CAPÍTULO VII
SEGMENTACIÓN ENCAUZADA (PIPELINE)

7.1 INTRODUCCIÓN

El "pipeline" ó segmentación encauzada, como se conoce en español, es una técnica utilizada en el diseño e implantación de microprocesadores en la cual múltiples instrucciones pueden ejecutarse simultáneamente. La técnica de la segmentación encauzada no reduce el tiempo que tarda una instrucción en ejecutarse, sólo incrementa el número de instrucciones que se ejecutan simultáneamente; es decir, mejora el rendimiento incrementando la productividad de las instrucciones en lugar del tiempo de ejecución de las instrucciones individuales.

Es fácil entender el concepto de segmentación encauzada si pensamos en una línea de ensamble, en donde cada etapa de la línea completa una parte del trabajo total. Al igual que en la línea de ensamble, el trabajo que se realiza para cada instrucción se descompone en partes más pequeñas; cada una de estas partes, denominadas etapas ó segmentos, necesitan una fracción del tiempo total para completar la instrucción.

Entre dos etapas de la línea de ensamble se coloca un registro de acoplo, también denominado registro de segmentación, que se encarga de guardar los datos y las señales de control necesarias para etapas posteriores. Por ejemplo, el registro de acoplo situado entre las etapas 2 y 3 de la figura 7.1 guarda los datos y las señales de control generadas en la etapa 2 para su uso posterior en la etapa 3. Gracias a los registros de acoplo es posible manejar múltiples instrucciones al mismo tiempo.

Figura 7.1. Etapas o segmentos.

De la figura 7.2 se observa que la ejecución de una instrucción consta de cuatro etapas o módulos: 1) traer la instrucción, 2) decodificarla, 3) traer operandos, y 4) ejecutarla. Si cada uno de estos módulos es independiente de los demás, entonces, en el tiempo '*n*' se empezaría a traer la instrucción I_n; al mismo tiempo se estaría decodificando la instrucción I_{n-1}, trayendo los operandos de la instrucción I_{n-2} y terminando de ejecutar la instrucción I_{n-3}. Esto significa que una vez que el cauce está lleno, idealmente, en cada ciclo de reloj se estaría ejecutando una instrucción.

Un aspecto de suma importancia en la segmentación encauzada es la duración del ciclo de reloj. Este reloj es el encargado de sincronizar todas las etapas de la segmentación, por lo tanto, debe ser lo suficientemente grande para acomodar las operaciones de la etapa más lenta.

Idealmente, la mejora de velocidad debido a la segmentación encauzada es igual al número de etapas, esto es, una arquitectura segmentada de cuatro etapas es cuatro veces más rápida que una arquitectura sin esta tecnología. Sin embargo, esta velocidad no se alcanza en la realidad ya que en algunas ocasiones el cauce tiene que ser llenado de nuevo, como por ejemplo, cuando ocurre un salto; en este caso, la secuencia de instrucciones que estaba dentro del cauce tiene que ser eliminada

para comenzar a ejecutar nuevas instrucciones a partir de la dirección de salto. Adicionalmente, debido a que las etapas están equilibradas imperfectamente, el tiempo por instrucción en una arquitectura segmentada no tiene el valor mínimo posible y la mejora en velocidad será menor que el número de etapas.

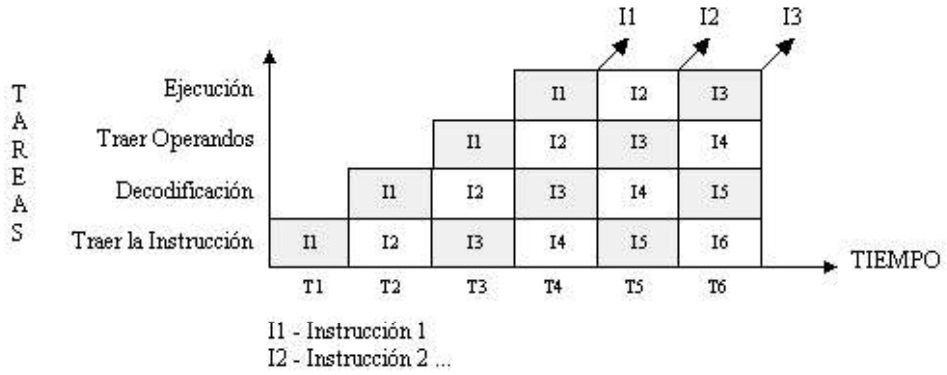


Figura 7.2. Diagrama de Tiempos / Tareas.

7.2 LA ARQUITECTURA SEGMENTADA DEL 68HC11

Si revisamos detenidamente el repertorio de instrucciones del 68HC11 notaremos que cada instrucción ejecuta una serie de pasos. En general, los pasos a seguir son los siguientes.

1. Traer de la memoria la instrucción que se desea ejecutar (a este paso se le conoce como ciclo fetch ó búsqueda de la instrucción)
2. Decodificación de la instrucción
3. Si la instrucción requiere leer un operando de la memoria, entonces se calcula la dirección efectiva de ese operando y se lee el dato de la memoria
4. Si lo requiere la instrucción, se leen de los registros internos del microprocesador los operandos necesarios
5. Ejecución, es decir, se realiza una operación en la unidad de procesos aritméticos con los operandos leídos anteriormente
6. Se guardan los resultados de la operación y se actualiza el registro de banderas

Observe que estos pasos son similares a los ejecutados en las cartas ASM para las instrucciones vistas en el capítulo VI. La arquitectura segmentada del 68HC11 también ejecutará estos mismos pasos, pero agrupados en las siguientes cuatro etapas.

1. Etapa IF (traer la instrucción / instruction fetch). La instrucción a ejecutar es leída de la memoria de instrucciones
2. Etapa ID (decodificación / instruction decode). Se decodifica la instrucción y se traen los operandos necesarios por la instrucción (tanto de memoria como de registros internos)
3. Etapa EX (ejecución / execution). Se procesan los operandos en la UPA (unidad de procesos aritméticos)
4. Etapa WB (post-escritura / write back). Se guardan resultados

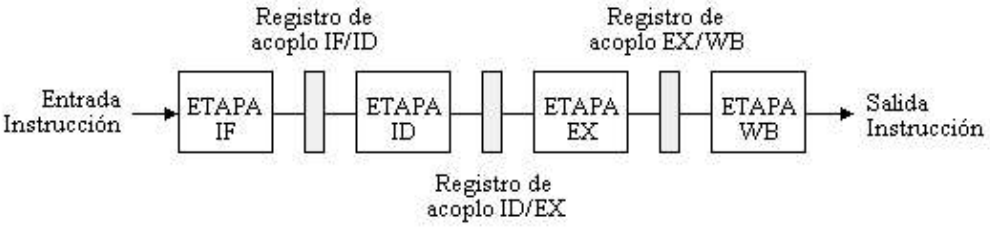


Figura 7.3. Etapas propuestas para la arquitectura segmentada del 68HC11.

A continuación se describen las tareas que se ejecutan en cada etapa y se presenta un diagrama de bloques con el hardware necesario para su implantación. Hay que tener presente que esta arquitectura no soporta todo el conjunto de instrucciones del 68HC11; algunas instrucciones no son soportadas y otras requieren redefinir el formato de la instrucción, ó agregar hardware adicional, para su posible implantación.

7.2.1 ETAPA 1 - LECTURA DE LA INSTRUCCIÓN

El primer paso que realiza todo microprocesador es leer de memoria la siguiente instrucción a ejecutar. Recordemos que en la arquitectura secuencial del 68HC11, descrita en el capítulo VI, el primer paso que se realiza para cada instrucción es su ciclo fetch, es decir, traer de memoria la instrucción a ejecutar. Enseguida, si la instrucción lo requería, también eran leídos de memoria los datos y/o las direcciones en memoria de los datos. Este proceso necesitaba acceder a memoria cierto número de veces, según la instrucción que se tratase.

Para una arquitectura segmentada, acceder tantas veces a la memoria complica el hardware y retrasa el comienzo de la siguiente instrucción a ejecutar. No olvide que la segmentación encauzada recomienda que el flujo de datos sea siempre hacia adelante, es decir, que se avance hacia etapas posteriores en el cauce; sin embargo, habrá etapas, como la de post-escritura, en la que se necesitará retroceder en el cauce.

Una manera de evitar los accesos a memoria repetidamente es leyendo en una sola pasada toda la información que la instrucción vaya a necesitar, esto es, leer el código de operación de la instrucción, leer los datos inmediatos y leer las direcciones de memoria. Para ello, el tamaño de la instrucción para el 68HC11 segmentado ha sido extendido a 32 bits, los cuales contendrán toda la información necesaria según el modo de direccionamiento del que se trate. Además, la memoria externa será separada en memoria de instrucciones o programa y en memoria de datos.

Como resultado del aumento en el tamaño de la instrucción, el formato de ésta se ha modificado de la siguiente manera.

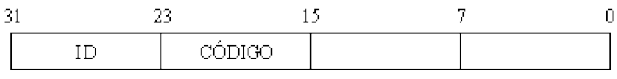


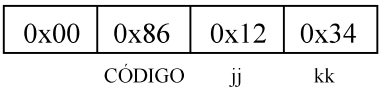
Figura 7.4. Formato general para las instrucciones del 68HC11 segmentado.

El campo ID, bits 31 al 24, sirven para especificar sobre qué registro índice se va a operar, es decir, se trata del registro índice IX ó del registro IY. El campo CÓDIGO, bits 23 al 16, guardan el código de operación de la instrucción, tal y como lo establece el conjunto de instrucciones del 68HC11. Y los bits 15 al 0 pueden almacenar el valor de un dato inmediato, una dirección, un desplazamiento, ó nada, según el modo de direccionamiento que se trate. Los formatos válidos para los bits 15 al 0 se muestran en la siguiente tabla.

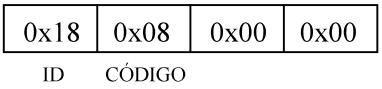
Bits 15-8	*Bits 7-0*	*Descripción*
hh	ll	Dirección de 16 bits
00	dd	Dirección de 8 bits
jj	kk	Dato inmediato de 16 bits
00	ii	Dato inmediato de 8 bits
00	ff	Desplazamiento de 8 bits sin signo
nn	mm	Máscara de 16 bits
uu	vv	Desplazamiento de 16 bits con signo
00	rr	Desplazamiento de 8 bits con signo

Tabla 7.1. Contenido para los 16 bits menos significativos del formato de instrucciones.

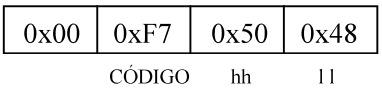
A continuación se analizan algunos ejemplos para dejar en claro el formato de instrucciones.



El campo código, 0x86, nos indica que la instrucción a ejecutar es *ldaa* con modo de direccionamiento inmediato. El dato inmediato, 0x1234, se encuentra guardado en los 16 bits menos significativos del formato de la instrucción.



El campo código, 0x08, junto con el campo ID, 0x18, nos indican que la instrucción a ejecutar es *iny*. Como *iny* utiliza el modo de direccionamiento inherente, el resto de los campos no son relevantes.



El campo código, 0xF7, nos indica que la instrucción a ejecutar es *stab* con modo de direccionamiento extendido. La dirección extendida, 0x5048, se encuentra guardada en los 16 bits menos significativos del formato de la instrucción.

La siguiente figura muestra los formatos de instrucción genéricos para los modos de direccionamiento directo y extendido.

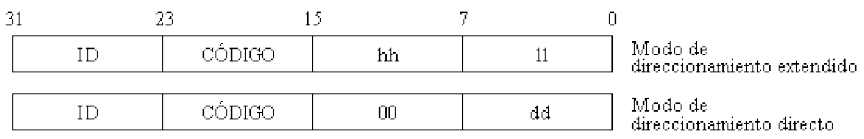


Figura 7.5. Formatos para los modos de direccionamiento directo y extendido.

Note que en ambos casos se están ocupando los 32 bits del formato de la instrucción, por lo tanto, resulta poco efectivo mantener ambos formatos, ya que con el modo extendido es posible manejar el modo directo. Algo similar ocurre con el formato de los desplazamientos con signo, y con el formato de los datos inmediatos de 8 y 16 bits. Aún con el conocimiento de que hay formatos repetidos ó innecesarios, éstos serán mantenidos con el fin de adaptar la nueva arquitectura al conjunto de instrucciones que ya habíamos manejado en el capítulo VI.

Finalmente, el hardware para la etapa de la lectura de la instrucción queda de la siguiente manera.

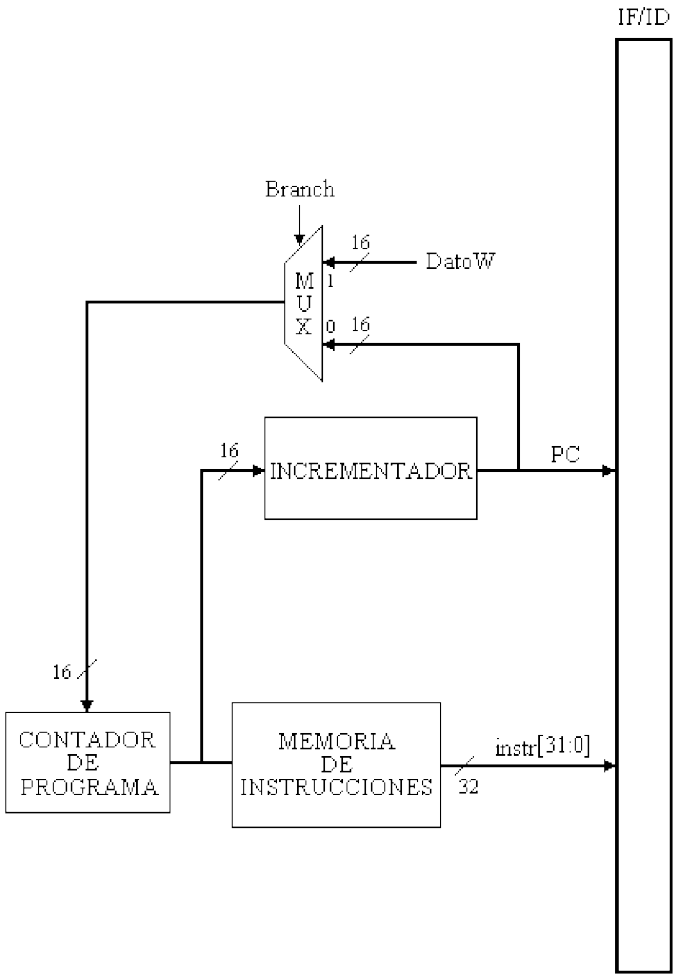


Figura 7.6. Hardware para la etapa IF - Lectura de la instrucción.

En esta etapa se lee de la memoria la instrucción a ejecutar y se almacena en el registro de segmentación IF/ID. La dirección de esta instrucción está dada por el contador de programa (PC); dicha dirección se incrementa y se vuelve a cargar en el PC para ser leída en el siguiente ciclo de reloj. La dirección incrementada también se guarda en el registro de segmentación IF/ID, pues es posible que la necesite otra instrucción posteriormente, por ejemplo, la instrucción *bra*.

7.2.2 ETAPA 2 - DECODIFICACIÓN DE LA INSTRUCCIÓN / CÁLCULO DE LA DIRECCIÓN EFECTIVA / LECTURA DE OPERANDOS

Durante esta etapa, la instrucción leída en la etapa anterior es decodificada. Una vez que se conoce la instrucción a ejecutar, el módulo de control genera las señales de control que gobernarán el hardware de esta etapa y de etapas posteriores. Por ejemplo, las señales de control que se emplean en la segunda etapa se encargan de informarle a los distintos componentes de esta etapa qué operaciones realizar. Estas operaciones consisten en lectura de operandos de registros, cálculo de direcciones, lectura de operandos de la memoria de datos, principalmente.

Obsérvese que la memoria de instrucciones o de programa ha sido separada de la memoria de datos, pues de estar unidas, la continúa lectura de operandos y escritura de resultados demoraría la lectura de la siguiente instrucción a ejecutar. El hardware para esta etapa se muestra en la figura 7.7.

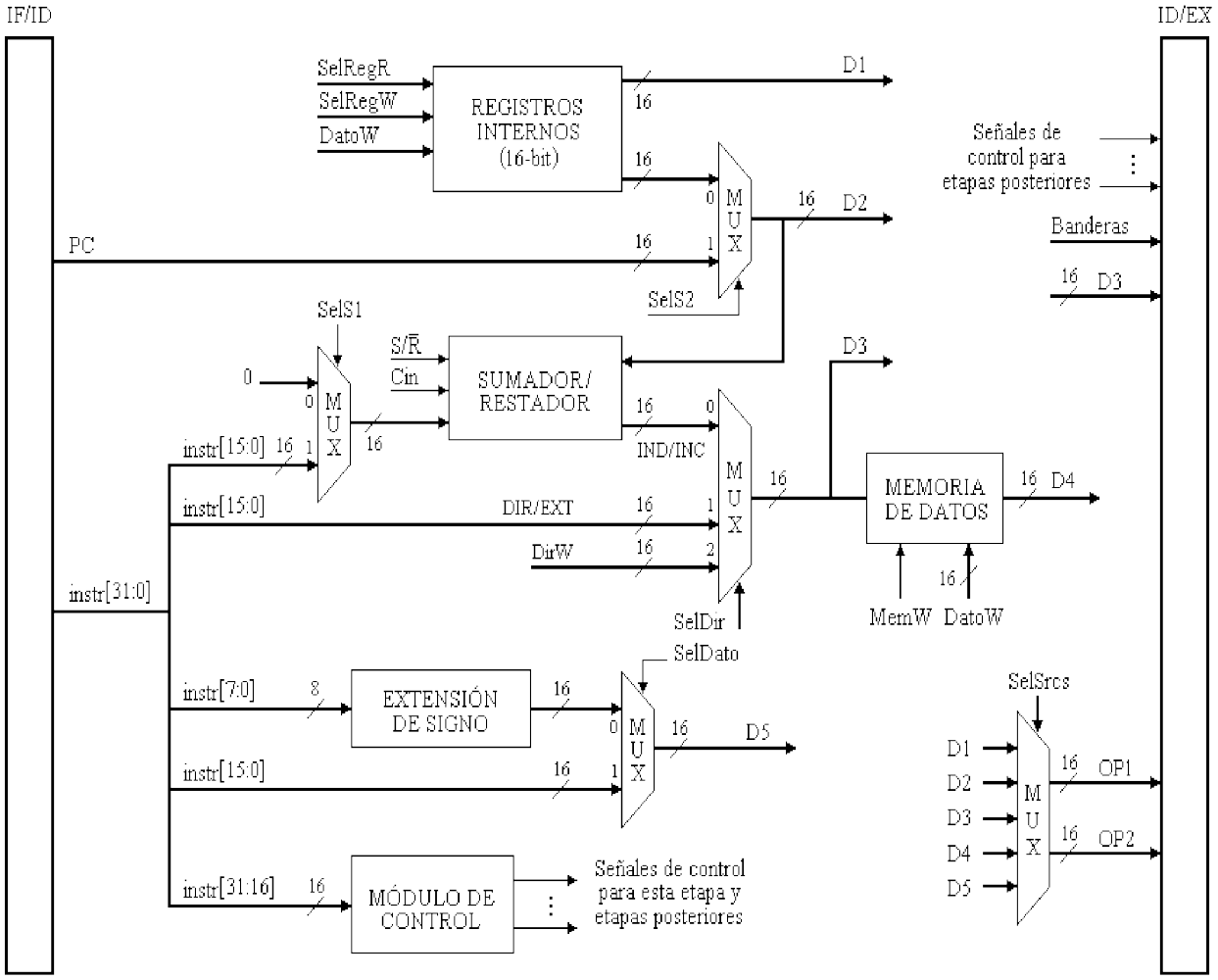


Figura 7.7. Hardware para la etapa ID - Decodificación de la instrucción y lectura de operandos.

El módulo de control de la figura 7.7 es el encargado de generar todas las señales de control. Algunas de estas señales serán utilizadas durante esta etapa y otras serán guardadas en los registros de segmentación para su utilización en etapas posteriores. La generación de las señales de control es posible gracias a la información que le brindan al módulo las líneas *instr*[31:16], pues estas líneas transportan el código de operación de la instrucción con la información suficiente para saber de qué instrucción se trata y su modo de direccionamiento.

Por otra parte, en el módulo de registros internos se encuentran los acumuladores A y B, y los registros internos IX, IY, SP y AUX. Todos estos registros son de 16 bits de tamaño. El siguiente diagrama muestra la disposición de todos ellos dentro del módulo.

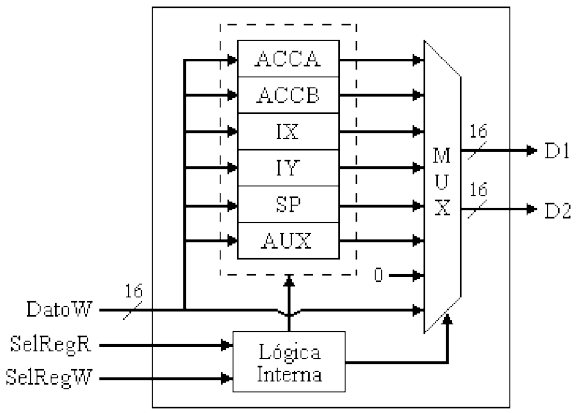


Figura 7.8. Módulo de registros internos.

La línea de control *SelRegR* le indica al módulo qué registros deseamos leer; la línea *SelRegW* le indica en qué registro se desea escribir el valor de *DatoW*; y las salidas *D1* y *D2* transportan los valores leídos de dichos registros. Posteriormente se describirá el funcionamiento de la lógica interna de este módulo, la cual nos permitirá adelantar datos, y evitar lecturas y escrituras al mismo tiempo en el mismo registro. A continuación se muestran las tablas para la escritura y lectura de los registros internos.

SelRegW	*Registro que se escribe*
0	Ninguno
1	ACCA
2	IX
3	IY
4	ACCB
5	AUX
6	SP

Tabla 7.2. Selección de los registros para escritura.

SelRegR	*Registro seleccionado para lectura*	
	D1	D2
0	0	0
1	ACCA	ACCB
2	ACCB	IX
3	ACCB	IY
4	ACCA	0
5	ACCB	0
6	ACCA	IX
7	ACCA	IY
8	AUX	0
9	0	IX
A	0	IY
B	0	SP
C	ACCA	SP
D	ACCB	SP
E	IX	SP
F	IY	SP

Tabla 7.3. Selección de los registros para lectura.

El módulo sumador/restador de esta etapa se utiliza para calcular incrementos, decrementos, y sobre todo, para el cálculo de la dirección efectiva en instrucciones con modo de direccionamiento indexado. Más adelante, se verá un ejemplo de este tipo de instrucción.

Las señales de control para el módulo sumador/restador son las siguientes: $S/\overline{R}$, permite seleccionar la operación a ejecutar, una suma (si vale uno) y una resta (si vale cero); y *Cin*, que es el acarreo de entrada al sumador. Note que este módulo obtiene sus operandos de dos multiplexores. El primer multiplexor utiliza la señal de control *SelS1*; si *SelS1* vale uno, entonces se elige el bus *instr*[15:0], pero si vale cero, entonces se elige el valor de cero. El segundo multiplexor utiliza la señal de control *SelS2*; si *SelS2* vale uno, entonces se elige el valor de *PC*, y si vale cero, se elige el contenido de alguno de los registros internos (*D2*).

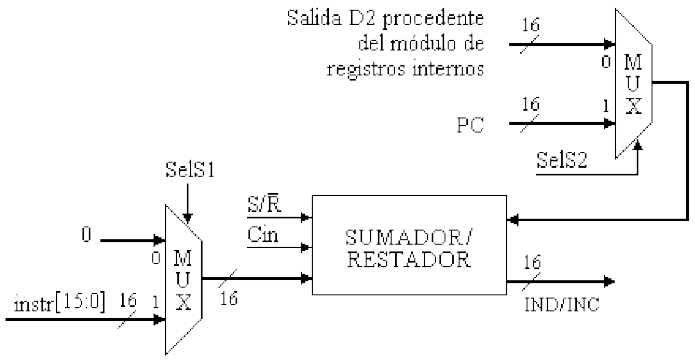


Figura 7.9. Módulo sumador / restador y sus señales de control.

En esta etapa también se encuentra la memoria de datos la cual es totalmente independiente de la memoria de instrucciones, de ella leeremos operandos, o bien, almacenaremos los resultados de nuestras operaciones, según sea el caso.

La única señal de control que se utiliza en la memoria es *MemW*, la cual indica si se realiza una operación de escritura o de lectura en ella. Si *MemW* vale uno la operación a efectuar será escritura, y si vale cero la operación será lectura. La dirección del dato a leer o escribir en memoria proviene de un multiplexor, esta dirección se selecciona con la línea de control *SelDir*. Cuando *SelDir*=0 la dirección que se elige proviene del módulo sumador/restador; si *SelDir*=1 proviene del bus *instr*[15:0]; y si *SelDir*=2 la dirección que se toma será *DirW*.

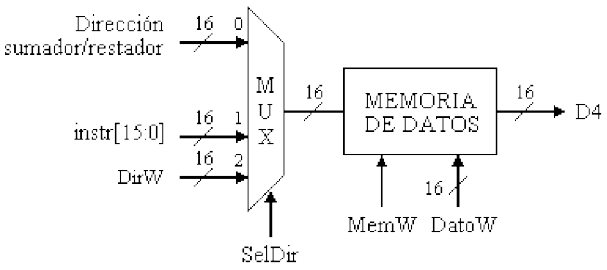


Figura 7.10. Memoria de datos y sus señales de control.

Por otra parte, las instrucciones de salto o de acceso relativo utilizan un desplazamiento de 8 bits con signo para calcular una dirección de salto. El módulo encargado de calcular esta dirección de salto es la UPA (unidad de procesos aritméticos) ubicada en la siguiente etapa. Sin embargo, como se verá más adelante, la UPA sólo ejecuta operaciones de 16 bits, por lo que el desplazamiento de 8 bits con signo necesitará ser extendido a 16 bits. El módulo encargado de esta tarea es el módulo de extensión de signo, quien tomará los 8 bits del desplazamiento con signo y los extenderá a 16 bits, repitiendo en los 8 bits más significativos el bit de signo del desplazamiento original, es decir,

Desplazamiento original $\Rightarrow$ Desplazamiento extendido

$$\underline{0}\,1\,1\,0\,0\,0\,1\,1 \Rightarrow 0\,0\,0\,0\,0\,0\,0\,\underline{0}\,0\,1\,1\,0\,0\,0\,1\,1$$
$$\underline{1}\,0\,1\,1\,1\,1\,0\,0 \Rightarrow 1\,1\,1\,1\,1\,1\,1\,\underline{1}\,1\,0\,1\,1\,1\,1\,0\,0$$

Finalmente, existe un multiplexor que selecciona los operandos para la siguiente etapa. Este multiplexor elige estos operandos de los datos presentes en los buses *D1*, *D2*, *D3*, *D4* y *D5*, según la instrucción que se trate. La selección de estos operandos se muestra en la siguiente tabla.

SelSrcs	*Operandos seleccionados*	
	OP1	OP2
0	0	0
1	D1	D2
2	D1	D4
3	D1	D5

4	D4	D3
5	D2	D5
6	D2	D4

Tabla 7.4. Selección de los operandos para la etapa de ejecución.

Los operandos que se seleccionaron utilizando el multiplexor anterior, la dirección efectiva y las señales de control para las etapas posteriores son guardados temporalmente en el registro de segmentación ID/EX.

En resumen, esta segunda etapa se encarga de realizar tres tareas: 1) la decodificación de la instrucción y la generación de las señales de control; 2) el cálculo de direcciones efectivas para datos en memoria; y 3) la lectura de operandos.

La decodificación es realizada por el módulo de control el cual revisa el código de operación de la instrucción (los dos bytes más significativos del formato de la instrucción), y con base en él, genera las señales de control necesarias para la etapa actual y para las etapas posteriores.

La dirección efectiva es una dirección en memoria de donde se lee un dato, o bien, una dirección en memoria en donde se guarda un dato. Para algunos modos de direccionamiento esta dirección no es inmediata. Por ejemplo, el modo de direccionamiento indexado, calcula la dirección efectiva sumando al contenido de un registro base un desplazamiento; en cambio, los modos de direccionamiento directo y extendido proporcionan la dirección efectiva de forma inmediata.

La lectura de operandos consiste en obtener los datos que se operarán en la siguiente etapa. Los operandos pueden provenir de los registros internos, de la memoria de datos, o bien, pueden estar contenidos en el mismo formato de la instrucción. Para obtener el contenido de algunos de los registros basta con indicar al módulo de registros internos qué registros se desean leer. Para obtener el operando de la memoria de datos es necesario contar con la dirección efectiva donde se encuentra ese dato. Y si el operando está contenido en el formato de la instrucción, como es el caso del direccionamiento inmediato, la línea de control *SelDato*, de uno de los multiplexores, permitirá su selección.

7.2.3 ETAPA 3 - EJECUCIÓN / CÁLCULO DE BANDERAS Y SALTOS

Esta etapa ejecuta tres tareas: 1) opera los operandos obtenidos en la etapa de decodificación (etapa 2); 2) actualiza el registro de estados o banderas; y 3) calcula la condición de salto.

El hardware para esta etapa se muestra en la figura 7.11.

La unidad de procesos aritméticos (UPA) se encarga de obtener el resultado entre los operandos según la operación establecida en *SelOp*. *SelOp* es una señal de control generada en la etapa anterior, pero que es empleada hasta esta etapa. En la UPA se realizan las operaciones lógicas, las operaciones aritméticas y los corrimientos, tal y como se muestra en la tabla 7.5.

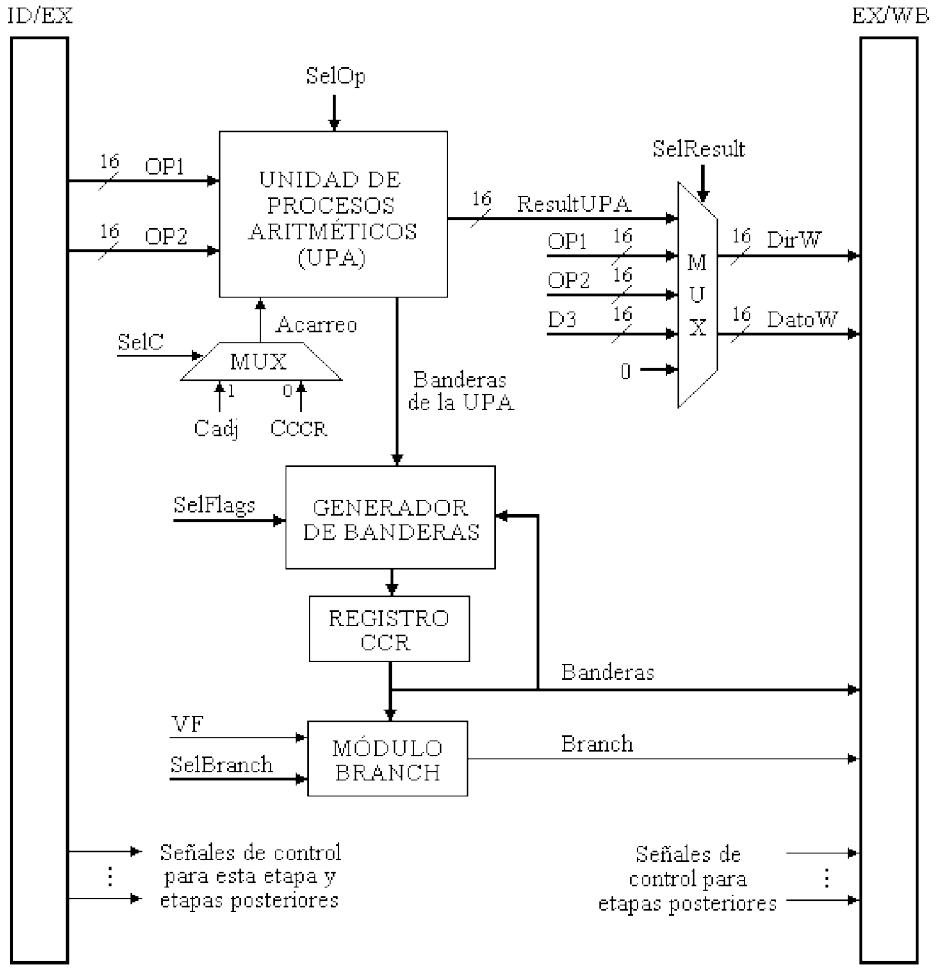


Figura 7.11. Hardware para la etapa EX - Ejecución.

SelOp	*Operación ejecutada*
0	Ninguna
1	OP1 + OP2 + Acarreo
2	OP1 – OP2 – $\overline{\text{Acarreo}}$
3	OP1 and OP2
4	OP1 or OP2
5	OP1 xor OP2
6	Corrimiento a la izquierda de OP1 con $B_0 = 0$
7	Corrimiento a la derecha de OP1 con $B_{15} = B_{15}$
8	OP2 – OP1 – $\overline{\text{Acarreo}}$
9	Corrimiento a la derecha de OP1 con $B_{15} = 0$
A	Rotación a la izquierda de OP1 con $B_0 = C_{CCR}$
B	Rotación a la derecha de OP1 con $B_{15} = C_{CCR}$

Tabla 7.5. Operaciones de la UPA.

El acarreo de entrada al módulo UPA es seleccionado mediante la señal de control *SelC*. Si *SelC* vale cero, el acarreo elegido proviene del registro de estados (C_{CCR}); si *SelC* vale uno, entonces el acarreo elegido es C_{adj}, el cual es generado por el módulo de control de la etapa 2 y es establecido a un cierto valor según la instrucción que se trate.

Las banderas que se modificaron tras la operación ejecutada en la UPA son guardadas en el registro de estados (CCR, Condition Code Register) por el módulo generador de banderas. La tabla 7.6 muestra la relación entre la señal de control *SelFlags* y las banderas que son actualizadas en el registro de estados por el módulo generador de banderas.

SelFlags	*Banderas que son actualizadas en el CCR*
0	No se modifica el CCR
1	N, Z, V=0
2	N, Z, V, C, H
3	N, Z, V, C
4	Z
5	C=0
6	I=0
7	V=0
8	C=1
9	I=1
A	V=1
B	N, Z, V=0, C=1
C	N, Z, V

Tabla 7.6. Banderas afectadas en CCR por el módulo generador de banderas.

En caso de ejecutar una instrucción de salto, la UPA calculará la dirección a donde probablemente se deba saltar, y el módulo Branch evaluará la condición de salto para determinar si en verdad se ejecuta el salto o no. Recuerde que las señales de control, *VF* y *SelBranch*, empleadas en el módulo Branch fueron generadas en la etapa anterior.

El módulo Branch genera una señal de salida denominada 'Branch'. Esta señal, generada a partir de la condición de salto y del valor de *VF*, nos permite saber si el salto se realiza o no. Para ello, la condición de salto debe ser evaluada; si el resultado de esta evaluación es igual al valor de *VF*, entonces la señal 'Branch' toma el valor de uno, si no, 'Branch' toma el valor de cero. La tabla 7.7 muestra la relación entre la señal de control *SelBranch* y la condición de salto que se evalúa.

SelBranch	*Condición a evaluar*
0	Se compara con cero
1	C
2	Z
3	$N \oplus V$
4	$Z + (N \oplus V)$

5	C + Z
6	N
7	V

Tabla 7.7. Condiciones de salto para el módulo Branch.

Por último, son guardados en el registro de segmentación EX/WB el resultado de la UPA, la dirección efectiva obtenida en la etapa 2, algunos valores de banderas, y las señales de control necesarias para la última etapa. A continuación se presenta la relación de la señal *SelResult* con las fuentes seleccionadas hacia el registro de segmentación EX/WB.

SelResult	*Fuentes seleccionadas*	
	DatoW	DirW
0	0	0
1	ResultUPA	D3
3	OP1	D3

Tabla 7.8. Fuentes seleccionadas por la señal de control SelResult.

7.2.4 ETAPA 4 – POST-ESCRITURA

En las arquitecturas segmentadas las instrucciones y los datos se desplazan generalmente de izquierda a derecha a través de las etapas, sin embargo, hay dos excepciones que se presentan en la etapa de post-escritura:

1. Cuando se guarda el resultado de la UPA en los registros o en memoria, haciendo que se retroceda a la etapa 2.
2. Cuando se selecciona el nuevo valor de PC en la etapa 1, valor que puede ser el PC incrementado, o bien, la dirección de salto calculada en la etapa 3.

En resumen, la etapa de post-escritura se encarga de actualizar los resultados obtenidos en etapas anteriores. Recuerde que las señales de control utilizadas en esta última etapa fueron generadas en la etapa 2 y se propagaron a través de los registros de segmentación hasta la etapa requerida.

La arquitectura completa del 68HC11 segmentado se muestra en la figura 7.12.

De la figura 7.12 notará que la señal de control *SelDir* se utiliza en dos etapas de la arquitectura, en la etapa 2 y en la etapa 4. Durante la etapa 2, la señal *SelDir* seleccionará la dirección de memoria de donde se leerá un dato. Esta dirección puede venir de dos buses, del bus IND/INC (si la instrucción utiliza acceso indexado), o del bus DIR/EXT (si la instrucción utiliza acceso directo o extendido). En cambio, en la etapa 4, la señal *SelDir* seleccionará la dirección en memoria en donde se guardará algún resultado, si es que la instrucción así lo especifica. Esta dirección provendrá exclusivamente del bus DirW.

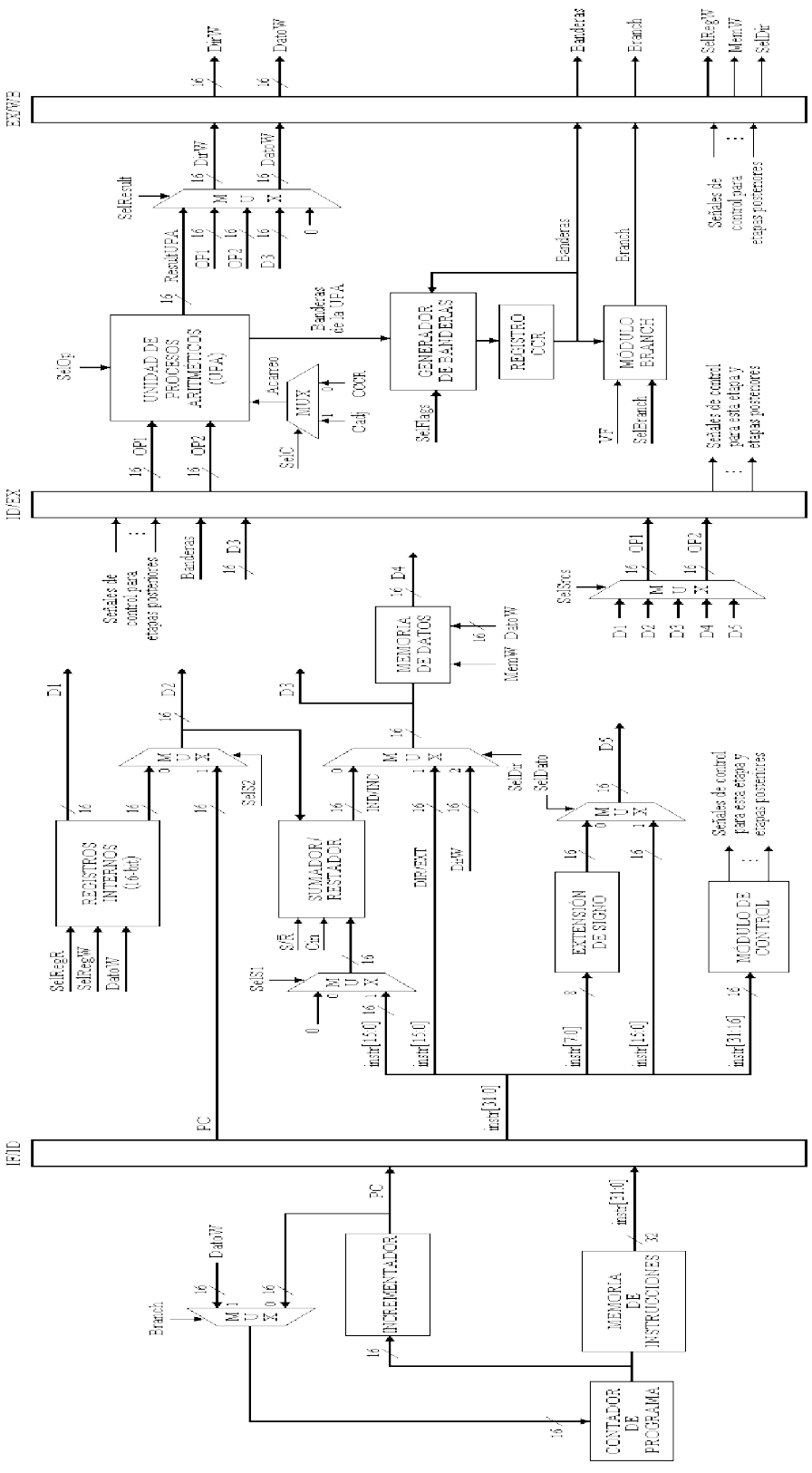


Figura 7.12. Arquitectura del 68HC11 utilizando la técnica de la segmentación encauzada.

La unidad de control ubicada en la etapa 2 es la encargada de generar los valores de la señal *SelDir* de acuerdo a la instrucción que se ejecuta. De esta manera, serán generados dos valores para la señal *SelDir*, el primer valor se utilizará durante la etapa 2, y el segundo valor será guardado en el registro de segmentación ID/EX junto con las señales de control para las etapas siguientes, y será utilizado hasta la etapa 4.

