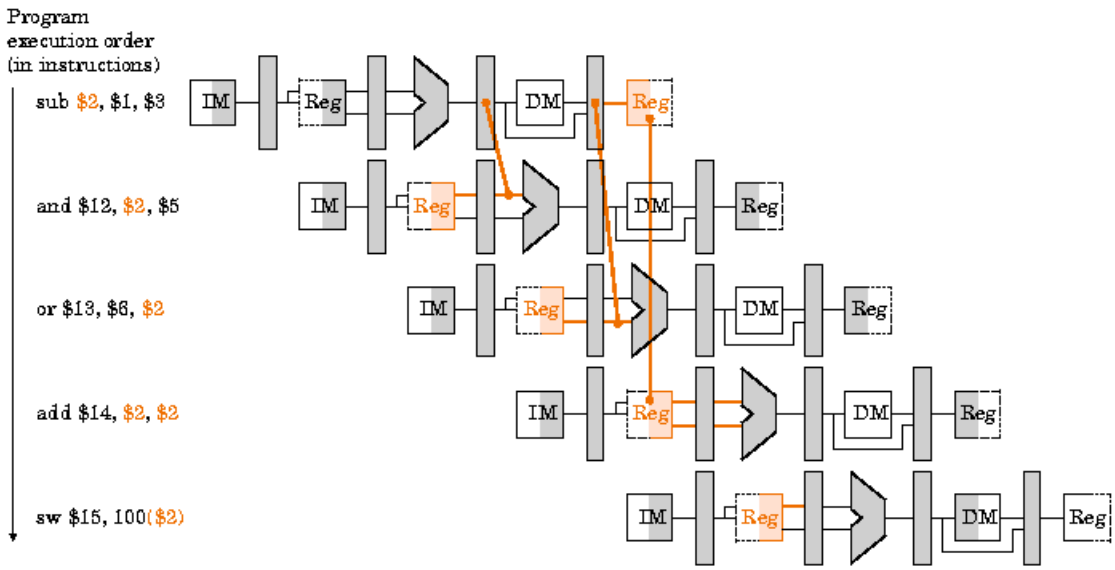

Fig. 5. 37 Ejecución de una secuencia con anticipaciones.

El ciclo de reloj debe ser suficiente para permitir la escritura y lectura del archivo de registros.

Puede notarse, en la figura 5.37, que la instrucción SW ya no tiene algún riesgo al usar el valor del registro $2 en el ciclo de reloj 6, puesto que su valor se escribió en el ciclo anterior.

Entonces, la anticipación puede hacerse de diferentes lugares, puede ocurrir que el dato a anticipar se encuentre en el registro EX/MEN o en el registro MEM/WB, y en ambos casos puede tratarse del primero o del segundo operando, incluso de ambos.

En todos los casos, las anticipaciones involucran a los operandos de la ALU y la ALU está en la etapa 3, en una implementación sin anticipaciones (fig. 5.38) los dos operandos se toman directamente del registro ID/EX. Al considerar las anticipaciones, se debe saber cuál será el registro destino en las instrucciones que están en las etapas MEM y WB, y si se va a escribir dicho registro. Si ese registro coincide con alguno de los operandos de la ALU, su valor debe regresarse a la etapa EX para que sustituya a ese operando. En la figura 5.39 se muestra la incorporación de la Unidad de anticipación (*Forwarding Unit).*

La unidad de anticipación compara cada uno de los operandos de la ALU (rs y rt) con el registro destino de la etapa MEM (EX/MEM.RegistroRd) y con el registro destino de la etapa WB (MEM/WB.RegistroRd) si existe igualdad entre un par de ellos, y la señal de escritura en registro está acertada en la etapas MEM (EX/MEM.RegWrite) o en la etapa

WB (MEM/WB.REgWrite) , la unidad de anticipación debe colocar el valor adecuado para sustituir a algún operando, controlando los multiplexores de anticipación.

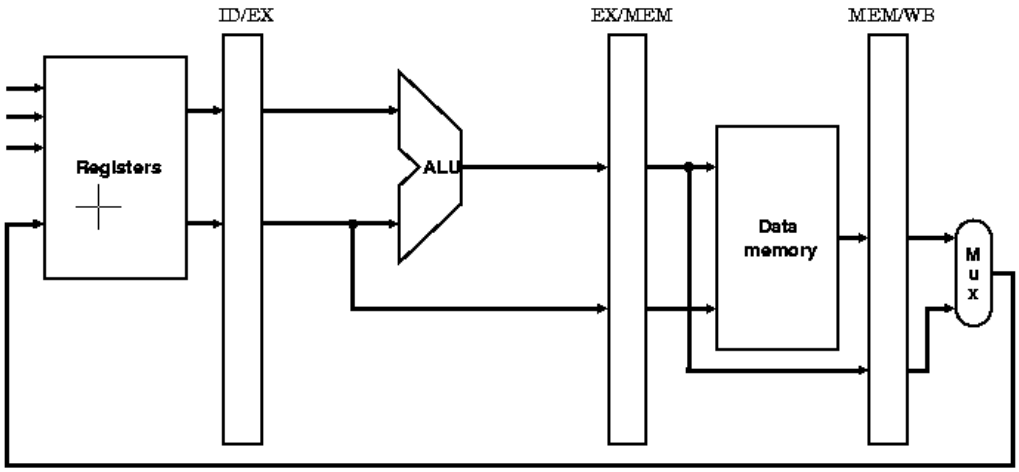

Fig. 5. 38 Segmentación sin anticipación.

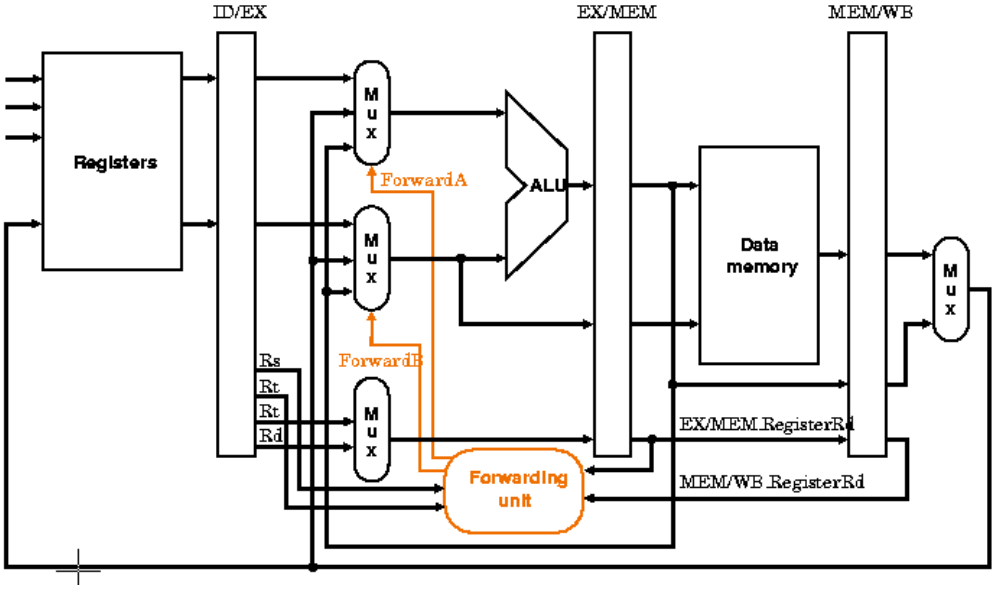

Fig. 5. 39 Incorporación de la Unidad de Anticipación.

Las decisiones que tomará la unidad de anticipación con respecto a la etapa MEM son:

if (EX/MEM.RegWrite **AND** EX/MEM.RegistroRd != 0 **AND**
EX/MEM.RegistroRd == ID/EX.RegistroRs **)**
ForwardA = 10

if (EX/MEM.RegWrite **AND** EX/MEM.RegistroRd != 0 **AND**
EX/MEM.RegistroRd == ID/EX.RegistroRt **)**
ForwardB = 10

Respecto a la etapa WB, la unidad de anticipación debe determinar:

if (MEM/WB.RegWrite **AND** MEM/WB.RegistroRd != 0 **AND**
MEM/WB.RegistroRd == ID/EX.RegistroRs **)**
ForwardA = 01

if (MEM/WB.RegWrite **AND** MEM/WB.RegistroRd != 0 **AND**
MEM/WB.RegistroRd == ID/EX.RegistroRt **)**
ForwardB = 01

Se garantiza que no se está intentando escribir en el registro 0, este registro siempre debe mantener el valor de 0 y debe estar protegido por hardware, pero aún con ello, si una instrucción intenta escribirlo y no se hace esta consideración, la unidad de anticipación provocará que ese resultado diferente de 0 sea considerado en la operación.

ForwardA y ForwardB son señales de dos bits que controlan los multiplexores de anticipación y determinan cuales serán los operandos de la ALU (Tabla 5.4).

Control	Origen	Explicación
ForwardA = 00	ID/EX	El 1er operando de la ALU viene del archivo de registros
ForwardA = 10	EX/MEM	El 1er operando de la ALU es anticipado del anterior resultado de la ALU
ForwardA = 01	MEM/WB	El 1er operando de la ALU es anticipado de la memoria de datos o de un previo resultado de la ALU
ForwardB = 00	ID/EX	El 2º operando de la ALU viene del archivo de registros
ForwardB = 10	EX/MEM	El 2º operando de la ALU es anticipado del anterior resultado de la ALU
ForwardB = 01	MEM/WB	El 2º operando de la ALU es anticipado de la memoria de datos o de un previo resultado de la ALU

Tabla 5. 4 Los valores de control para los multiplexores de Anticipación.