7.2.5 REPRESENTACIÓN GRÁFICA DE LA SEGMENTACIÓN ENCAUZADA

La segmentación encauzada puede ser difícil de comprender, ya que múltiples instrucciones se ejecutan simultáneamente en un sólo camino de datos en cada ciclo de reloj. Para ayudar a comprenderlo hay dos estilos básicos de diagramas: los diagramas de múltiples ciclos de reloj y los diagramas de un sólo ciclo de reloj.

En los diagramas de múltiples ciclos de reloj el tiempo avanza de izquierda a derecha en el diagrama, y las instrucciones avanzan de arriba hacia abajo. Utilizamos estos diagramas para dar visiones generales de las situaciones en la segmentación.

Los diagramas de un sólo ciclo de reloj muestran el estado del camino de datos durante un sólo ciclo de reloj. Generalmente todas las instrucciones de la segmentación encauzada se identifican por rótulos debajo de sus respectivas etapas. Estos diagramas se utilizan para mostrar detalladamente lo que ocurre en la segmentación durante cada ciclo de reloj.

7.3 CONJUNTO DE INSTRUCCIONES

En esta sección se analizará el comportamiento de algunas de las instrucciones del 68HC11 en cada una de las etapas de la segmentación. Para ilustrar con detalle el progreso de estas instrucciones utilizaremos los diagramas de un sólo ciclo de reloj.

7.3.1 INSTRUCCIÓN LDAA (Acceso Inmediato)

Instrucción: LDAA #Dato_16Bits
Operación: ACCA ⇐ (Memoria)
Código: 0086
Descripción: Carga en el registro ACCA un dato inmediato de 16 bits contenido en memoria.
Banderas: N=1 si el MSB[9] del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

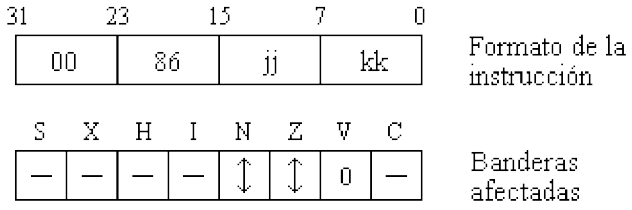


Figura 7.13. Formato de la instrucción LDAA y banderas que afecta.

El comportamiento de la instrucción *ldaa* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *ldaa*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.14).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

El modo de direccionamiento para esta instrucción es inmediato, es decir, el mismo formato de la instrucción, guardado en el registro de segmentación IF/ID, contiene el dato a cargar en el registro

[9] MSB = Bit más significativo.

ACCA; por lo tanto, no es necesario calcular ninguna dirección efectiva. Sin embargo, observe que la carga del dato inmediato en el registro ACCA no puede hacerse en la etapa 2, ya que *DatoW* que es el bus de datos con los resultados obtenidos proviene de la etapa 4. Así que deberemos esperar hasta la etapa 4 para hacer la escritura del dato inmediato en el registro ACCA.

Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	0
	SelS1	0
	$S/\overline{R}$	1
	Cin	0
	SelS2	0
	SelDato	1
	SelScrs	3
	SelDir	0
Etapa 3	SelOp	4
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	1
	SelBranch	0
	VF	1
Etapa 4	SelRegW	1
	MemW	0
	SelDir	0

Tabla 7.9. Señales de control para la instrucción LDAA.

Por el momento sólo analizaremos las señales correspondientes a la etapa 2, el resto de ellas serán analizadas en su respectiva etapa. La señal SelRegR=0 le informa al módulo de registros internos qué registros leer, en este caso, se leen valores de cero. Por otra parte, la señal $S/\overline{R}$ =1 le indica al módulo sumador/restador que realice la suma entre los operandos seleccionados con SelS2=0, SelS1=0, y el acarreo de entrada Cin=0. La señal SelS2=0 elige el segundo dato que genera el módulo de registros y la señal SelS1=0 elige el valor de cero.

La señal SelDato=1 selecciona el dato inmediato proveniente del formato de la instrucción y lo asigna al bus D5. Con SelScrs=3 se guarda el dato inmediato y el dato contenido en el bus D1 en el registro de segmentación ID/EX. Finalmente, la señal SelDir=0 selecciona el resultado obtenido por el módulo sumador/restador; este valor también se guarda en el registro de segmentación. Observe que para esta instrucción la dirección efectiva contenida en el bus D3 no se utiliza. Las señales de control para las etapas 3 y 4 también son guardadas en ID/EX (véase la figura 7.15).

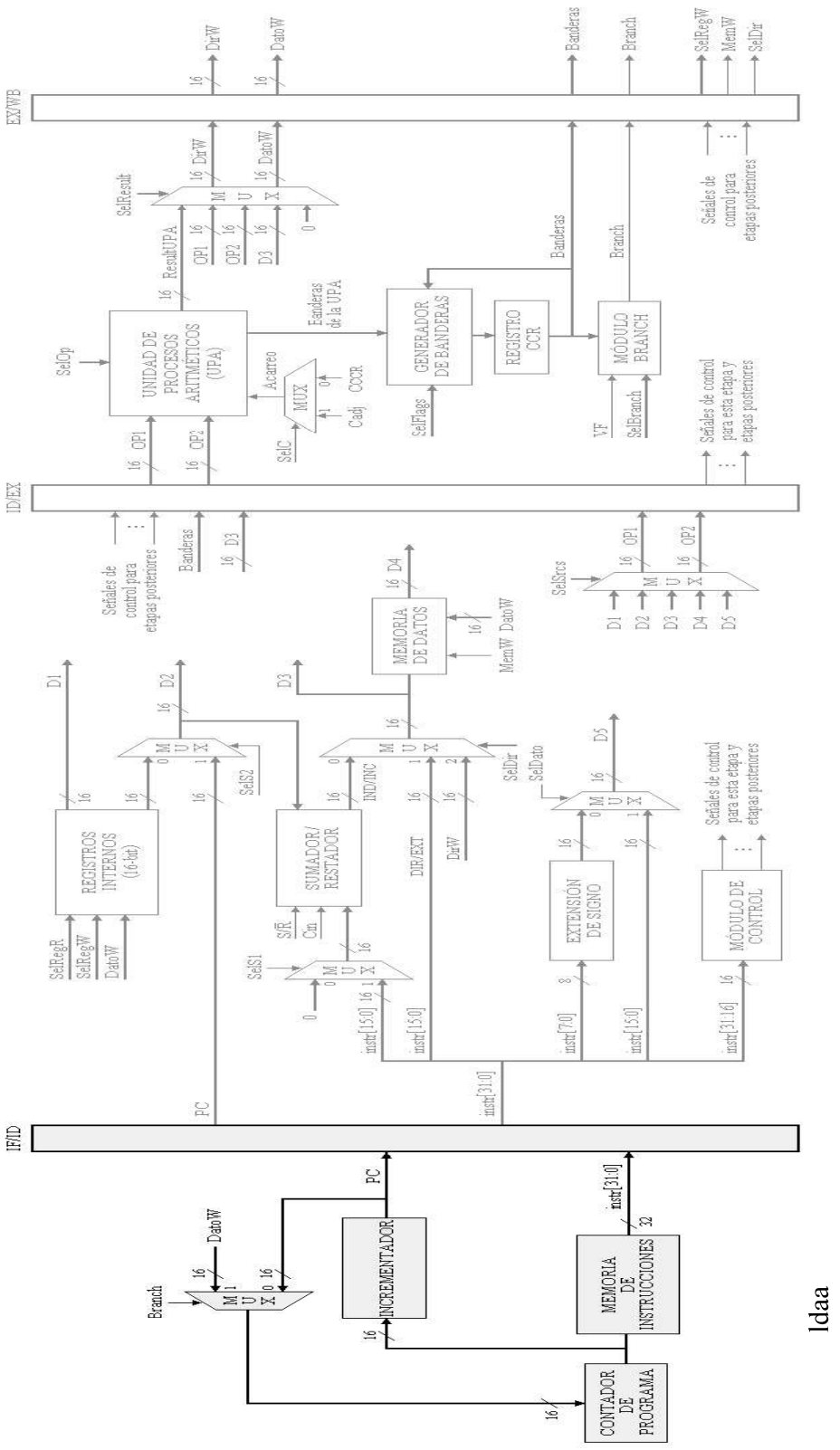


Figura 7.14. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción LDAA es leída de la memoria de instrucciones.

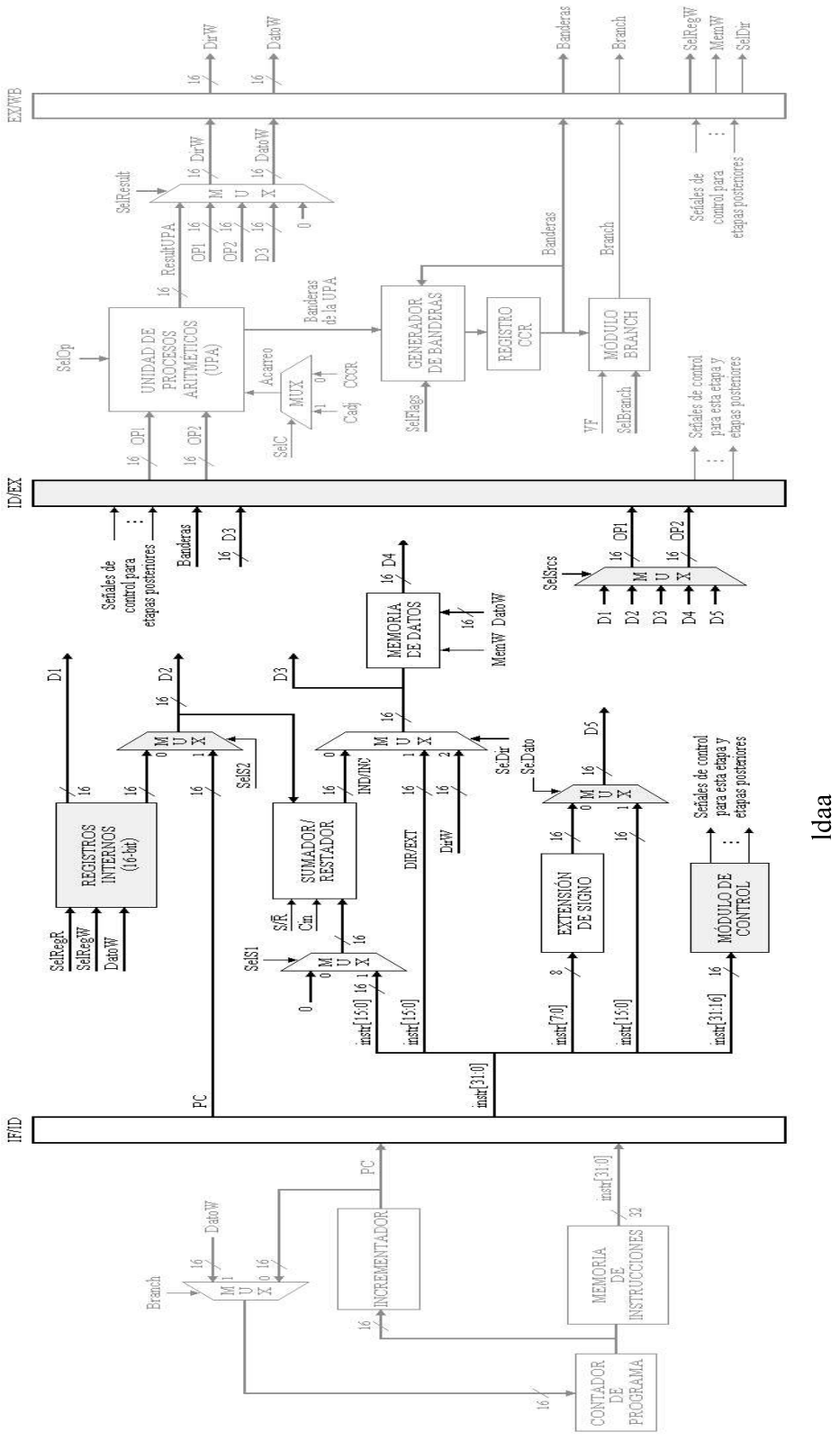


Figura 7.15. En el segundo ciclo de reloj la instrucción LDAA se encuentra en la etapa de decodificación. En esta etapa se extrae un dato inmediato almacenado en el formato de la instrucción, y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

El dato inmediato (contenido en el campo OP2) y el valor de cero (contenido en el campo OP1) son procesados por la UPA. La señal de control SelOp le indica a la UPA la operación que debe ejecutar entre estos dos operandos, para este caso es una OR lógica. De igual manera, son generados los valores de las banderas para esta operación. Observe que para la operación OR los valores de las señales de control SelC y Cadj no son utilizados.

El módulo generador de banderas toma los valores de las banderas generadas por la UPA, y actualiza el registro de estados (CCR) con esos nuevos valores según la especificación de la señal SelFlags. Por otra parte, el módulo Branch revisa la condición de salto y genera la señal Branch de acuerdo a las señales de control que recibe. Para este caso SelBrach=0 y VF=1, es decir, el valor de VF es comparado contra cero, como estos valores son diferentes, entonces, la señal Branch que se genera vale cero.

Por último, en el registro de segmentación EX/WB se guardan: el resultado de la UPA, la dirección efectiva contenida en el bus D3, algunos valores de las banderas y las señales de control necesarias para la última etapa (post-escritura). Véase la figura 7.16.

Etapa 4. Post-escritura

En esta última etapa el dato inmediato contenido en el campo DatoW del registro de segmentación EX/WB es guardado en el registro ACCA localizado en la etapa 2. La señal de control SelRegW le indica al módulo de registros internos en qué registro guardar este resultado.

Note que para la instrucción *ldaa*, las señales MemW y SelDir no son utilizadas. MemW indica qué operación efectuar en la memoria: lectura o escritura; mientras que la señal SelDir elige el bus de donde provendrá la dirección de memoria en donde se guardará DatoW. Más adelante se analizarán estas señales siguiendo otra instrucción como ejemplo (véase la figura 7.17).

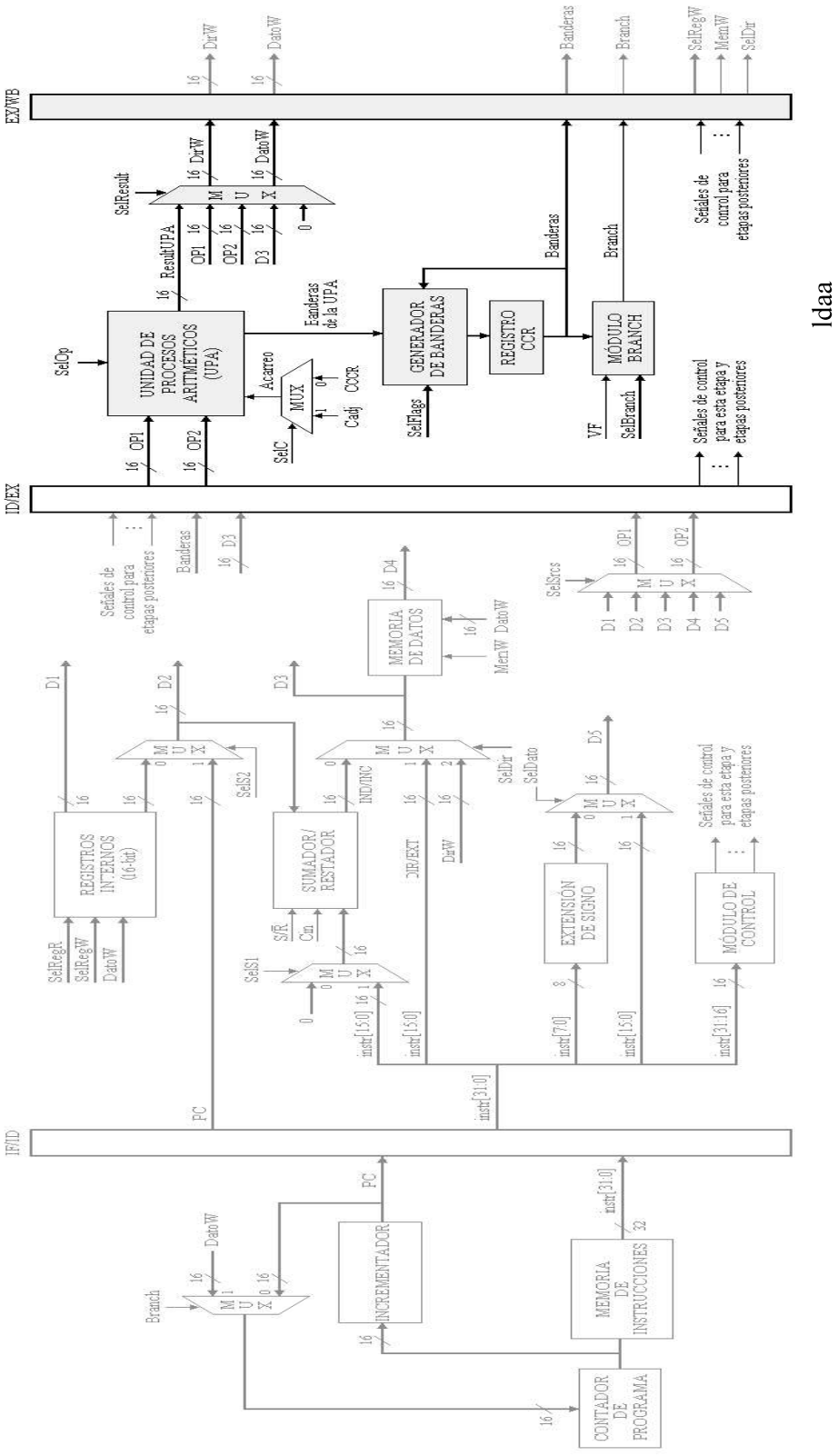


Figura 7.16. En el tercer ciclo de reloj la instrucción LDAA se encuentra en la etapa de ejecución. Durante esta etapa los operandos seleccionados en la etapa anterior son operados en la UPA, se calculan los valores de las banderas afectadas por esta instrucción y se evalúa la condición de salto.

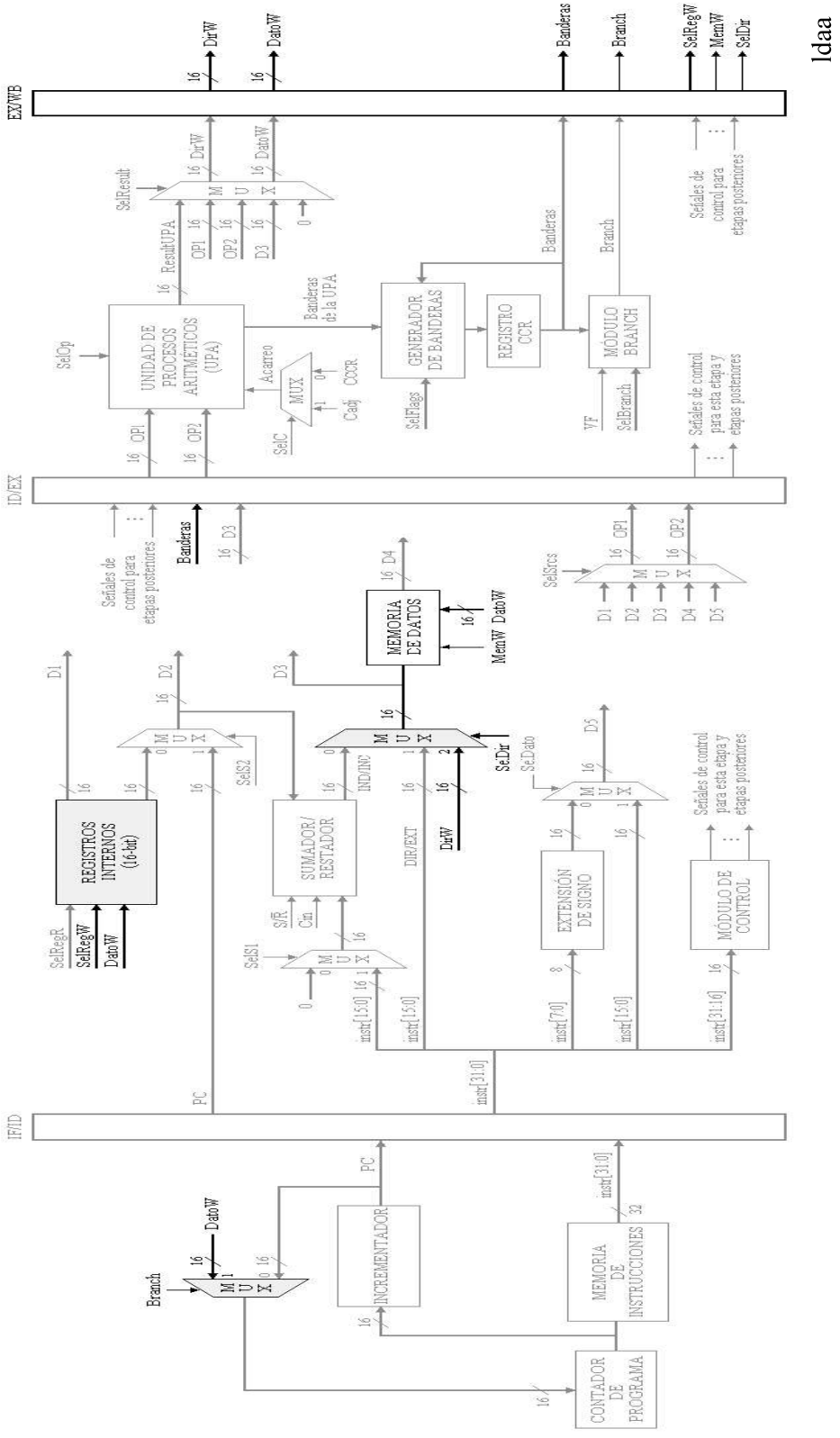


Figura 7.17. En el cuarto ciclo de reloj la instrucción LDAA se encuentra en la etapa de post-escritura. En esta última etapa, el dato inmediato, contenido en el bus DatoW, es guardado en el registro ACCA.

7.3.2 INSTRUCCIÓN ABA (Acceso Inherente)

Instrucción: ABA
Operación: ACCA ⇐ (ACCA) + (ACCB)
Código: 001B
Descripción: Suma el contenido del registro ACCA al contenido del registro ACCB, y el resultado lo guarda nuevamente en ACCA.

Banderas:
$H = ACCA7 \bullet ACCB7 + ACCB7 \bullet \overline{RES7} + \overline{RES7} \bullet ACCA7$
N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
$V = ACCA15 \bullet ACCB15 \bullet \overline{RES15} + \overline{ACCA15} \bullet \overline{ACCB15} \bullet RES15$
$C = ACCA15 \bullet ACCB15 + ACCB15 \bullet \overline{RES15} + \overline{RES15} \bullet ACCA15$

donde,
ACCA15 = Bit más significativo del dato contenido en el registro ACCA
ACCB15 = Bit más significativo del dato contenido en el registro ACCB
RES15 = Bit más significativo del resultado de la suma

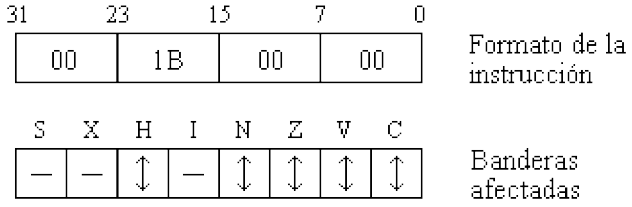


Figura 7.18. Formato de la instrucción ABA y banderas que afecta.

El comportamiento de la instrucción *aba* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *aba*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (ver figura 7.19).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

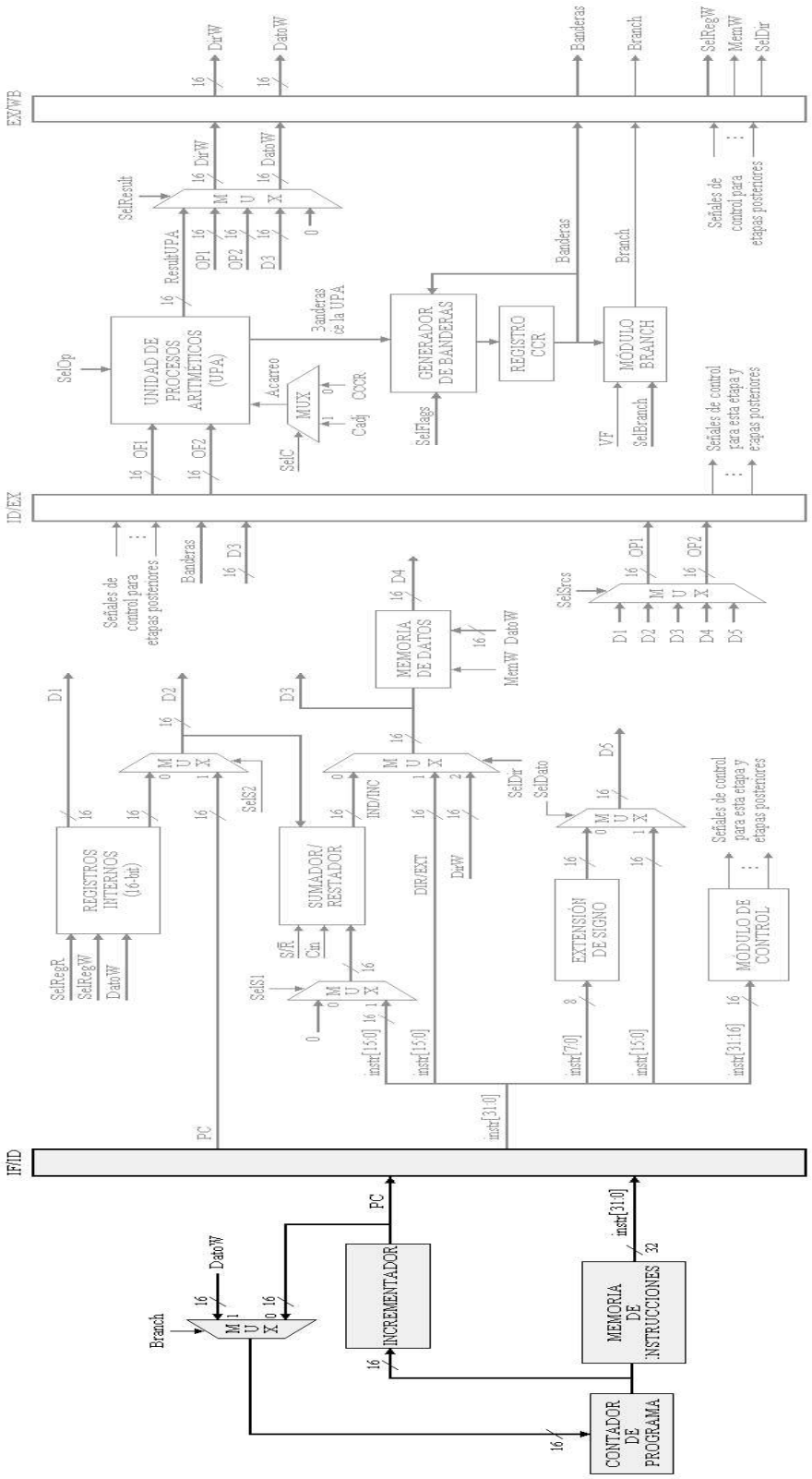
El modo de direccionamiento para esta instrucción es inherente, es decir, el código de la instrucción es suficiente para saber el tipo de instrucción y la tarea que debe ejecutar. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	1
	SelS1	0
	$S/\overline{R}$	1
	Cin	0
	SelS2	0
	SelDato	1
	SelScrs	1
	SelDir	0
Etapa 3	SelOp	1
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	2
	SelBranch	0
	VF	1
Etapa 4	SelRegW	1
	MemW	0
	SelDir	0

Tabla 7.10. Señales de control para la instrucción ABA.

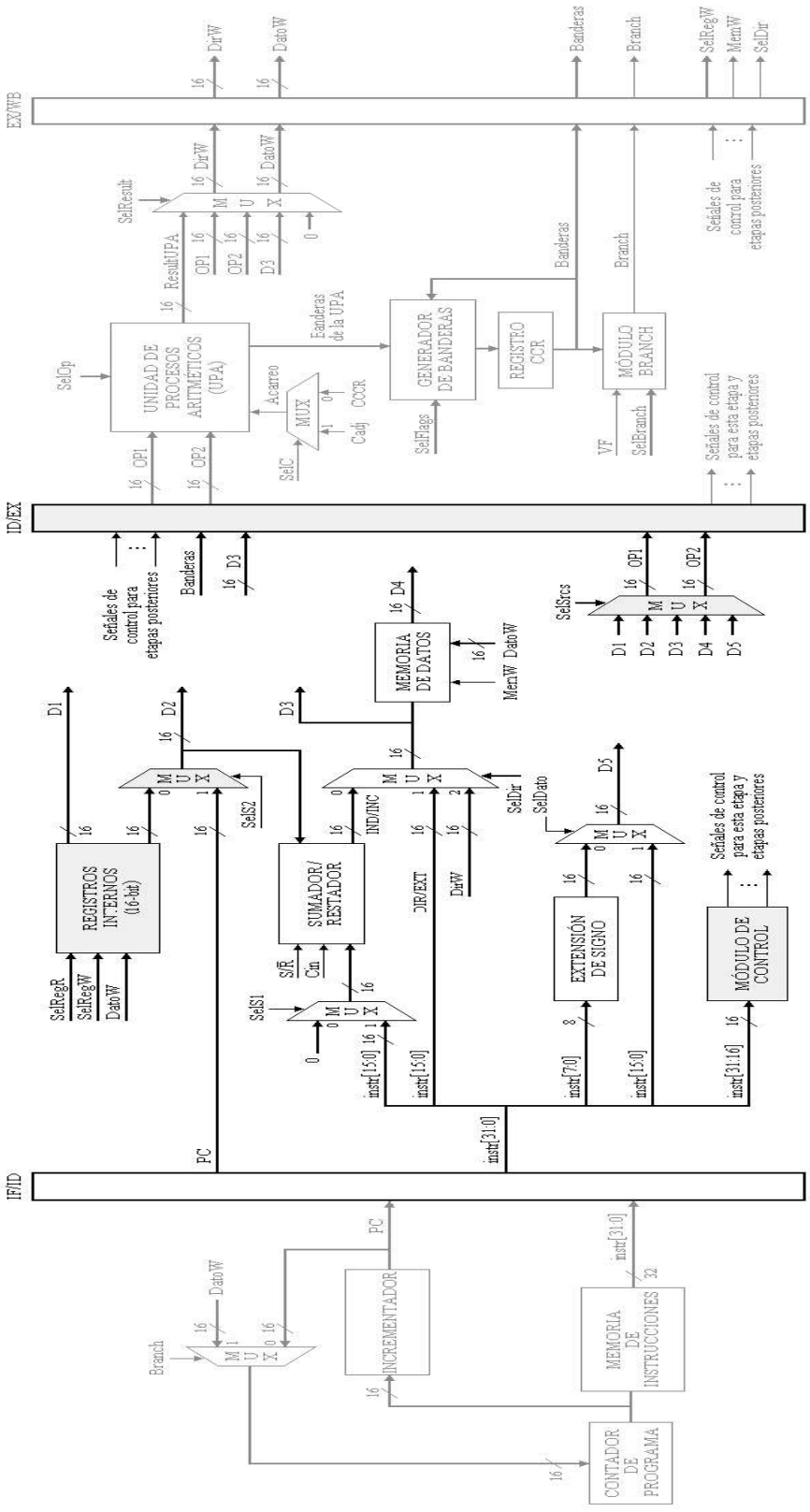
La señal SelRegR=1 permite leer el contenido de los registros ACCA y ACCB. Con SelS2=0 seleccionamos la segunda fuente del módulo de registros, de manera que, el contenido de ACCB está disponible en el bus D2 y el contenido de ACCA en el bus D1.

Para esta instrucción no es necesario calcular la dirección efectiva de los operandos en memoria, ya que éstos son leídos de los registros internos; por lo tanto, los datos seleccionados por medio de las señales SelDir y SelDato son ignorados por esta instrucción. Los datos que verdaderamente utilizaremos son los operandos de los buses D1 y D2, los cuales son seleccionados por medio de la señal SelSrcs=1, y guardados en el registro de segmentación ID/EX. Las señales de control para las etapas 2 y3 también son guardadas en el registro de segmentación, así como la dirección efectiva del bus D3, que como se mencionó anteriormente, no se utiliza para esta instrucción (ver figura 7.20).



aba

Figura 7.19. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción ABA es leída de la memoria de instrucciones.



aba

Figura 7.20. En el segundo ciclo de reloj la instrucción ABA se encuentra en la etapa de decodificación. Durante esta etapa se leen los contenidos de los registros ACCA y ACCB del módulo de registros internos, y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

En esta etapa, los operandos contenidos en los campos OP1 y OP2 del registro de segmentación son procesados por la UPA. La señal de control SelOp=1 hace que la UPA calcule la suma entre OP1, OP2 y un acarreo de entrada, el cual es seleccionado por medio de la señal SelC=1. Recuerde que la operación de adición en la UPA se define de la siguiente manera: $OP1 + OP2 + Acarreo$, por lo tanto, es necesario obligar a que el valor del acarreo sea cero, pues con ello se garantiza que el resultado de la suma no es afectado por un acarreo indeseado.

La UPA también calculará los valores de las banderas de acuerdo con la especificación de la instrucción. Estos valores de banderas son leídos por el módulo generador de banderas, el cual con base en la señal SelFlags, actualizará las banderas en el registro de estados (registro CCR).

El módulo Branch revisa la condición de salto dada por la señal SelBranch; si al evaluar la condición ésta resulta ser igual al valor de la señal VF, entonces, la señal Branch se activa, de lo contrario, Branch vale cero. Para este caso, Branch vale cero pues la condición de salto vale cero y VF vale uno, visto de otra manera, no se realiza el salto, lo cual suena lógico ya que la instrucción *aba* no es una instrucción de salto.

Finalmente, la señal SelResult selecciona el resultado de la UPA y la dirección efectiva guardada en el campo D3 del registro de segmentación ID/EX, y las guarda en el registro de segmentación EX/WB. En este registro también se guardan algunas banderas y las señales de control necesarias para la última etapa (ver figura 7.21).

Etapa 4. Post-escritura

Sólo falta guardar el resultado obtenido en la etapa anterior en el registro ACCA, para esto, la señal SelRegW es colocada a 1. Las señales SelDir y MemW valen cero ya que no deseamos escribir nada en memoria (ver figura 7.22).

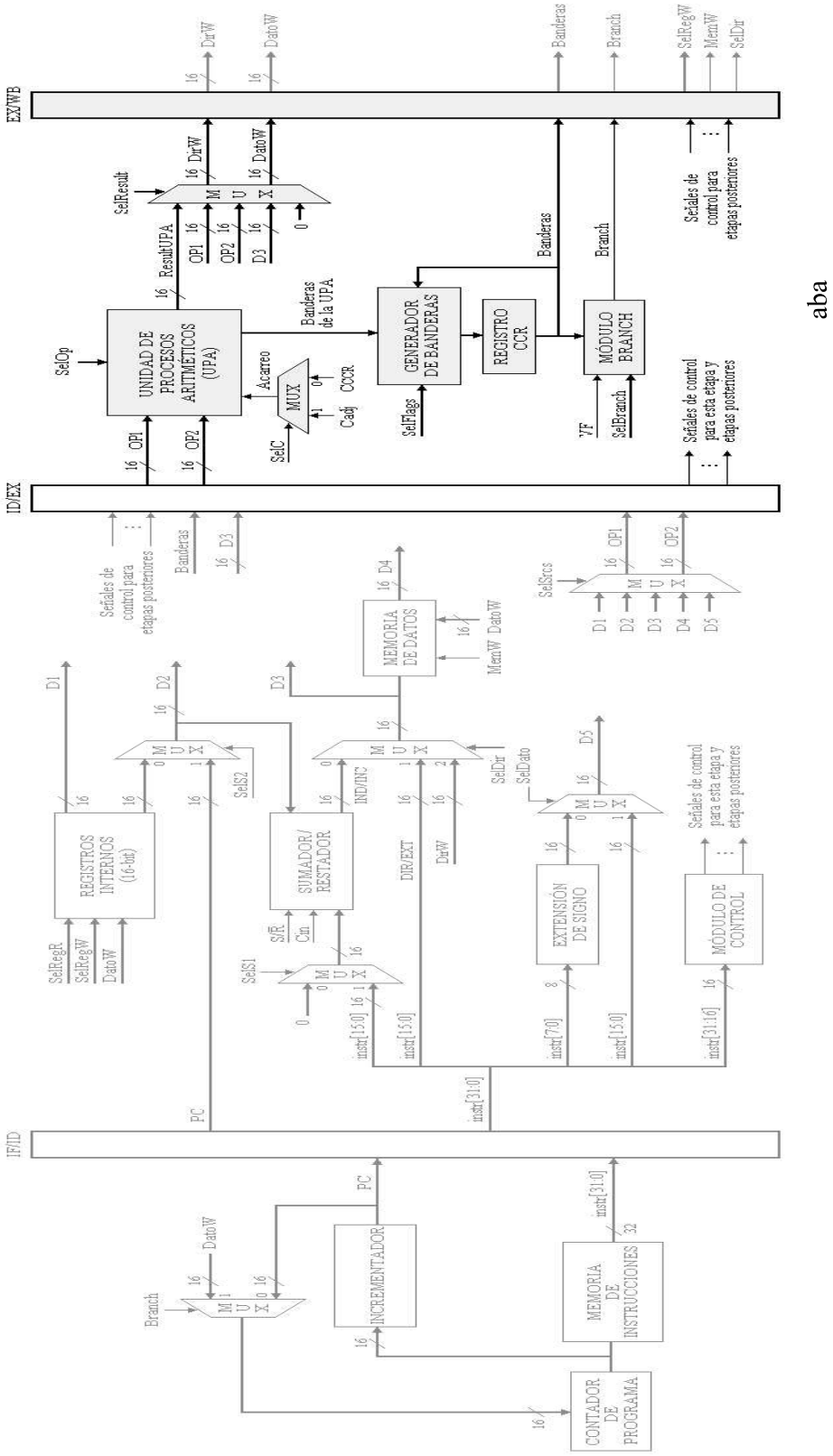


Figura 7.21. En el tercer ciclo de reloj la instrucción ABA se encuentra en la etapa de ejecución. Durante esta etapa se calcula la suma entre los operandos seleccionados (los contenidos de ACCA y ACCB), se calculan los valores de las banderas afectadas y se evalúa la condición de salto.

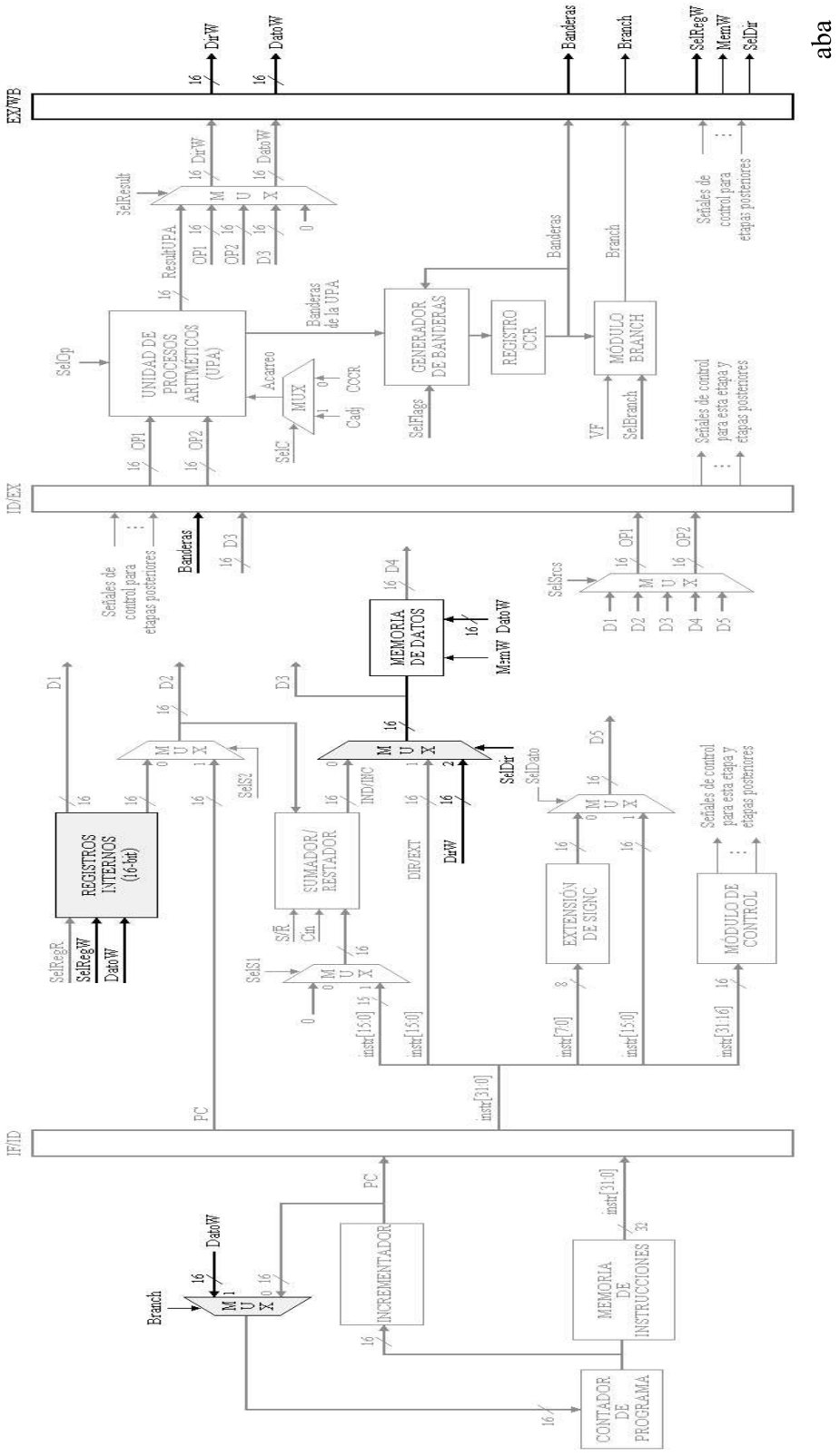


Figura 7.22. En el cuarto ciclo de reloj la instrucción ABA se encuentra en la etapa de post-escritura. En esta última etapa, el resultado de la suma, contenido en el bus DatoW, es guardado en el registro ACCA.

7.3.3 INSTRUCCIÓN ANDB (Acceso Extendido)

Instrucción: ANDB Dirección_16Bits
Operación: ACCB ⇐ (ACCB) • (Memoria)
Código: 00F4
Descripción: Ejecuta la operación AND lógica entre el contenido del registro ACCB y un dato contenido en memoria. El resultado se guarda en ACCB.
Banderas: N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

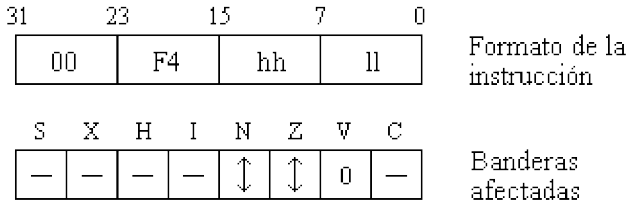


Figura 7.23. Formato de la instrucción ANDB y banderas que afecta.

El comportamiento de la instrucción *andb* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *andb*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.24).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

Esta instrucción utiliza el modo de direccionamiento extendido, es decir, uno de los operandos necesarios por la instrucción se encuentra almacenado en memoria. La dirección efectiva en donde se aloja el dato en memoria se obtiene directamente del formato de la instrucción. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	5
	SelS1	0
	$S/\overline{R}$	1
	Cin	0

	SelS2	0
	SelDato	1
	SelScrs	2
	SelDir	1
Etapa 3	SelOp	3
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	1
	SelBranch	0
	VF	1
Etapa 4	SelRegW	4
	MemW	0
	SelDir	0

Tabla 7.11. Señales de control para la instrucción ANDB.

El primer operando se lee del registro ACCB por medio de la señal SelRegR=5, y el segundo operando se lee de la memoria de datos. La dirección efectiva para el segundo operando se extrae del formato de la instrucción, esta dirección se pasa a la memoria mediante SelDir=1, y el dato contenido en esa dirección es leído y colocado en el bus D4. A continuación, con SelScrs=2 se escriben los datos del bus D1 (con el contenido del registro ACCB) y del bus D4 (con el contenido de memoria) en el registro de segmentación ID/EX. También guardamos en el registro ID/EX las señales de control para las etapas 3 y 4, así como la dirección efectiva contenida en el bus D3 (véase la figura 7.25).

Etapa 3. Ejecución / Cálculo de banderas y saltos

En esta etapa, la UPA calcula la AND lógica entre los datos contenidos en los campos OP1 y OP2 del registro de segmentación ID/EX, para ello, la señal SelOp es colocada a 2. Debido a que la operación AND no utiliza un acarreo de entrada, entonces, éste es ignorado.

El módulo generador de banderas actualiza los valores de las banderas de acuerdo a la señal SelFlags. El módulo Branch revisa la condición de salto dada por la señal SelBranch y VF, y genera la señal Branch. La señal SelResult selecciona el resultado de la UPA y la dirección efectiva, y los guarda en el registro de segmentación EX/WB; también son guardadas las señales de control necesarias para la última etapa y algunos valores de banderas (véase la figura 7.26).

Etapa 4. Post-escritura

En esta etapa se guarda el resultado de la operación AND en el registro ACCB, para ello, la señal SelRegW es colocada a 4. Las señales SelDir y MemW valen cero ya que no deseamos escribir nada en memoria (véase la figura 7.27).

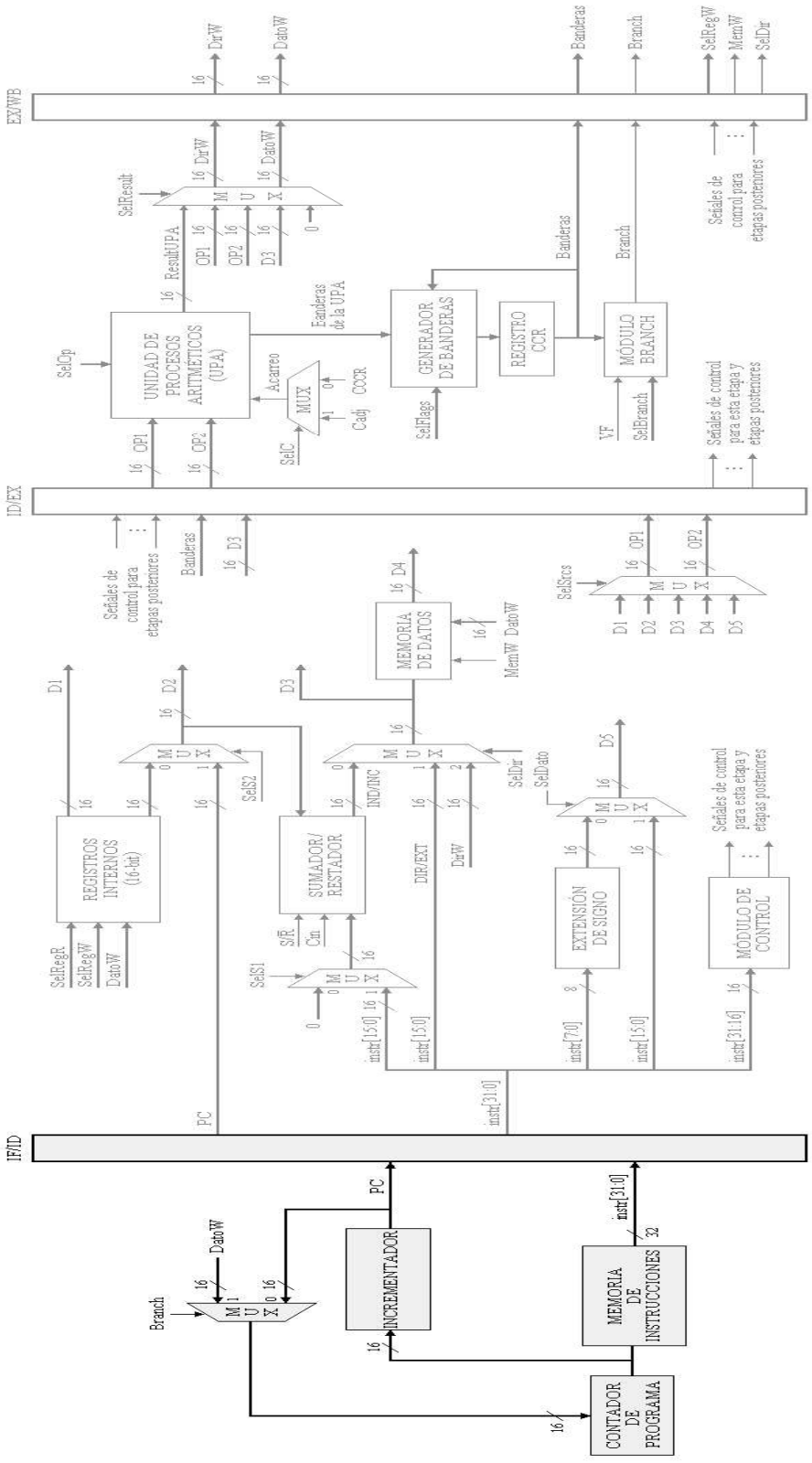


Figura 7.24. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción ANDB es leída de la memoria de instrucciones.

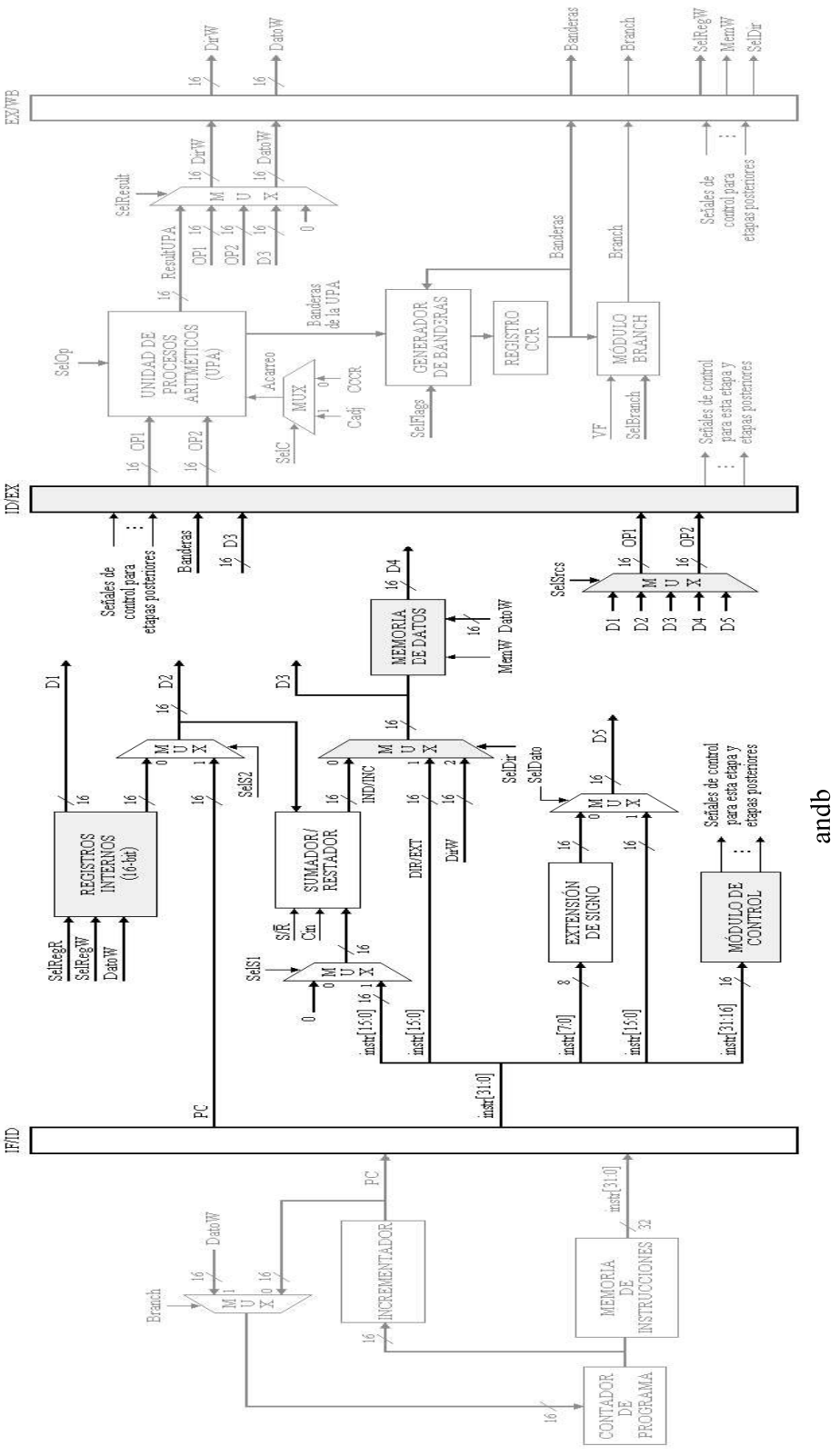


Figura 7.25. En el segundo ciclo de reloj la instrucción ANDB se encuentra en la etapa de decodificación. Durante esta etapa se obtienen los operandos que necesita la instrucción, el contenido del registro ACCB y un dato contenido en memoria, y se generan las señales de control para las etapas siguientes.

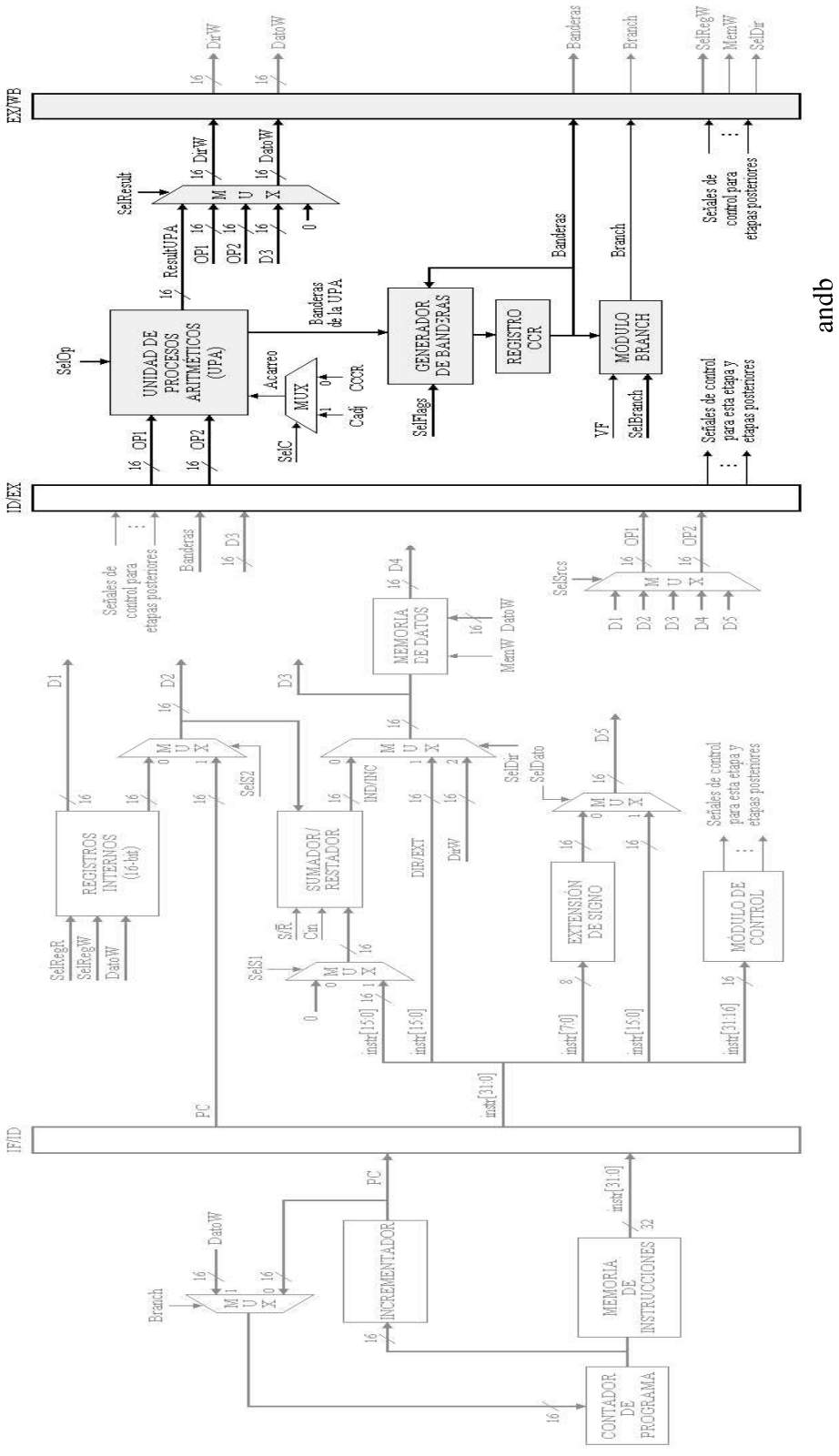


Figura 7.26. En el tercer ciclo de reloj la instrucción ANDB se encuentra en la etapa de ejecución. Durante esta etapa se calcula la operación lógica AND entre los operandos seleccionados (el registro ACCB y dato de memoria), se actualizan los valores de las banderas afectadas y se evalúa la condición de salto.

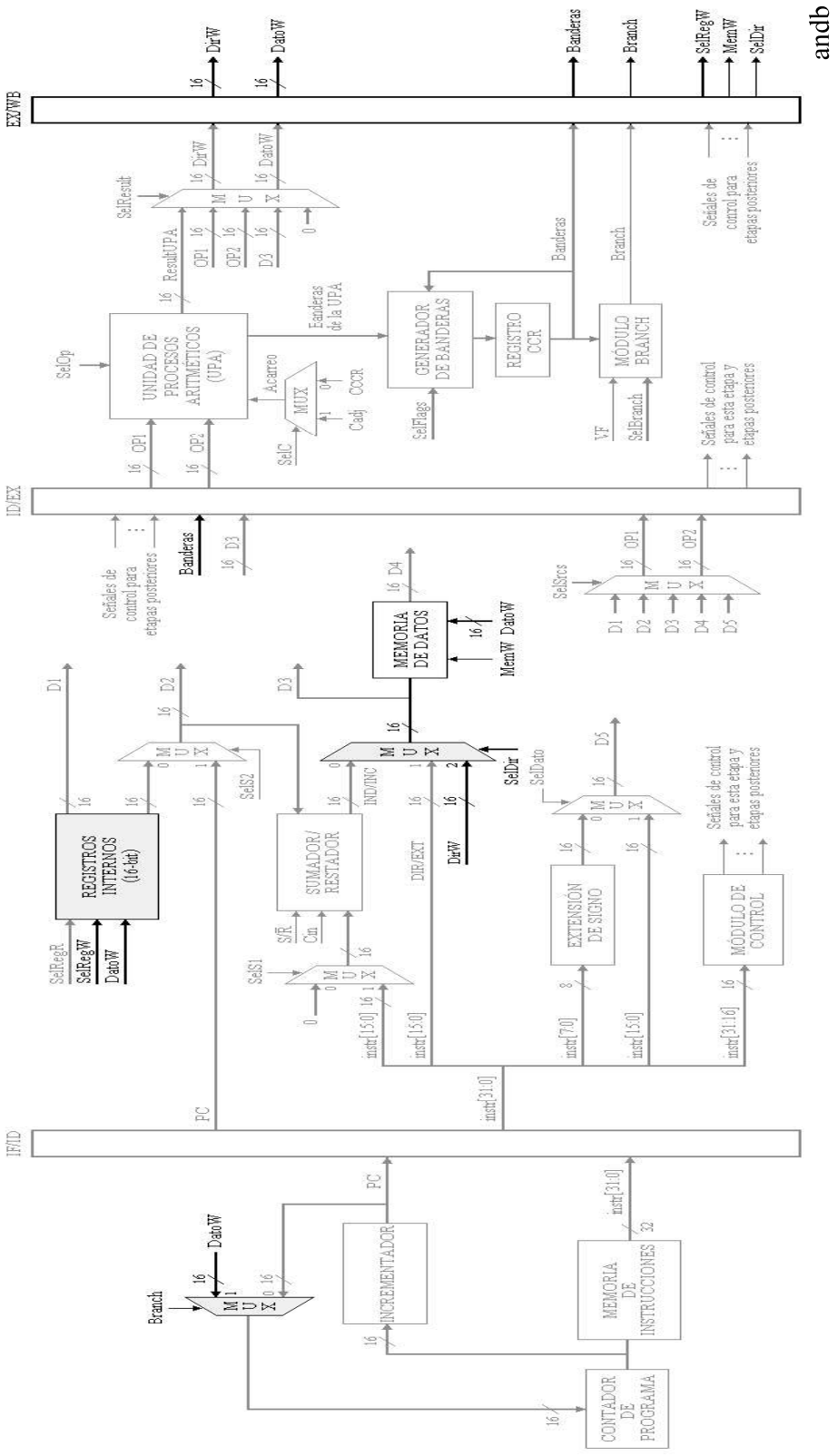


Figura 7.27. En el cuarto ciclo de reloj la instrucción ANDB se encuentra en la etapa de post-escritura. En esta última etapa, el resultado de la operación lógica, contenido en el bus DatoW, es guardado en el registro ACCB.

7.3.4 INSTRUCCIÓN ASL (Acceso Indexado)

Instrucción: ASL Desplazamiento_8Bits_sin_signo, Y
Operación: (Memoria) ⇐ (Memoria) << 1
Código: 1868
Descripción: Realiza un corrimiento hacia la izquierda del dato contenido en memoria. El bit de acarreo del registro CCR es cargado con el bit más significativo del dato de memoria, mientras que el bit 0 del dato de memoria es cargado con cero.
Banderas: N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
$V = N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ para N y C después del corrimiento.
C=1 si el MSB del dato de memoria (antes del corrimiento) está encendido, C=0 en caso contrario.

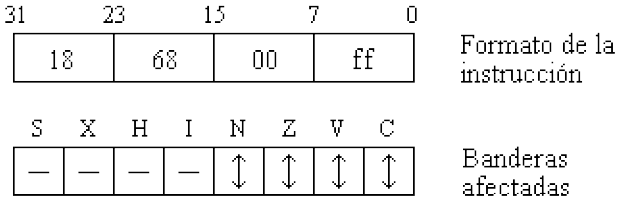


Figura 7.28. Formato de la instrucción ASL y banderas que afecta.

El comportamiento de la instrucción *asl* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *asl*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.30).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

Las señales de control para las etapas 2, 3 y 4 se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	A
	SelS1	1
	$S/\overline{R}$	1
	Cin	0

	SelS2	0
	SelDato	1
	SelScrs	4
	SelDir	0
Etapa 3	SelOp	6
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	3
	SelBranch	0
	VF	1
Etapa 4	SelRegW	0
	MemW	1
	SelDir	2

Tabla 7.12. Señales de control para la instrucción ASL.

Esta instrucción utiliza el modo de direccionamiento indexado, por lo tanto, la dirección efectiva del operando en memoria se calcula sumando al contenido del registro índice IY el desplazamiento de 8 bits sin signo.

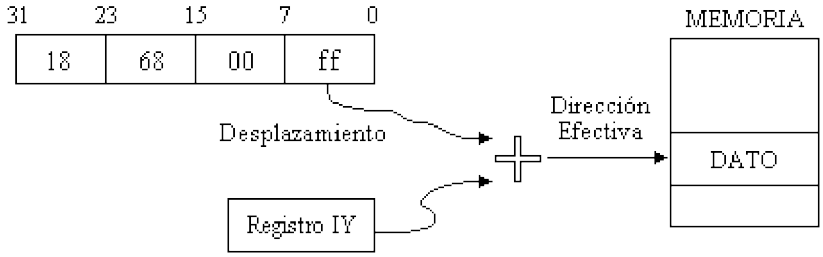


Figura 7.29. Cálculo de la dirección efectiva para el modo de direccionamiento indexado.

La señal SelRegR=A lee el contenido del registro índice IY del módulo de registros internos. La señal SelS1=1 selecciona el desplazamiento de 8 bits sin signo proveniente del formato de la instrucción, y el módulo sumador/restador suma al desplazamiento el dato seleccionado con SelS2=0, es decir, con el contenido del registro IY. De esta manera, el módulo sumador/restador calcula la dirección efectiva del dato en memoria. A continuación, la señal SelDir=0 pasa esta dirección a la memoria, y el dato contenido en dicha localidad es leído y colocado en el bus D4. En esta ocasión la señal SelDato vale uno, pero el dato que selecciona es ignorado.

Finalmente, la señal SelSrcs=4 toma los contenidos de los buses D4 y D3, y los guarda en los campos OP1 y OP2 del registro de segmentación ID/EX, respectivamente. Recuerde que las señales de control para las etapas siguientes y la dirección efectiva también son guardadas en este registro. Note que para esta instrucción sí es necesario guardar la dirección efectiva, pues de no hacerlo, no se sabrá en donde almacenar el resultado del corrimiento que se obtendrá en la siguiente etapa (véase la figura 7.31).

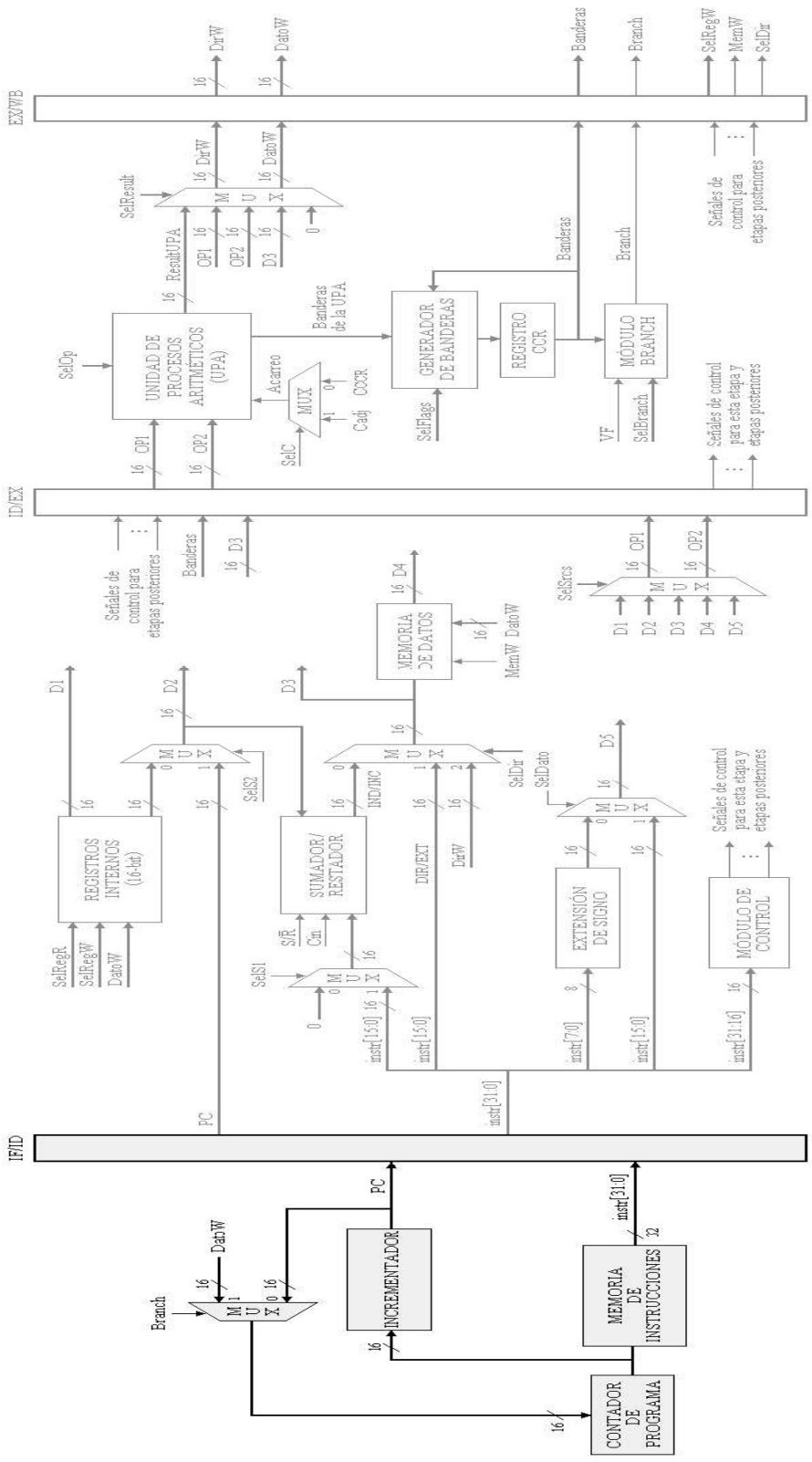
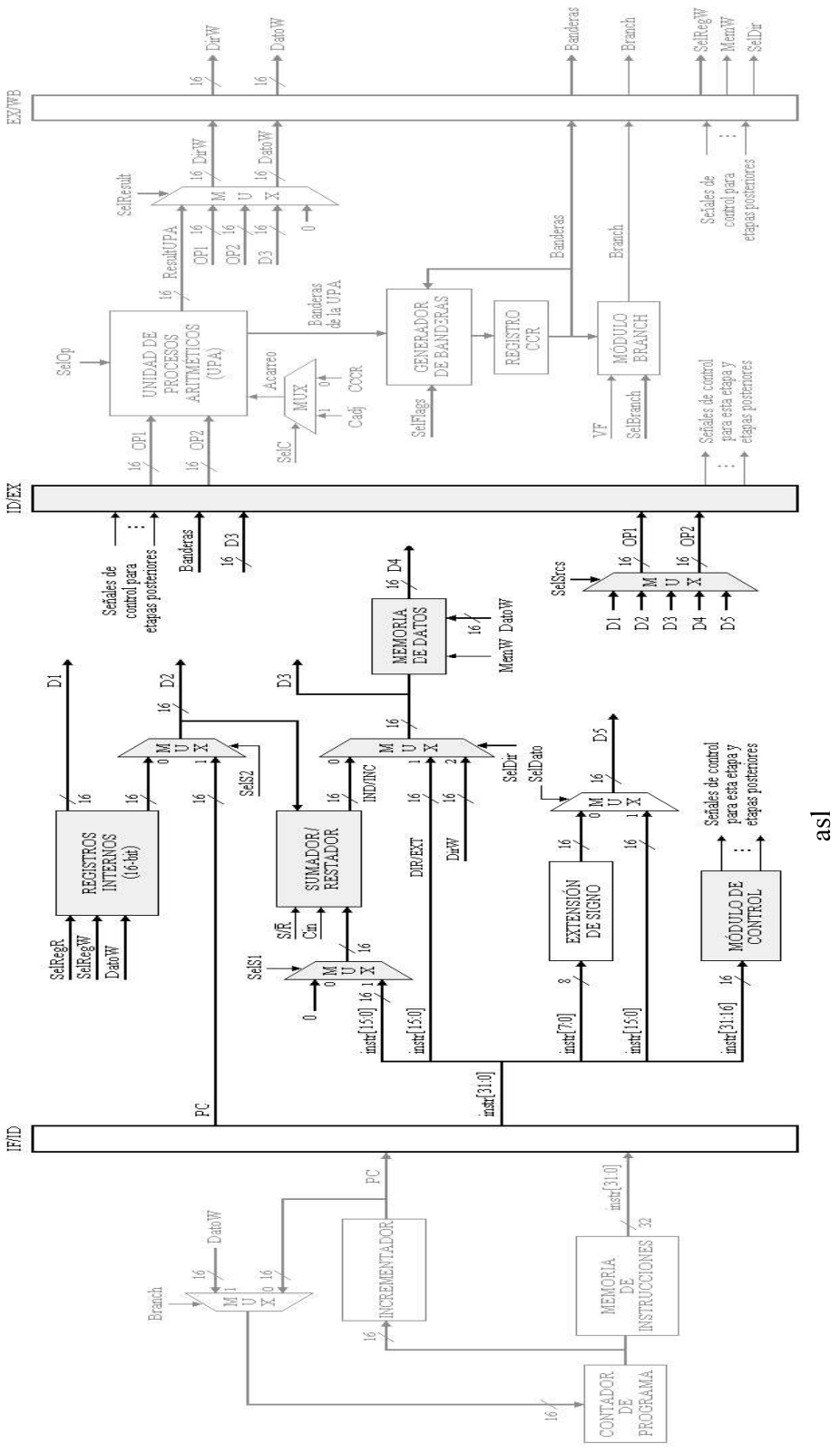


Figura 7.30. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción ASL es leída de la memoria de instrucciones.

asl



asl

Figura 7.31. En el segundo ciclo de reloj la instrucción ASL se encuentra en la etapa de decodificación. Durante esta etapa se calcula la dirección efectiva de un operando en memoria, se lee dicho operando y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

La operación de corrimiento a la izquierda y el cálculo de las banderas, tal y como lo establece la especificación de la instrucción, se realizan en la UPA con SelOp=6. El resultado de esta operación junto con la dirección efectiva (contenida en el campo D3 del registro ID/EX) son seleccionadas por medio de la señal SelResult=1, y guardadas en el registro de segmentación EX/WB en el flanco de subida del reloj. Las señales de control para la última etapa y algunas banderas también son guardadas en el registro EX/WB.

Por otra parte, el módulo generador de banderas junto con la señal de control SelFlags, actualizan las banderas calculadas por la UPA en el registro CCR. Y el módulo Branch junto con las señales SelBranch y VF, determinan si se ejecuta un salto o no (véase la figura 7.32).

Etapa 4. Post-escritura

En esta última etapa se guarda el resultado de la UPA en memoria. La localidad de memoria en donde se almacenará el resultado es la misma de donde fue leído el dato, es decir, en la dirección dada por IY + desplazamiento, la cual está guardada en el campo DirW del registro de segmentación EX/WB.

Para guardar un dato en memoria se necesita activar la señal MemW, la cual nos permite realizar una operación de escritura en ella. También es necesario colocar la señal SelDir a 2, pues así se seleccionará el bus DirW con la dirección de la localidad de memoria en donde se guardará el resultado (DatoW). Observe que para esta instrucción no se actualiza ningún registro interno, por eso, SelRegW es colocado a cero (véase la figura 7.33).

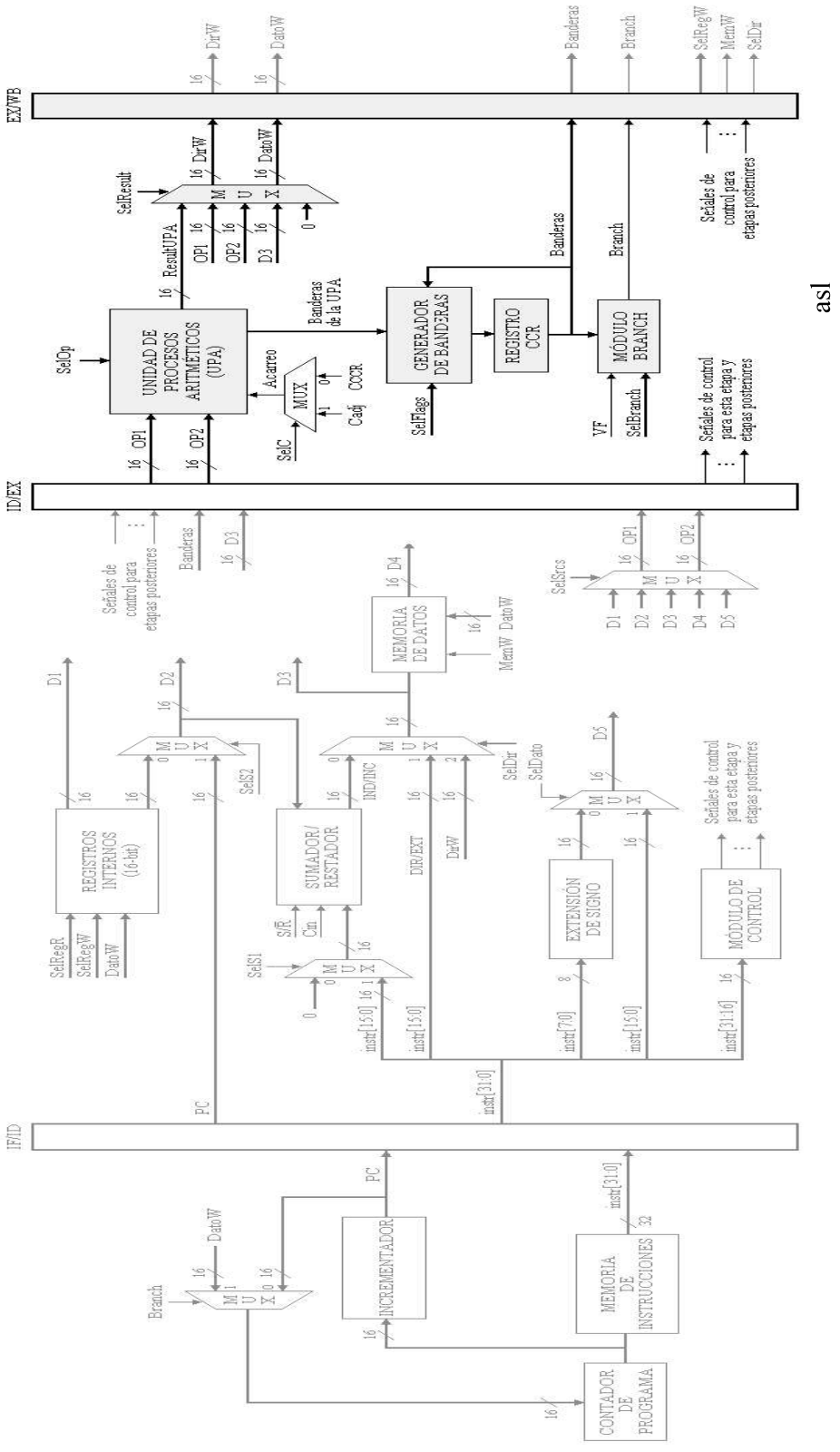


Figura 7.32. En el tercer ciclo de reloj la instrucción ASL se encuentra en la etapa de ejecución. Durante esta etapa se calcula el corrimiento del dato leído de memoria en la etapa anterior, se actualizan los valores de las banderas afectadas y se evalúa la condición de salto.

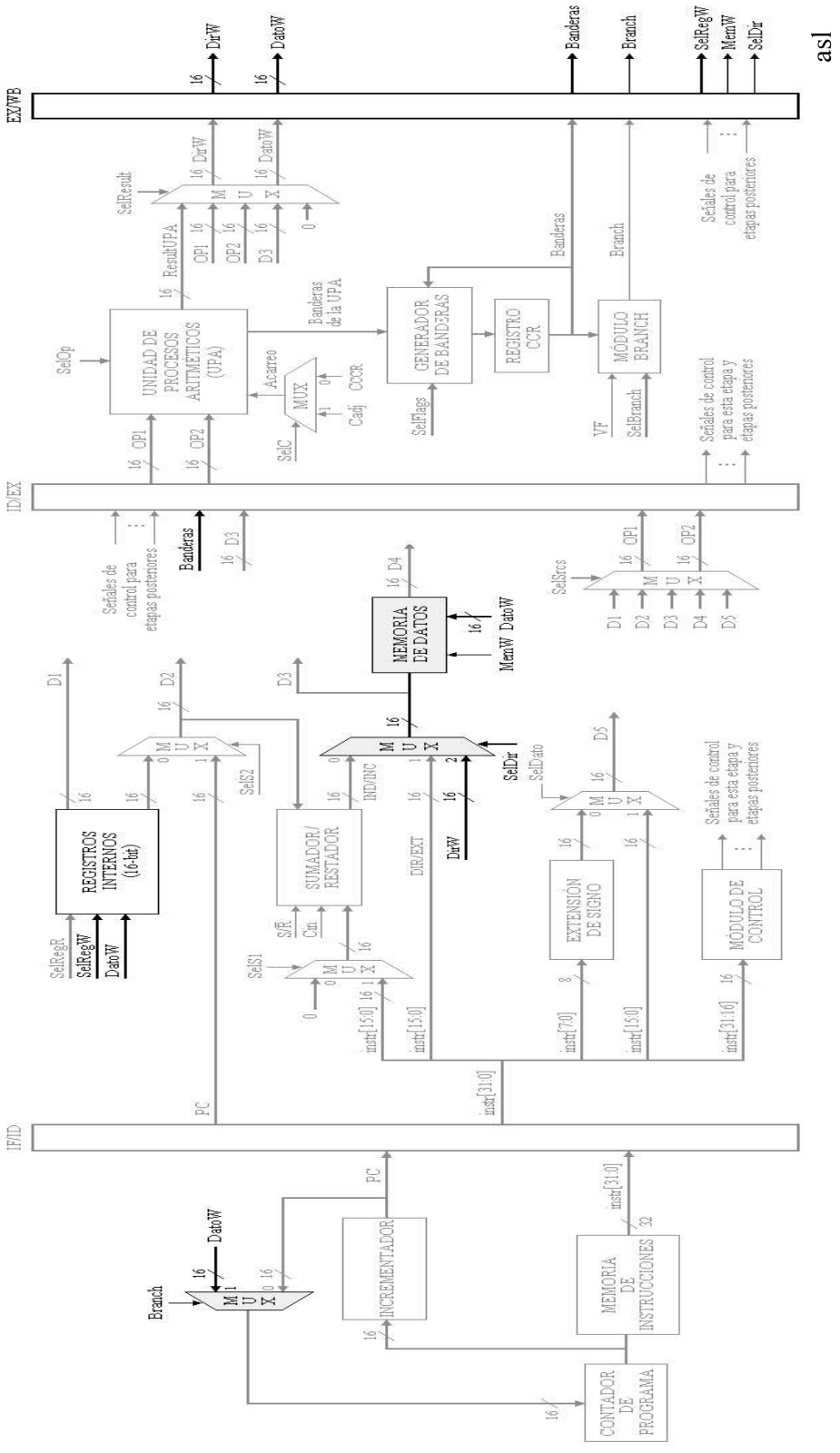


Figura 7.33. En el cuarto ciclo de reloj la instrucción ASL se encuentra en la etapa de post-escritura. En esta última etapa, el resultado del corrimiento, calculado en la etapa anterior y contenido en el bus DatoW, es guardado en la localidad de memoria dada por DirW.

7.3.5 INSTRUCCIÓN STAA (Acceso Extendido)

Instrucción: STAA Dirección_16Bits
Operación: (Memoria) ⇐ (ACCA)
Código: 00B7
Descripción: Almacena el contenido del registro ACCA en memoria. El contenido de ACCA permanece sin cambios.
Banderas: N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

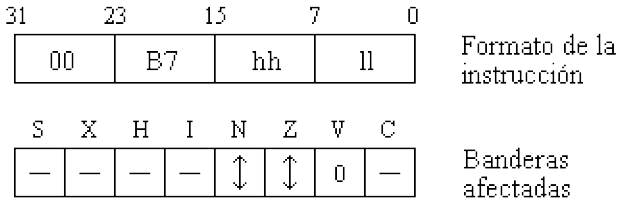


Figura 7.34. Formato de la instrucción STAA y banderas que afecta.

El comportamiento de la instrucción *staa* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *staa*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.35).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

En esta instrucción el modo de direccionamiento es extendido. La dirección de la localidad de memoria, en donde se guardará el contenido del registro ACCA, se obtiene directamente del formato de la instrucción. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	4
	SelS1	1
	$S/\overline{R}$	1
	Cin	0

	SelS2	0
	SelDato	1
	SelScrs	1
	SelDir	0
Etapa 3	SelOp	4
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	1
	SelBranch	0
	VF	1
Etapa 4	SelRegW	0
	MemW	1
	SelDir	2

Tabla 7.13. Señales de control para la instrucción STAA.