Una complicación se presenta si la etapa WB presenta un resultado para un registro y la etapa MEM presenta un resultado diferente para el mismo registro, y ese registro es un operando de la ALU. Por ejemplo, para el siguiente código:

```
add    $1, $1, $2
add    $1, $1, $3
add    $1, $1, $4
. . .
```

Cuando la segunda suma está en la etapa EXE, anticipa sin problema el valor del registro $1 de la etapa MEM (el cual fue generado por la primera suma).

Sin embargo, cuando la tercera suma alcanza la etapa EXE, tiene dos posibilidades para anticipar el valor de $1, en la etapa MEM se encuentra el valor generado por la segunda suma y en la etapa WB se encuentra el valor generado por la primera.

En ese caso, el resultado que debe anticiparse es el de la etapa MEM porque es el resultado más reciente. Por lo tanto, deben jerarquizarse las decisiones que toma la unidad de anticipación. Combinando ambas decisiones para el primer operando de la ALU se tiene:

```
if ( EX/MEM.RegWrite AND EX/MEM.RegistroRd != 0 AND
     EX/MEM.RegistroRd == ID/EX.RegistroRs )
        ForwardA = 10;

else if ( MEM/WB.RegWrite AND MEM/WB.RegistroRd != 0  AND
     MEM/WB.RegistroRd == ID/EX.RegistroRs )
        ForwardA = 01;

else
        FordwardA = 00;
```

Es similar para el segundo operando.

En la figura 5.40 se muestra el camino de los datos segmentado y el control, con la unidad de anticipación. Es importante aclarar que solo se esboza el camino de los datos, se omiten algunos detalles, como el multiplexor que se presenta en la segunda entrada de la ALU o el mismo control de la ALU, por que el punto importante en esta sección es el estudio de la unidad de anticipación.

Para concluir con el estudio de la anticipación, se evalúa la ejecución de la siguiente secuencia de código en un esquema con anticipación (Los registros con dependencias de datos se resaltan en rojo):

```
sub     $2, $1, $3
and     $4, $2, $5
or      $4, $4, $2
add     $9, $4, $2
```

Para la ejecución de esta secuencia se requiere de ocho ciclos de reloj.

En el primer ciclo, la instrucción SUB entra a la etapa IF de la segmentación, en este ciclo no ocurren aspectos relevantes con respecto a la anticipación.

En el segundo ciclo de reloj, la instrucción SUB avanza a la etapa ID y la instrucción AND ingresa a la etapa IF, similar al ciclo anterior, no hay aspectos relevantes en cuanto a la anticipación, sin embargo, el control generará las señales con las que se hará la escritura del resultado de la instrucción SUB.

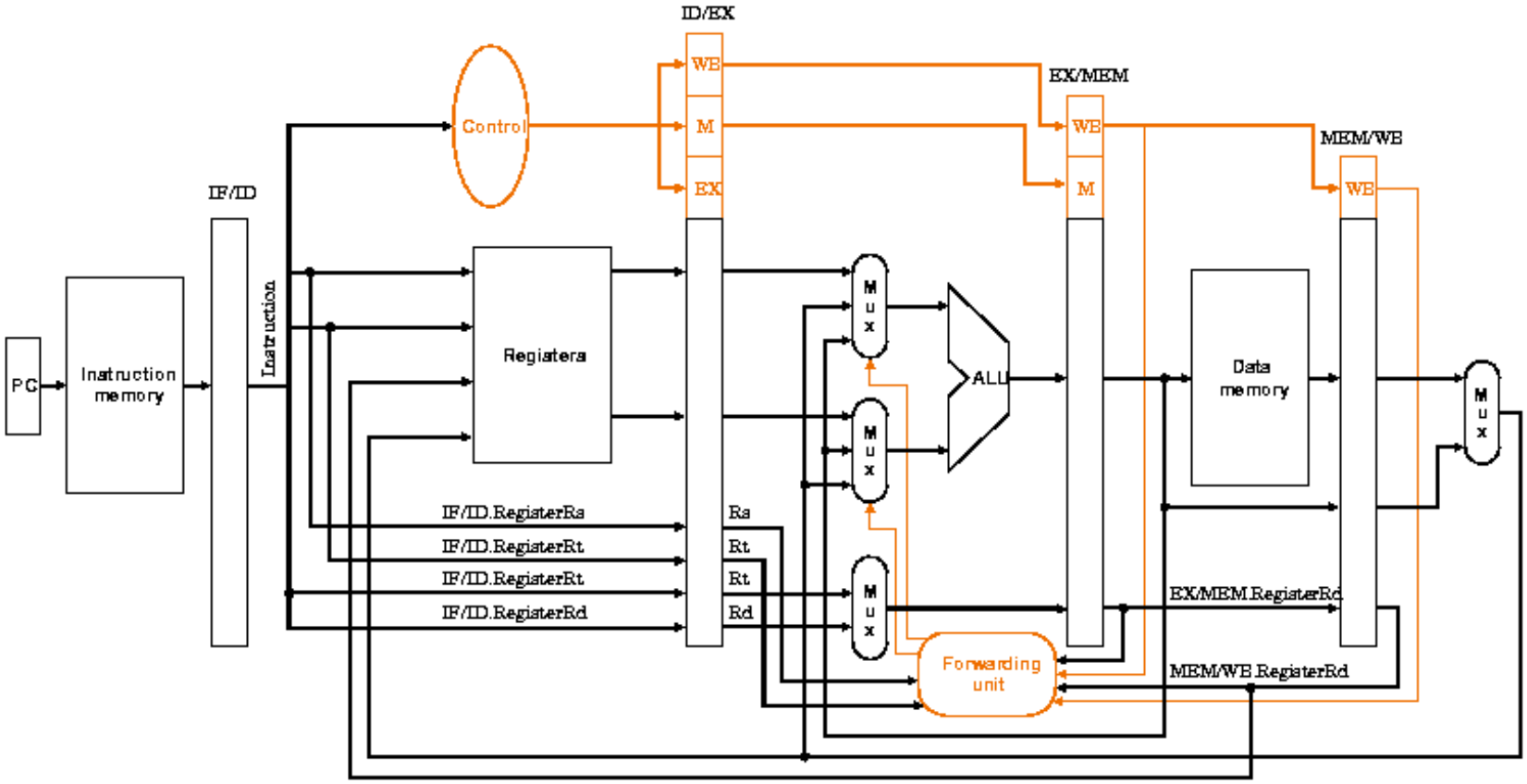

Figura 5.40 Se incorpora la Unidad de Anticipación a la implementación segmentada.

El tercer ciclo de reloj se muestra en la figura 5.41, la instrucción SUB llegó a la etapa EX y la ALU hace la resta de $1 con $3, aún no se puede determinar si hay dependencia de datos, puesto que se desconocen las instrucciones anteriores. La AND avanzó a la etapa ID y la OR ingresó a la etapa IF.

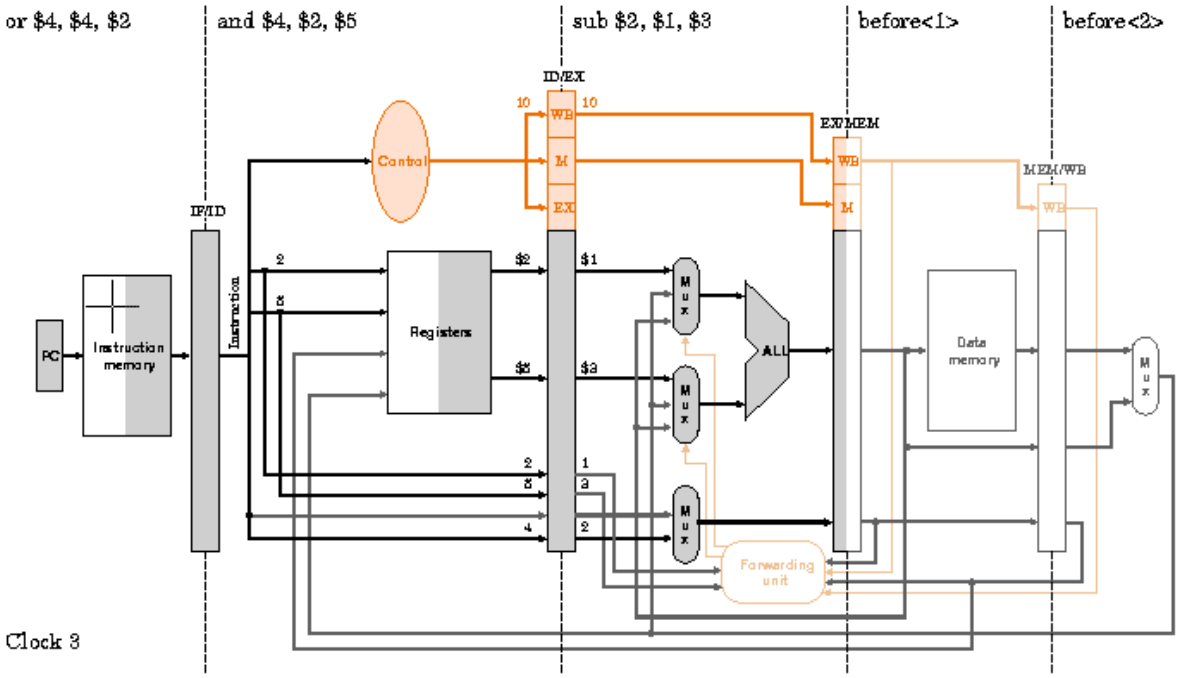

Figura 5.41 Ciclo de reloj 3 de la secuencia de instrucciones bajo consideración.