El dato que deseamos guardar en la memoria se lee del módulo de registros internos por medio de la señal SelRegR=4. La dirección efectiva en donde se guardará el contenido del registro ACCA se pasa por el módulo sumador/restador, quien suma al dato seleccionado con SelS1=1 el dato seleccionado con SelS2=0, es decir, se suma a la dirección de 16 bits un cero. El resultado final, la misma dirección de 16 bits, es seleccionada por medio de la señal SelDir=0.

De esta manera, en el bus D1 se tendrá el contenido del registro ACCA, en el bus D2 un cero, y en el bus D3 la dirección efectiva de 16 bits. Todos estos datos son guardados en el registro de segmentación ID/EX mediante la señal SelSrcs=1. Recuerde que también se guardan en el registro de segmentación ID/EX las señales de control para las etapas siguientes.

Note que para la instrucción *staa* el resultado seleccionado por la señal SelDato carece de importancia (véase la figura 7.36).

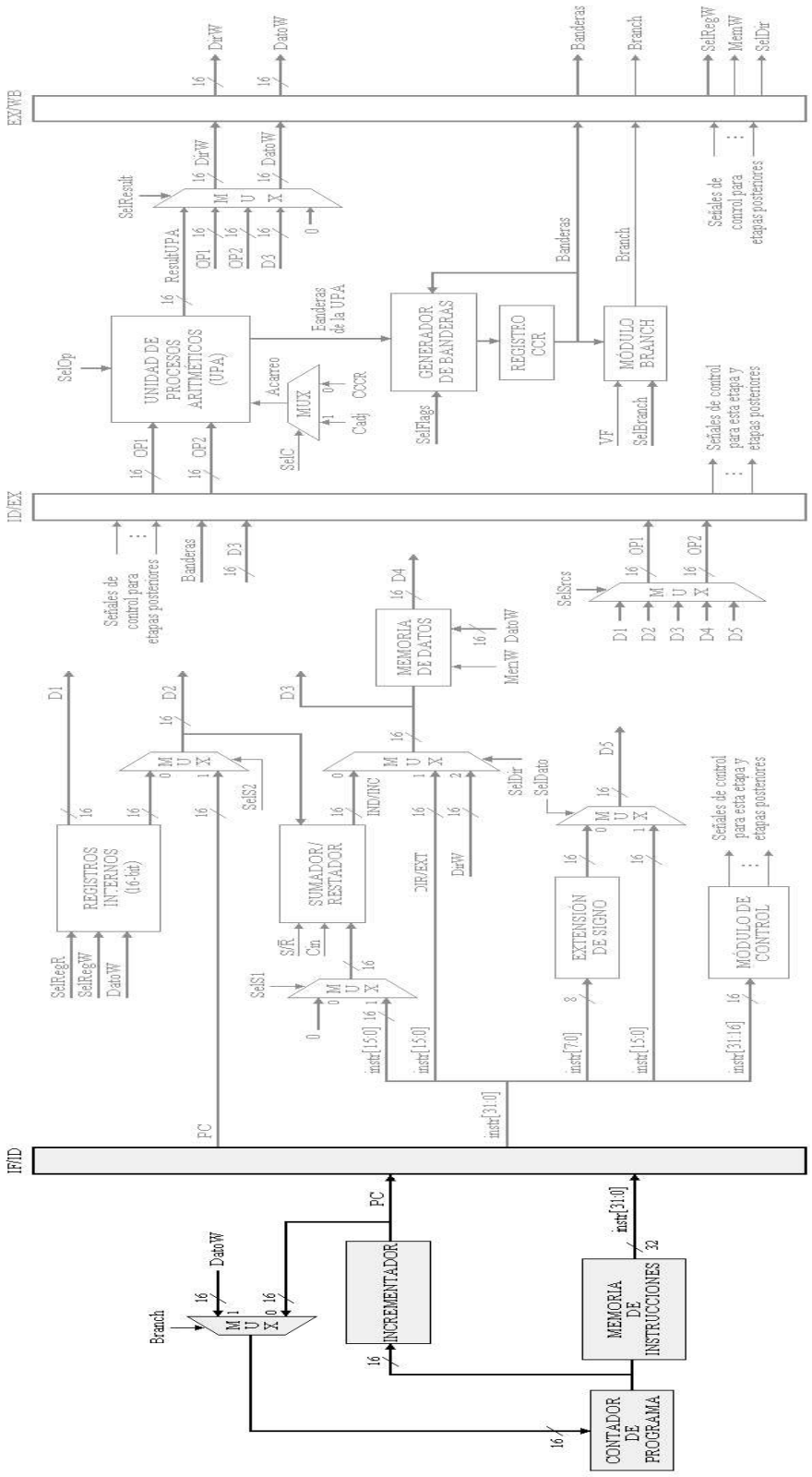


Figura 7.35. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción STAA es leída de la memoria de instrucciones.

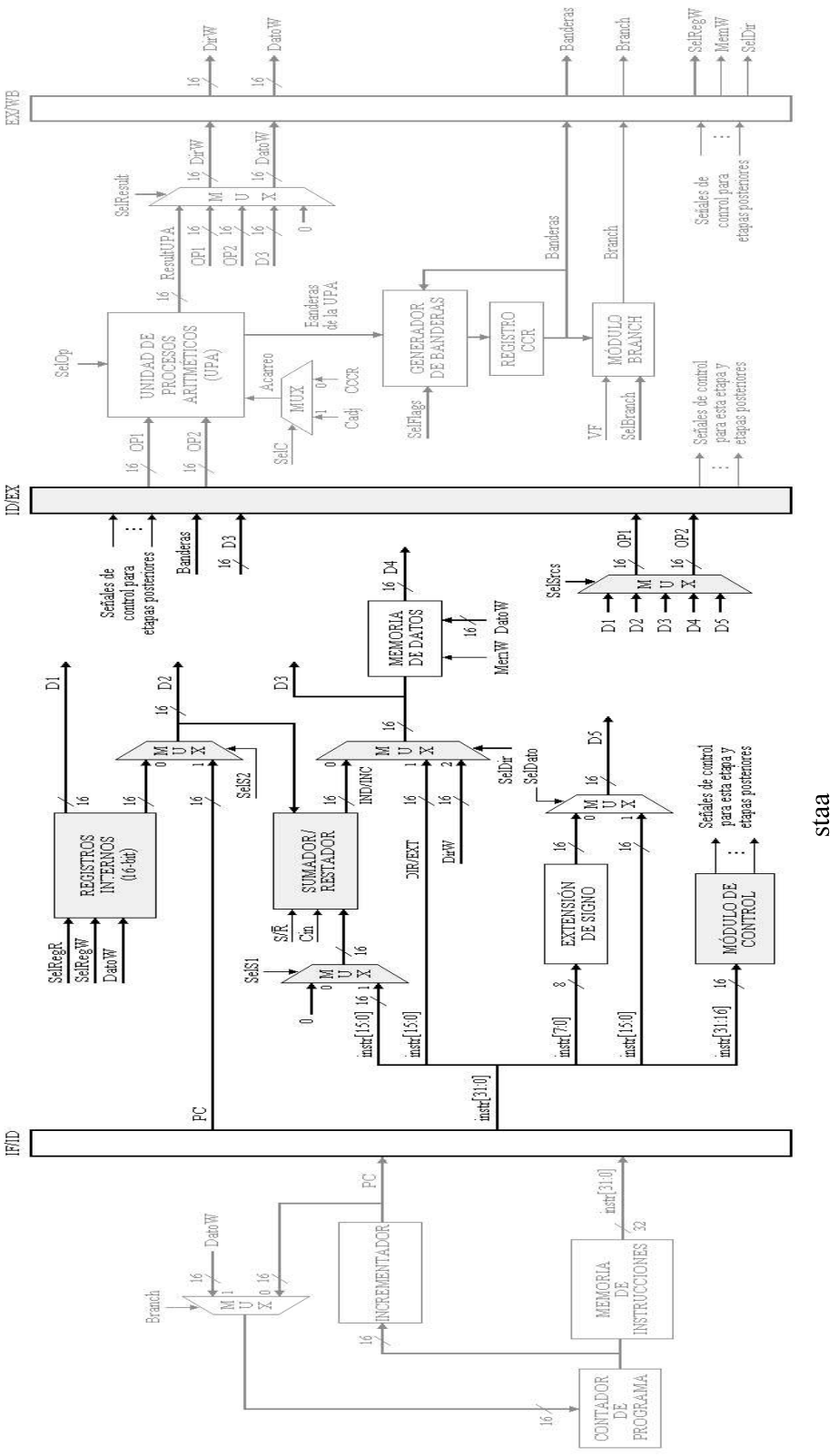


Figura 7.36. En el segundo ciclo de reloj la instrucción STAA se encuentra en la etapa de decodificación. Durante esta etapa se lee el contenido del registro ACCA, se extrae la dirección efectiva en donde se almacenará dicho contenido, y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

El dato contenido en el campo OP1 del registro de segmentación ID/EX corresponde al contenido del registro ACCA, y el dato contenido en el campo OP2 corresponde al valor de cero. La señal SelOp=4 opera los contenidos de estos dos campos utilizando la operación lógica OR, de manera que el dato en OP1 no es modificado. A simple vista esta operación puede parecer innecesaria, sin embargo, sí tiene una razón de ser, pues calcula los valores de las banderas especificados por la instrucción.

Las banderas afectadas son actualizadas en el registro CCR según la señal SelFlags; y como esta instrucción no es una instrucción de salto, entonces, colocamos las señales SelBranch a cero y VF a uno para que el módulo Branch evite generar una señal de salto.

La señal SelResult=1 selecciona el resultado de la operación lógica OR y lo guarda en el registro de segmentación EX/WB junto con la dirección efectiva (contenida en el campo D3 del registro de segmentación ID/EX). También son guardadas en el registro EX/WB algunas banderas y las señales de control necesarias para la última etapa (véase la figura 7.37).

Etapa 4. Post-escritura

En esta última etapa se guarda el contenido del registro ACCA en memoria. El contenido de dicho registro está almacenado en el campo DatoW del registro de segmentación EX/WB, y la localidad de memoria en donde es almacenado el dato también está guardada en el registro EX/WB, pero en el campo DirW.

Finalmente, para guardar el dato en memoria se activa la señal MemW, la cual permite realizar una operación de escritura en la memoria. También es necesario colocar la señal SelDir a 2, pues con ello se seleccionará el bus DirW con la dirección de la localidad de memoria en donde se guardará el contenido del registro ACCA. Observe que para esta instrucción no se actualiza ningún registro interno, por eso la señal SelRegW se coloca a cero (véase la figura 7.38).

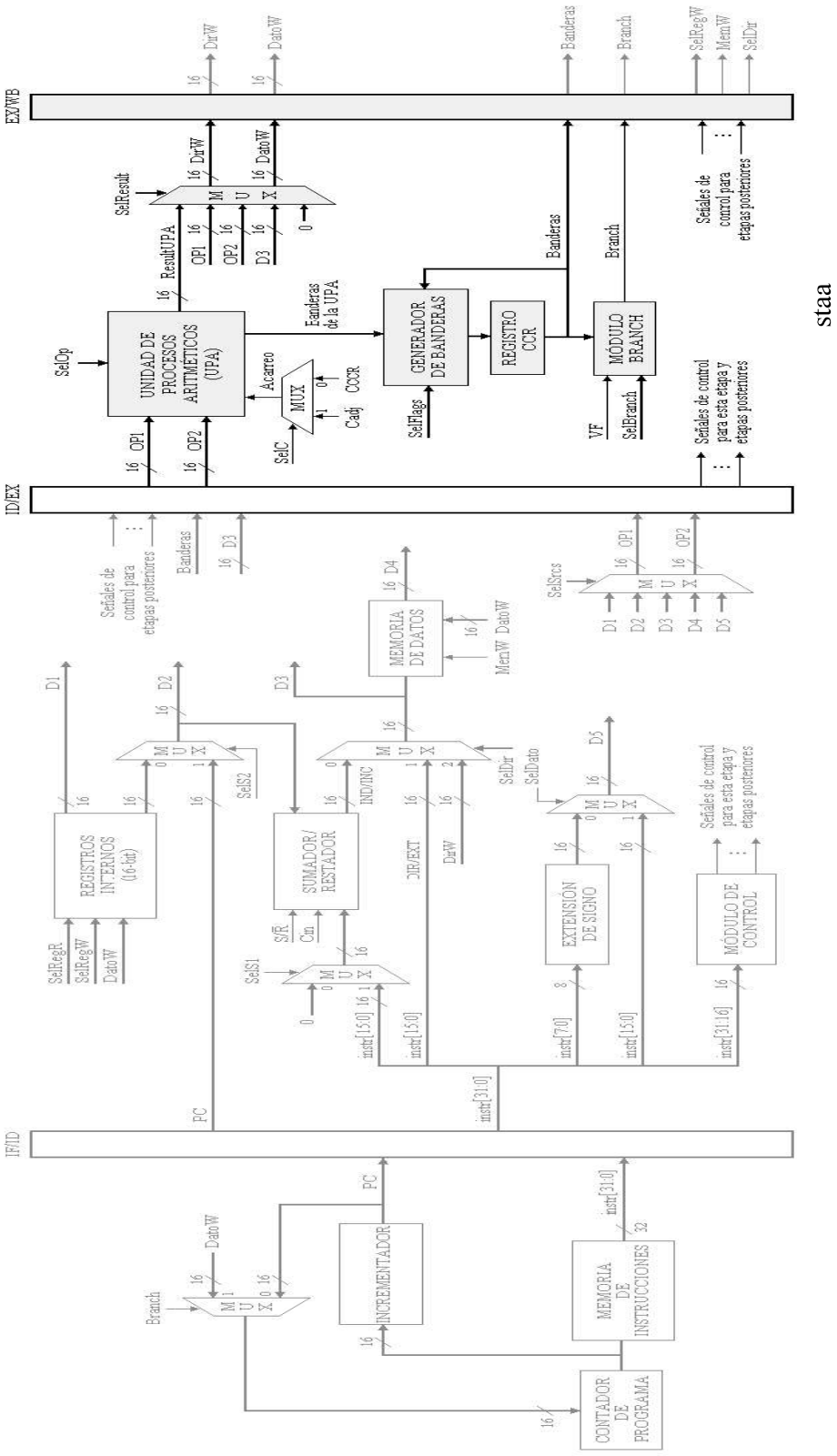


Figura 7.37. En el tercer ciclo de reloj la instrucción STAA se encuentra en la etapa de ejecución. Durante esta etapa se actualizan los valores de las banderas afectadas y se evalúa la condición de salto.

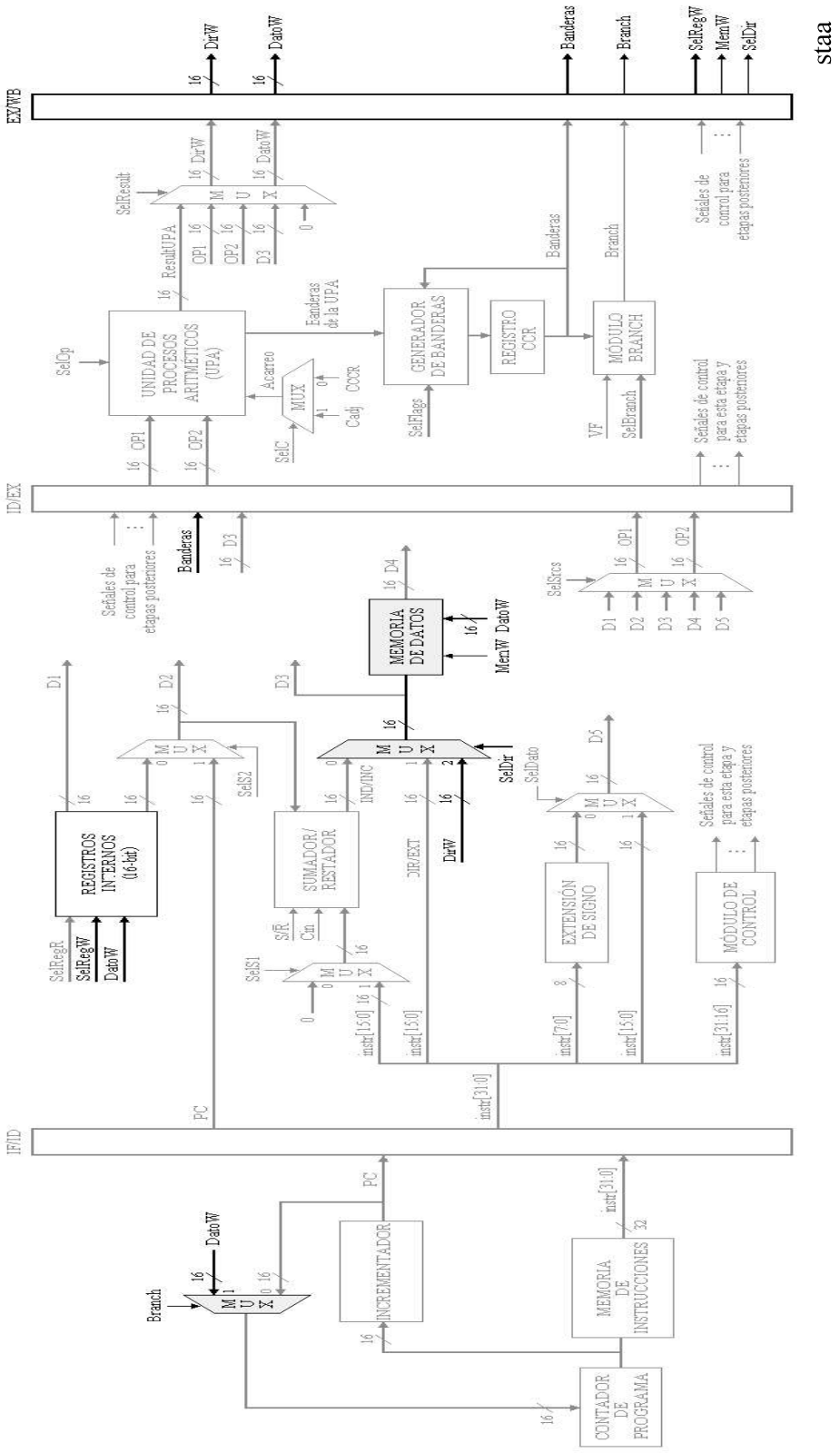


Figura 7.38. En el cuarto ciclo de reloj la instrucción STAA se encuentra en la etapa de post-escritura. En esta última etapa, el contenido del registro ACCA presente en el bus DatoW, es guardado en la localidad de memoria dada por DirW.

7.3.6 INSTRUCCIÓN BRA (Acceso Relativo)

Instrucción: BRA Desplazamiento
Operación: PC ⇐ (PC) + 1 + Desplazamiento
Código: 0020
Descripción: Salto incondicional a la dirección: PC + 1 + Desplazamiento; donde el desplazamiento es un número de 8 bits en complemento a dos.
Banderas: Ninguna bandera es afectada.

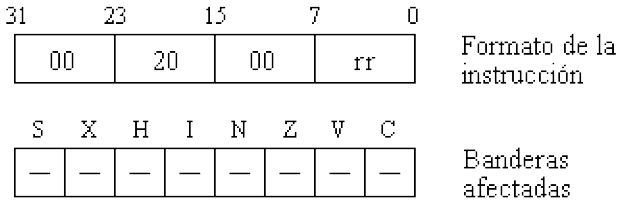


Figura 7.39. Formato de la instrucción BRA y banderas que afecta.

El comportamiento de la instrucción *bra* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *bra*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.40).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

La dirección de salto se obtiene sumando al contenido del registro PC el valor del desplazamiento. Sin embargo, antes de calcular esta dirección de salto es necesario que PC apunte a la dirección de la siguiente instrucción en memoria, es decir, a PC + 1. Una vez que PC apunta a la dirección correcta podremos sumarle el desplazamiento. Quizá ahora resulta más claro el por qué de guardar en el registro de segmentación IF/ID el PC incrementado. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	0
	SelS1	0
	$S/\overline{R}$	1
	Cin	0

	SelS2	1
	SelDato	0
	SelScrs	5
	SelDir	0
Etapa 3	SelOp	1
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	0
	SelBranch	0
	VF	0
Etapa 4	SelRegW	0
	MemW	0
	SelDir	0

Tabla 7.14. Señales de control para la instrucción BRA.

Como se mencionó anteriormente, para calcular la dirección de salto se necesitan sumar el valor del PC incrementado y el valor del desplazamiento. Dicha suma será ejecutada por la unidad de procesos aritméticos hasta la siguiente etapa, por lo tanto, la etapa de decodificación deberá enviar a la UPA los operandos adecuados.

El PC incrementado se obtiene del campo PC del registro IF/ID y el desplazamiento se extrae directamente del formato de la instrucción. El PC incrementado es seleccionado con la señal SelS2=1 y asignado al bus D2, mientras que el desplazamiento es seleccionado con SelDato=0 y asignado al bus D5. Antes de que el desplazamiento sea asignado al bus D5, es necesaria su extensión de 8 bits a 16 bits (recuerde que la UPA sólo efectúa operaciones de 16 bits), para ello, el módulo de extensión de signo repite en los 8 bits más significativos del nuevo desplazamiento el bit de signo del desplazamiento original.

El PC incrementado y el desplazamiento extendido son seleccionados con SelScrs=5 y guardados en el registro de segmentación ID/EX junto con las señales de control para las etapas posteriores. Note que para esta instrucción no se leen datos de registros internos ni de la memoria de datos (véase la figura 7.41).

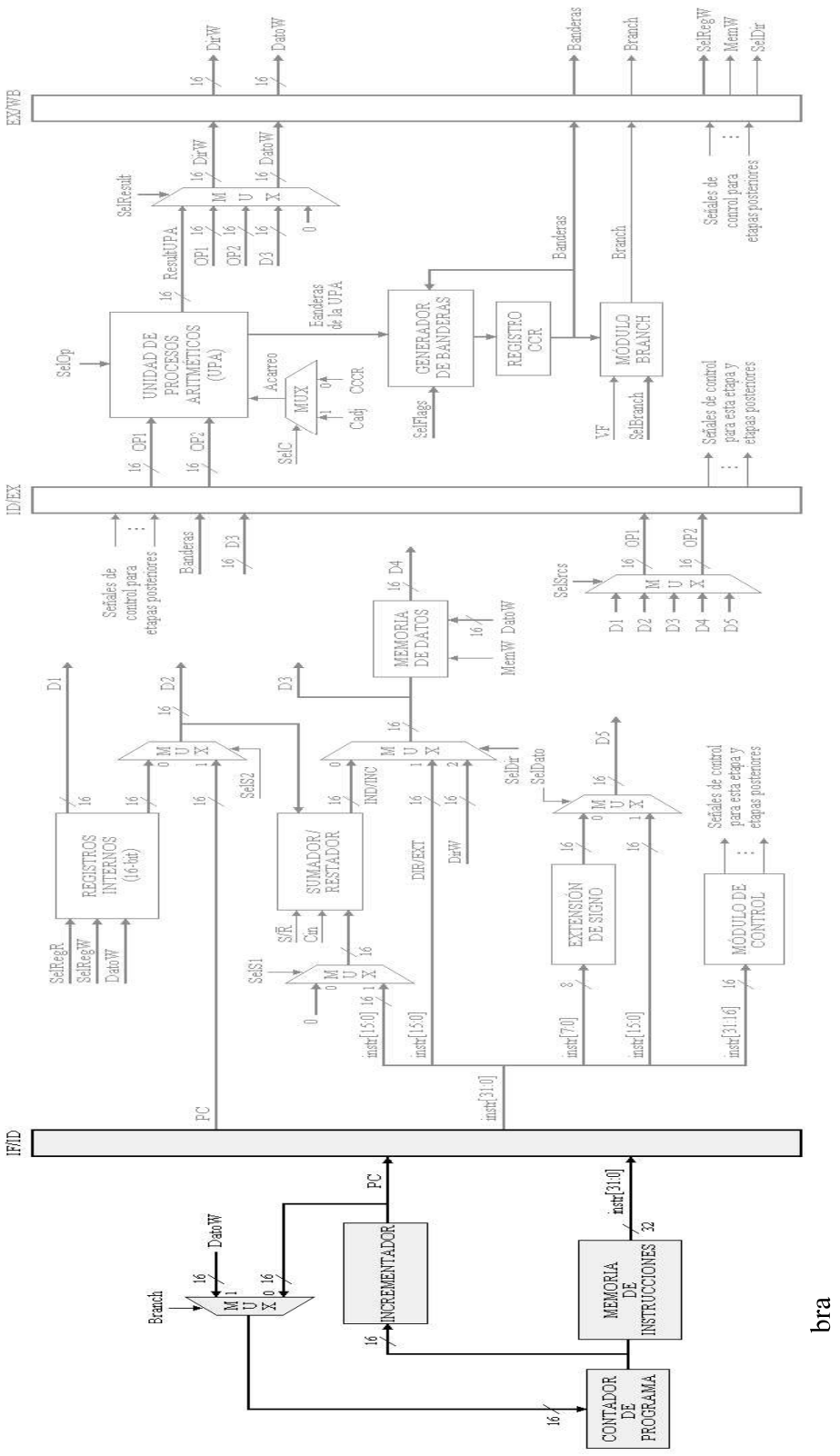
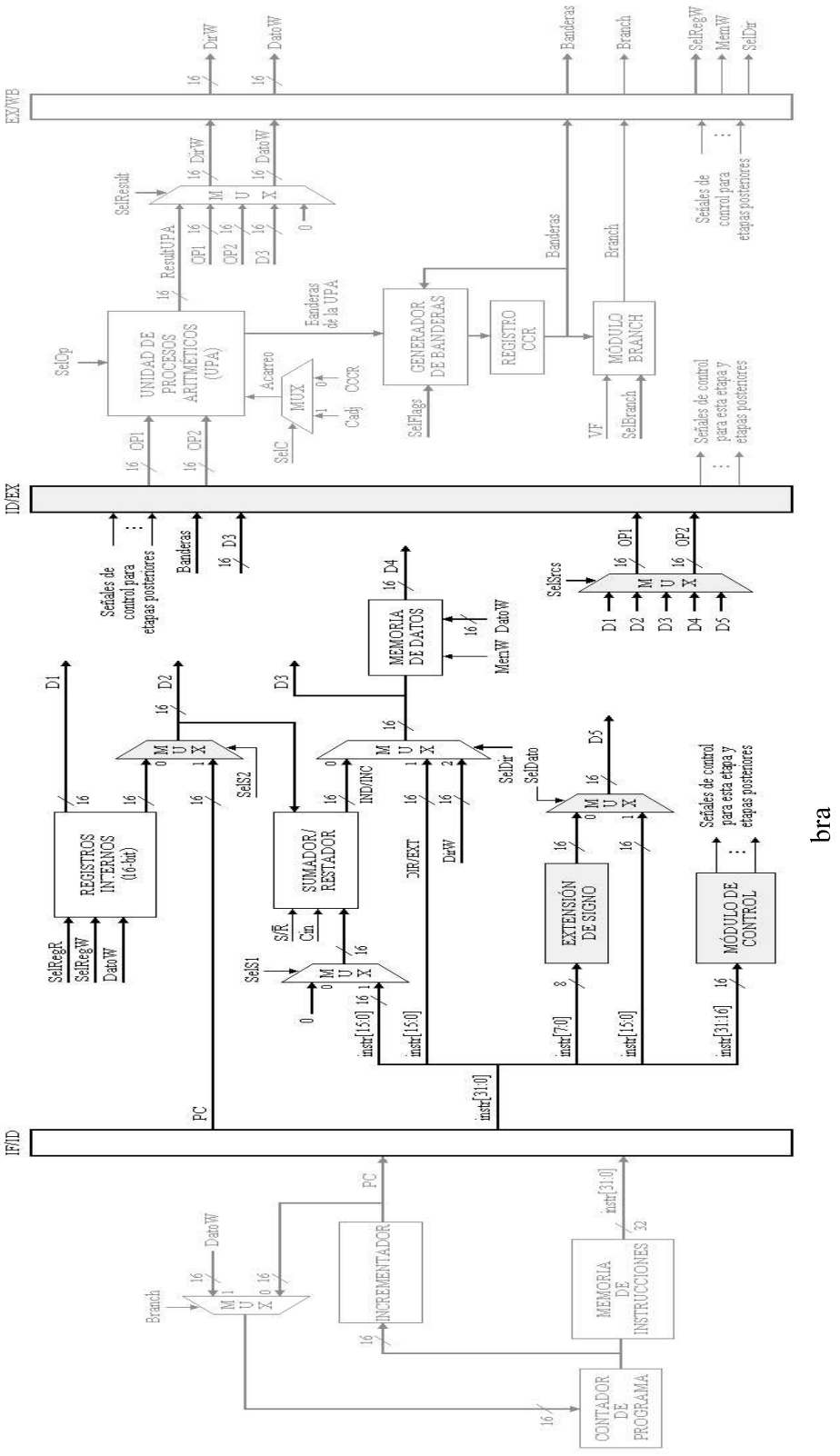


Figura 7.40. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción BRA es leída de la memoria de instrucciones.



bra

Figura 7.41. En el segundo ciclo de reloj la instrucción BRA se encuentra en la etapa de decodificación. Durante esta etapa se extiende el desplazamiento de 8 bits con signo a 16 bits, se obtiene el valor del PC incrementado y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

En esta etapa se calcula la dirección de salto, para ello, el valor del PC incrementado y el desplazamiento extendido son sumados junto con el acarreo C_{adj}, el cual se coloca a cero para evitar modificar el resultado de la suma. La selección del acarreo se realiza por medio de la señal SelC=1, mientras que la suma se ejecuta con SelOp=1.

El resultado de la suma corresponde al nuevo valor del PC, es decir, la dirección a donde el programa debe saltar, así que este valor es guardado en el campo DatoW del registro de segmentación EX/WB por medio de la señal SelResult=1.

Por otra parte, se sabe que esta instrucción es una instrucción de salto incondicional, es decir, se debe ejecutar el salto, o visto de otra forma, la señal Branch debe colocarse a uno. Para obligar a que Branch valga uno hay que asegurar que el valor de la condición de salto sea igual al valor de VF. Si se utiliza la condición de salto SelBranch=0, esto es, se intenta comparar contra el valor de cero, entonces, la única forma de activar a Branch es colocando a cero a VF.

Finalmente, la instrucción *bra* no modifica valores de banderas en el registro CCR, por lo tanto, SelFlags se coloca a cero (véase la figura 7.42).

Etapa 4. Post-escritura

Las instrucciones de salto no guardan resultados en los registros internos ni en la memoria de datos, por lo tanto, las operaciones de escritura en ellos son desactivadas. La única señal que se utiliza en esta cuarta etapa es la señal Branch que se generó en la etapa anterior, la cual selecciona la dirección de salto contenida en el campo DatoW del registro de segmentación EX/WB, para guardarla en el registro contador de programa (véase la figura 7.43).

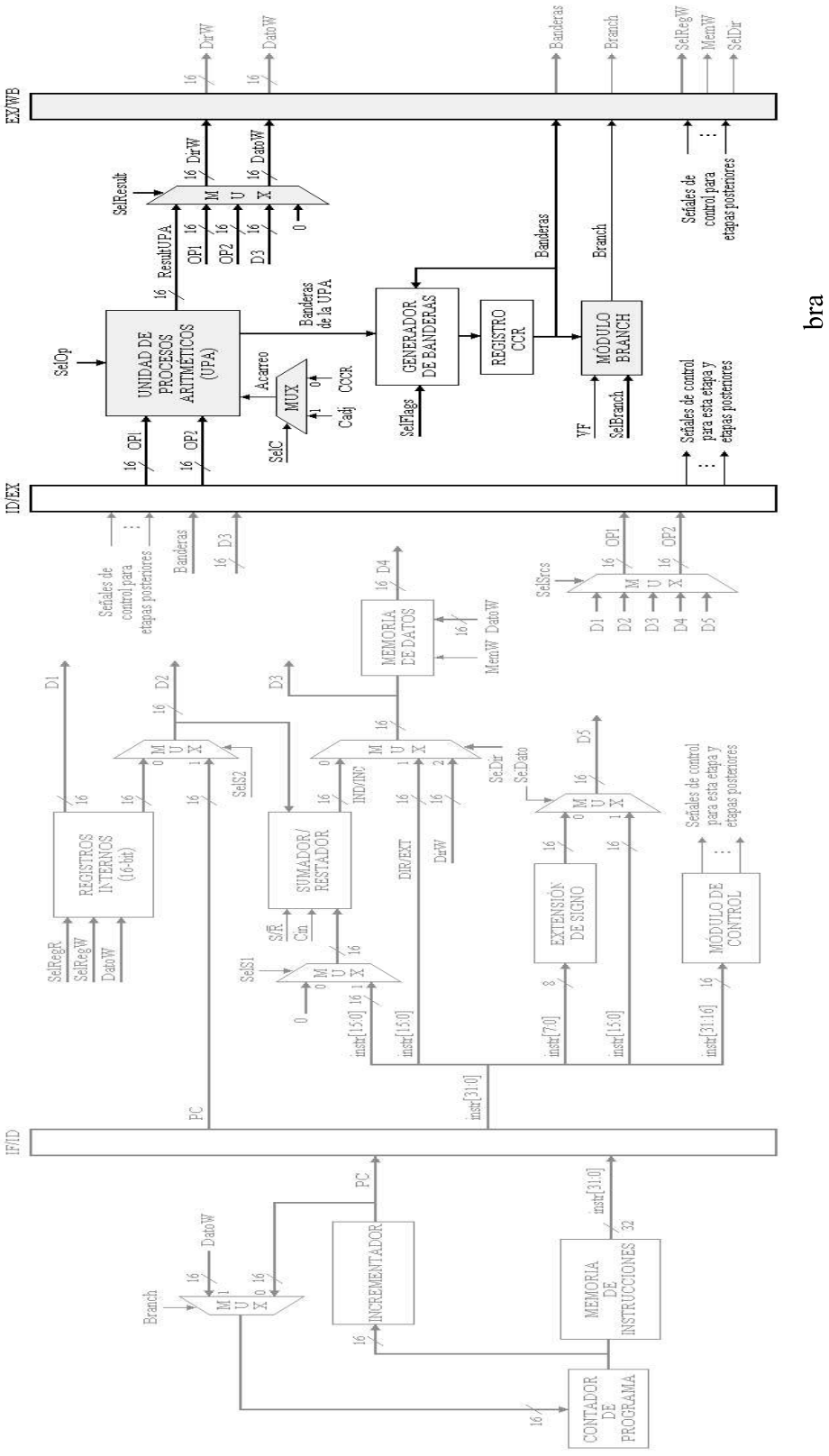


Figura 7.42. En el tercer ciclo de reloj la instrucción BRA se encuentra en la etapa de ejecución. Durante esta etapa la UPA calcula la dirección de salto y el módulo Branch evalúa la condición de salto.

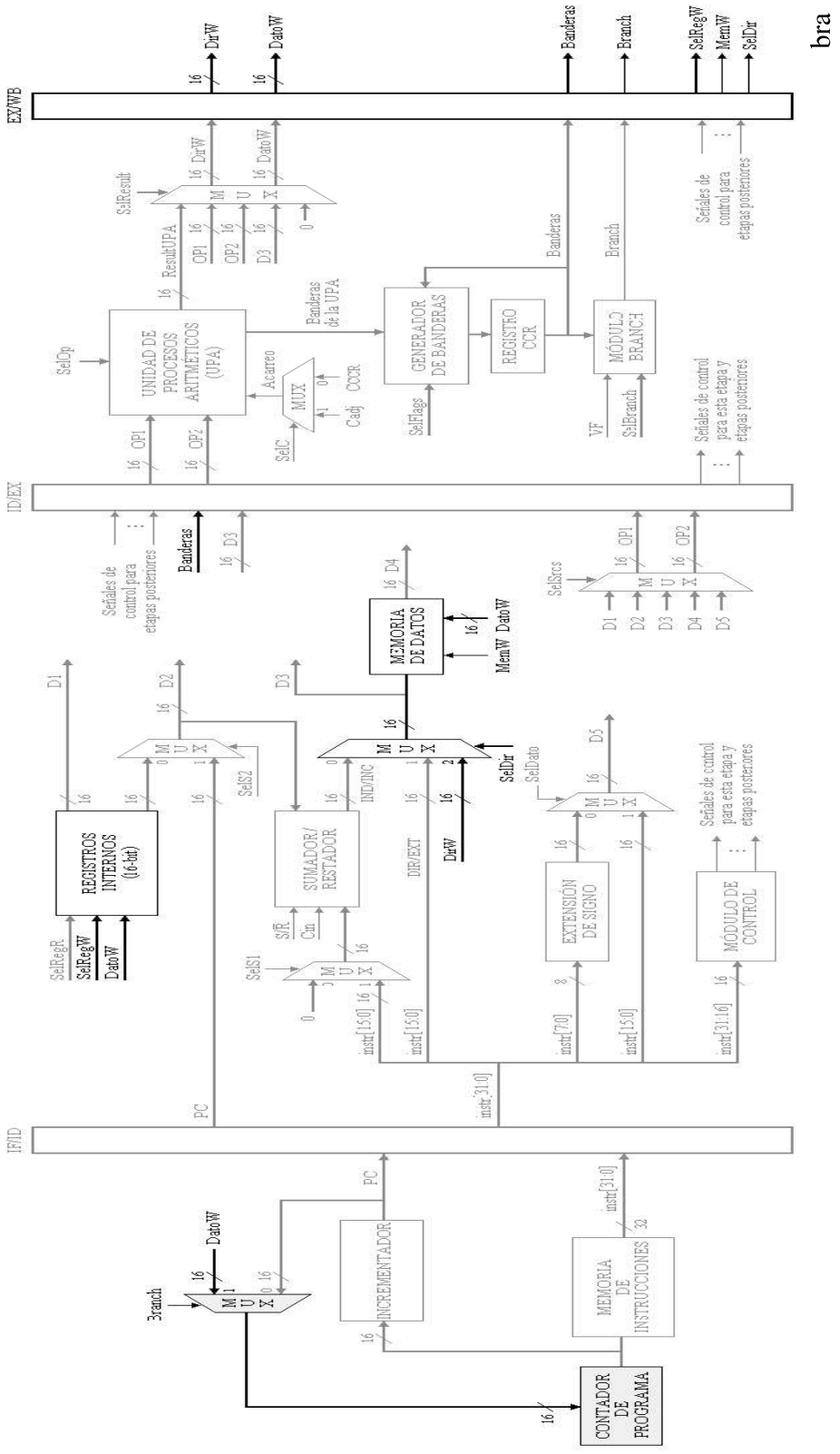


Figura 7.43. En el cuarto ciclo de reloj la instrucción BRA se encuentra en la etapa de post-escritura. En esta última etapa, la dirección de salto, contenida en el bus DatoW, es guardada en el registro contador de programa.

7.3.7 RESUMEN DE INSTRUCCIONES

A continuación se presenta una tabla con las señales de control para las instrucciones vistas anteriormente y para otras instrucciones que tienen soporte en esta arquitectura segmentada.

Instrucciones		*Señales de Control*																		
		Etapa 2								Etapa 3							Etapa 4		
Mnemónico	Instr[31:16]	SelRegR	SelS1	S/$\overline{R}$	Cin	SelS2	SelDato	SelScrs	SelDir	SelOp	SelResult	SelC	Cadj	SelFlags	SelBranch	VF	SelRegW	MemW	SelDir
aba (INH)	001B	1	0	1	0	0	1	1	0	1	1	1	0	2	0	1	1	0	0
aby (INH)	183A	3	0	1	0	0	1	1	0	1	1	1	0	0	0	1	3	0	0
adcb (DIR)	00D9	5	0	1	0	0	1	2	1	1	1	0	0	2	0	1	4	0	0
anda (EXT)	00B4	4	0	1	0	0	1	2	1	3	1	1	0	1	0	1	1	0	0
andb (EXT)	00F4	5	0	1	0	0	1	2	1	3	1	1	0	1	0	1	4	0	0
andb (IND,X)	00E4	2	1	1	0	0	1	2	0	3	1	1	0	1	0	1	4	0	0
asl (IND,Y)	1868	A	1	1	0	0	1	4	0	6	1	1	0	3	0	1	0	1	2
asr (IND,X)	0067	9	1	1	0	0	1	4	0	7	1	1	0	3	0	1	0	1	2
asrb (INH)	0057	5	0	1	0	0	1	1	0	7	1	1	0	3	0	1	4	0	0
bcc (REL)	0024	0	0	1	0	1	0	5	0	1	1	1	0	0	1	0	0	0	0
bita (IMM)	0085	4	0	1	0	0	1	3	0	3	0	1	0	1	0	1	0	0	0
bra (REL)	0020	0	0	1	0	1	0	5	0	1	1	1	0	0	0	0	0	0	0
cba (INH)	0011	1	0	1	0	0	1	1	0	2	0	1	1	3	0	1	0	0	0
clr (EXT)	007F	0	0	1	0	0	1	2	1	3	1	1	0	3	0	1	0	1	2
cmpb (DIR)	00D1	5	0	1	0	0	1	2	1	2	0	1	1	3	0	1	0	0	0
com (IND,X)	0063	9	1	1	0	0	1	2	0	2	1	1	0	B	0	1	0	1	2
comb (INH)	0053	5	0	1	0	0	1	1	0	8	1	1	0	B	0	1	4	0	0
cpy (IND,Y)	18AC	A	1	1	0	0	1	6	0	2	0	1	1	3	0	1	0	0	0
dec (EXT)	007A	0	0	1	0	0	1	2	1	8	1	1	0	C	0	1	0	1	2
des (INH)	0034	B	0	1	0	0	1	1	0	8	1	1	0	0	0	1	6	0	0
incb (INH)	005C	5	0	1	0	0	1	1	0	1	1	1	1	C	0	1	4	0	0
ldaa (IMM)	0086	0	0	1	0	0	1	3	0	4	1	1	0	1	0	1	1	0	0
ldy (IND,Y)	18EE	A	1	1	0	0	1	2	0	4	1	1	0	1	0	1	3	0	0
neg (EXT)	0070	0	0	1	0	0	1	2	1	2	1	1	1	3	0	1	0	1	2
nop (INH)	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
ror (IND,Y)	1866	A	1	1	0	0	1	4	0	B	1	0	0	3	0	1	0	1	2
sbcb (IMM)	00C2	5	0	1	0	0	1	3	0	2	1	0	0	3	0	1	4	0	0
staa (EXT)	00B7	4	1	1	0	0	1	1	0	4	1	1	0	1	0	1	0	1	2
stx (IND,X)	00EF	9	1	1	0	0	1	1	0	4	1	1	0	1	0	1	0	1	2
tst (IND,Y)	186D	A	1	1	0	0	1	2	0	8	0	1	1	3	0	1	0	0	0
tsx (INH)	0030	B	0	1	0	0	1	1	0	1	1	1	1	0	0	1	2	0	0
tys (INH)	1835	A	0	1	0	0	1	1	0	8	1	1	0	0	0	1	6	0	0

Tabla 7.15. Señales de control para algunas instrucciones del 68HC11.

7.3.8 EJECUCIÓN DE MÚLTIPLES INSTRUCCIONES

Hasta el momento sólo hemos visto la ejecución de instrucciones de manera individual, por lo tanto, no hemos observado el potencial de las arquitecturas segmentadas.

El siguiente ejemplo intenta mostrar este potencial mediante la ejecución simultánea de cuatro instrucciones: ldaa, ldab, inx y staa. Tome en cuenta las siguientes condiciones iniciales: la dirección en memoria de la primera instrucción a ejecutar es 0x0400, el contenido del registro IX vale 0x2232, y la señal Branch vale cero para el primer ciclo de reloj. Además, considere que las instrucciones anteriormente ejecutadas en el cauce no almacenan resultados en registros ni en memoria.

```
0x0400    ldaa   #0080
0x0401    ldab   #4000
0x0402    inx
0x0403    staa   1000
```

El ejemplo es muy sencillo, la primera instrucción carga en el registro ACCA un dato inmediato de 16 bits en formato hexadecimal; la segunda instrucción carga en el registro ACCB otro dato inmediato; la tercera instrucción incrementa en una unidad el contenido del registro IX; y la última instrucción guarda en la dirección de memoria 0x1000 el contenido del registro ACCA. La secuencia de ejecución de estas cuatro instrucciones se presenta en el siguiente diagrama de múltiples ciclos de reloj (véase la figura 7.44).

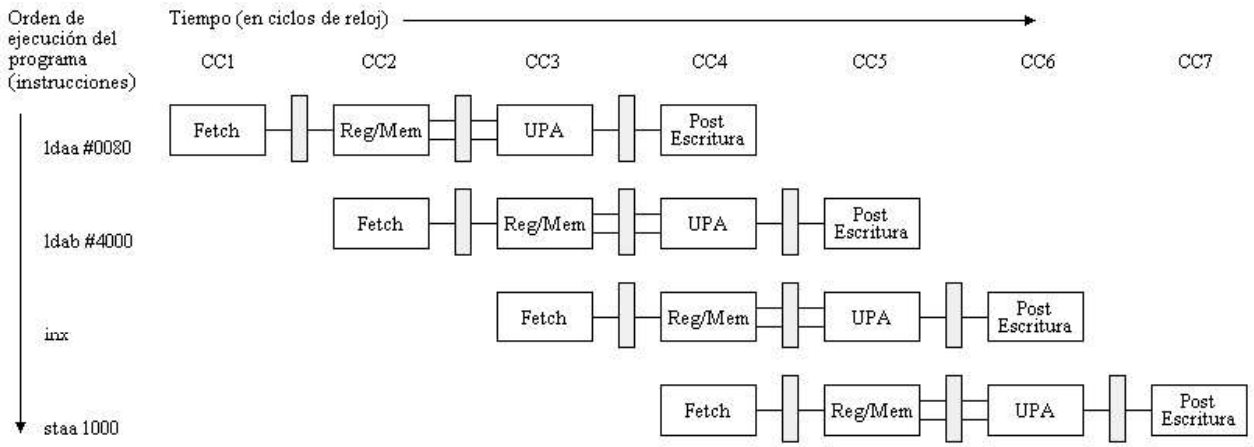


Figura 7.44. Diagrama de segmentación de múltiples ciclos de reloj para las dos instrucciones del ejemplo.

También se anexan los diagramas de un sólo ciclo de reloj para observar en detalle lo que ocurre en cada etapa de la segmentación (véanse las figuras 7.45 a la 7.51). En estos últimos diagramas se resaltan los componentes que intervienen en el ciclo de reloj descrito.

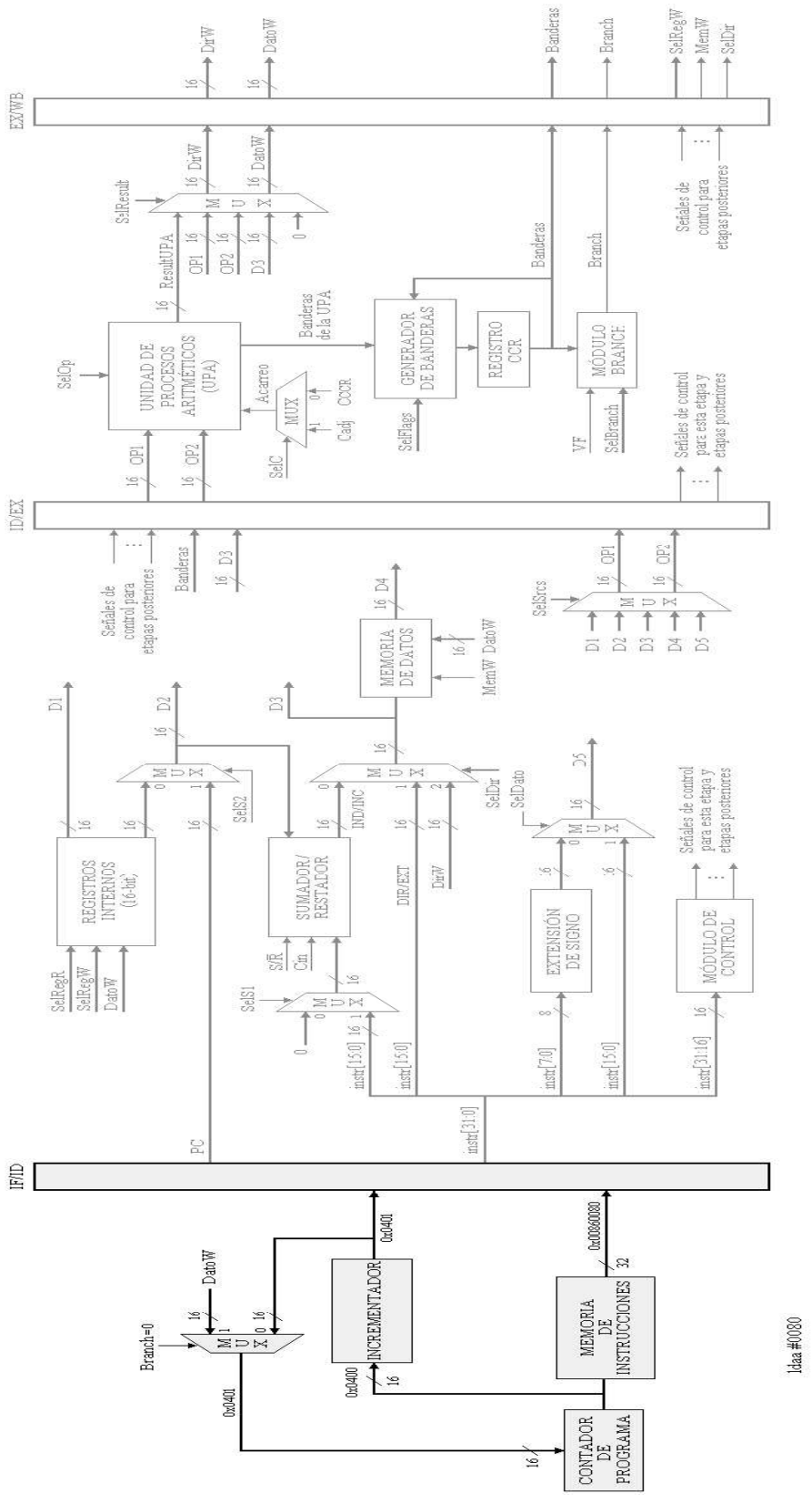


Figura 7.45. Diagrama de segmentación correspondiente al ciclo de reloj 1. En este ciclo de reloj se lee la instrucción *ldaa* de la dirección 0x0400 de la memoria de instrucciones.

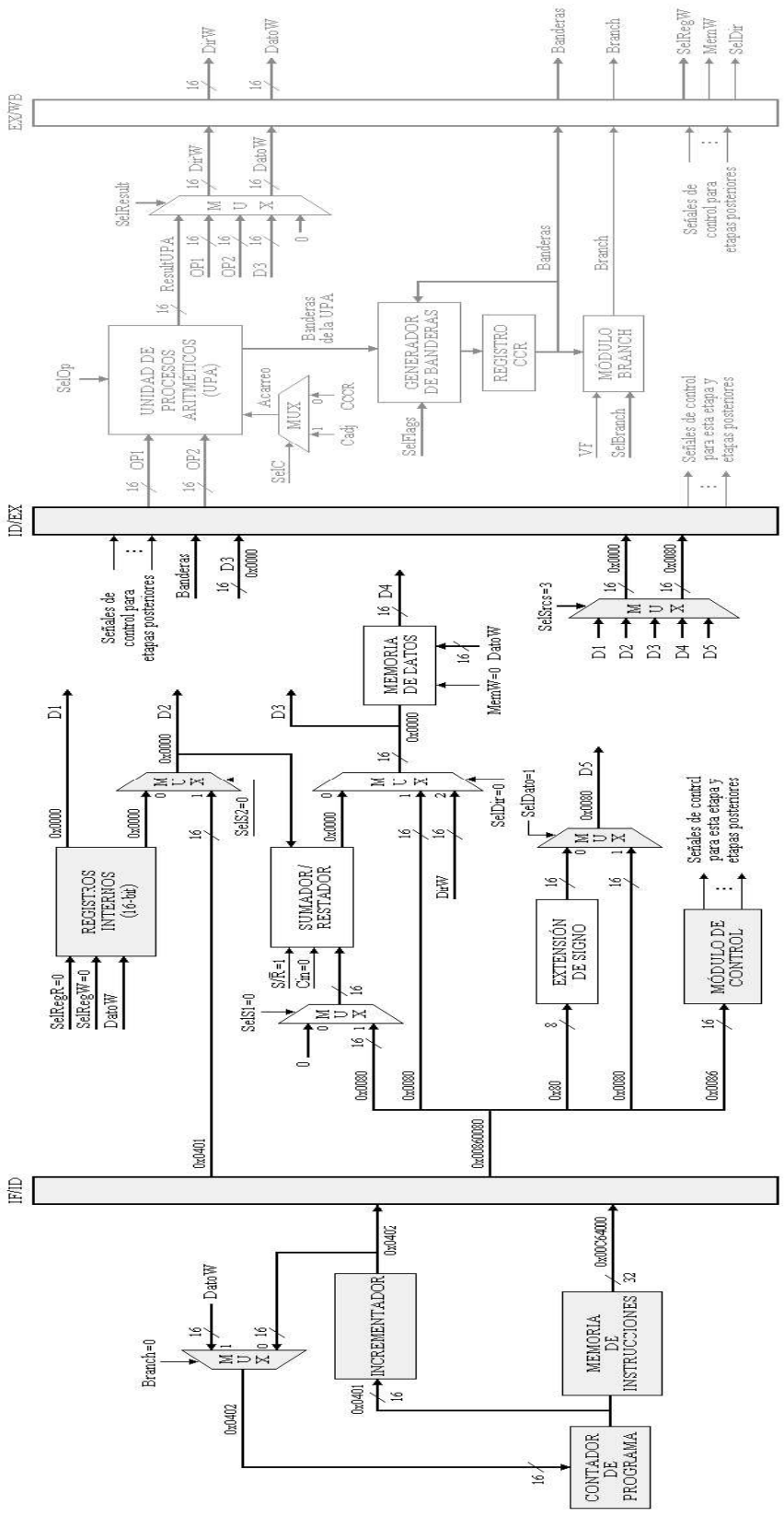


Figura 7.46. Diagrama de segmentación correspondiente al ciclo de reloj 2. En este ciclo de reloj la instrucción *ldaa* está siendo decodificada; los operandos requeridos por esta instrucción son leídos y guardados en el registro de segmentación ID/EX junto con las señales de control para las etapas posteriores. En el mismo ciclo de reloj, en la etapa 1, se lee de la memoria la instrucción *ldab*.

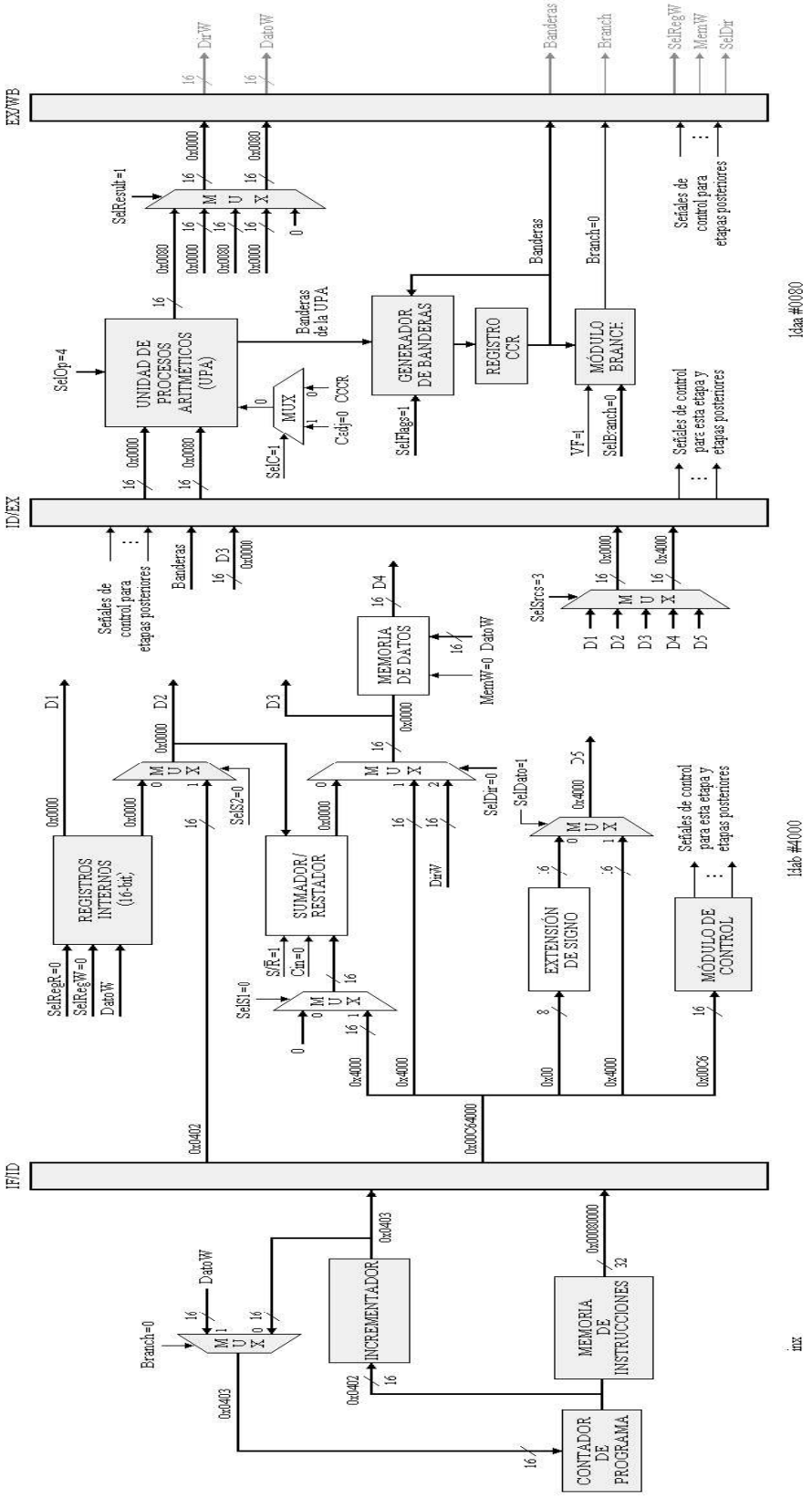


Figura 7.47. Diagrama de segmentación correspondiente al ciclo de reloj 3. La instrucción *ldaa* ahora se encuentra en la etapa 3; en esta etapa, la UPA calcula los resultados con base en los operandos y en las señales de control que le fueron proporcionados por la etapa anterior. En la etapa 2, la instrucción *ldab* es decodificada y son obtenidos los operandos que esta instrucción necesitará en las etapas siguientes. Finalmente, en la etapa 1, se lee la instrucción *inx* de la memoria de instrucciones.

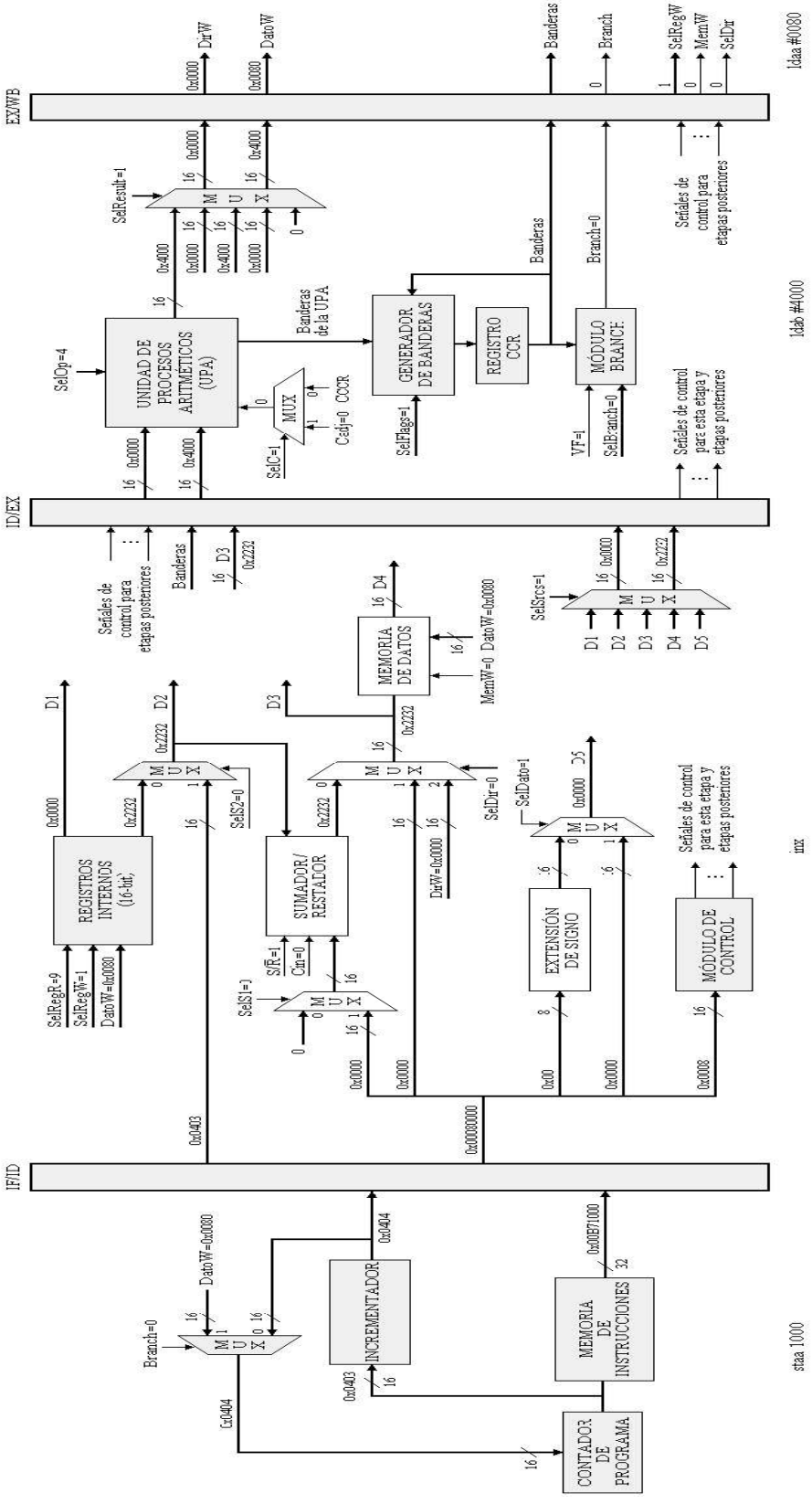


Figura 7.48. Diagrama de segmentación correspondiente al ciclo de reloj 4. En este ciclo de reloj el cauce está a toda su capacidad ejecutando una instrucción diferente en cada una de sus etapas. En la etapa de post-escritura, la instrucción *ldaa* guarda el dato inmediato 0x0080 en el registro ACCA; en la etapa de ejecución, se calculan resultados para la instrucción *ldab*; en la etapa de decodificación, se traen los operandos y las señales de control para la instrucción *inx*; y en la etapa de lectura de instrucción, se lee de la memoria de instrucciones la instrucción *staa*.

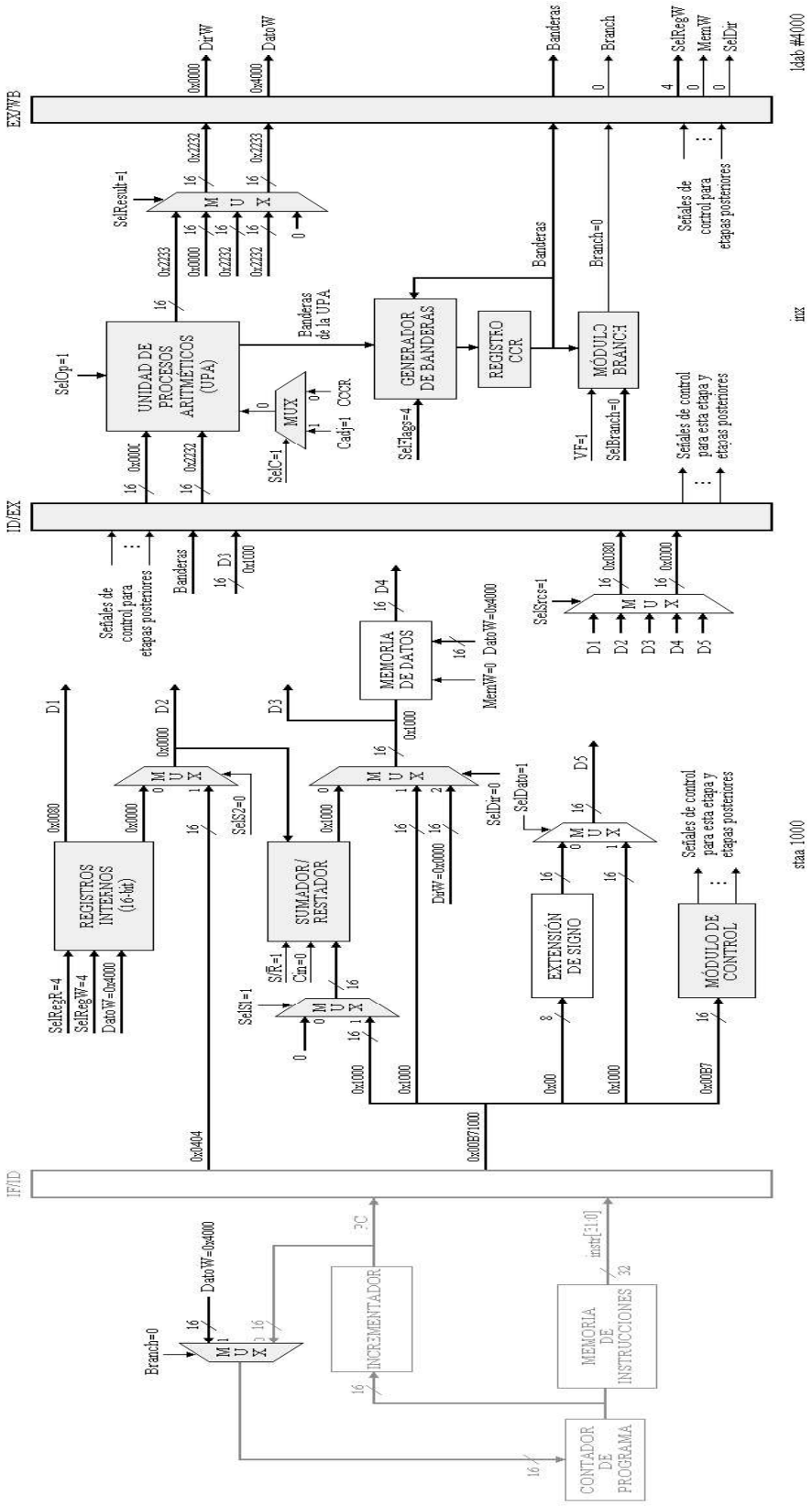


Figura 7.49. Diagrama de segmentación correspondiente al ciclo de reloj 5. En la etapa de post-escritura, la instrucción *ldab* guarda el dato inmediato 0x4000 en el registro ACCB; en la etapa de ejecución, la instrucción *inx* calcula el incremento; y en la etapa de decodificación, se traen los operandos y las señales de control para la instrucción *staa*. Observe que de tener más instrucciones, en la etapa 1, se estaría leyendo de memoria la siguiente instrucción a ejecutar.

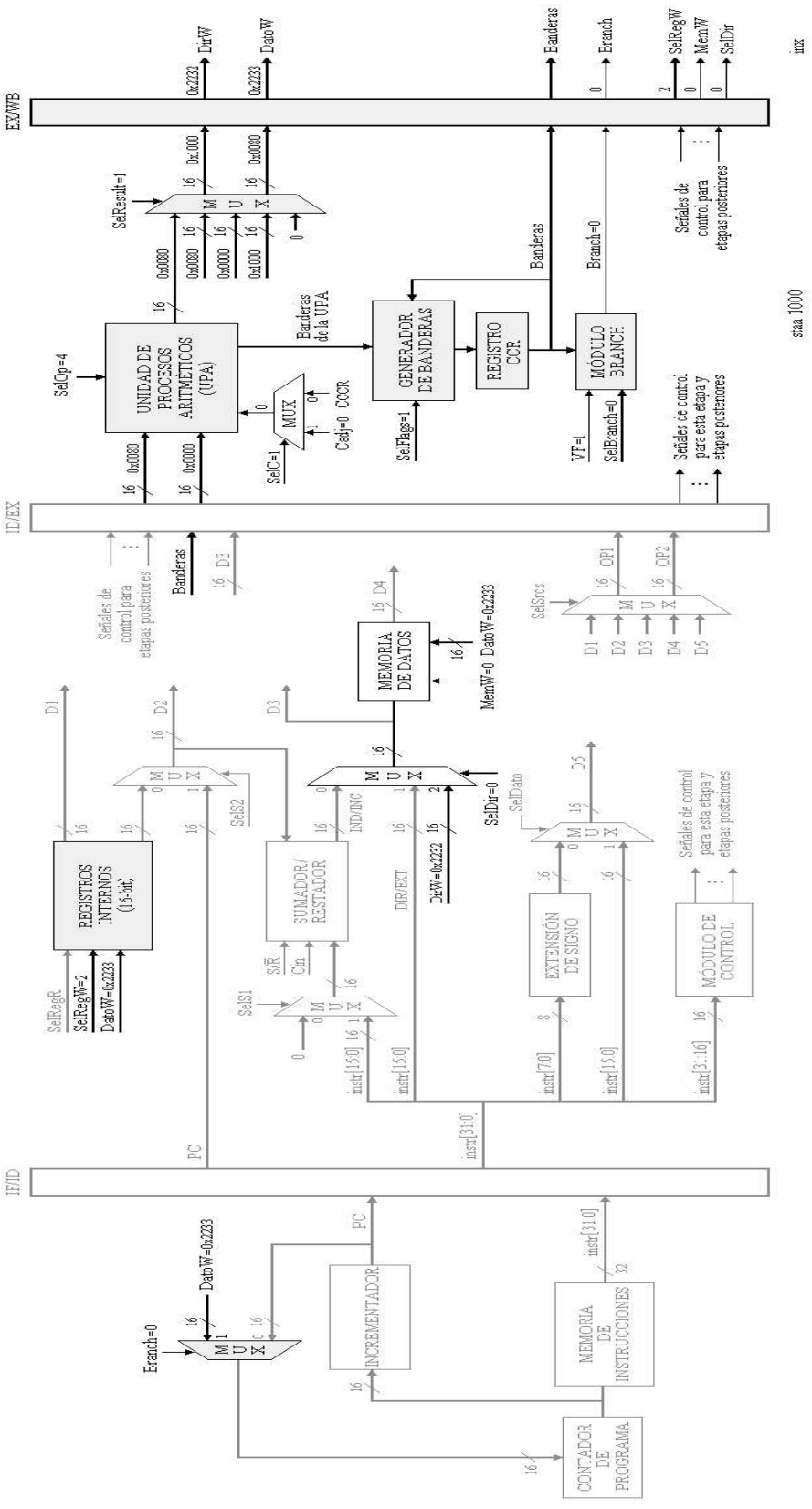


Figura 7.50. Diagrama de segmentación correspondiente al ciclo de reloj 6. En la última etapa, la instrucción *inx* guarda el valor incrementado en el registro IX; mientras, la instrucción *staa* se encuentra en la etapa de ejecución calculando resultados.

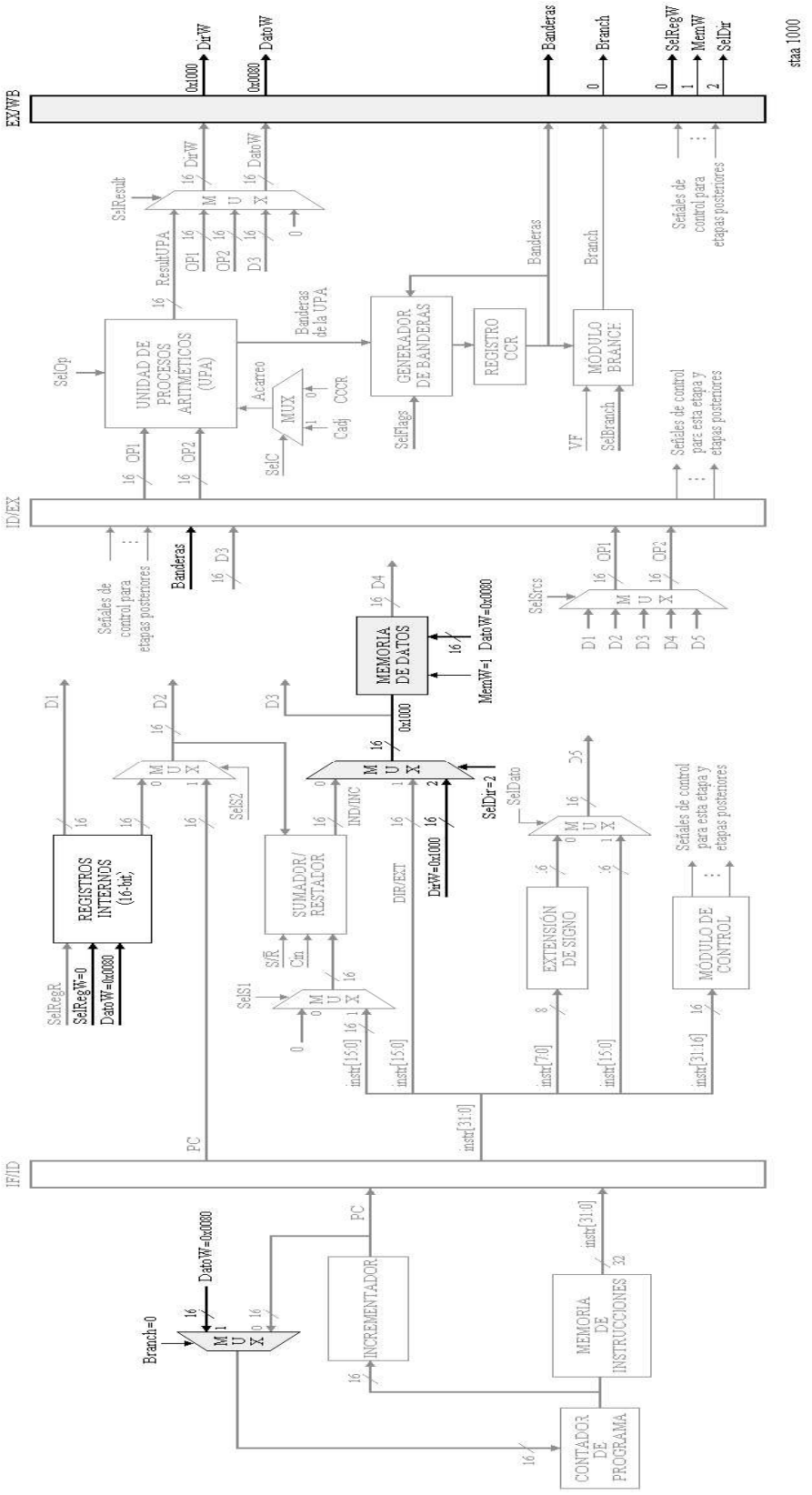


Figura 7.51. Diagrama de segmentación correspondiente al ciclo de reloj 7. En este último ciclo de reloj la instrucción *staa* se encuentra en la etapa de post-escritura guardando, en la dirección 0x1000 de la memoria de datos, el valor leído del registro ACCA.

7.4 RIESGOS POR DEPENDENCIAS DE DATOS

El ejemplo de la sección anterior muestra la potencia de una arquitectura segmentada para un caso ideal; es decir, las instrucciones que se ejecutaron eran totalmente independientes una de la otra, en otras palabras, ninguna de ellas utilizaba los resultados calculados por una instrucción anterior. Ahora es el momento de dejar los casos ideales y observar qué ocurre con los programas reales, por ejemplo, examine los siguientes dos ejemplos.

Riesgos por dependencias de datos: Ejemplo 1

```
aba          ; El resultado obtenido por esta instrucción es escrito en el registro ACCA
anda #6011   ; Uno de los operandos utilizados por esta instrucción, el valor de ACCA,
             ; depende del resultado guardado por la instrucción aba
oraa #FF00   ; Uno de los operandos utilizados por esta instrucción, el valor de ACCA,
             ; depende del resultado guardado por la instrucción anda
staa 1000    ; El dato guardado por staa depende del resultado escrito por oraa
```

Observe que las últimas tres instrucciones son dependientes de los resultados calculados por las instrucciones anteriores, entonces, ¿cómo afecta la dependencia de datos a la ejecución de instrucciones en el cauce?. Para responder esta interrogante nos apoyaremos en el siguiente diagrama de múltiples ciclos de reloj, que muestra cómo se ejecuta la secuencia de instrucciones del ejemplo 1 sobre el modelo de arquitectura de la figura 7.12.

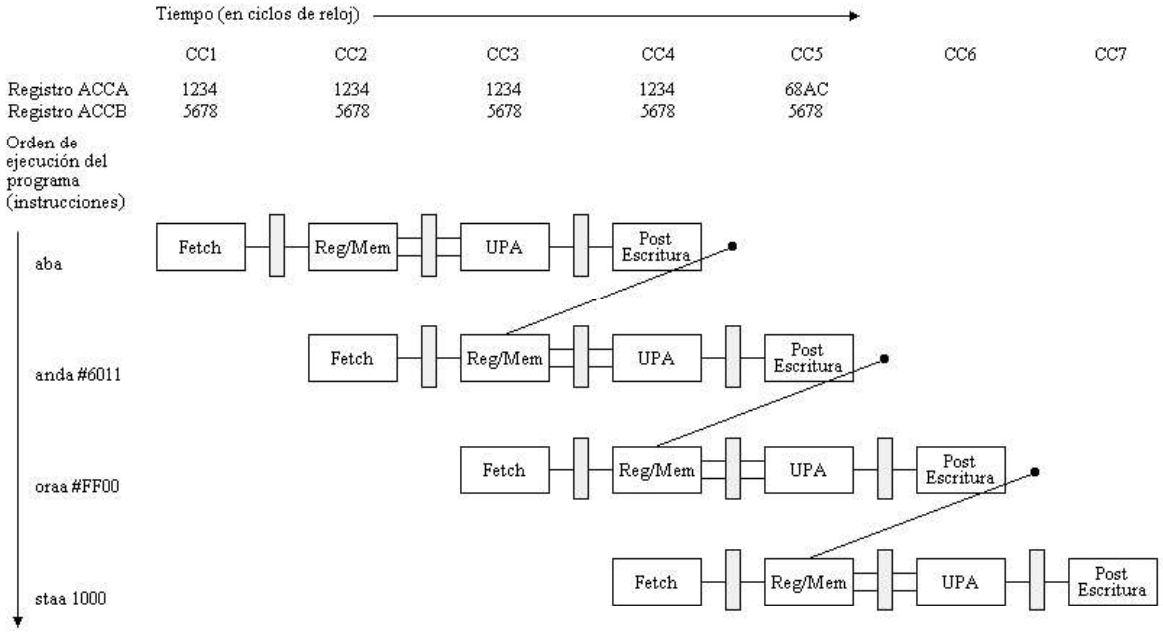


Figura 7.52. Diagrama de múltiples ciclos de reloj para el ejemplo 1. Las líneas trazadas entre las etapas muestran los riesgos por dependencias de datos; estas líneas se trazan desde caminos de datos superiores hacia inferiores, esto es, desde instrucciones anteriores hacia posteriores.

Antes de comenzar a ejecutar la instrucción *aba*, suponga que los registros ACCA y ACCB contienen los valores hexadecimales 0x1234 y 0x5678, respectivamente. En la figura 7.52 los valores de estos registros se muestran en la parte superior del diagrama.

Ahora observe detenidamente el diagrama. La instrucción *aba* requiere de cuatro ciclos de reloj para tener listo el resultado de la suma; este resultado es escrito en el registro ACCA durante el ciclo de reloj CC4, y hasta el ciclo de reloj CC5 podrá ser leído por otra instrucción.

Por otra parte, la instrucción *anda* necesita el resultado de la instrucción anterior para calcular su propio resultado. Observe que cuando la instrucción *anda* lee el contenido del registro ACCA durante el ciclo de reloj CC3, ACCA tiene un valor que no corresponde al resultado de la instrucción *aba*, por lo tanto, la instrucción *anda* ejecutará la operación AND sobre operandos incorrectos. Para que la instrucción *anda* lea el operando adecuado debe esperar hasta el ciclo de reloj CC5, pues es cuando el registro ACCA tiene el resultado correcto. A estas situaciones en donde se leen datos que serán escritos más tarde se les denominan riesgos por dependencias de datos o conflictos por dependencias de datos (data hazards), y son la razón por la complican el diseño de arquitecturas segmentadas de alto rendimiento.

Algo similar ocurre para la instrucción *oraa*, necesita del resultado escrito por la instrucción *anda* para operar correctamente; este resultado se escribe durante el ciclo de reloj CC5 y hasta el ciclo de reloj CC6 estará disponible para lectura. De esta manera, cuando la instrucción *oraa* lee el contenido del registro ACCA en el ciclo de reloj CC4, éste aún conserva el valor 0x1234, el cual, en ese mismo ciclo de reloj, está siendo actualizado con el resultado de la instrucción *aba*. Nuevamente se tiene una situación de riesgo por dependencia de datos.

Finalmente, la instrucción *staa* guarda en memoria el resultado obtenido por la instrucción *oraa*. Un nuevo riesgo por dependencia de datos se hace presente, ya que la instrucción *staa* lee, durante el ciclo de reloj CC5, el contenido del registro ACCA con el resultado de la instrucción *aba* y no con el resultado de la instrucción *oraa*, el cual será escrito hasta el ciclo de reloj CC7.

Note que al final de la secuencia de instrucciones la única instrucción que se calculó correctamente fue *aba* y las demás, debido a las dependencias de datos, guardaron y obtuvieron resultados incorrectos.

Riesgos por dependencias de datos: Ejemplo 2

```
ldab #1234      ; Carga en el registro ACCB el dato inmediato 0x1234
ldaa #5678      ; Carga en el registro ACCA el dato inmediato 0x5678
aba             ; Los operandos que utiliza esta instrucción dependen de los valores escritos
                ; por las instrucciones ldab y ldaa
abx             ; Uno de los operandos, el valor de ACCB, depende de la instrucción ldab
```

A continuación se analiza esta secuencia de instrucciones utilizando el diagrama de múltiples ciclos de reloj mostrado en la figura 7.53.