El cuarto ciclo de reloj se muestra en la figura 5.42, la instrucción SUB llega a la etapa MEM y la instrucción AND avanza a la etapa EX, en esta etapa, la unidad de anticipación detecta que $2 es un registro destino en la etapa MEM y que es el primer operando en la etapa EX, por lo que realiza la anticipación. De manera que en la etapa EX, el primer operando no se toma del archivo de registros. La instrucción OR llega a la etapa ID y la ADD ingresa a la segmentación.

En el quinto ciclo de reloj (figura 5.43), la instrucción OR llega a la etapa EX, la ALU va a trabajar con $4 y $2, sin embargo $4 es el registro destino de la instrucción AND que está en la etapa MEM y $2 es el registro destino de la SUB que está en la etapa WB, de manera que los dos operandos para la OR deben anticiparse, en este ciclo concluye la instrucción SUB e ingresa una instrucción desconocida. La ADD avanza a la etapa ID.

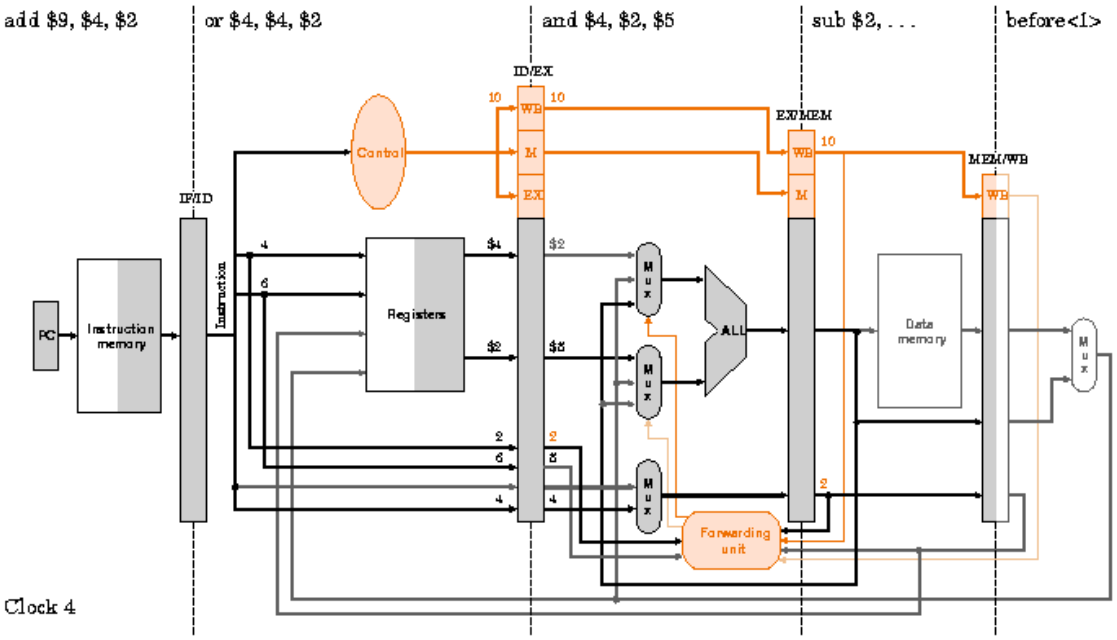

Figura 5.42 Ciclo de reloj 4 de la secuencia de instrucciones bajo consideración.

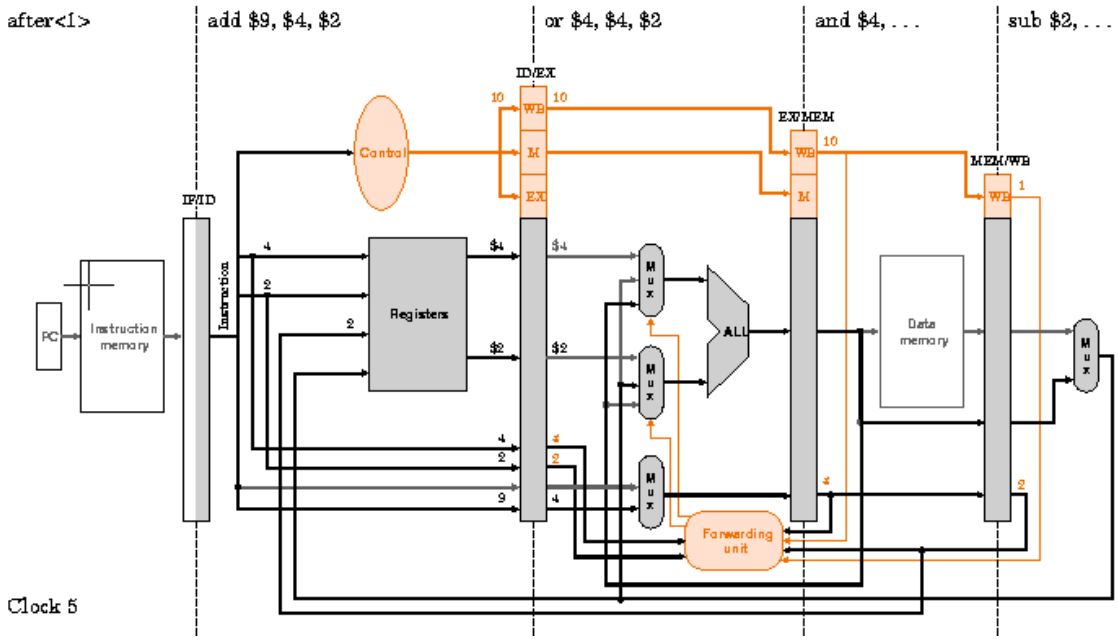

Figura 5.43 Ciclo de reloj 5 de la secuencia de instrucciones bajo consideración.

El sexto ciclo de reloj se muestra en la figura 5.44, la instrucción ADD avanza a la etapa EX, su primer operando es $4 y existen dos posibilidades para su anticipación, por que $4 es el registro destino de la OR que está en la etapa MEM y de la AND que está en la etapa WB. Se anticipa al que está en la etapa MEM puesto que es el último que se generó.

En los ciclos 7 y 8 la ADD avanza por las etapas MEM y WB hasta culminar su ejecución, e instrucciones desconocidas entran a la segmentación.

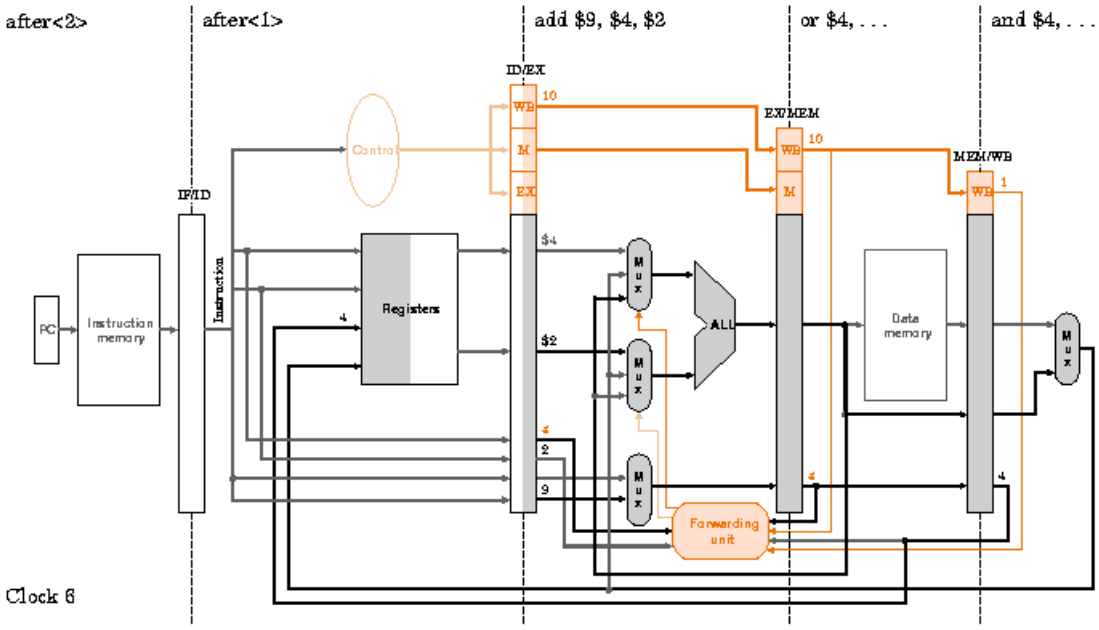

Figura 5.44 Ciclo de reloj 6 de la secuencia de instrucciones bajo consideración.

5.6 Detenciones

Como se describió en la sección 5.4, la anticipación no puede resolver aquellas situaciones en las que se realizará la carga de un registro (LW) y el dato a cargar es un operando de la instrucción siguiente. En la figura 5.45 se muestra un ejemplo del problema, se tiene una instrucción que cargará un dato de memoria al registro $2 seguida por una AND cuyo primer operando es $2.