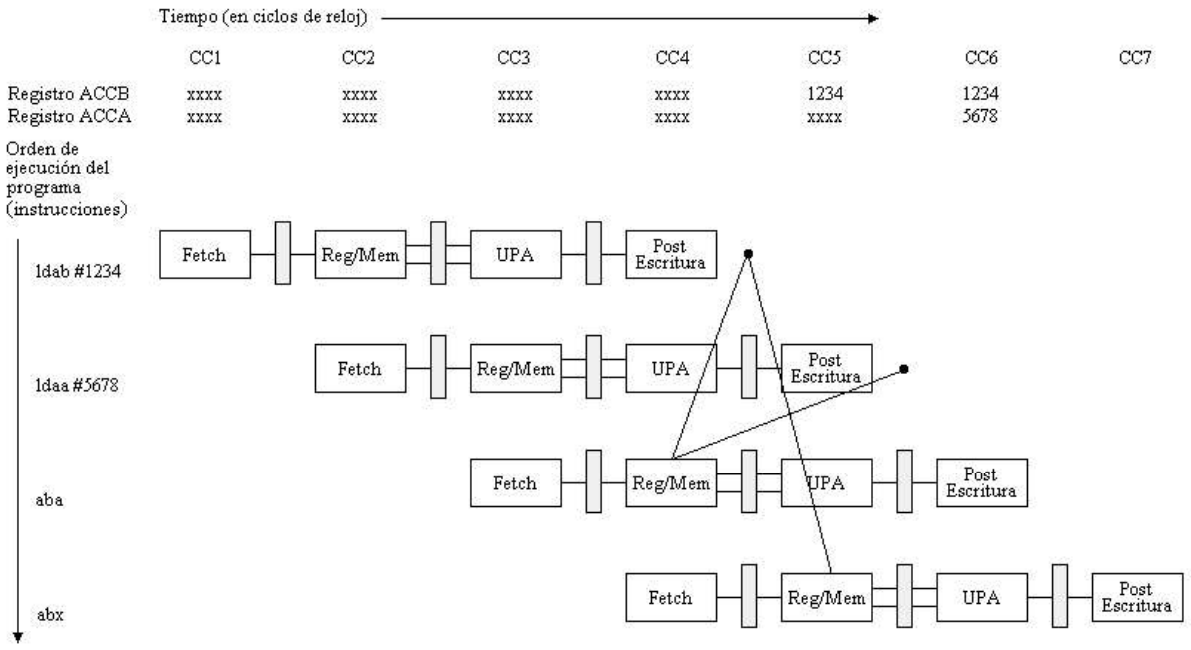


Figura 7.53. Diagrama de múltiples ciclos de reloj para el ejemplo 2. Las líneas trazadas entre las etapas muestran las dependencias de datos, sin embargo, sólo las que retroceden en el tiempo representan riesgos.

La instrucción *ldab* carga en el registro ACCB el valor hexadecimal 0x1234; la carga es ejecutada durante el ciclo de reloj CC4, pero hasta el ciclo de reloj CC5 el nuevo valor estará disponible para lectura. De manera similar, la instrucción *ldaa* carga en el registro ACCA el valor hexadecimal 0x5678, el cual estará disponible para lectura en el ciclo de reloj CC6. Note que la instrucción *ldaa* no utiliza ningún resultado calculado por la instrucción *ldab*, por lo tanto, decimos que estas instrucciones son independientes.

Por otra parte, la instrucción *aba* opera sobre los valores contenidos en los registros ACCB y ACCA, los cuales son modificados por las instrucciones *ldab* y *ldaa*; es decir, la instrucción *aba* es dependiente de las instrucciones *ldab* y *ldaa*. Observe que cuando la instrucción *aba* lee sus operandos durante el ciclo de reloj CC4, éstos aún no han sido escritos por las instrucciones anteriores; por lo tanto, los operandos leídos en este tiempo no corresponden a los valores adecuados, en otras palabras, se tiene una situación de riesgo por dependencia de datos.

Por último, la instrucción *abx* también depende de la instrucción *ldab*, sin embargo, cuando la instrucción *abx* está en busca de sus operandos en el ciclo de reloj CC5, en particular del contenido del registro ACCB, éste ya está disponible para lectura.

7.5 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS

Existen varios esquemas que nos permiten resolver los riesgos por dependencias de datos. Algunos de ellos son implantados en software y otros en hardware, sin embargo, para nosotros, los esquemas en hardware serán los que tengan mayor importancia.

7.5.1 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS POR MEDIO DE SOFTWARE

Este método consiste en dar reglas al compilador de manera que no genere secuencias de instrucciones como las secuencias de los ejemplos de la sección anterior. Por ejemplo, para la secuencia del ejemplo 1, el compilador podría insertar dos instrucciones independientes entre las instrucciones *aba* y *anda*, haciendo que desaparezca el riesgo. Algo similar se haría para eliminar el riesgo entre las instrucciones *anda* y *oraa*, y las instrucciones *oraa* y *staa*.

Cuando no se puedan encontrar instrucciones independientes, el compilador podría insertar instrucciones *nop* garantizando así la independencia de datos entre instrucciones. La abreviatura *nop* significa no operación, porque esta instrucción no lee ningún registro, no modifica ningún dato y no escribe ningún resultado.

A continuación se presentan las secuencias de instrucciones para los ejemplos 1 y 2 de la sección 7.4 utilizando este esquema de control de riesgos por dependencias de datos.

Control de riesgos por medio de software: Ejemplo 1

```
aba          ; Suma ACCA y ACCB, el resultado se guarda en ACCA
nop          ; La dependencia de datos entre las instrucciones aba y anda se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
anda #6011   ; AND lógica entre el valor 0x6011 y el resultado de aba
nop          ; La dependencia de datos entre las instrucciones anda y oraa se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
oraa #FF00   ; OR lógica entre el valor 0xFF00 y el resultado de anda
nop          ; La dependencia de datos entre las instrucciones oraa y staa se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
staa 1000    ; Guarda el resultado obtenido por la instrucción oraa
```

Control de riesgos por medio de software: Ejemplo 2

```
ldab #1234   ; Carga en ACCB el valor 0x1234
ldaa #5678   ; Carga en ACCA el valor 0x5678
nop          ; La dependencia de datos entre las instrucciones ldab, ldaa y aba se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
aba          ; Suma ACCA y ACCB, el resultado se guarda en ACCA
abx          ; Suma IX y ACCB, el resultado se guarda en IX
```

Aunque las nuevas secuencias de instrucciones funcionan adecuadamente en la arquitectura segmentada de la figura 7.12, estas instrucciones *nop* ocupan ciclos de reloj que no realizan trabajo útil. Idealmente, el compilador encontrará instrucciones para ayudar al cálculo en lugar de insertar instrucciones inactivas.

7.5.2 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS POR MEDIO DE DETENCIONES

El esquema más sencillo para resolver los riesgos por dependencias de datos en hardware es detener las instrucciones en el cauce hasta que se resuelva el riesgo. Este tipo de detenciones se conocen con el sobrenombre de burbujas (bubbles); con esta estrategia, primero se detecta un riesgo por dependencia de datos, y después se detienen las instrucciones en el cauce (se insertan burbujas) hasta que se resuelve el riesgo.

Si realiza una inspección más estricta de la arquitectura segmentada mostrada en la figura 7.12, notará que un riesgo por dependencia de datos se presenta cuando una instrucción trata de leer en su etapa 2 el mismo registro que una instrucción anterior intenta escribir en su etapa 4. De esta manera, se tienen dos condiciones que permiten determinar si existen riesgos por dependencia de datos o no; estas condiciones son las siguientes.

1. ID/EX.SelRegW := Etapa2.SelRegR
2. EX/WB.SelRegW := Etapa2.SelRegR

La notación anterior nos permite describir con mayor precisión en qué parte de la arquitectura se presenta el riesgo por dependencia de datos. Por ejemplo, la primera condición revisa el valor de la señal de escritura que guarda una instrucción anterior en el registro de segmentación ID/EX, si este valor corresponde a alguno de los registros que intenta leer una instrucción posterior en su etapa 2, entonces se generará un riesgo. Recuerde que la etapa 4 de la segmentación se dedica a la escritura de registros, por lo tanto, las señales de control necesarias para hacer la escritura son guardadas temporalmente en los registros de segmentación ID/EX y EX/WB.

La segunda condición revisa el valor de la señal de escritura guardada por una instrucción anterior en el registro de segmentación EX/WB, si este valor corresponde a alguno de los registros que intenta leer una instrucción posterior en su etapa 2, entonces se generará un riesgo.

Para dejar en claro estas dos condiciones se revisarán nuevamente los ejemplos de la sección 7.4. También es conveniente que revise las figuras 7.52 y 7.53 para cualquier duda sobre las etapas en las que se presentan y detectan los riesgos.

Control de riesgos por medio de detenciones: Ejemplo 1

En este ejemplo existen tres riesgos por dependencias de datos. El primer riesgo se presenta entre las instrucciones *aba* y *anda*, y corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR. Esto significa, que durante la etapa de decodificación de la instrucción *anda* (ciclo de reloj CC3) se está intentando leer el registro ACCA, el cual aún no ha sido actualizado por la instrucción *aba*, ya que la señal de control encargada de actualizar este registro, SelRegW, se encuentra almacenada en el registro de segmentación ID/EX.

El segundo riesgo se presenta entre las instrucciones *anda* y *oraa*, y al igual que el riesgo anterior, corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR. Esto quiere decir, que la instrucción *oraa* también está intentando leer un registro (durante el ciclo de reloj CC4) que aún no

ha sido actualizado por una instrucción anterior, ya que la señal de control encargada de actualizar dicho registro, SelRegW, aún se encuentra almacenada en el registro de segmentación ID/EX.

Finalmente, el tercer riesgo se presenta entre las instrucciones *oraa* y *staa*, y también corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR; es decir, la instrucción *staa* está tratando de leer un registro (en el ciclo de reloj CC5) que aún no ha sido actualizado por la instrucción *oraa*.

Control de riesgos por medio de detenciones: Ejemplo 2

En este ejemplo existen dos riesgos por dependencias de datos. El primer riesgo se presenta entre las instrucciones *ldab* y *aba*, y corresponde a la condición EX/WB.SelRegW := Etapa2.SelRegR. Esto significa, que durante la etapa de decodificación de la instrucción *aba* (ciclo de reloj CC4) se está intentando leer el registro ACCB, el cual no ha sido actualizado por la instrucción *ldab*, ya que la señal de control encargada de actualizar este registro, SelRegW, se encuentra almacenada en el registro de segmentación EX/WB.

El segundo riesgo se presenta entre las instrucciones *ldaa* y *aba*, y corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR. Esto significa, que durante la etapa de decodificación de la instrucción *aba* (ciclo de reloj CC4) se está intentando leer el registro ACCA, el cual no ha sido actualizado por la instrucción *ldaa*, ya que la señal de control encargada de actualizar este registro, SelRegW, se encuentra almacenada en el registro de segmentación ID/EX.

Por último, observe que entre las instrucciones *ldab* y *abx* no existe riesgo, ya que cuando *abx* lee el contenido del registro ACCB durante el ciclo de reloj CC5, éste ya está disponible.

Cómo implantar las detenciones en hardware

Ya que se sabe en dónde y cuándo se presentan los riesgos por dependencias de datos, entonces se pueden detectar y eliminar. La detección de un riesgo, consiste en comparar las señales de control que se estudiaron arriba; mientras que su eliminación, consiste en detener a la instrucción dependiente hasta que se termine de ejecutar la instrucción causante de la dependencia.

Para detener las instrucciones en el cauce se necesita conseguir el mismo efecto que produce la ejecución de la instrucción *nop*; para ello, se agrega una unidad de detenciones, la cual detecta los riesgos y genera, durante la etapa 2, las señales de control propias de la instrucción *nop*; en términos de diseñadores de hardware, se dice que se inserta una burbuja. Sin embargo, si una instrucción en la etapa 2 es detenida, entonces la instrucción de la etapa 1 también debe ser detenida; de lo contrario, se pierde la instrucción buscada en la etapa 1. Para que esto no ocurra, el contenido del registro PC (contador de programa) y el contenido del registro de segmentación IF/ID no deben modificarse; de esta manera, la etapa 1 traerá la siguiente instrucción a ejecutar utilizando el mismo valor de PC, y la etapa 2 continuará leyendo la misma instrucción del registro de segmentación IF/ID.

La figura 7.54 muestra la nueva arquitectura segmentada utilizando el esquema de detenciones para reducir los riesgos por dependencias de datos.

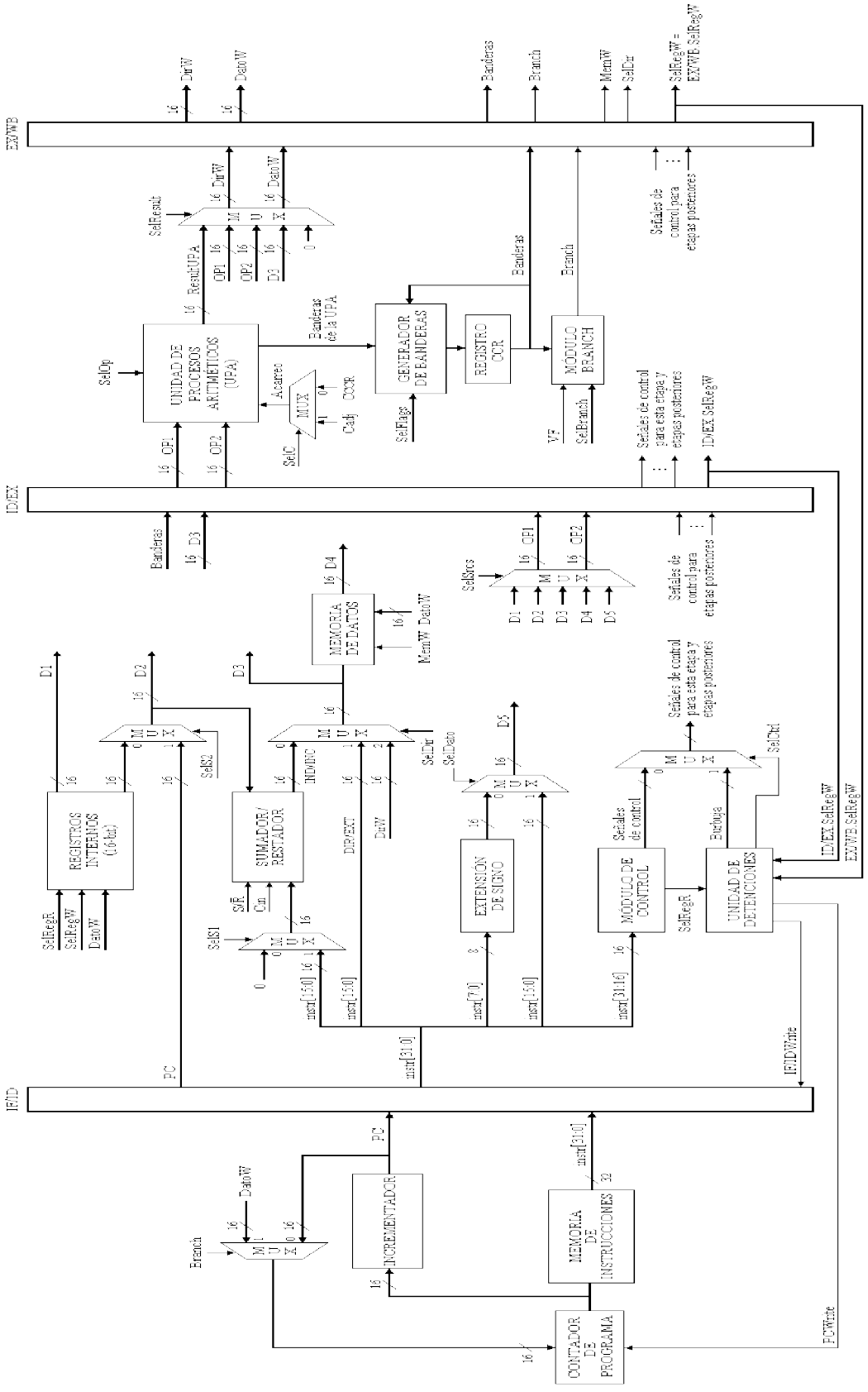


Figura 7.54. Arquitectura segmentada que utiliza el esquema de detenciones para reducir los riesgos por dependencias de datos.

El funcionamiento de la unidad de detenciones es muy simple. Primero compara el valor de la señal SelRegR, proveniente de la unidad de control de la etapa 2, contra los valores de las señales SelRegW, guardadas en los registros de segmentación ID/EX y EX/WB. Si se encuentra alguna correspondencia entre estas señales, significa que se está tratando de leer un registro que no ha sido actualizado, en otras palabras, se detecta un riesgo por dependencia de datos.

Una vez registrado el riesgo, la unidad de detenciones debe generar una burbuja; es decir, debe asignar ciertos valores a las señales de control, de manera que se obtenga el mismo efecto de una instrucción *nop*. La asignación de estos valores se realiza por medio de la señal SelCtrl, la cual seleccionará las señales de control del módulo de detenciones en lugar de las señales de control del módulo de control. Recuerde que algunas señales de control son utilizadas en la etapa 2 y otras son guardadas en el registro de segmentación ID/EX para etapas posteriores. Adicionalmente, se cuenta con las señales PCWrite e IF/IDWrite que habilitan las operaciones de escritura en el registro contador de programa y en el registro de segmentación IF/ID, respectivamente.

La siguiente tabla muestra las condiciones para detectar los riesgos por dependencias de datos, así como las señales de salida, generadas por la unidad de detenciones, que permiten eliminarlos.

Condiciones de entrada		*Señales de salida*		
SelRegR	ID/EX.SelRegW ó EX/WB.SelRegW	PCWrite	IF/IDWrite	SelCtrl
1	1, 4	0	0	1
2	2, 4	0	0	1
3	3, 4	0	0	1
4	1	0	0	1
5	4	0	0	1
6	1, 2	0	0	1
7	1, 3	0	0	1
8	5	0	0	1
9	2	0	0	1
A	3	0	0	1
B	6	0	0	1
C	1, 6	0	0	1
D	4, 6	0	0	1
E	2, 6	0	0	1
F	3, 6	0	0	1
Cualquier otra combinación no presente en esta tabla		1	1	0

Tabla 7.16. Condiciones para detectar los riesgos por dependencias de datos y señales de salida generadas por la unidad de detenciones para eliminar estos riesgos.

La figura 7.55 muestra un diagrama de múltiples de ciclos de reloj usando el esquema de detenciones para resolver los riesgos por dependencias de datos del ejemplo 2. Este ejemplo fue elegido porque en él se presentan las dos condiciones de riesgo por dependencia de datos.

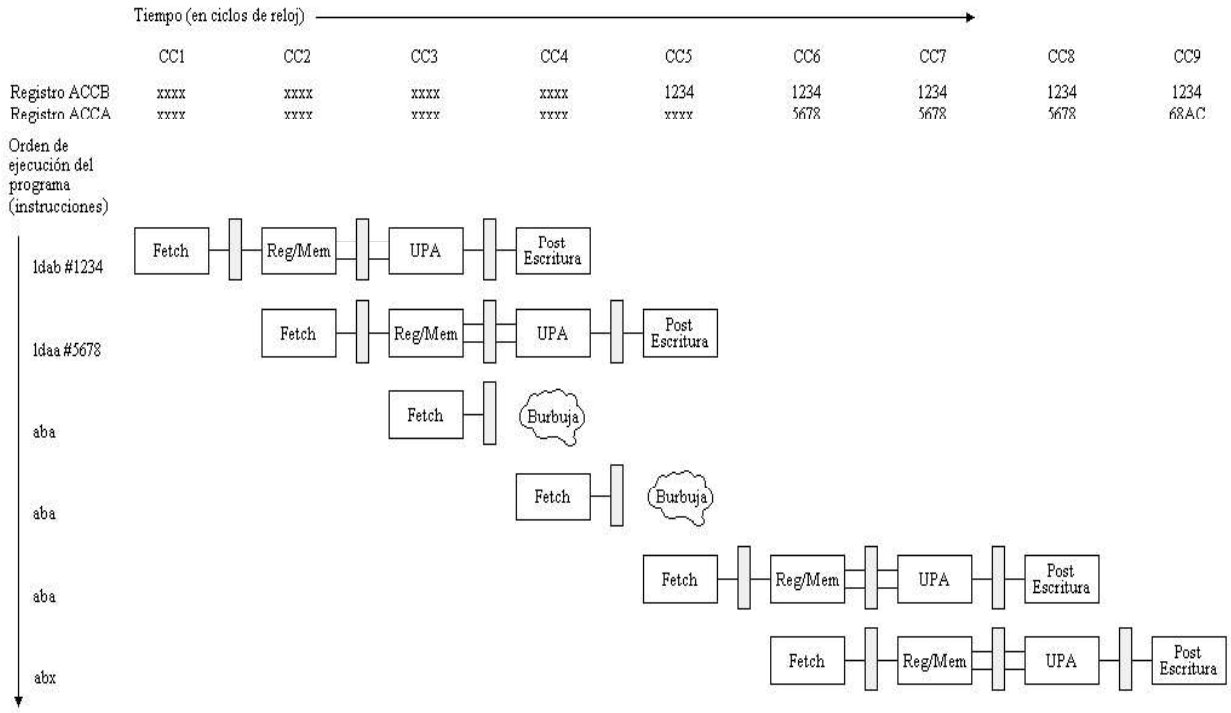


Figura 7.55. Diagrama de múltiples ciclos de reloj para el ejemplo 2; en él se utiliza el esquema de detenciones para resolver los riesgos por dependencias de datos.

Cuando la instrucción *aba* intenta leer los registros ACCA y ACCB durante el ciclo de reloj CC4, la unidad de detenciones detecta la presencia de un riesgo e inserta una burbuja. La detección del riesgo se logra gracias a la señal de lectura SelRegR de la instrucción *aba* y a las señales de escritura SelRegW de las instrucciones *ldab* y *ldaa*, las cuales se encuentran almacenadas temporalmente en los registros de segmentación EX/WB e ID/EX, respectivamente. Observe que en este instante, la unidad de detenciones detecta las dos condiciones de riesgo para la misma instrucción: la primera condición, ID/EX.SelRegW := Etapa2.SelRegR, se presenta entre las instrucciones *aba* y *ldaa*; mientras que la segunda condición, EX/WB.SelRegW := Etapa2.SelRegR, se presenta entre las instrucciones *aba* y *ldab*.

Una vez detectado el riesgo, la unidad de detenciones activa la señal SelCtrl para seleccionar las señales de control de la burbuja. No olvide que las señales de control para la burbuja son generadas por la unidad de detenciones y producen el mismo efecto que la instrucción *nop*. Por otra parte, las señales PCWrite e IF/IDWrite son colocadas a cero para deshabilitar las operaciones de escritura en los registros PC e IF/ID, de esta manera, en el siguiente ciclo de reloj, se intentará decodificar nuevamente la instrucción que originó el riesgo, es decir, *aba*.

En el ciclo de reloj CC5 el registro ACCB ya fue actualizado. En este mismo ciclo de reloj, la instrucción *ldaa* está por escribir el registro ACCA, la burbuja insertada en el ciclo anterior está en la etapa de ejecución, y la instrucción *aba* se intenta decodificar por segunda ocasión. Debido a que la instrucción *ldaa* no ha terminado su ejecución, la unidad de detenciones vuelve a encontrar un riesgo por dependencia de datos, ya que la señal SelRegR de la instrucción *aba* coincide con la señal

de escritura SelRegW de la instrucción *ldaa*, la cual está guardada en el registro de segmentación EX/WB. En consecuencia, la unidad de detenciones inserta otra burbuja en la etapa 2 y deshabilita las operaciones de escritura en los registros PC e IF/ID.

En el ciclo de reloj CC6, la burbuja insertada en el ciclo CC4 está en la etapa de post-escritura, la burbuja insertada en el ciclo CC5 está en la etapa de ejecución, y la instrucción *aba* se intenta decodificar por tercera vez. En esta ocasión, la unidad de detenciones no detecta ningún riesgo entre la instrucción *aba* y las burbujas de las etapas 3 y 4; por lo tanto, el riesgo ha sido eliminado y la ejecución de instrucciones en el cauce continúa de manera normal hasta que un nuevo riesgo es detectado. No olvide que las burbujas no leen operandos, no calculan resultados y no escriben en registros; es decir, son independientes de cualquier instrucción.

Detenciones debido a accesos múltiples a memoria

Suponga que en el ciclo de reloj X una instrucción en la etapa 4 intenta guardar un resultado en la memoria de datos, y en ese mismo instante, una instrucción en la etapa 2 intenta leer un dato de la misma memoria. Un nuevo tipo de riesgo se presenta en la arquitectura segmentada, ya que la memoria de datos ubicada en la etapa 2, sólo puede leer ó escribir datos en un instante dado, pero nunca los dos. Para poder detectar y eliminar este nuevo riesgo, se extenderá la unidad de detenciones de la figura 7.54. El nuevo modelo de arquitectura, que incorpora detenciones debido a accesos múltiples a memoria, y detenciones por dependencias de datos, se muestra en la figura 7.56.

Para detectar un riesgo debido a accesos múltiples a memoria se utilizan las señales de control SelSrcs y EX/WB.MemW; estas dos señales son suficientes para determinar si en la memoria se está intentando leer y escribir simultáneamente. Por ejemplo, si la señal SelSrcs selecciona al bus D4, significa que la memoria será accedida para lectura, ya que el bus D4 es el bus por donde se leen datos de la memoria; por otra parte, si la señal MemW, proveniente del registro de segmentación EX/WB, está encendida, significa que un dato será escrito en memoria. Cuando las dos condiciones anteriores se presentan, entonces se genera un riesgo por accesos múltiples a memoria.

Una vez detectado el riesgo, el siguiente paso es eliminarlo utilizando la técnica de las detenciones. Esta técnica consiste en detener la instrucción que intenta leer un dato de memoria, y permitir que la instrucción que intenta escribir en ella termine de ejecutar su tarea; para ello, se debe generar una burbuja en la etapa 2 tal y como se realiza para los riesgos por dependencias de datos, y se deben deshabilitar las escrituras en los registros PC e IF/ID para no perder la instrucción que se trataba de ejecutar al momento de detectar el riesgo. Si observa cuidadosamente la figura 7.54 notará que la generación de la burbuja en la etapa 2 ya fue implantada cuando se resolvieron los riesgos por dependencias de datos; por lo tanto, lo único que hace falta, es permitir que la instrucción que va a escribir en memoria termine de ejecutar su trabajo. Para esto, ha sido agregada una señal de control cuyo nombre es SelD, y es generada por la unidad de detenciones tras detectar un riesgo por accesos múltiples a memoria. La señal SelD presenta el siguiente comportamiento: si SelD=0, entonces se selecciona la señal SelDirCtrl, que es la señal SelDir que genera el módulo de control para la etapa 2; y si SelD=1, entonces se selecciona la señal SelDir proveniente del registro de segmentación EX/WB, dando prioridad a la escritura en memoria.

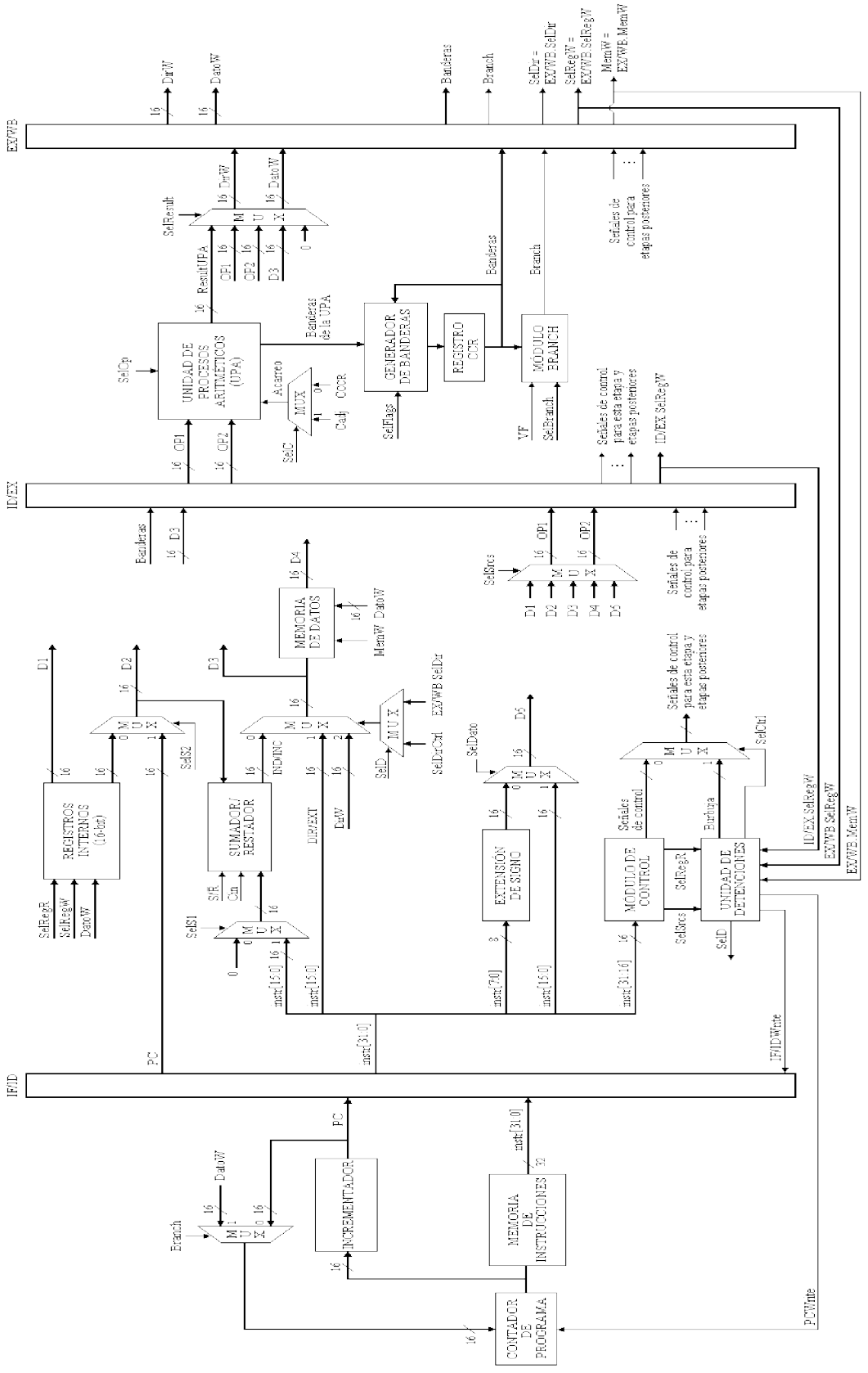


Figura 7.56. Arquitectura segmentada que utiliza el esquema de detenciones para reducir los riesgos por dependencias de datos y los accesos múltiples a memoria.

La siguiente tabla muestra las condiciones para detectar los riesgos por accesos múltiples a memoria, así como las señales de salida generadas por la unidad de detenciones, que permiten eliminar los riesgos de este tipo.

Condiciones de entrada		*Señales de salida*			
SelSrcs	EX/WB.MemW	PCWrite	IF/IDWrite	SelD	SelCtrl
2	1	0	0	1	1
4	1	0	0	1	1
6	1	0	0	1	1
2	0	1	1	0	0
4	0	1	1	0	0
6	0	1	1	0	0

Tabla 7.17. Condiciones para detectar los riesgos por accesos múltiples a memoria y señales de salida generadas por la unidad de detenciones para eliminar estos riesgos.

Tenga en cuenta que tanto la tabla 7.16 como la tabla 7.17 están implantadas dentro de la unidad de detenciones, la cual dará prioridad a los riesgos por accesos múltiples a memoria antes que a los riesgos por dependencias de datos.[10]

Finalmente, las figuras 7.57 a 7.60 muestran de manera detallada los eventos que ocurren en la arquitectura para el programa de la figura 7.55; estas figuras utilizan el esquema de detenciones tratado en la figura 7.56. Considere como condición inicial que la dirección en memoria de la primera instrucción a ejecutar es 0x0400.

7.5.3 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS POR MEDIO DE ANTICIPACIONES

Detener las instrucciones en el cauce garantiza la ejecución de instrucciones dependientes muy próximas de manera correcta, sin embargo, el costo de las detenciones afecta negativamente al rendimiento.

En el ejemplo 1 de la sección 7.4 se observó que la instrucción *anda* necesitaba del resultado de la instrucción *aba* para operar correctamente, ya que dicho resultado era uno de los operandos de la instrucción *anda*. Si analiza detenidamente la figura 7.52 notará que el resultado de la instrucción *aba* se utiliza hasta la etapa de ejecución de la instrucción *anda* (ciclo de reloj CC4); en este momento, el resultado de *aba* no ha sido escrito en el registro ACCA, sin embargo, ya está disponible en el campo DatoW del registro de segmentación EX/WB. Lo mismo ocurre cuando la instrucción *oraa* intenta calcular su resultado en el ciclo de reloj CC5; el resultado de la instrucción *anda* no ha sido guardado en el registro ACCA, pero ya está disponible en el registro de segmentación EX/WB.

[10] Para poder implantar las tablas 7.16 y 7.17 en la unidad de detenciones, el número de salidas para ambos casos debe ser el mismo; por lo tanto, la tabla 7.16 también debe anexar la señal de salida SelD, la cual tomará el valor de cero para cualquier caso de riesgo por dependencia de datos.

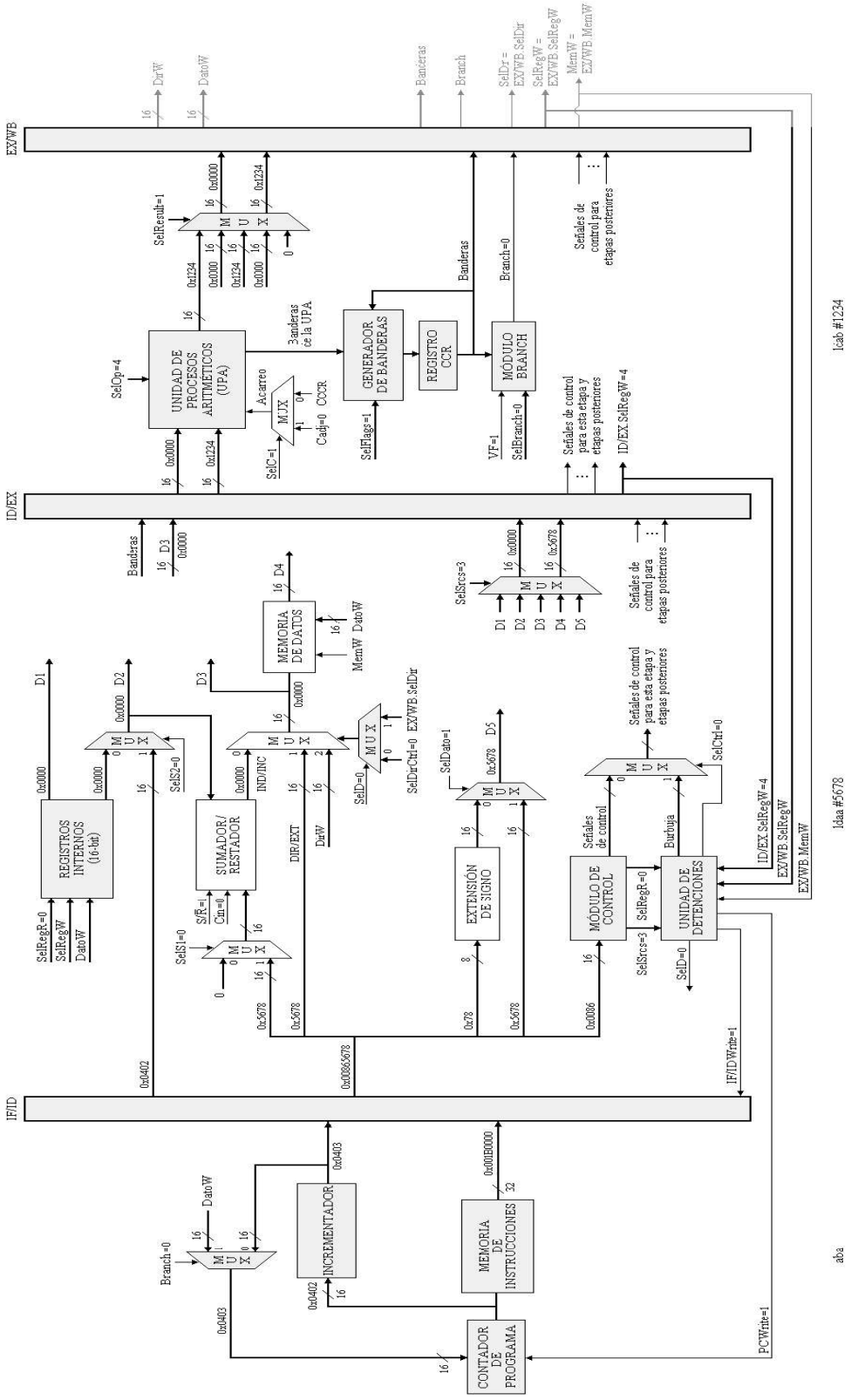


Figura 7.57. Diagrama de segmentación correspondiente al ciclo de reloj 3. La instrucción *ldab* se encuentra en la etapa 3, la instrucción *ldaa* en la etapa 2 y la instrucción *aba* en la etapa 1; hasta el momento no se han presentado riesgos por dependencias de datos o por accesos múltiples a memoria.

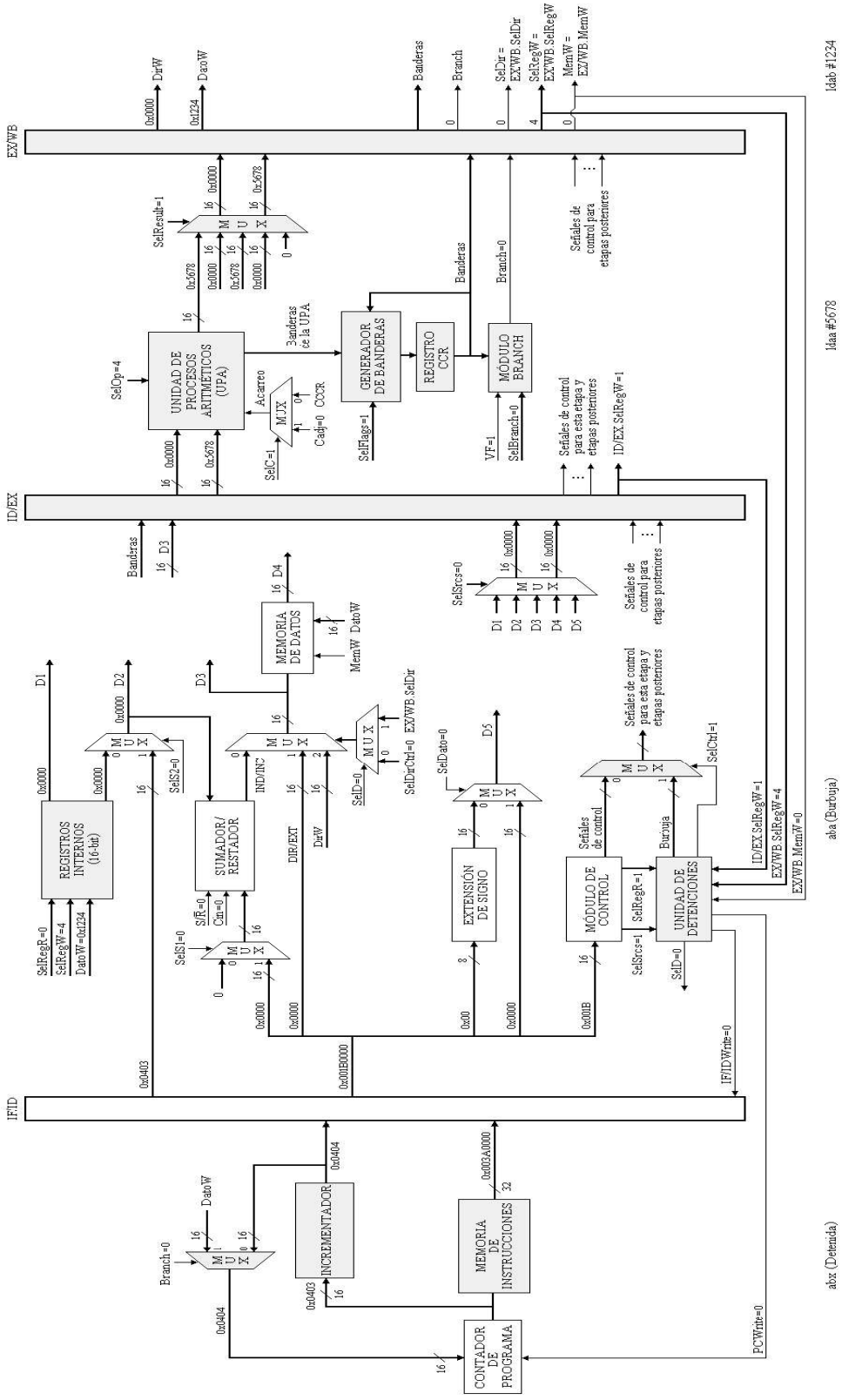


Figura 7.58. Diagrama de segmentación correspondiente al ciclo de reloj 4. La instrucción *ldab* se encuentra en la etapa de post-escritura intentado escribir un resultado en el registro ACCB; mientras, la instrucción *ldaa* está en la etapa 3, la instrucción *aba* en la etapa 2 y la instrucción *abx* en la etapa 1. En este instante, la unidad de detenciones detecta un riesgo por dependencia de datos, ya que la instrucción de la etapa 2 intenta leer un registro que está siendo escrito en este mismo momento por la instrucción *ldab*. También es detectado un riesgo por dependencia de datos entre la instrucción *aba* y la instrucción *ldaa*, pues se intenta leer el registro ACCA que aún no ha sido actualizado. Por lo tanto, la unidad de detenciones inserta una burbuja en la etapa 2 y detiene las operaciones de escritura en los registros PC e IF/ID.