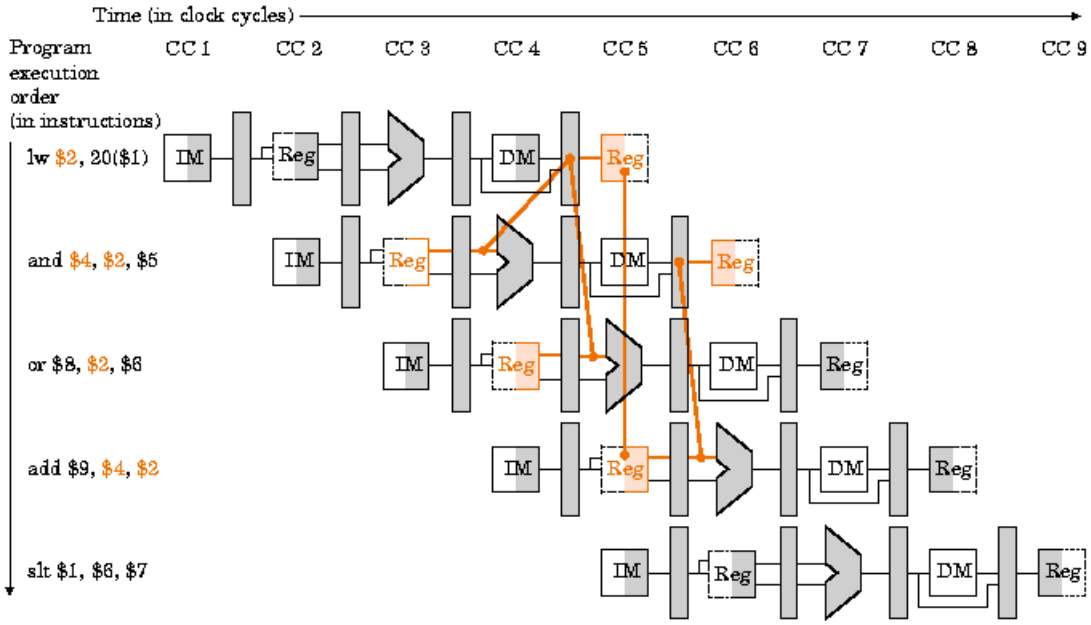

Figura 5.45 Se muestra la ejecución de una secuencia en la que los riesgos no pueden ser resueltos por la unidad de anticipación.

En el ciclo de reloj 4, el dato que se escribirá en $2 se está leyendo de la memoria y la ALU lo requiere en ese mismo ciclo, no es posible anticipar el valor de $2 por que se escribirá en el registro de segmentación hasta el final del ciclo y se podrá disponer de él hasta el siguiente ciclo. Por lo tanto, se debe *detener* a la instrucción AND por un ciclo de reloj, para que en el ciclo de reloj 5, con ayuda de la unidad de anticipación, pueda operar sobre el valor correcto de $2.

De manera que, además de la unidad de anticipación, es necesario agregar una *unidad de detección de riesgos (Hazard detection Unit)*, que opere en la etapa ID, de manera que pueda insertar una burbuja entre la carga y la instrucción que usará el dato a cargar. Esta nueva unidad evaluará la siguiente condición:

if (ID/EX.MemRead **AND**
(ID/EX.RegistroRt = IF/ID.RegistroRs OR
ID/EX.RegistroRt = IF/ID.RegistroRt) **)**
Inserta una Burbuja a la segmentación

La primera condición detecta si la instrucción que está en la etapa EX es una carga; las siguientes dos condiciones evalúan si el registro a cargar (ID/EX.RegistroRt) coincide con alguno de los dos operandos de la instrucción que está en la etapa ID (IF/ID.RegistroRs o IF/ID.RegistroRt), si se cumplen las condiciones, se debe detener la ejecución de la instrucción que está en la etapa ID por un ciclo de reloj y en su lugar insertar una burbuja, esto se muestra en la figura 5.46.

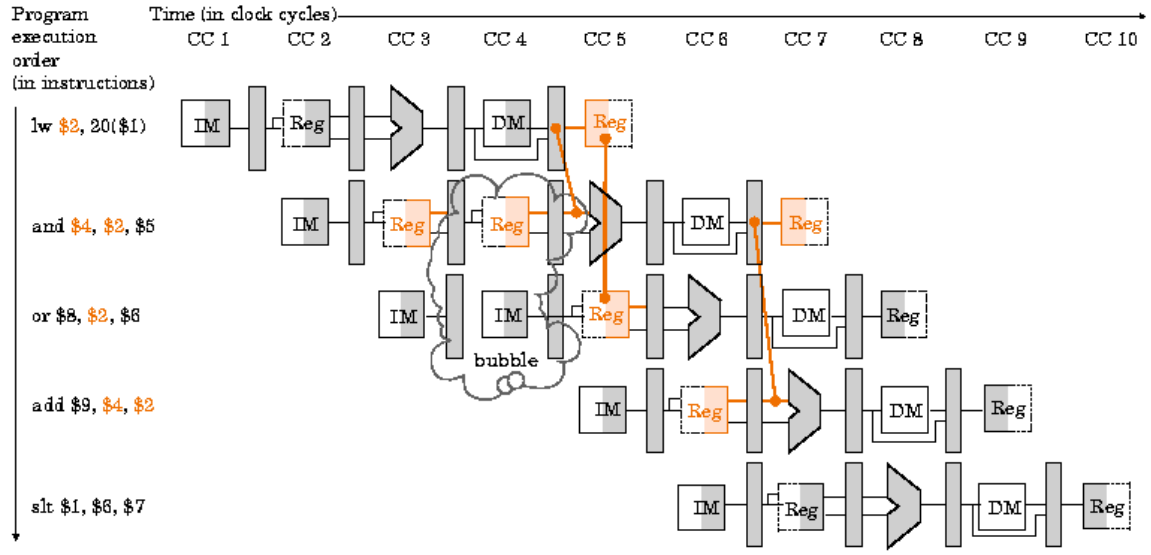

Figura 5.46 Se muestra la ejecución de la misma secuencia en la que se genera una burbuja para resolver los riesgos por dependencias de datos.

Una burbuja es un conjunto de señales que viajan sin realizar alguna operación, no escriben en registros o en memoria. Su efecto es muy similar al de una instrucción NOP, sólo que las burbujas se insertan al nivel de Hardware, retrasando a las instrucciones siguientes por un ciclo de reloj, mientras que las instrucciones NOP son parte del software.

Entonces, en la etapa ID se debe agregar un multiplexor por medio del cual la unidad de detección de riesgos pueda determinar si envía las señales de control generadas por la unidad de control o envía una burbuja (señales con valor cero).

Sin embargo, cuando se envíe una burbuja, las instrucciones que están en la etapa IF o en la etapa ID se deben detener, es decir, no deben avanzar en la segmentación. Anteriormente en estas etapas no existía alguna señal de control, puesto que no se tomaban decisiones sobre el tipo de instrucción. Ahora, es necesario agregar una señal que controle la escritura del contador del programa (PC) y del registro de segmentación IF/ID, de manera que si no se habilitan mantendrán su información actual y por lo tanto, detendrán el avance de las dos instrucciones que están en esa etapa. La habilitación de escritura de estas señales será controlada por la Unidad de Detección de Riesgos.

En la figura 5.47 se muestra la implementación segmentada: Camino de datos y control, con la unidad de detección de riesgos y la unidad de anticipación. En las siguientes tres figuras (5.48 a 5.50) se muestran algunos ciclos de la ejecución del código:

```
lw    $2, 20($1)
and   $4, $2, $5
or    $4, $4, $2
add   $9, $4, $2
```

Puede notarse en esas figuras como en el ciclo de reloj 4 las instrucciones AND y OR se detienen y se incorpora la burbuja. También puede notarse como la burbuja avanza por la segmentación sin afectar la ejecución del resto del programa. Y además, se observa que no hay conflictos con la unidad de anticipación.

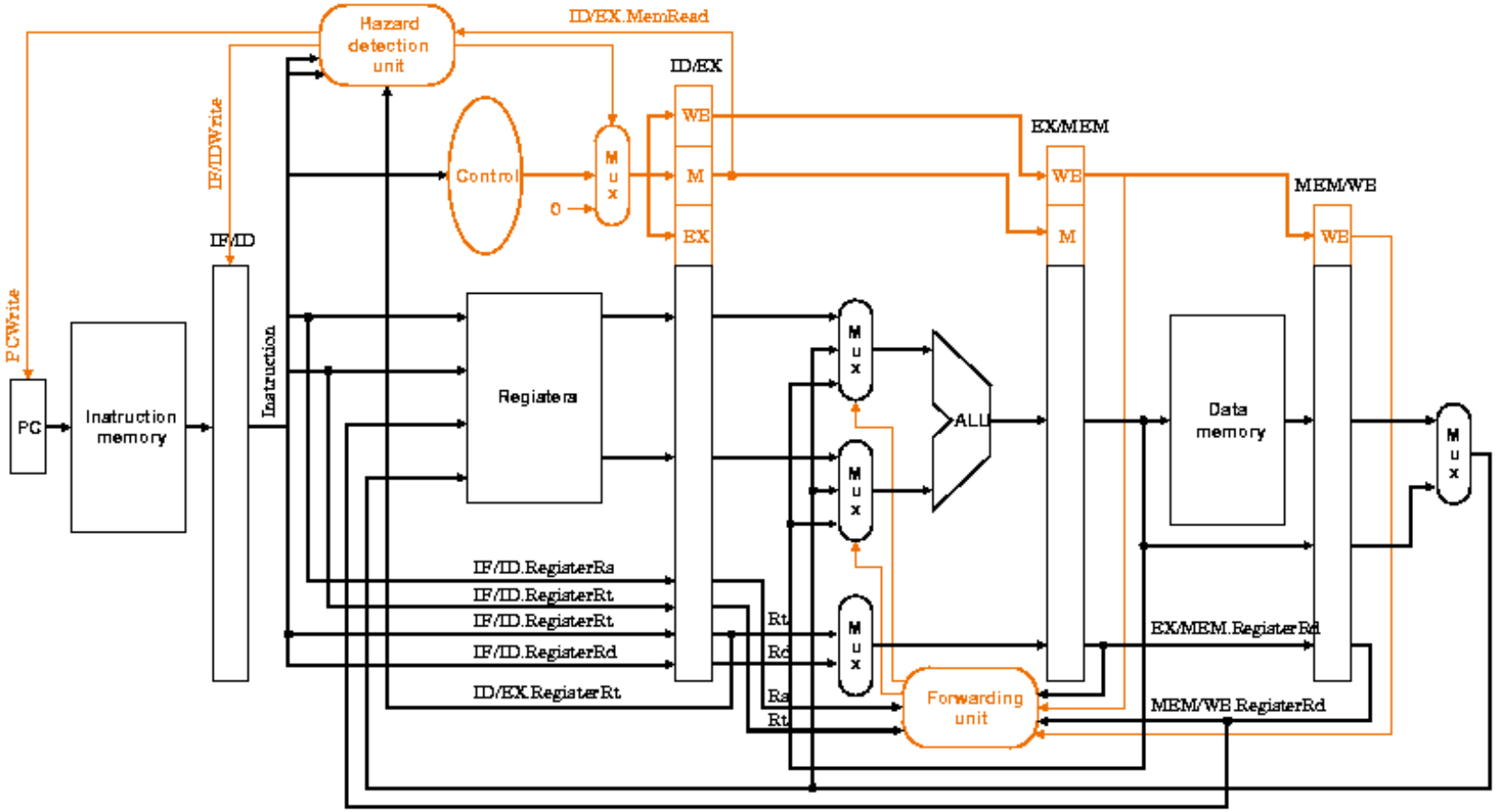

Figura 5.47 Una implementación segmentada: Camino de datos y control, con la unidad de detección de riesgos y la unidad de anticipación.