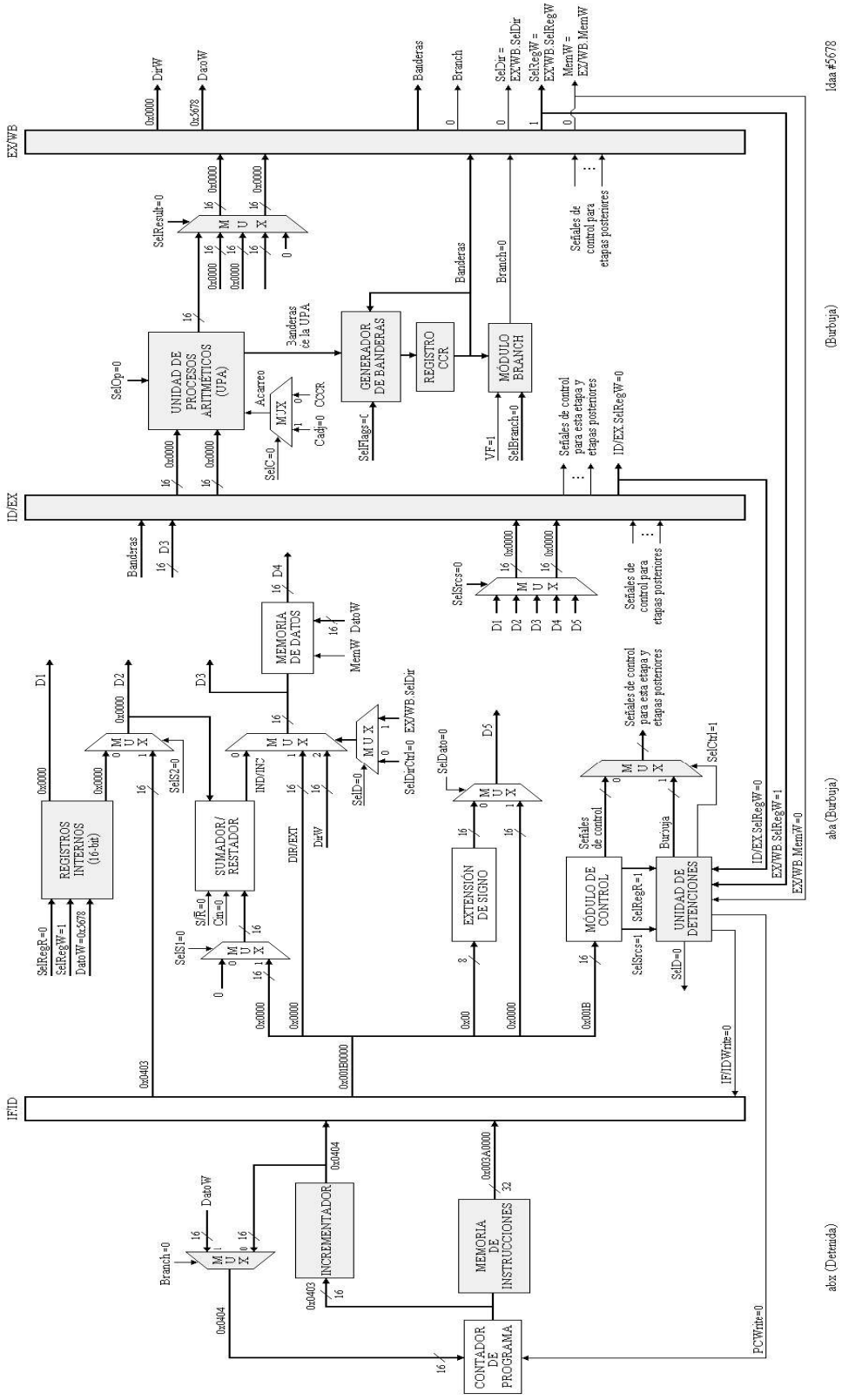


Figura 7.59. Diagrama de segmentación correspondiente al ciclo de reloj 5. La instrucción *ldab* ha terminado de ejecutarse; la instrucción *ldaa* se encuentra escribiendo un resultado en el registro ACCA; y las instrucciones *aba* y *abx* aún continúan detenidas en las etapas 2 y 1, respectivamente. Observe que en la etapa 3 está la burbuja que se insertó en el ciclo de reloj anterior; las operaciones que se ejecutan en esta etapa corresponden a las de una instrucción *nop*. Por otra parte, la unidad de detenciones vuelve a detectar un riesgo por dependencia de datos entre las instrucciones *ldaa* y *aba*, ya que se intenta leer un registro que está siendo escrito en este mismo instante. Por lo tanto, nuevamente es insertada una burbuja en la etapa 2 y son detenidas las operaciones de escritura en los registros PC e IF/ID.

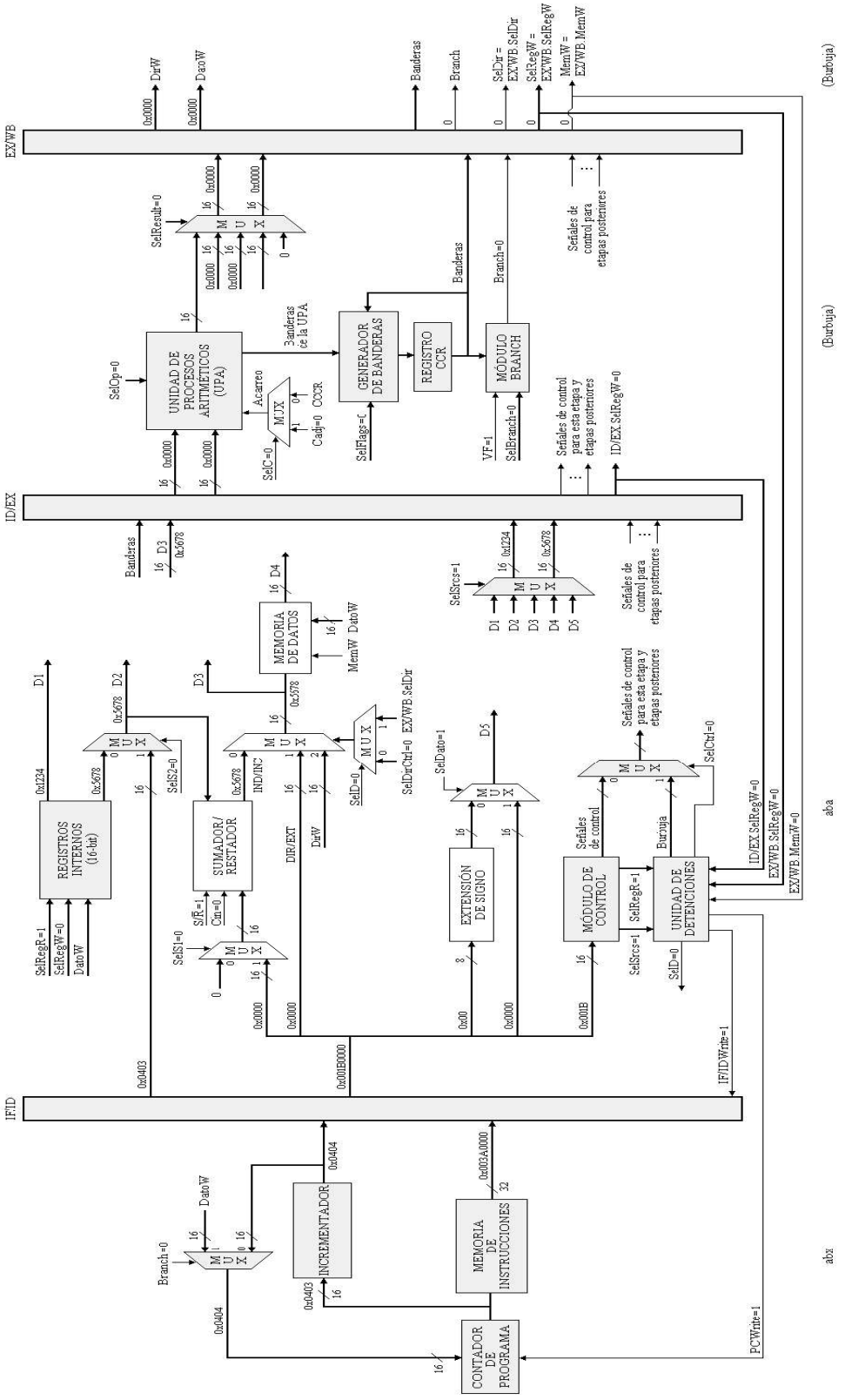


Figura 7.60. Diagrama de segmentación correspondiente al ciclo de reloj 6. La instrucción *ldaa* ya terminó de ejecutarse; y en las etapas 3 y 4 continúa la ejecución de las burbujas anteriormente insertadas. En este momento, la unidad de detenciones no detecta ningún riesgo, por lo que las instrucciones *aba* y *abx* continúan ejecutándose de manera normal.

La disponibilidad del dato requerido en el registro de segmentación EX/WB sugiere un atajo que puede reducir las pérdidas de tiempo debido a las detenciones, pues es posible leer los resultados del registro de segmentación en lugar de esperar a que la etapa 4 escriba los resultados en los registros reales. Por lo tanto, si se toman los resultados del registro de segmentación EX/WB hacia las entradas de la UPA, entonces, las instrucciones en el cauce puede proceder sin detenciones. Esta técnica, que utiliza resultados temporales en lugar de esperar que los registros reales sean escritos, se denomina anticipación (en inglés forwarding o bypassing).

Para implantar este método, se utiliza una unidad de anticipación en la etapa de ejecución, la cual detecta las dependencias de datos entre instrucciones y anticipa, en caso de dependencia, el resultado del registro de segmentación EX/WB hacia alguna de las entradas de la UPA. La figura 7.61 muestra la nueva arquitectura segmentada con la unidad de anticipación incorporada.

La unidad de anticipación necesita dos señales de control para detectar los riesgos por dependencias de datos: la primera es la señal SelRegR que proviene del registro de segmentación ID/EX, es decir, es la señal de lectura de registros de la instrucción actual; y la segunda es la señal SelRegW que proviene del registro de segmentación EX/WB, es decir, es la señal de escritura de registros de la instrucción anterior. Al comparar estas dos señales se puede determinar si el resultado guardado en el registro de segmentación EX/WB corresponde al dato requerido por la instrucción actual en la etapa 3. Si es así, existen dos multiplexores colocados a la entrada de la UPA que adelantan el resultado del registro de segmentación EX/WB hacia una de las entradas de la UPA, o bien, eligen los operandos guardados en el registro de segmentación ID/EX.

La tabla 7.18 muestra cómo la unidad de anticipación detecta los riesgos por dependencias de datos y los elimina.

Condiciones de entrada		*Señales de salida*		*Condiciones de entrada*		*Señales de salida*	
ID/EX SelRegR	EX/WB SelRegW	SelA	SelB	ID/EX SelRegR	EX/WB SelRegW	SelA	SelB
1	1	1	0	9	2	0	1
1	4	0	1	A	3	0	1
2	4	1	0	B	6	0	1
2	2	0	1	C	1	1	0
3	4	1	0	C	6	0	1
3	3	0	1	D	4	1	0
4	1	1	0	D	6	0	1
5	4	1	0	E	2	1	0
6	1	1	0	E	6	0	1
6	2	0	1	F	3	1	0
7	1	1	0	F	6	0	1
7	3	0	1	Combinaciones no presentes en esta tabla		0	0
8	5	1	0				

Tabla 7.18. Condiciones para detectar los riesgos por dependencias de datos y señales de salida generadas por la unidad de anticipación para eliminar estos riesgos.

A continuación se analizan algunas de las combinaciones mostradas en la tabla 7.18.

Suponga que la señal SelRegR del registro de segmentación ID/EX vale uno, al igual que la señal SelRegW del registro de segmentación EX/WB. Gracias a la señal SelRegR, la unidad de anticipación sabe qué registros leyó la instrucción actual en la etapa de decodificación; para este caso, los registros leídos fueron ACCA y ACCB, cuyos contenidos están almacenados en los campos OP1 y OP2, respectivamente, del registro de segmentación ID/EX. Y gracias a la señal SelRegW, se puede determinar si alguno de los registros leídos en la etapa anterior aún no había sido actualizado; para este caso, el registro ACCA aún no había sido actualizado. Esto significa que el dato guardado en el campo OP1 del registro de segmentación ID/EX no tiene el valor correcto de ACCA, sin embargo, como se estudió anteriormente, el resultado necesario está almacenado en el registro de segmentación EX/WB.

Tras esta situación, la unidad de anticipación, por medio de la señal SelA=1, adelanta dicho resultado hacia la primera entrada de la UPA; de esta manera, el valor de ACCA, guardado en el registro de segmentación ID/EX, es reemplazado por el resultado guardado en el registro de segmentación EX/WB. Note que el segundo operando de la UPA, el contenido del registro ACCB, sí tiene el valor correcto, por lo tanto, es seleccionado directamente del registro de segmentación ID/EX por medio de la señal SelB=0.

Ahora suponga que SelRegR es igual a uno y SelRegW es igual a cuatro. Gracias a la señal SelRegR, la unidad de anticipación sabe que los operandos guardados en el registro de segmentación ID/EX corresponden a los contenidos de los registros ACCA y ACCB; y gracias a la señal SelRegW, sabe que el contenido leído de ACCB aún no había sido actualizado por la instrucción anterior. Por lo tanto, el dato guardado en el campo OP2 del registro de segmentación ID/EX no es el valor correcto de ACCB, pero el resultado almacenado temporalmente en el campo DatoW del registro de segmentación EX/WB, sí lo es.

En este caso, la unidad de anticipación, por medio de la señal SelB=1, adelanta el resultado guardado en el registro de segmentación EX/WB hacia la segunda entrada de la UPA; de esta manera, el valor leído de ACCB es reemplazado por el resultado correcto. Note que el primer operando de la UPA, el contenido del registro ACCA, sí tiene el valor correcto, por lo tanto, es seleccionado directamente del registro de segmentación ID/EX por medio de la señal SelA=0.

El módulo de registros internos

Note que la unidad de anticipación sólo detecta una condición de riesgo por dependencia de datos. Otra condición de riesgo no es posible porque suponemos que el módulo de registros internos suministra el resultado correcto si una instrucción en la etapa 2 intenta leer el mismo registro que otra instrucción en su etapa 4 intenta escribir, entonces, se puede decir que el módulo de registros internos es otra forma de anticipación. De esta manera, la unidad de anticipación y el módulo de registros internos se encargarán de resolver las dos condiciones de riesgos por dependencias de datos que se estudiaron en la sección 7.5.2.

La tabla 7.19 muestra el funcionamiento de la lógica interna del módulo de registros.

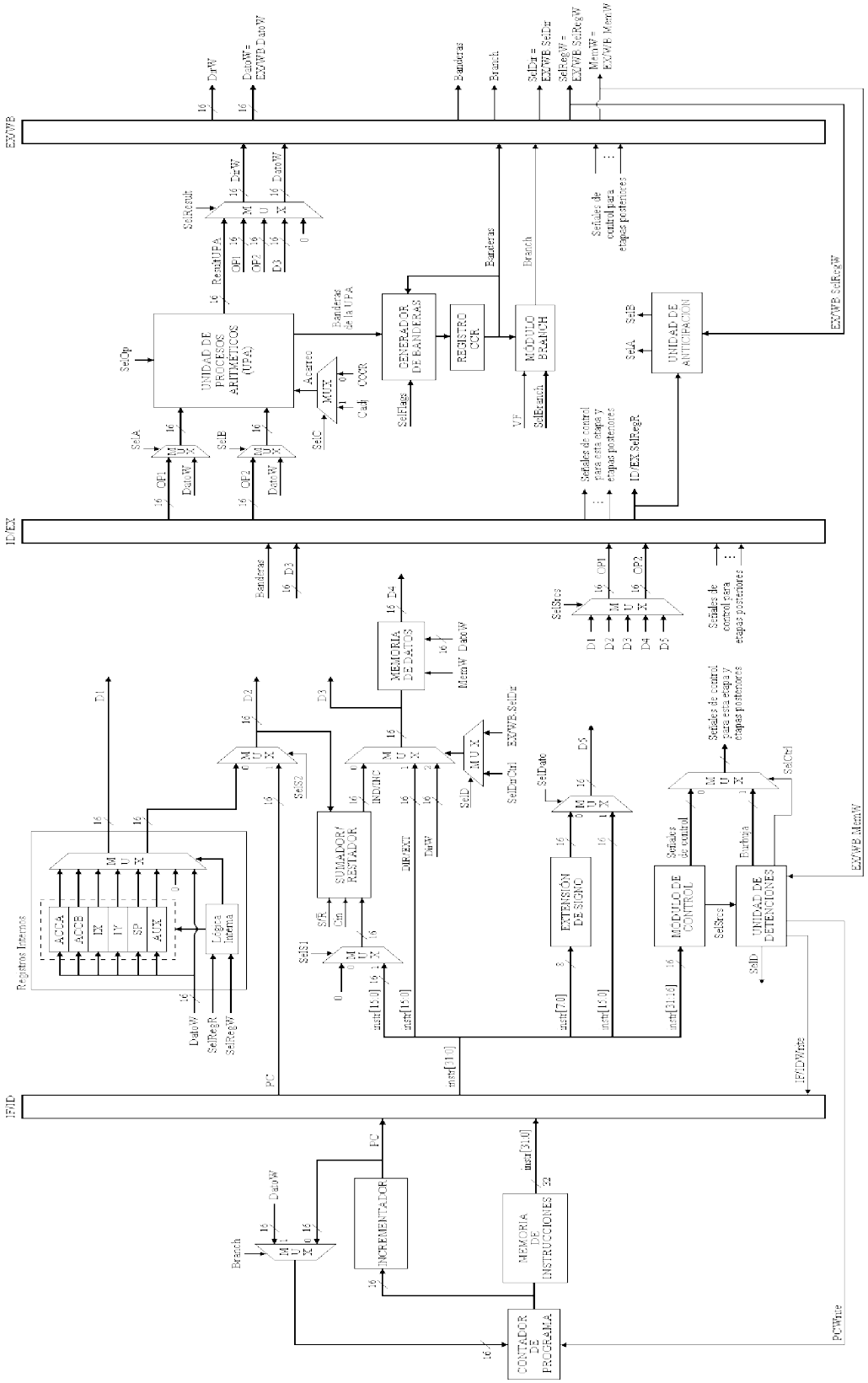


Figura 7.61. Esta arquitectura segmentada emplea el esquema de anticipaciones para eliminar los riesgos por dependencias de datos y el esquema de detenciones para resolver los accesos múltiples a memoria.

Condiciones de entrada		*Señales de salida*		*Condiciones de entrada*		*Señales de salida*	
SelRegR	EX/WB SelRegW	D1	D2	SelRegR	EX/WB SelRegW	D1	D2
1	1	DatoW	ACCB	9	2	0	DatoW
1	4	ACCA	DatoW	A	3	0	DatoW
2	4	DatoW	IX	B	6	0	DatoW
2	2	ACCB	DatoW	C	1	DatoW	SP
3	4	DatoW	IY	C	6	ACCA	DatoW
3	3	ACCB	DatoW	D	4	DatoW	SP
4	1	DatoW	0	D	6	ACCB	DatoW
5	4	DatoW	0	E	2	DatoW	SP
6	1	DatoW	IX	E	6	IX	DatoW
6	2	ACCA	DatoW	F	3	DatoW	SP
7	1	DatoW	IY	F	6	IY	DatoW
7	3	ACCA	DatoW	Combinaciones no		Consultar tablas	
8	5	DatoW	0	presentes en la tabla		7.2 y 7.3	

Tabla 7.19. Lógica interna del módulo de registros. Al detectar un riesgo por dependencia de datos se adelanta el contenido del bus DatoW hacia alguna de las salidas del módulo.

Si una instrucción intenta leer el mismo registro que otra instrucción intenta escribir, entonces, la lógica interna del módulo de registros adelanta el resultado, contenido en el campo DatoW del registro de segmentación EX/WB, hacia alguna de las salidas del módulo, D1 ó D2.

Por ejemplo, suponga que las señales SelRegR y SelRegW valen uno. La señal SelRegR indica que se intentan leer los contenidos de los registros ACCA y ACCB; mientras que la señal SelRegW indica que se intenta escribir el resultado del bus DatoW en el registro ACCA. Dada esta condición, la lógica interna del módulo de registros sabe que el contenido del registro ACCA aún no ha sido actualizado con el resultado del bus DatoW. Por lo tanto, en lugar de leer el contenido del registro ACCA, la lógica interna adelanta el contenido del bus DatoW hacia la primera salida del módulo de registros; es decir, hacia D1, que es por donde se lee el contenido de ACCA según la tabla 7.2. Observe que la segunda salida del módulo de registros, D2, se asigna con el contenido del registro ACCB, ya que éste contiene un valor actualizado. Posteriormente, durante el flanco de subida del reloj, el registro ACCA será actualizado con el valor DatoW.

Algo similar ocurre para la siguiente combinación, SelRegR=1 y SelRegW=4. En este caso, se intentan leer los contenidos de los registros ACCA y ACCB, y se intenta escribir el contenido del bus DatoW en el registro ACCB. Dada esta condición, la lógica interna sabe que ACCB no ha sido actualizado con el resultado de DatoW; por lo tanto, el contenido del bus DatoW es adelantado hacia la salida D2, que es por donde se lee el contenido del registro ACCB según la tabla 7.2. Note que la primera salida del módulo de registros, D1, se asigna con el contenido del registro ACCA, ya que éste contiene un valor actualizado. Finalmente, no olvide que en el flanco de subida del reloj el registro ACCB será actualizado con el valor del bus DatoW.

Para el resto de las combinaciones en la tabla se sigue un razonamiento similar; y en caso de que no se presente un riesgo, el módulo de registros funcionará de acuerdo a las tablas 7.2 y 7.3.

Detenciones debido a accesos múltiples a memoria

Gracias al esquema de anticipaciones, los riesgos por dependencias de datos son resueltos sin retrasos de tiempo. Desafortunadamente, los riesgos debidos a accesos múltiples a memoria no pueden ser anticipados de la misma forma, ya que la memoria de datos de la etapa 2 sólo puede leer ó escribir datos en un instante dado. Por lo tanto, se incorpora una unidad de detenciones en la arquitectura segmentada de la figura 7.61 para resolver los riesgos por accesos múltiples a memoria; recuerde que el funcionamiento de esta unidad de detenciones se explicó en la tabla 7.17.

Control de riesgos por medio de anticipaciones: Ejemplo 2

Nuevamente recurrimos al ejemplo 2 de la sección 7.4 porque muestra los dos tipos de anticipación que se estudiaron:

1. anticipación vía el registro de segmentación EX/WB hacia la UPA; y
2. anticipación vía el registro de segmentación EX/WB hacia el módulo de registros internos

La figura 7.62 presenta el diagrama de múltiples ciclos de reloj para este ejemplo; además, las figuras 7.63 a 7.65 muestran detalladamente los eventos que ocurren en cada etapa de la arquitectura durante los ciclos de reloj CC3 a CC5 de la figura 7.62.

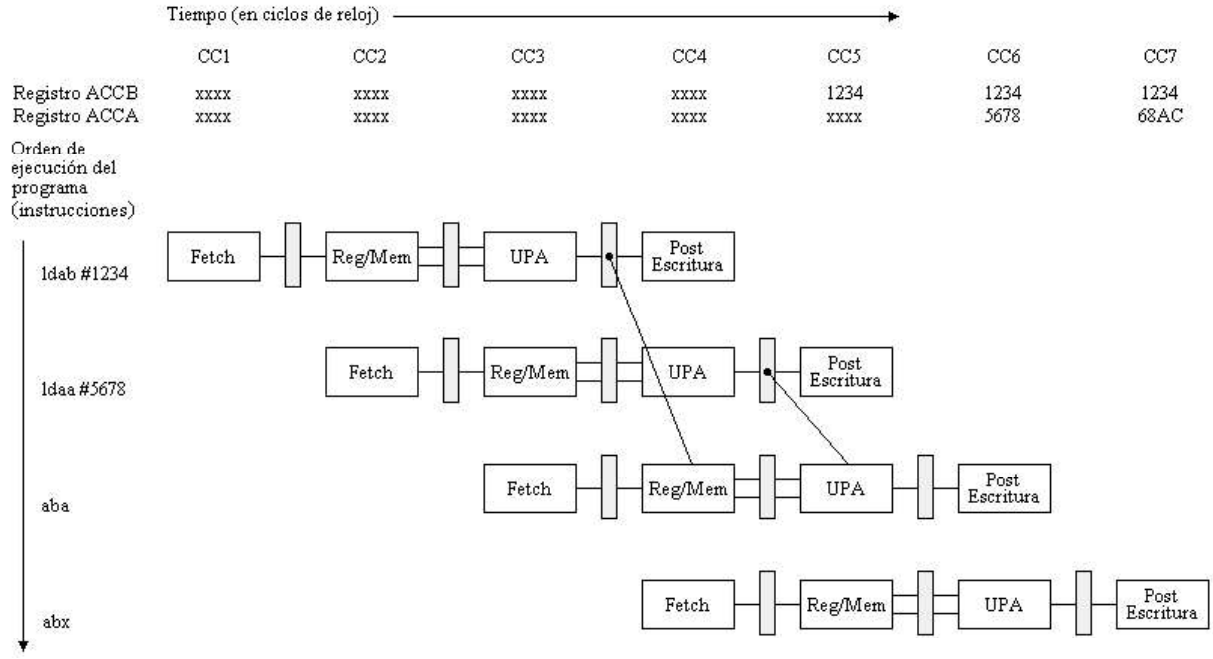


Figura 7.62. Diagrama de múltiples ciclos de reloj para el ejemplo 2. Se utiliza el esquema de anticipación para resolver los riesgos por dependencia de datos.

Observe cuidadosamente las instrucciones en el ciclo de reloj CC4 de la figura 7.62. La instrucción *ldab* se encuentra en la etapa de post-escritura intentando escribir un dato en el registro ACCB; la instrucción *ldaa* se encuentra en la etapa de ejecución; la instrucción *aba* en la etapa de lectura de operandos; y la instrucción *abx* está siendo leída de la memoria de instrucciones.

Note que la instrucción *aba* lee sus operandos en este mismo ciclo de reloj; es decir, lee los contenidos de los registros ACCA y ACCB, los cuales no han sido actualizados con los valores de las instrucciones de carga anteriores. En este momento, la lógica interna del módulo de registros detecta que se intenta leer y escribir el registro ACCB, por lo tanto, el dato que se va a escribir es adelantado del registro de segmentación EX/WB hacia alguna de las salidas del módulo de registros. De esta manera, uno de los operandos empleados por la instrucción *aba*, el contenido del registro ACCB, ya tiene el valor correcto, pero el operando correspondiente al contenido del registro ACCA aún no lo tiene (véase la figura 7.64).

En el siguiente ciclo de reloj, CC5, la instrucción *ldaa* intenta escribir un dato en el registro ACCA, la instrucción *aba* se encuentra en la etapa de ejecución, y la instrucción *abx* en la etapa de lectura de operandos. En este mismo instante, la unidad de anticipación, colocada en la etapa de ejecución, compara los valores de las señales SelRegR y SelRegW de las instrucciones *aba* y *ldaa*, respectivamente. Gracias a estas señales la unidad de anticipación sabe que el contenido del registro ACCA, leído en la etapa anterior, no corresponde al valor actualizado, ya que en el registro de segmentación EX/WB se tiene el resultado y la señal de escritura para este registro. Por lo tanto, el resultado del registro de segmentación EX/WB es anticipado hacia una de las entradas de la UPA, reemplazando el valor de ACCA leído en la etapa anterior. Finalmente la instrucción *aba* tiene los operandos correctos sobre los que operará la UPA (véase la figura 7.65).

7.6 RIESGOS POR SALTOS

Hasta el momento hemos limitado nuestro interés a riesgos que involucran operaciones aritméticas y transferencias de datos (tanto a memoria como a registros), pero existe otro tipo de riesgo que se suele presentar debido a los saltos, los cuales son cambios en el flujo de control del programa. Por ejemplo, suponga que tiene la siguiente secuencia de instrucciones:

```
0x0400    bmi 07
0x0401    anda #0013
0x0402    orab #FFFF
0x0403    adda #0010
.....     .....
0x0408    ldaa #1000
0x0409    ldab #8080
```

La primera instrucción, *bmi*, es una instrucción de salto condicional. Esta instrucción revisa el valor de la bandera de negativo del registro de estados, si su valor es igual a uno, entonces se realiza un salto hacia la instrucción ubicada en la localidad de memoria 0x0408, si no, la siguiente instrucción a ejecutar es la de la localidad 0x0401.

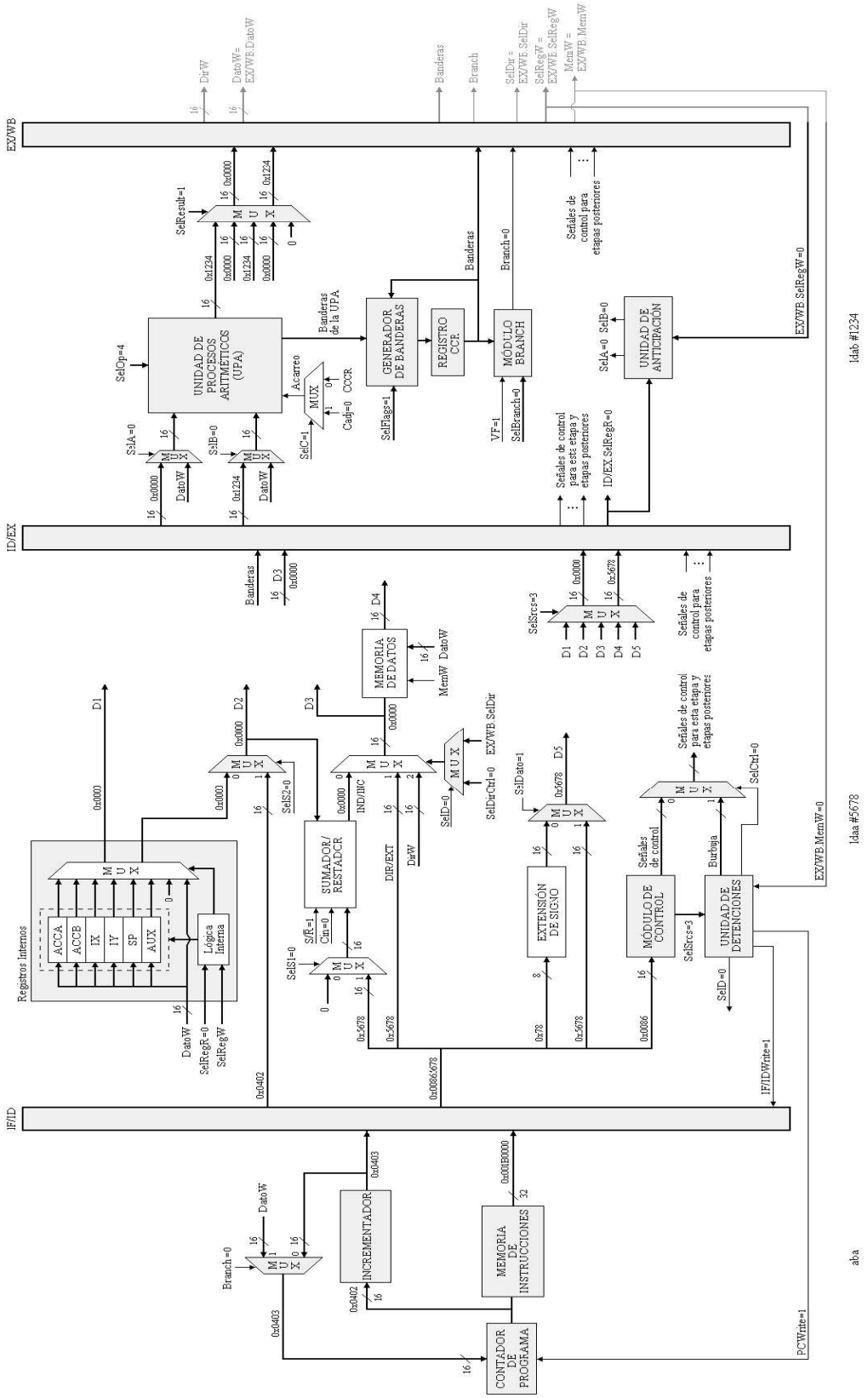


Figura 7.63. Diagrama de segmentación correspondiente al ciclo de reloj 3. Hasta el momento, la ejecución de las instrucciones en el cauce ha transcurrido sin contratiempos. La instrucción *ldab* se encuentra en la etapa de ejecución (etapa 3), la instrucción *ldaa* en la etapa de lectura de operandos (etapa 2) y la instrucción *aba* en la etapa de traer una instrucción de memoria (etapa 1).

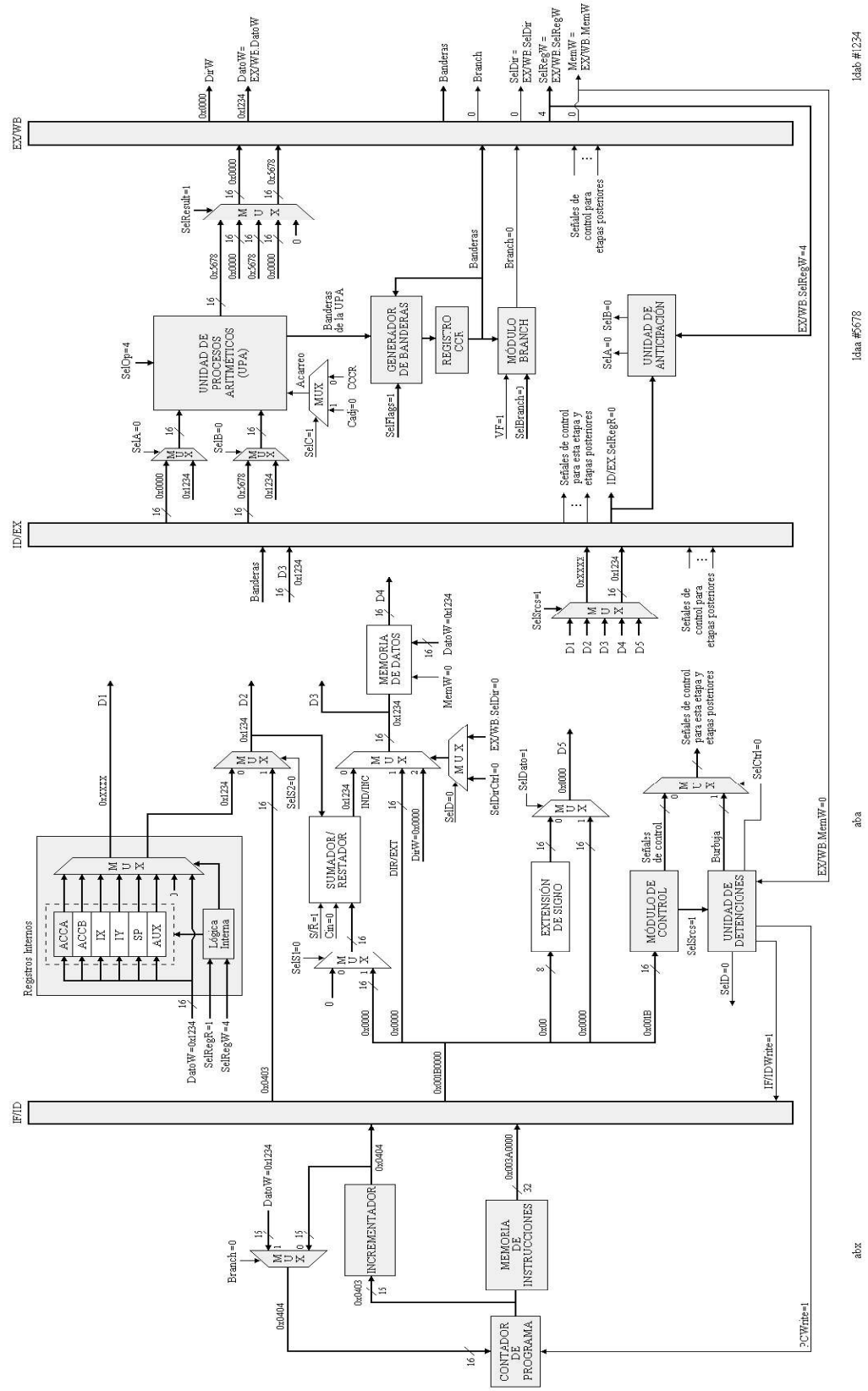


Figura 7.64. Diagrama de segmentación correspondiente al ciclo de reloj 4. La instrucción *ldab* intenta escribir un resultado en el registro ACCB, al mismo tiempo, la instrucción *aba* intenta leer los contenidos de los registros ACCA y ACCB; por lo tanto, el módulo de registros internos adelanta el resultado de la instrucción *ldab* del registro de segmentación EX/WB hacia el bus de salida D2 del módulo de registros. Observe que sólo es adelantado el dato del registro ACCB, por lo que el dato leído del registro ACCA no corresponde al valor de carga de la instrucción *ldaa*. Por otro lado, en el mismo ciclo de reloj, las instrucciones *ldaa* y *abx* están siendo ejecutadas sin contratiempos en las etapas 3 y 1, respectivamente.

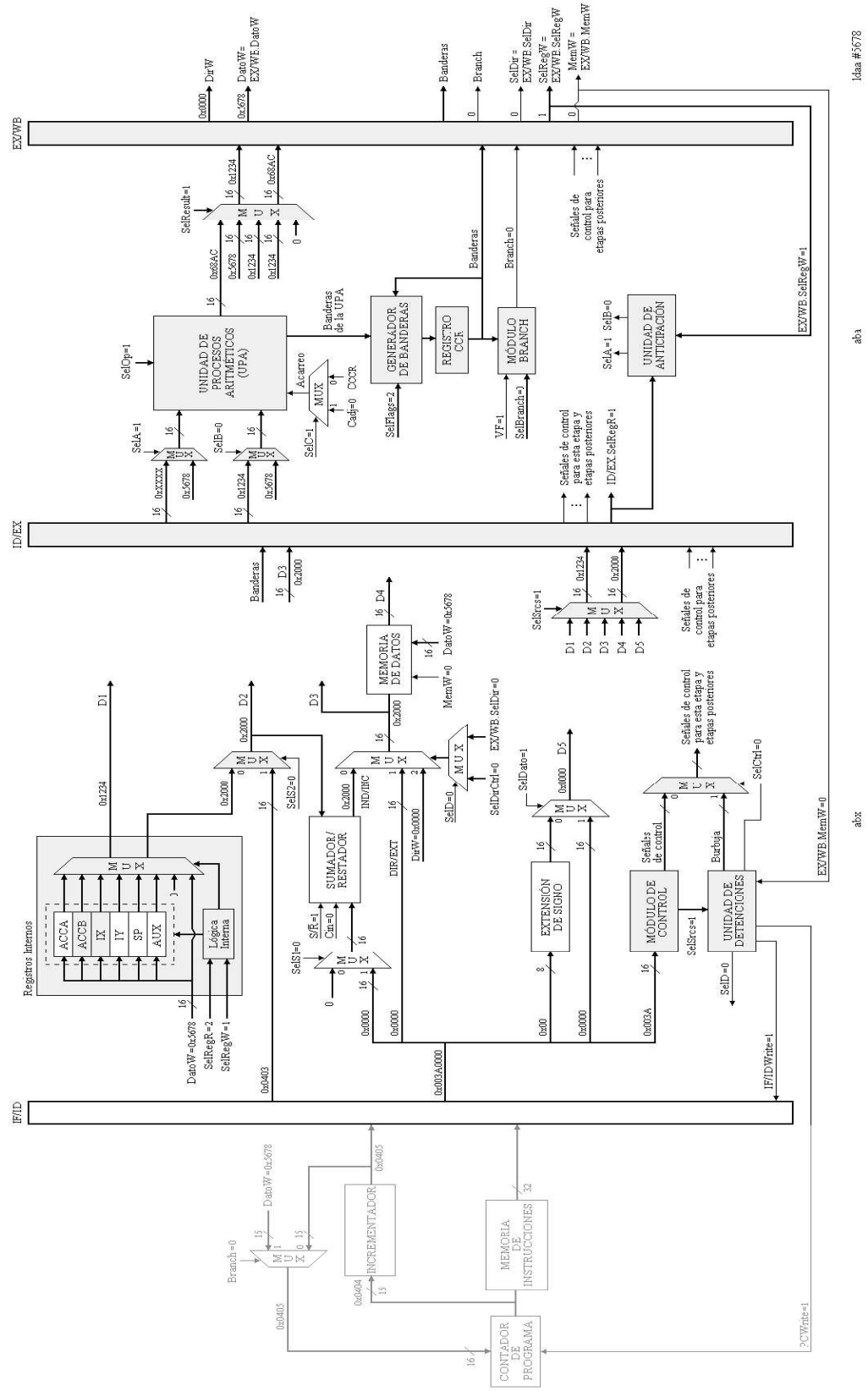


Figura 7.65. Diagrama de segmentación correspondiente al ciclo de reloj 5. La instrucción *ldaa* intenta escribir un resultado en el registro ACCA, en este instante, la unidad de anticipación se da cuenta de que el dato correspondiente al contenido del registro ACCA, que se leyó en la etapa anterior, no es el correcto; por lo tanto, la unidad de anticipación adelanta el resultado del registro de segmentación EX/WB hacia la primera entrada de la UPA, reemplazando el dato no actualizado por el valor correcto. La ejecución del resto de las instrucciones continuará sin problemas hasta que un nuevo riesgo sea detectado.

Recuerde que los saltos son resueltos hasta la última etapa de la segmentación (consulte la sección 7.3.6 para mayor información); en esta última etapa, con base en el valor de la señal Branch, se puede determinar si se realiza el salto o no. Como el salto es resuelto hasta la última etapa de la segmentación, significa que las etapas 3, 2 y 1 deben estar ejecutando las instrucciones *anda*, *oraa* y *adda*, respectivamente. Cuando el salto no es efectuado, la secuencia de ejecución continúa sin contratiempos ejecutando las instrucciones *anda*, *oraa* y *adda*. Pero, si se realiza el salto, el flujo de control del programa debe ser transferido a la localidad de memoria 0x0408; por lo tanto, las instrucciones que se ejecutaban en las etapas anteriores deben ser descartadas del cauce. Entonces surgen algunas preguntas: ¿cómo descartar esas instrucciones del cauce si el salto se realiza? ó ¿cómo evitar su ejecución para no tener problemas?. Las respuestas a estas preguntas se encontrarán al estudiar los siguientes dos métodos que nos permiten resolver los riesgos por saltos:

1. Detenciones
2. Suponer que el salto no es realizado

7.6.1 DETENCIONES

Este método plantea que la solución para eliminar los riesgos por saltos es detenerse hasta que el salto es resuelto, es decir, detener las instrucciones posteriores al salto hasta que se concluya la instrucción de salto. Observe que este método es similar al método de detenciones que se utilizó para resolver los riesgos por dependencias de datos, en el cual se detenían las instrucciones posteriores hasta resolver el riesgo. Recuerde que la desventaja de este método es la penalización de varios ciclos de reloj que se presenta cuando el salto no se realiza. Esta penalización se muestra en el diagrama de múltiples ciclos de reloj de la figura 7.66.