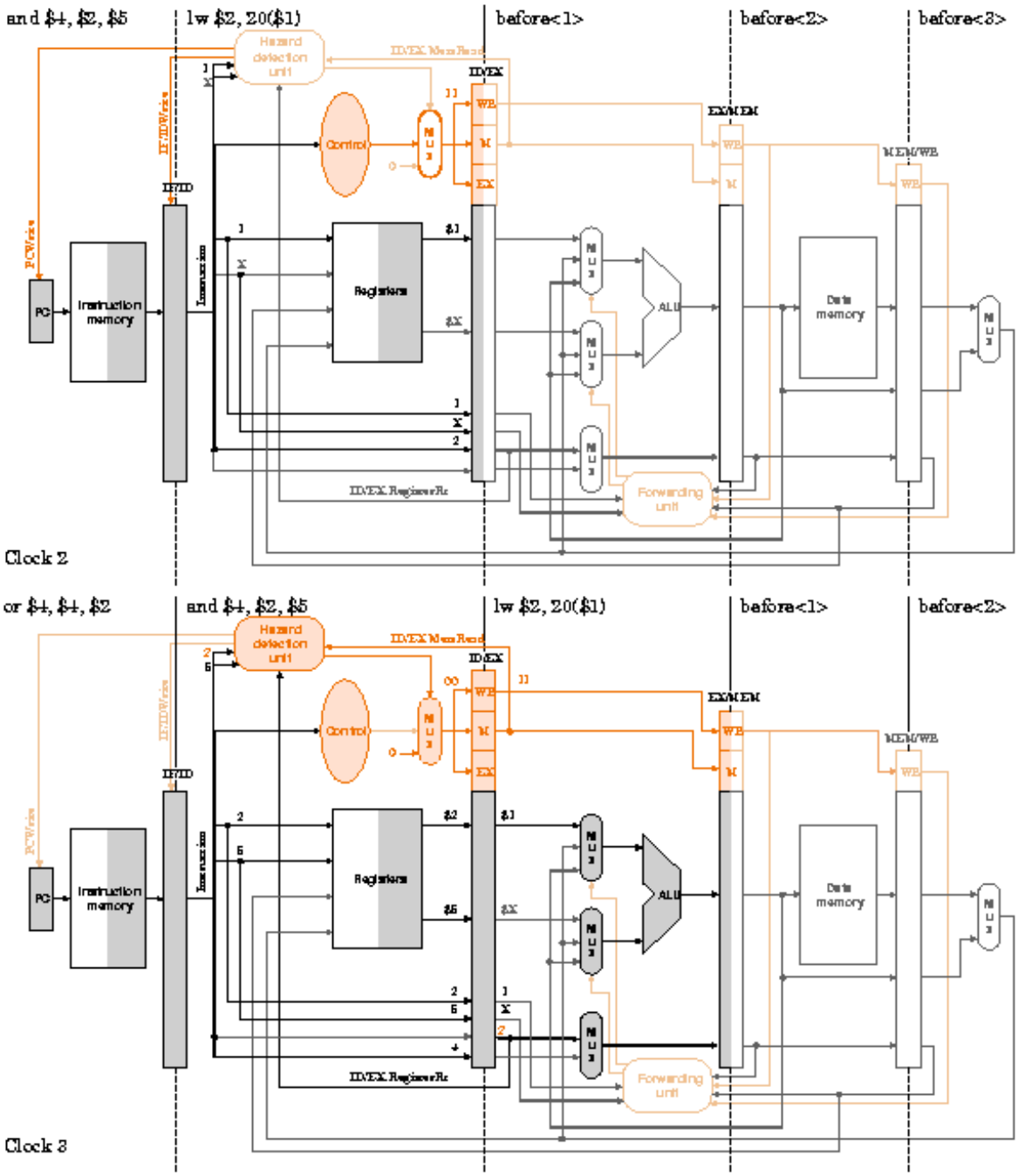

Figura 5.48 Ciclos de reloj 2 y 3 de la ejecución de la secuencia anterior. Es en el ciclo 3 en el que la unidad de detección de riesgos inserta la burbuja y detiene a las instrucciones de las dos primeras etapas.

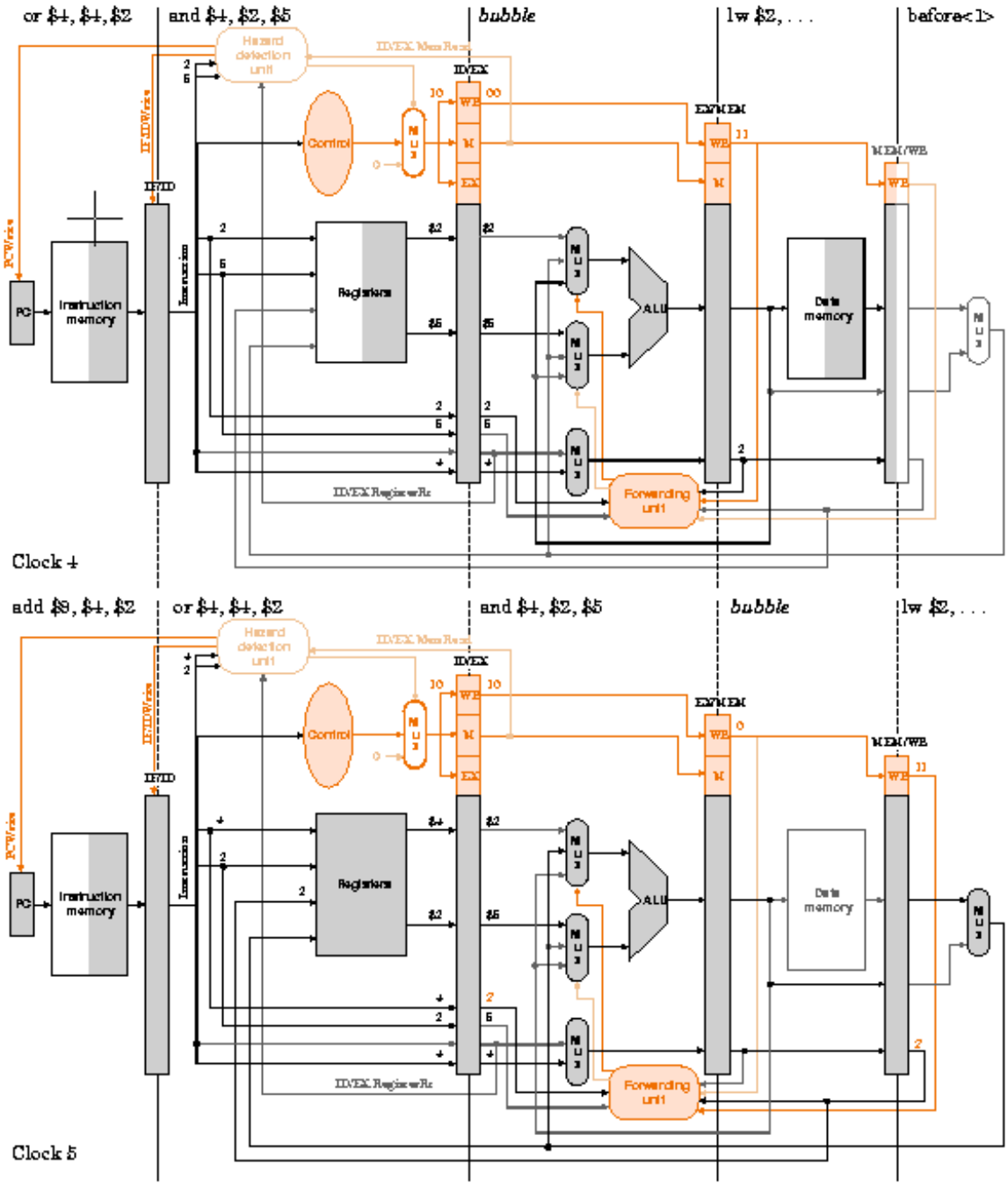

Figura 5.49 Ciclos de reloj 4 y 5. Es en el ciclo 4 la burbuja llega a la etapa EX, la instrucción LW llega a la etapa MEM y las instrucciones AND y OR se mantienen en las etapas ID e IF respectivamente. En el ciclo 5 la AND llega a la etapa EX y la unidad de anticipación resuelve las dependencias de datos.

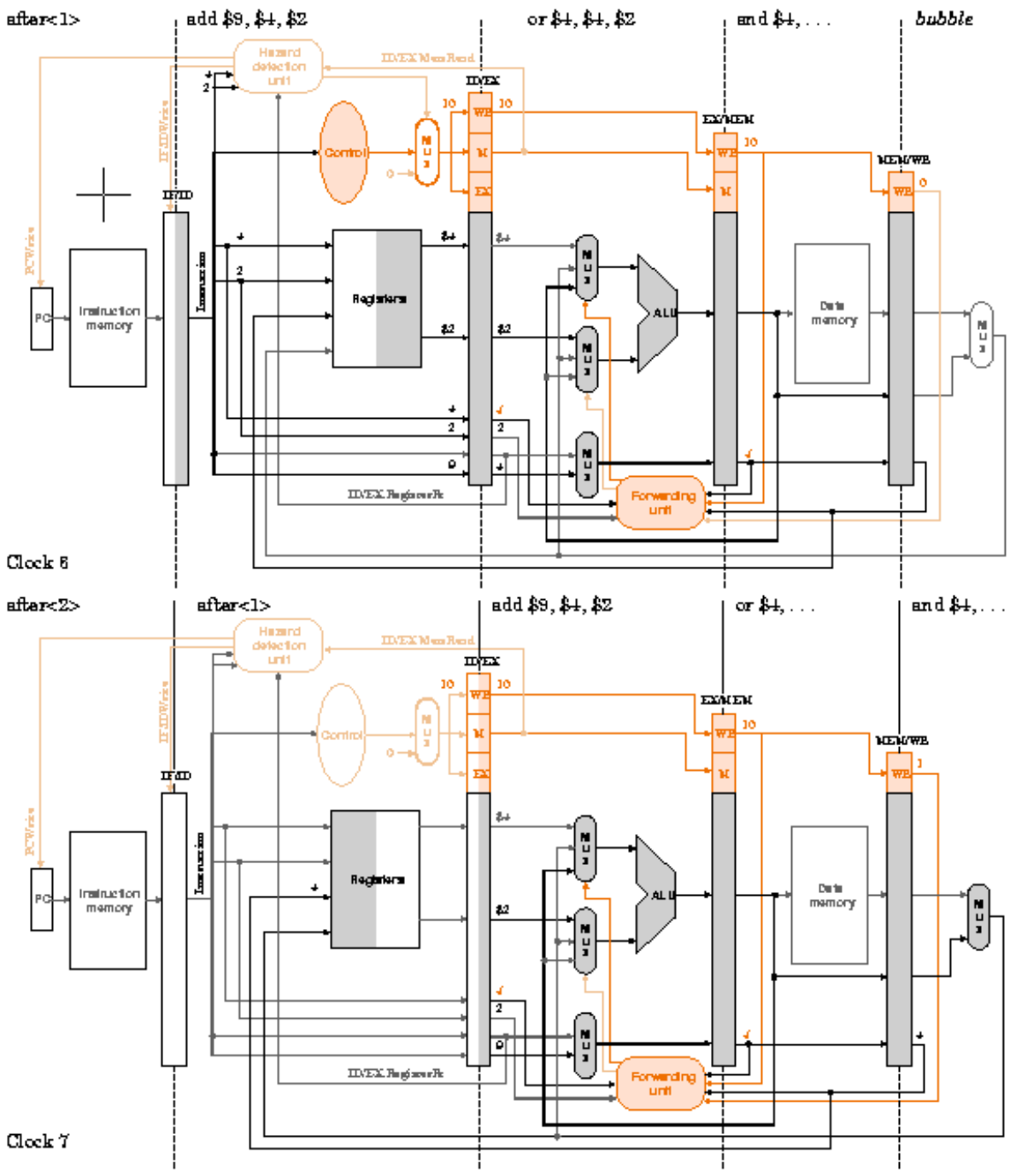

Figura 5.50 Ciclos de reloj 6 y 7. En estos ciclos la unidad de detección de riesgos ya no genera burbujas, las dependencias de datos las puede resolver la unidad de anticipación.

5.7 Riesgos en Brincos

Existe un problema en los brincos condicionales, al depender de la comparación de dos registros, cuando se determine la realización del brinco ya habrá otras instrucciones en la segmentación.

En la figura 5.23 se muestra que la decisión de la ejecución de un brinco se realiza en la etapa MEM, por que en la etapa anterior un sumador calculó la dirección destino del brinco y la ALU hizo la resta de los dos registros para la posible generación de la bandera de *zero*. De manera que cuando la instrucción BEQ llega a la etapa MEM, ya se tienen los argumentos para determinar si el brinco se realizará.

No es complicado modificar el valor del PC para continuar en la instrucción ubicada en la dirección destino del brinco, el problema es que en las etapas IF, ID y EX hay otras instrucciones que no deben continuar con su ejecución. En la figura 5.51 se muestra el impacto que tienen los brincos en la segmentación.

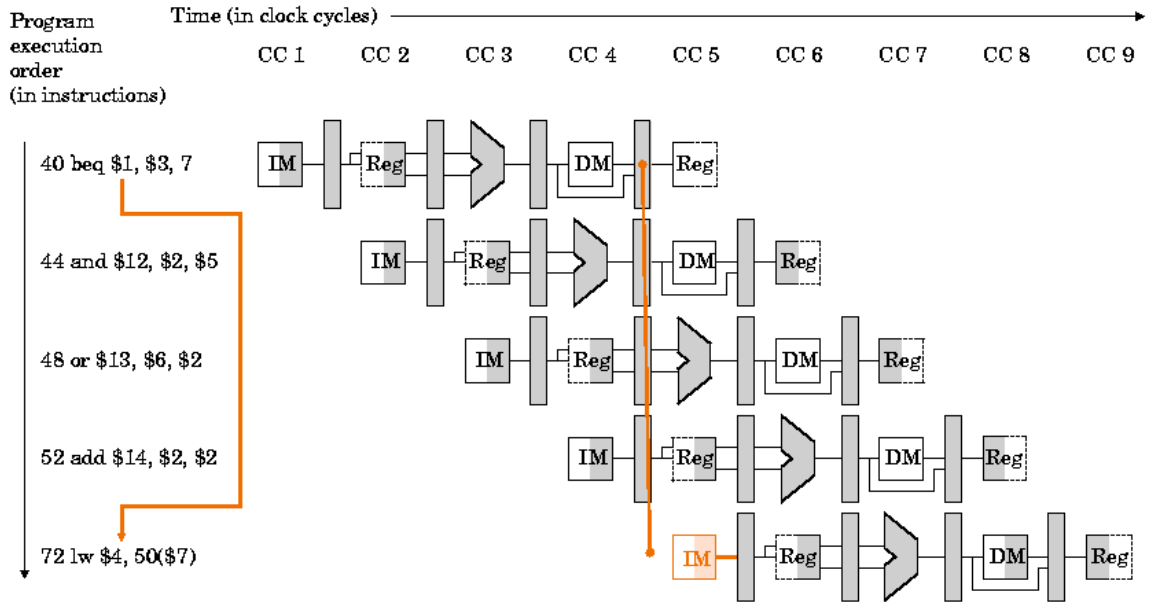

Figura 5.51 El impacto de los brincos en la segmentación.

La instrucción BEQ está en la dirección 40, y al ejecutase el PC automáticamente tomará en valor de 44. La etiqueta en BEQ corresponde a la constante 7, la cual al desplazarse a la izquierda en 2 (después de la extensión en signo) toma el valor de 28, de manera que el destino del brinco corresponde a la dirección 72 (44 + 28).

Una solución para eliminar las instrucciones que están en las etapa IF, ID y EX, consiste en agregar una unidad funcional en la etapa MEM, la cual al detectar que se trata de un brinco que si se realizará, se encargue de *inicializar* a los registros de segmentación IF/ID, ID/EX y EX/MEM (en la práctica los registros incluyen una señal de Reset), buscando que sus valores se sustituyan con 0s, con ello se sustituirán a esas tres instrucciones por

instrucciones no importa y en el siguiente ciclo de reloj se incorporará la instrucción ubicada en la dirección destino del brinco.

Los saltos incondicionales (J) solo requieren eliminar una instrucción, ya que una vez que se detecte el opcode de una instrucción J, el valor del PC se debe sustituir por el destino del salto y se debe limpiar el registro IF/ID para eliminar la instrucción que está en la etapa IF.

En la figura 5.52 se muestra la consideración de los brincos y saltos, se aprecia que el registro IF/ID puede ser reiniciado por dos condiciones diferentes.

La solución anterior es efectiva y bastante fácil de implementar, el problema es que por cada brinco condicional realizado se perderán los 3 ciclos de reloj siguientes, reduciendo el rendimiento de la implementación. Entonces, es necesario buscar otra solución que minimice los ciclos de reloj desperdiciados. Definitivamente en la etapa IF no se puede determinar si se trata de un brinco condicional, porque en esta etapa apenas se está capturando a la instrucción. Sin embargo, si en la etapa ID se detecta que se trata de un brinco y se determina que éste se debe realizar, sólo se inicializaría al registro IF/ID y con ello solo se perdería un ciclo de reloj.

Esta idea parece congruente, pero requiere que en la etapa ID se agregue el hardware necesario para identificar a los brincos y determinar si se van a realizar. El sumador que estaba en la etapa EX, encargado de calcular la dirección destino del brinco, se puede trasladar sin problema a la etapa ID, para disponer inmediatamente de esa información.

Para la comparación de los dos registros, ya no puede utilizarse a la ALU por que trasladar la ALU a la etapa ID involucraría la eliminación de una etapa y por lo tanto, aumentaría la duración del ciclo de reloj (por que dos unidades funcionales principales trabajarían en un ciclo de reloj). De manera que se requiere de un módulo rápido dedicado a comparar el contenido de los registros, este módulo puede basarse en un conjunto de compuertas XOR, que comparen bit a bit y que sus salidas se conecten por medio de una compuerta AND. El módulo de comparación sobre igual se puede ubicar en la salida de los valores que están en los registros. Con ello, tan pronto se hizo la lectura de registros, ya se puede determinar si son iguales o diferentes.

En la figura 5.53 se muestra el hardware resultante de trasladar la evaluación de los brincos a la etapa ID. Con estos cambios, cuando un brinco se va a efectuar, habrá un retraso de un ciclo de reloj. Cabe aclarar que a diferencia de la figura 5.52, en la figura 5.53 solo se plantea la idea general, para que los brincos se resuelvan en la etapa 2 se deben hacer otras consideraciones prácticas.

Al trasladar la ejecución de los brincos a la etapa ID, los ciclos de reloj 3 y 4 de la secuencia mostrada en la figura 5.51 tendrían el comportamiento que se exhibe en la figura 5.54, sólo se anula una instrucción, pero ahora lo hace la unidad de control.

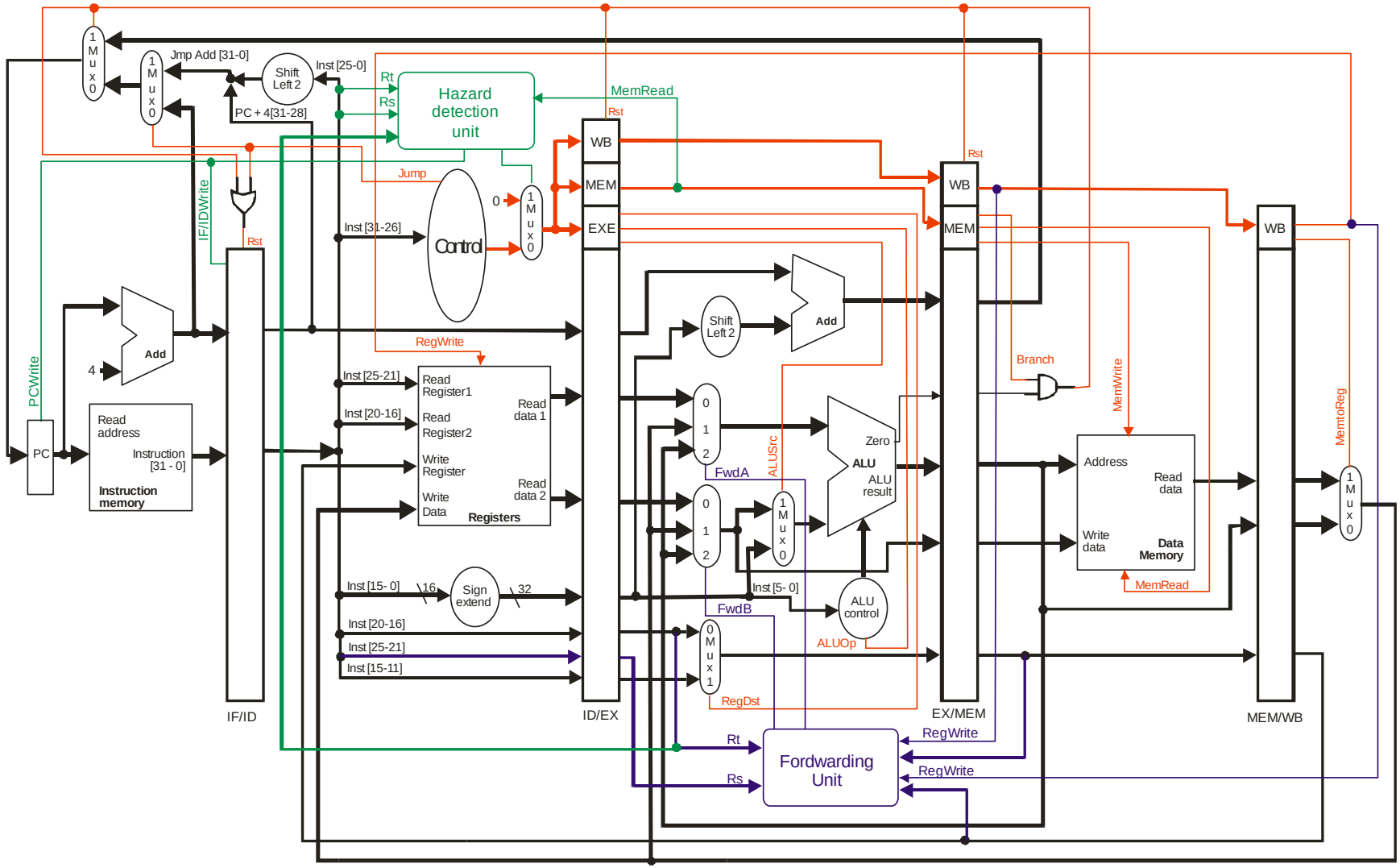

Figura 5.52 Implementación segmentada que resuelve a BEQ en la etapa MEM y a J en la etapa ID.

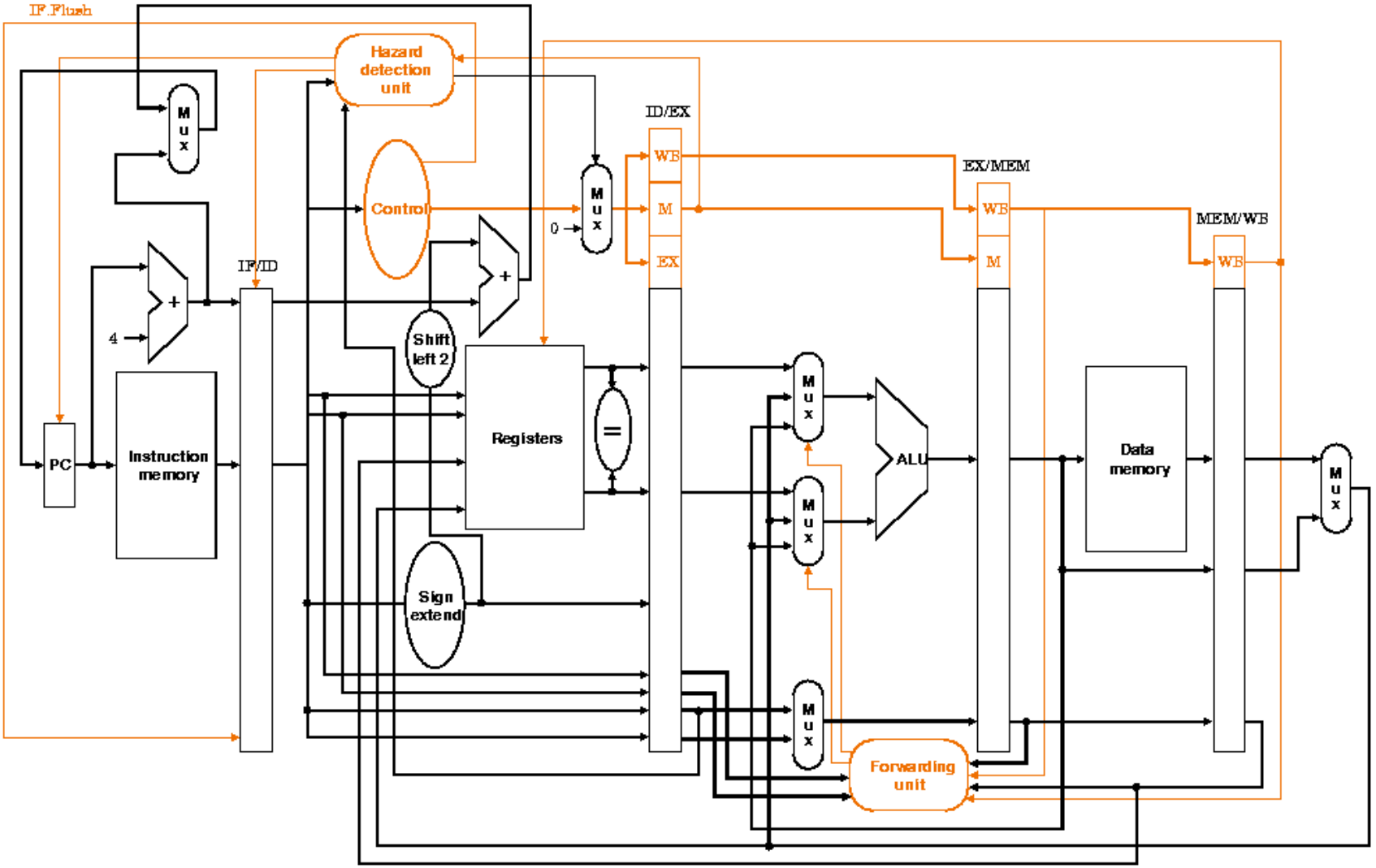

Figura 5.53 Modificaciones del Hardware para decidir sobre los brincos en la etapa ID.

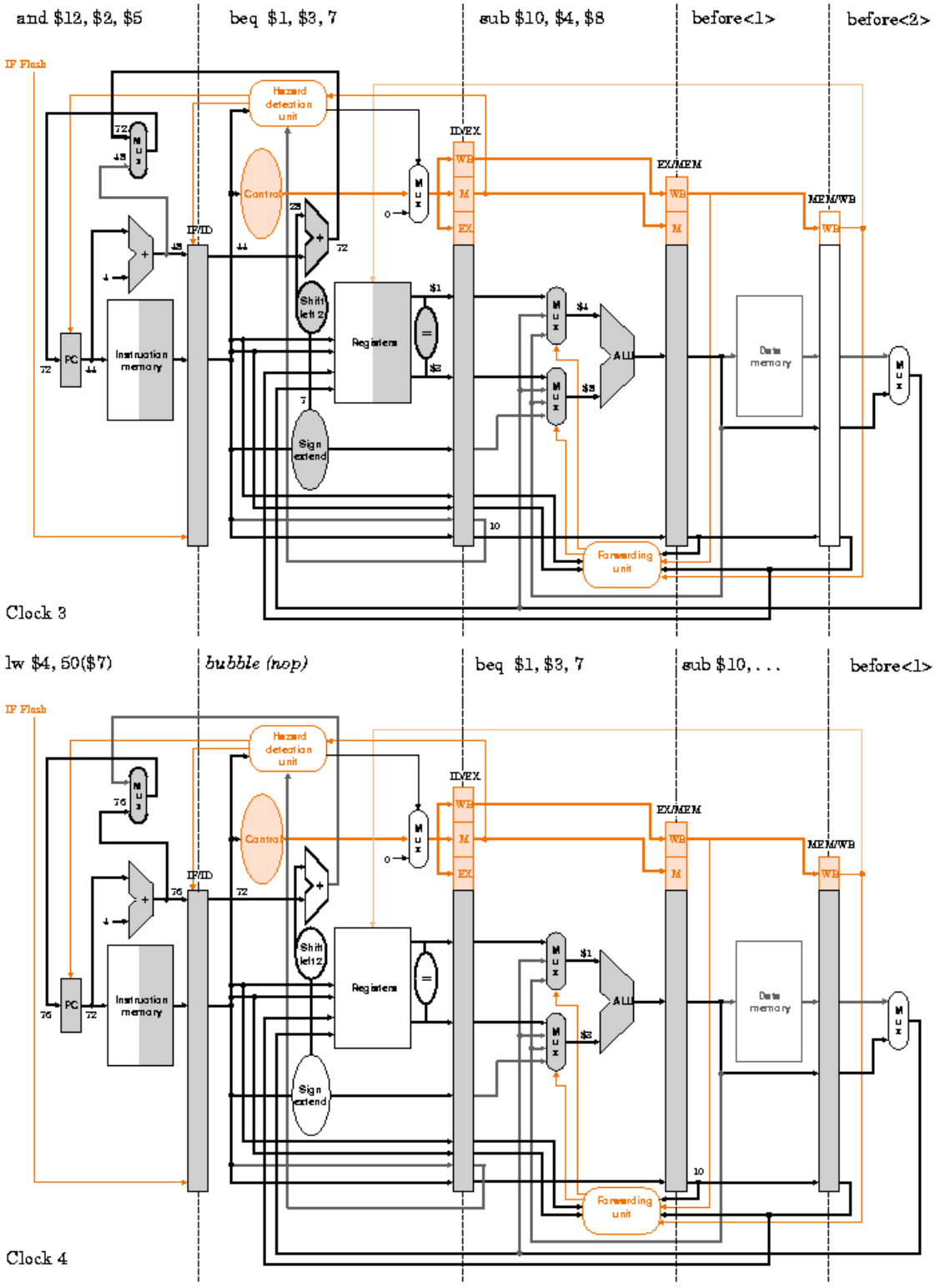

Fig. 5.54 Ciclos de reloj 3 y4 de una secuencia de código con un brinco que si se realiza.

Ejercicio: Comparación de Implementaciones.

Comparar el rendimiento para una implementación de un solo ciclo de reloj (M_{SC}), una implementación multiciclo (M_{MC}) y una implementación segmentada (M_{SEG}). Considerando un programa de prueba con la siguiente distribución de instrucciones:

Tipo	Frecuencia
Cargas	20 %
Almacenamientos	15 %
Brincos	20 %
Saltos	5 %
Aritmético - Lógicas	40 %

Los tiempos de operación para las principales unidades funcionales son: 2 ns para un acceso a memoria, 2 ns para la operación de la ALU, 1 ns para lectura o escritura de registros.

Para la implementación segmentada suponer que la mitad de las instrucciones de carga están inmediatamente seguidas por una instrucción que usa el valor obtenido de la memoria, que una cuarta parte de los brincos condicionales se realizaron produciendo un retraso de 1 ciclo de reloj y que los saltos incondicionales también se ejecutan en la etapa 2 requiriendo de 1 ciclo de reloj adicional.

Respuesta:

Para una implementación de un sólo ciclo, la duración del ciclo (T_{SC}) está determinada por el tiempo requerido para una carga (son las instrucciones más lentas):

$$T_{SC} = t_{\text{lectura de la instrucción}} + t_{\text{lectura del registro base}} + t_{\text{cálculo de la dirección a accesar}} + t_{\text{lectura de memoria}} + t_{\text{escritura en registro}} = 2 \text{ ns} + 1 \text{ ns} + 2 \text{ ns} + 2 \text{ ns} + 1 \text{ ns} = 8 \text{ ns}$$

Entonces, si son N instrucciones, el tiempo de ejecución correspondería a:

$$\text{Tiempo de ejecución}_{SC} = 8 \times N \text{ ns}$$

Para una implementación de múltiples ciclos, el tiempo del ciclo es de 2ns, porque es lo que requieren las unidades funcionales más lentas.

De manera que una carga tardaría 5 x 2 ns = 10 ns, los almacenamientos e instrucciones tipo-R requieren de 4 ciclos, por lo que tardarían 4 x 2ns = 8 ns, los brincos y saltos requieren de 3 ciclos, por lo que tardarían 3 x 2 ns = 6 ns. El tiempo de ejecución lo obtenemos con base a la tabla anterior y a la duración de cada instrucción:

$$\text{Tiempo de ejecución}_{MC} = 0.2 \times N \times 10\text{ns} + 0.15 \times N \times 8\text{ns} + 0.2 \times N \times 6\text{ns} + 0.05 \times N \times 6\text{ns} + 0.4 \times N \times 8\text{ns} = 7.9 \times N \text{ ns}$$

La implementación multiciclos es más rápida que la implementación de un sólo ciclo, aunque por muy poco:

$$\frac{\text{Rendimiento}_{\text{MC}}}{\text{Rendimiento}_{\text{SC}}} = \frac{8 \times N \times ns}{7.9 \times N \times ns} = 1.012$$

La máquina multiciclos es *1.012 veces más rápida* que la máquina de un sólo ciclo. No se observan ventajas significativas porque no se consideraron instrucciones complejas, es decir, instrucciones que requieran de muchos ciclos de reloj, y que en el caso de la implementación de un sólo ciclo aumentarían considerablemente la duración del ciclo.

Para la implementación segmentada, la duración del ciclo también es de 2 ns, sin embargo, una vez que termina la primera instrucción, se requerirá de un ciclo para concluir con la siguiente, con las siguientes excepciones:

La mitad de las instrucciones de carga (10 % de N) provocan la inserción de una burbuja, requieren de 2 ciclos. Una cuarta parte de los brincos si se realiza (5 % de N), anulando a la instrucción siguiente y todos los saltos (5% de N) producen también un retraso de un ciclo. Entonces, el tiempo de ejecución se obtiene como:

$$\begin{aligned}\text{Tiempo de ejecución}_{\text{SEG}} &= t_{\text{espera de la 1a instrucción}} + t_{\text{para culminar cada instrucción}} \\ &\quad + t_{\text{retrazos por algunos tipos de instrucciones}} \\ &= 4 \text{ x } 2\text{ns} + 2 \text{ ns x N} + 0.2 \text{ x N x } 2\text{ns} = 8\text{ns} + 2.4 \text{ x N ns}\end{aligned}$$

Entonces, para comparar que tan rápida es una implementación segmentada, respecto a una implementación multiciclos, se tiene:

$$\frac{\text{Rendimiento}_{\text{SEG}}}{\text{Rendimiento}_{\text{MC}}} = \frac{7.9 \times N \times ns}{8ns + 2.4 \times N \times ns} = \frac{7.9}{\frac{8}{N} + 2.4} = 3.29$$

Suponiendo N >> 8, lo cual es correcto para un programa real. La máquina segmentada es 3.29 *veces más rápida* que la máquina multiciclos.

Con respecto a la máquina de un sólo ciclo, tenemos:

$$\frac{\text{Rendimiento}_{\text{SEG}}}{\text{Rendimiento}_{\text{SC}}} = \frac{8 \times N \times ns}{8ns + 2.4 \times N \times ns} = \frac{8}{\frac{8}{N} + 2.4} = 3.33$$

La máquina segmentada es 3.33 *veces más rápida* que la máquina multiciclos.

Se demuestra que la implementación afecta directamente el rendimiento de un sistema. Se han revisado tres implementaciones diferentes de una misma arquitectura, y en los tres casos, se tienen rendimientos distintos.

TAREA 12

1. ¿Qué riesgos resuelve la unidad de anticipación y como los resuelve?

2. Por medio de una representación del múltiples ciclos de la segmentación, representar la ejecución de las siguientes instrucciones:

```
        Add   $2, $5, $4
        Add   $4, $2, $5
        Lw    $5, 100($2)
        Add   $3, $5, $4
        Beq   $8, $8, s1
        And   $1, $2, $3
        Or    $4, $5, $6
s1:     Sub   $7, $8, $9
```

   Mostrar las líneas por medio de las cuales se resuelven las dependencias de datos y/o mostrar la inserción de burbujas si es que se requieren.

3. Considerando los cambios hechos al HW para que los brincos puedan resolverse en la etapa 2, se tiene otra situación de riesgo: Uno de los operandos de un brinco es un resultado de una instrucción tipo R (previa al brinco):

```
        Add   $3, $4, $5
        Beq   $7, $3, etiqueta
```

   ¿Cómo puede resolverse esta situación?

   ¿Habría que hacer otra consideración si el operando en conflicto resulta de una carga?

```
        Lw    $3, 24 ( $5 )
        Beq   $7, $3, etiqueta
```

   ¿Cuántos ciclos de reloj adicionales se requieren en cada caso?