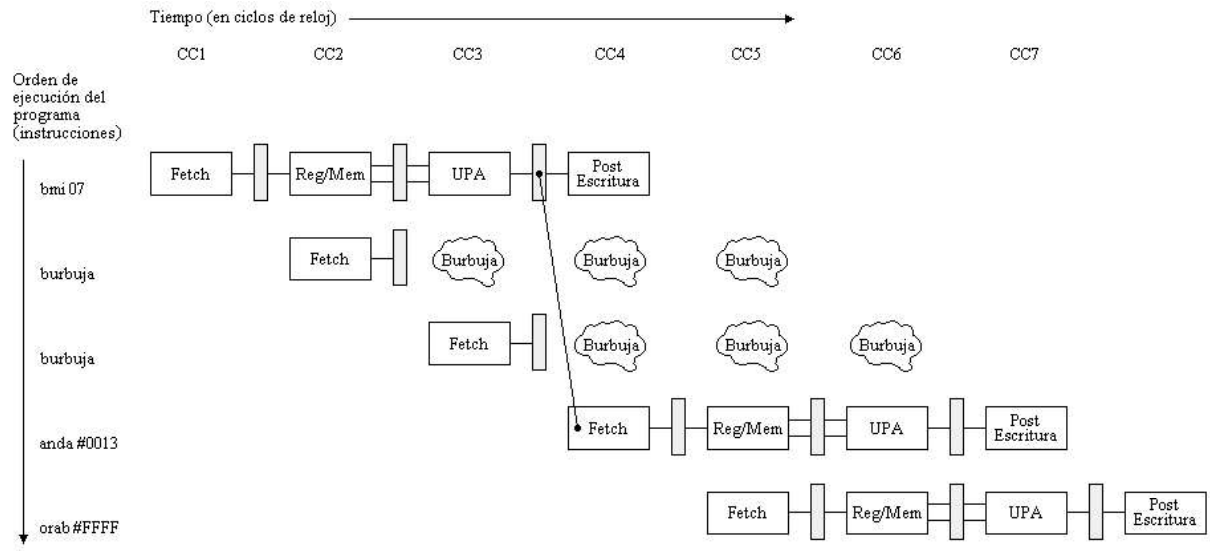


Figura 7.66. Diagrama de múltiples ciclos de reloj en donde se utiliza el esquema de detenciones para resolver los riesgos por saltos. Este esquema espera a que la instrucción de salto termine de ejecutarse para después continuar con la ejecución de las demás instrucciones; note que en este caso el salto no se realiza.

7.6.2 SUPONER QUE EL SALTO NO ES REALIZADO

Una mejora al esquema de detenciones es suponer que el salto no se realiza, por lo tanto, se continúa avanzando en la ejecución del flujo secuencial de instrucciones. La figura 7.67 muestra esta idea utilizando el ejemplo planteado al inicio de esta sección.

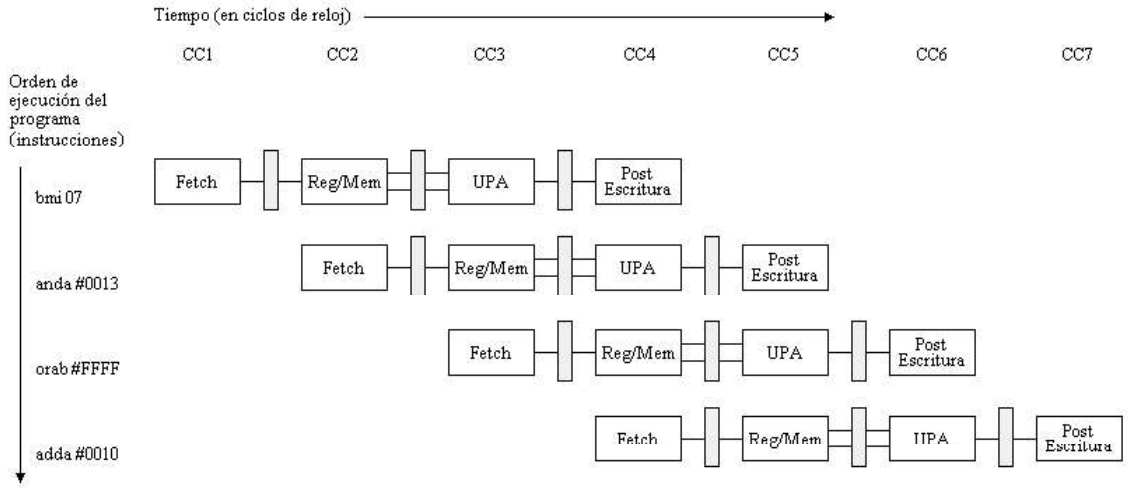


Figura 7.67. En este segundo esquema, si el salto no se realiza, no se desperdicia tiempo esperando a que el cauce vuelva a llenarse con las instrucciones siguientes.

En caso de que se realice el salto, las instrucciones posteriores a la instrucción de salto, las cuales están siendo ejecutadas, son descartadas (se limpia el cauce); y la ejecución de instrucciones reinicia a partir de la dirección destino del salto. Esta idea se expresa en la figura 7.68.

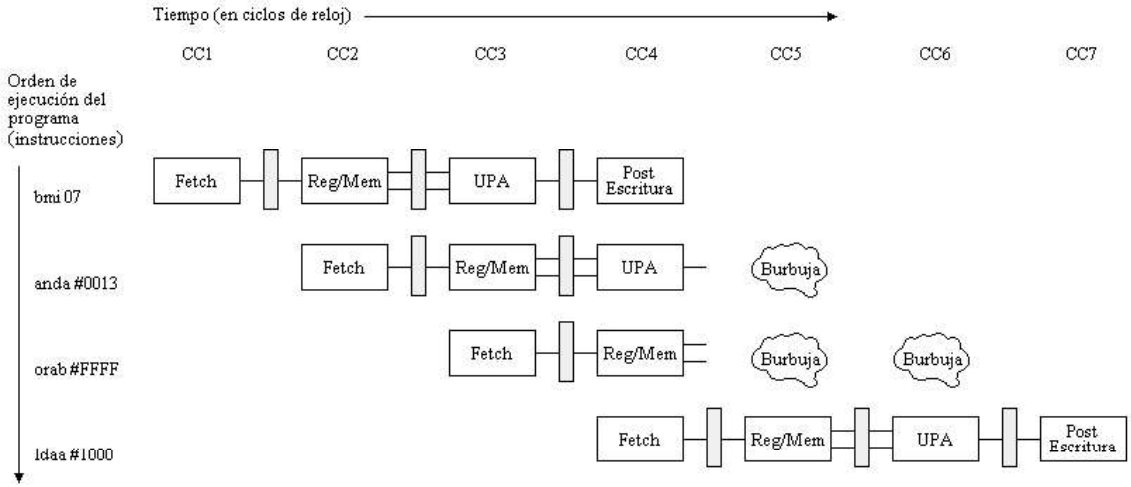


Figura 7.68. Si el salto se realiza, las instrucciones posteriores al salto que están siendo ejecutadas (*anda* y *oraa*) son descartadas, y el flujo de control de programa se transfiere a la dirección destino del salto (*ldaa*).

El método que se utiliza para descartar las instrucciones (limpiar el cauce) es muy parecido al método de generación de burbujas. Estas burbujas son insertadas en las últimas etapas de la segmentación, es decir, los contenidos de los registros de segmentación ID/EX y EX/WB son reemplazados con nuevas señales de control, obteniendo el mismo efecto de la instrucción *nop*.

Nuevamente la unidad encargada de detectar los saltos y descartar las instrucciones cuando se realiza el salto es la unidad de detenciones. Tres nuevas señales de control son agregadas a esta unidad: Branch, que indica cuándo realizar el salto; isBranch, que indica si la instrucción que se está ejecutando en la etapa 4 corresponde a una instrucción de salto; y EXFlush, que coloca a ceros el contenido del registro de segmentación EX/WB. Adicionalmente, la unidad de detenciones incorpora las señales de control necesarias para detectar y eliminar los riesgos debidos a accesos múltiples a memoria.

El funcionamiento de la nueva unidad de detenciones se resume en la tabla 7.20; y el diagrama de la figura 7.69 muestra la nueva arquitectura segmentada del 68HC11 utilizando el esquema planteado en la sección 7.6.2 para resolver los riesgos por saltos.

Condiciones de entrada				*Señales de salida*				
SelSrcs	EX/WB MemW	EX/WB Branch	EX/WB isBranch	PCWrite	IF/IDWrite	SelD	SelCtrl	EXFlush
2, 4, 6	1	0	0	0	0	1	1	0
No Importa	0	1	1	1	1	0	1	1
Combinaciones no presentes en la tabla				1	1	0	0	0

Tabla 7.20. Condiciones para detectar los riesgos por accesos múltiples a memoria y los riesgos por saltos, y señales de salida generadas por la unidad de detenciones para eliminar estos riesgos.

Finalmente, las figuras 7.70 y 7.71 muestran detalladamente los eventos que ocurren en la arquitectura durante los ciclos de reloj CC4 y CC5 de la figura 7.68.

En el ciclo de reloj CC4, la instrucción *bmi* se encuentra en la etapa de post-escritura intentando ejecutar un salto a la dirección 0x0408; en consecuencia, la instrucción contenida en dicha dirección, ldaa #1000, es leída de la memoria de instrucciones de la etapa 1. Como el salto sí es realizado, entonces las instrucciones *anda* y *oraa* de las etapas 3 y 2 respectivamente, deben ser descartadas por la unidad de detenciones. Esta unidad detecta el riesgo por salto y genera las señales de control necesarias para limpiar los registros de segmentación ID/EX y EX/WB durante el flanco de subida del reloj (véase la figura 7.70).

En el ciclo de reloj CC5, las instrucciones *anda* y *oraa* ya fueron eliminadas del cauce, en su lugar son insertadas dos burbujas en las etapas 4 y 3. Observe que el resultado de limpiar los registros de segmentación ID/EX y EX/WB es la generación de estas dos burbujas. Por otra parte, la secuencia de ejecución continúa a partir de la dirección de salto, por ello, ahora la instrucción *ldaa* se encuentra en la etapa 2 y la instrucción *ldab* en la etapa 1 (véase la figura 7.71).

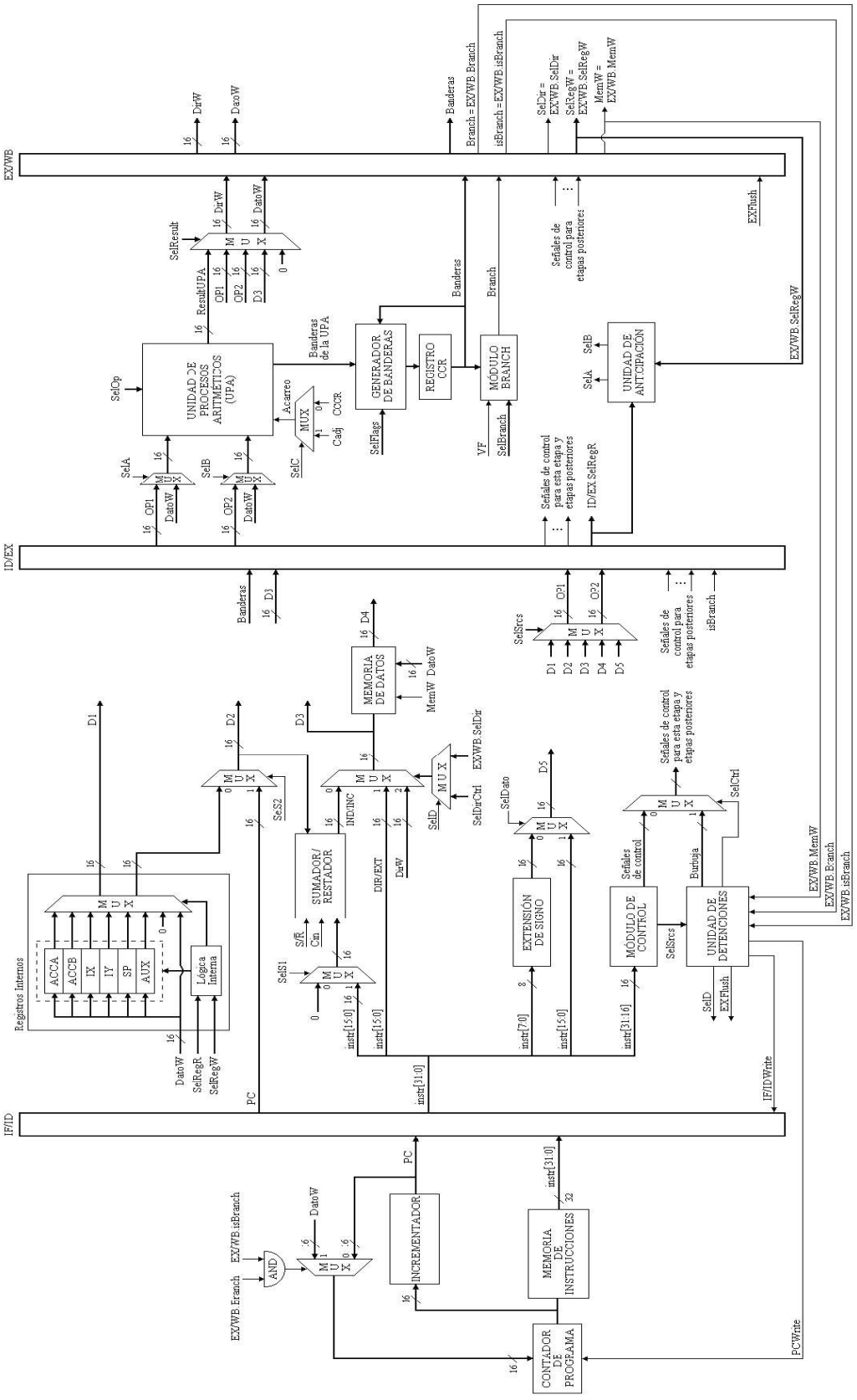


Figura 7.69. Esta arquitectura segmentada utiliza el esquema de anticipaciones para eliminar los riesgos por dependencias de datos, el esquema de detenciones para resolver los accesos múltiples a memoria, y el esquema de saltos de la sección 7.6.2 para resolver los riesgos por saltos.

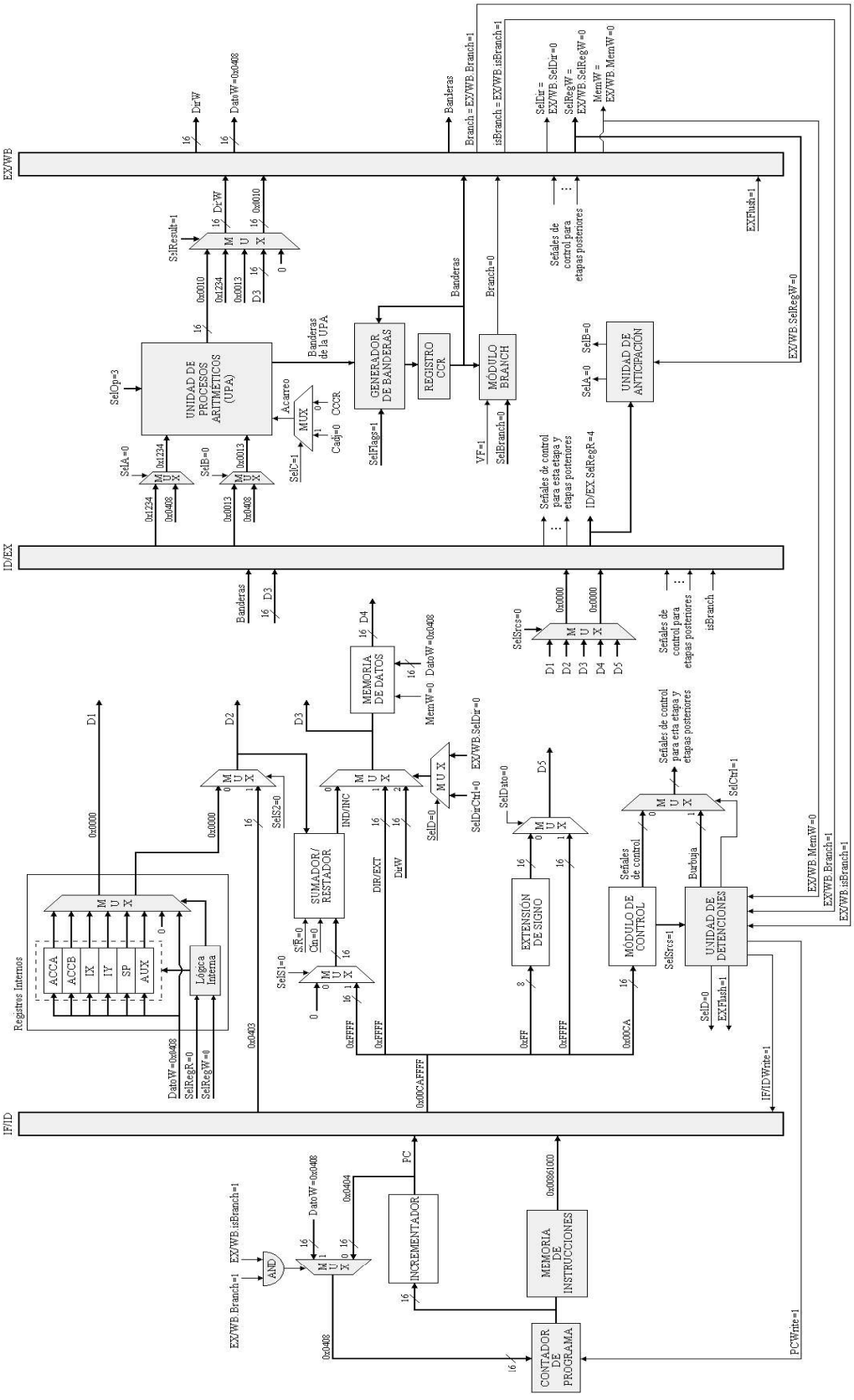


Figura 7.70. Ciclo de reloj 4. La instrucción *bmi* ejecuta un salto a la dirección 0x0408, leyendo de memoria a la instrucción *ldaa*. Como sí se realiza el salto, la unidad de detenciones detecta el riesgo y prepara las señales de control necesarias para descartar a las instrucciones *anda* y *oraa*.

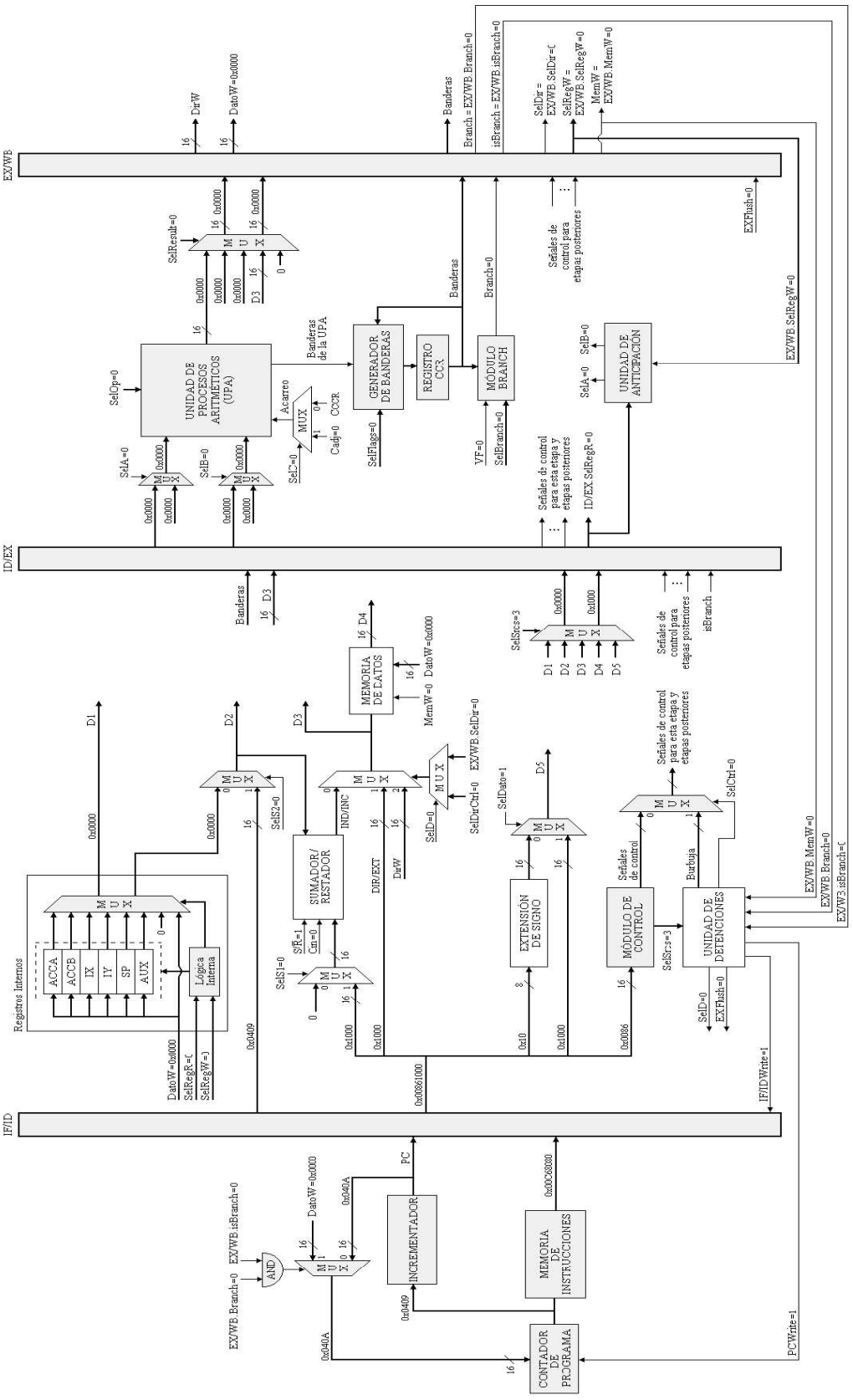


Figura 7.71. Ciclo de reloj 5. Las instrucciones *anda* y *oraa* ya fueron eliminadas del cauce, en su lugar fueron insertadas dos burbujas en las etapas 4 y 3. La secuencia de ejecución de instrucciones continúa a partir de la dirección de salto.

7.7 INTERRUPCIONES

Como se ha visto a lo largo de este capítulo, el control es el aspecto más complicado en el diseño de un procesador, pero la parte más ardua del control es la implantación de las interrupciones. Una interrupción es un evento que proviene del exterior del procesador y provoca un cambio inesperado en el flujo de control del programa. Las interrupciones las utilizan los dispositivos de entrada/salida para comunicarse con el procesador, y de manera similar a los saltos, cambian el flujo normal de ejecución de las instrucciones.

Inicialmente, las interrupciones se crearon para manejar los eventos inesperados como las peticiones de servicio provenientes de los dispositivos de entrada/salida. Actualmente, este mecanismo se ha extendido para manejar también los eventos que se generan internamente en el procesador, como por ejemplo, desbordamientos aritméticos, instrucciones indefinidas, entre otras. Los diseñadores de hardware utilizan el término "excepción" para referirse a estos tipos de interrupciones internas.

Para nuestro caso particular, el único tipo de interrupciones que se atenderán serán las solicitadas por los dispositivos de entrada/salida. Para su implantación, se han colocado en la arquitectura dos nuevos módulos: el módulo PCTrap y la unidad de interrupciones. La figura 7.72 incorpora el hardware necesario para el manejo de este tipo de interrupciones.

El módulo PCTrap está compuesto por un registro de 16 bits, el cual almacena la dirección de regreso de la interrupción. Recuerde que antes de cambiar el flujo de control del programa hacia la rutina de atención a la interrupción, el procesador debe guardar la dirección de la próxima instrucción a ejecutar, con el fin de regresar a ejecutar esa instrucción una vez que la interrupción haya sido atendida. La señal de entrada SavePC habilitará al módulo para guardar una copia de la dirección de regreso, la cual es obtenida del campo PC del registro de segmentación IF/ID.

La unidad de interrupciones se encarga de informarle al procesador cuándo un dispositivo de entrada/salida necesita atención. Si un dispositivo externo requiere de los servicios del procesador, basta que active la línea $\overline{\text{IRQ}}$ ó $\overline{\text{XIRQ}}$; en caso de que ambas líneas estén activadas, la interrupción $\overline{\text{XIRQ}}$ tendrá prioridad sobre la interrupción $\overline{\text{IRQ}}$. Si el procesador puede atender a la interrupción, la unidad de interrupciones proporciona una dirección de salto a la localidad de memoria en donde se encuentra la rutina de atención a la interrupción. Un aspecto de gran importancia que permite simplificar el diseño de la unidad de interrupciones es que el procesador no maneja interrupciones anidadas, es decir, mientras se esté atendiendo una interrupción no será posible aceptar otra.

Las condiciones de entrada para la unidad de interrupciones están dadas por las líneas EnaINT, isBranch, Branch, $\overline{\text{IRQ}}$ y $\overline{\text{XIRQ}}$, las cuales se describen a continuación.

- Señal de entrada EnaINT. Cuando la unidad de interrupciones acepta atender una petición de servicio, ésta es deshabilitada con el fin de que no puedan ser atendidas otras interrupciones hasta que la interrupción en curso termine su ejecución. La señal EnaINT, proveniente del registro de segmentación EX/WB, permite habilitar nuevamente a la unidad de interrupciones una vez que termina de atender la interrupción en curso, es decir, cuando se

ejecuta una instrucción de regreso de interrupción (RTI). La señal EnaINT es generada por la unidad de control, y es activada sólo cuando se ejecuta una instrucción de regreso de interrupción (RTI).

- Señal de entrada isBranch. Esta señal le informa a la unidad de interrupciones que la instrucción que se está ejecutando en la última etapa de la segmentación corresponde a una instrucción de salto. Es importante conocer el estado de esta señal porque en algunos casos no está permitido ejecutar una interrupción si un salto está en progreso. Recuerde que la señal isBranch proviene del registro de segmentación EX/WB, y es generada por la unidad de control siempre y cuando la instrucción que decodifica es una instrucción de salto.

- Señal de entrada Branch. Si la instrucción que se ejecuta en la última etapa de la segmentación es una instrucción de salto, entonces, la señal Branch le informa a la unidad de interrupciones si en verdad se realiza el salto o no. La señal Branch también proviene del registro de segmentación EX/WB, y es generada por el módulo Branch ubicado en la etapa de ejecución. Una interrupción no es atendida por la unidad de interrupciones si se presenta una condición de salto al mismo tiempo, es decir, no es posible atender una interrupción si el procesador está ejecutando una instrucción salto en la última etapa de la segmentación. Esto significa que la ejecución de un salto tendrá mayor prioridad que la atención a una interrupción externa. Una vez terminado el salto, la unidad de interrupciones podrá atender al dispositivo causante de la interrupción.

- Señal de entrada $\overline{\text{IRQ}}$. La señal $\overline{\text{IRQ}}$ es activada cuando el dispositivo externo conectado a esta línea requiere de la atención del procesador.

- Señal de entrada $\overline{\text{XIRQ}}$. La señal $\overline{\text{XIRQ}}$ es activada cuando el dispositivo externo conectado a esta línea requiere de la atención del procesador. La interrupción $\overline{\text{XIRQ}}$ tiene mayor prioridad que la interrupción $\overline{\text{IRQ}}$.

Una vez validada la condición de entrada, la unidad de interrupciones contesta a través de las líneas SelINT y SavePC, las cuales se describen a continuación.

- Señal de salida SelINT. La señal SelINT selecciona la procedencia de una dirección de salto. Si SelINT=1, se selecciona la dirección DirINT que corresponde a la dirección de inicio de la rutina de atención a la interrupción. En cambio, si SelINT=0, la dirección que se selecciona proviene del incrementador, o bien, del bus DatoW. La dirección de inicio de la rutina de interrupción, DirINT, es proporcionada por la unidad de interrupciones.

- Señal de salida SavePC. Una vez que la unidad de interrupciones decide atender a un dispositivo externo, es necesario guardar la dirección en memoria de la siguiente instrucción a ejecutar, para que una vez atendida la interrupción, el procesador continúe ejecutando el programa a partir de esa instrucción. La señal SavePC habilita al registro PCTrap para guardar la dirección de regreso de la interrupción.

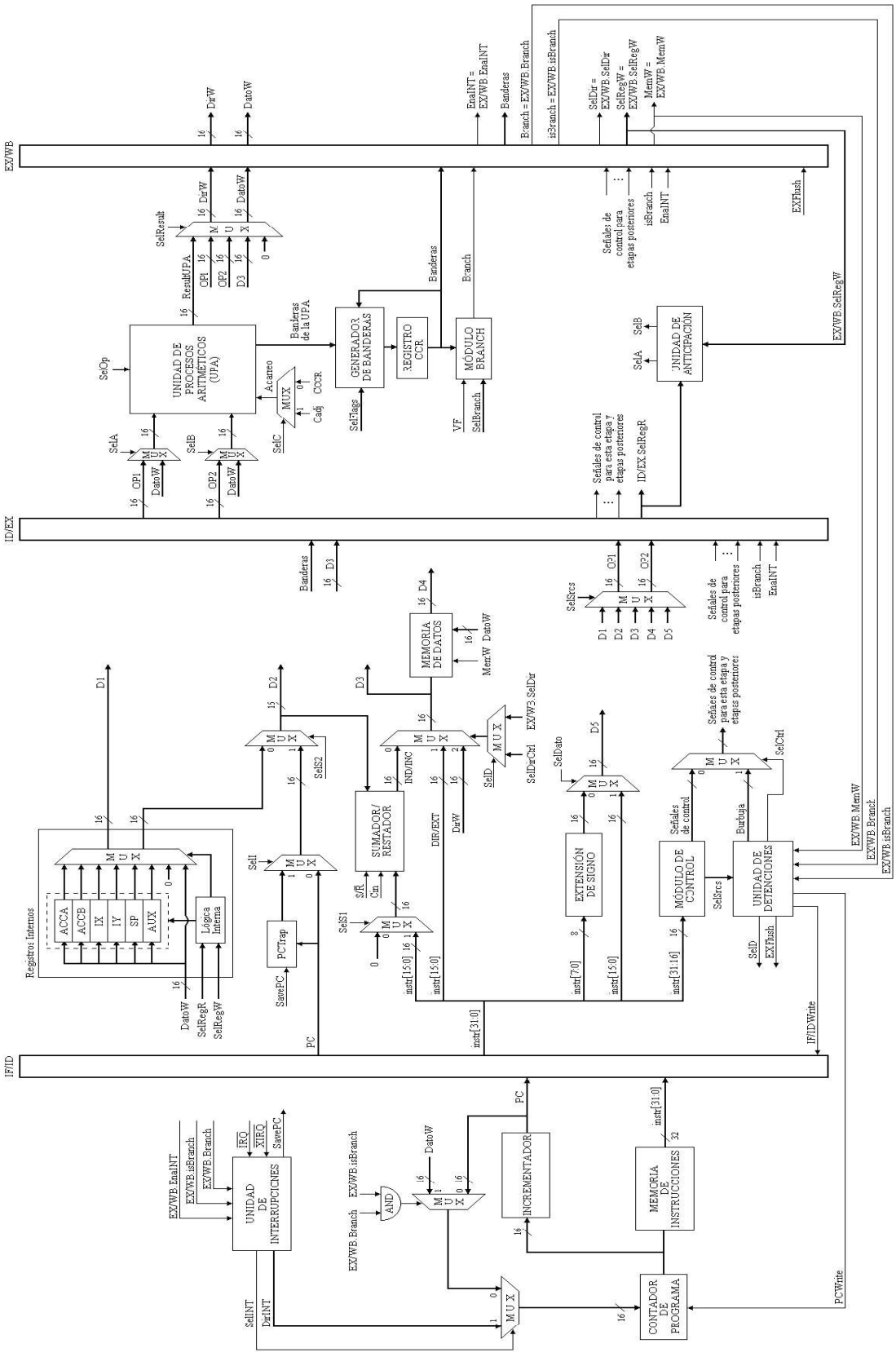


Figura 7.72. Esta arquitectura utiliza el esquema de anticipaciones para eliminar los riesgos por dependencias de datos, el esquema de detenciones para resolver los accesos múltiples a memoria, y el esquema de saltos de la sección 7.6.2 para resolver los riesgos por saltos. Adicionalmente, incluye una unidad de interrupciones para atender peticiones de servicio de dispositivos externos.

La tabla 7.21 muestra la lógica interna de la unidad de interrupciones.

Condiciones de entrada						*Señales de salida*		*Comentario*
EX/WB EnaINT	EX/WB Branch	EX/WB isBranch	$\overline{\text{IRQ}}$	$\overline{\text{XIRQ}}$		SelINT	SavePC	
0	x	0	x	0		1	1	Atiende interrupción [A.1]
0	x	0	0	1		1	1	Atiende interrupción [A.2]
0	0	1	x	0		1	1	Atiende interrupción [B.1]
0	0	1	0	1		1	1	Atiende interrupción [B.2]
0	1	1	x	0		0	0	Salto [C.1]
0	1	1	0	1		0	0	Salto [C.1]
0	x	x	1	1		0	0	No hay interrupción
1	0	x	x	x		0	0	Condición inválida
1	1	0	x	x		0	0	Condición inválida
1	1	1	x	x		0	0	Regreso de interrupción [D.1]

El valor lógico 'x' significa no importa

Tabla 7.21. Funcionamiento de la unidad de interrupciones.

A.1. Si la unidad de interrupciones está habilitada para atender peticiones de servicio, entonces, ésta da prioridad en el servicio al dispositivo conectado a la línea $\overline{\text{XIRQ}}$. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{\text{XIRQ}}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de 16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

A.2. Si la unidad de interrupciones está habilitada para atender peticiones de servicio, entonces, ésta atiende al dispositivo conectado a la línea $\overline{\text{IRQ}}$. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{\text{IRQ}}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de 16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

B.1. En esta condición ocurren dos eventos al mismo tiempo: 1) existe una petición de interrupción, y 2) se está ejecutando una instrucción de salto. Como la instrucción de salto no realiza el salto, la unidad de interrupciones puede atender la interrupción $\overline{\text{XIRQ}}$ siempre y cuando no haya otra interrupción en ejecución. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{\text{XIRQ}}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de

16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

B.2. En esta condición ocurren dos eventos al mismo tiempo: 1) existe una petición de interrupción, y 2) se está ejecutando una instrucción de salto. Como la instrucción de salto no realiza el salto, la unidad de interrupciones puede atender la interrupción $\overline{IRQ}$ siempre y cuando no haya otra interrupción en ejecución. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{IRQ}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de 16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

C.1. En esta condición ocurren dos eventos al mismo tiempo: 1) existe una petición de interrupción, y 2) se está ejecutando un salto. Debido a que la instrucción de salto sí realiza el salto, entonces, la unidad de interrupciones no puede atender la interrupción, ya que de hacerlo, se guardaría una dirección errónea de regreso de la interrupción. Por lo tanto, para evitar este tipo de inconvenientes, la unidad de interrupciones le otorga mayor prioridad a los saltos que a las interrupciones.

D.1. La interrupción está terminando su ejecución. Cuando se presenta esta condición, la instrucción de regreso de interrupción, RTI, ejecuta un salto hacia la dirección de regreso de la interrupción. En este mismo instante, la unidad de interrupciones vuelve a habilitarse para permitir la atención de nuevas interrupciones, sin embargo, note que no se puede comenzar la atención de una nueva interrupción en este mismo ciclo de reloj, ya que el salto que ejecuta la instrucción RTI tiene mayor prioridad que la atención a una nueva interrupción.

Además del módulo PCTrap y de la unidad de interrupciones, fue anexado un multiplexor a la salida del módulo PCTrap. Este multiplexor utiliza la señal de control SelI, la cual es generada por el módulo de control y permite seleccionar la dirección guardada en el registro PCTrap, o bien, la dirección guardada en el campo PC del registro de segmentación IF/ID. La señal SelI será colocada a uno cuando se desee recuperar la dirección de regreso de la interrupción, es decir, durante la ejecución de una instrucción RTI.

Atención a la interrupción

Considere el siguiente ejemplo.

```
0x0400      ldaa #1234
0x0401      ldab #5678
0x0402      aba
0x0403      abx
0x0404      oraa #1000
….          ….      ….
```

Como es sabido, antes de atender una petición de interrupción se deben guardar los contenidos de los registros IY, IX, ACCA y ACCB, así como la dirección de regreso de la interrupción. La dirección de regreso de la interrupción se guarda en el registro PCTrap, mientras que los contenidos de los registros se guardan en el área de la pila de la memoria de datos. Una vez salvado el estado de la arquitectura se comenzarán a ejecutar las instrucciones propias de la interrupción. La rutina de atención a la interrupción tiene la siguiente apariencia:

```
Rutina_INT:

0xFD00      push_IY
0xFD01      push_IX
0xFD02      push_ACCA
0xFD03      push_ACCB
            ....    ....
0xFEFC      pull_ACCB
0xFEFD      pull_ACCA
0xFEFE      pull_IX
0xFEFF      pull_IY
0xFF00      rti
0xFF01      instrucción_x1
0xFF02      instrucción_x2
            ....    ....
```

Como puede observar, al inicio de la rutina de atención a la interrupción hay varias instrucciones *push*, las cuales guardan en la pila los contenidos de los registros del procesador. Después de las instrucciones *push* es colocado el código que atiende a la interrupción; y finalmente, antes de ejecutar la instrucción de regreso de interrupción, son restaurados los registros guardados en la pila por medio de las instrucciones *pull*.

La figura 7.73 muestra los eventos que acontecen cuando se atiende una llamada a interrupción.

El comienzo de la interrupción ocurre en el ciclo de reloj CC4. En este mismo instante, la instrucción *ldaa* está en ejecución en la última etapa de la segmentación sin ocasionar conflictos con la atención a las interrupciones; por lo tanto, la unidad de interrupciones puede atender la interrupción solicitada. En consecuencia, el módulo PCTrap guarda una copia de la dirección de regreso de la interrupción, la cual está almacenada temporalmente en el campo PC del registro de segmentación IF/ID. Observe que para nuestro ejemplo, la dirección de regreso de la interrupción, 0x0403, corresponde a la instrucción *abx*. Esto significa que antes de comenzar a ejecutar las instrucciones de la rutina de interrupción, las instrucciones *ldaa*, *ldab* y *aba* terminarán su ejecución.

En el mismo ciclo de reloj, en la etapa 1, el registro PC es cargado con la dirección de inicio de la rutina de interrupción, 0xFD00. Dicha dirección es generada por la unidad de interrupciones y seleccionada por medio de la señal SelINT=1; por lo tanto, también en el ciclo de reloj CC4, se comienzan a traer las primeras instrucciones de la rutina de interrupción. De acuerdo a la rutina de interrupción que se estudió con anterioridad, las primeras instrucciones que se ejecutarían serían las instrucciones *push*.

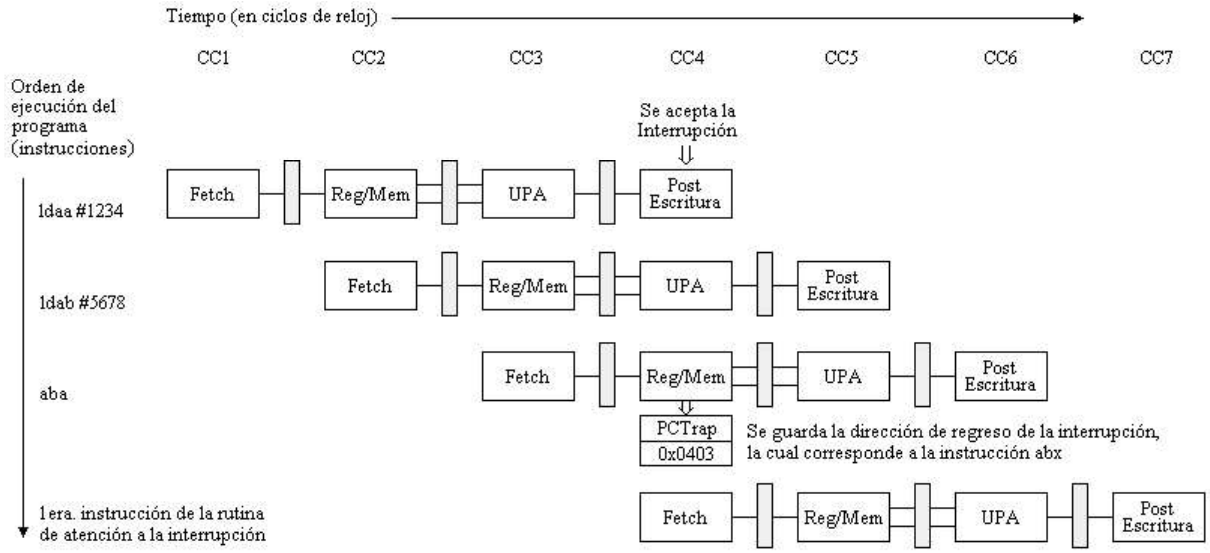


Figura 7.73. Atención a la interrupción.

Regreso de interrupción

Una vez atendido el dispositivo causante de la interrupción y restaurados los registros de la arquitectura, es ejecutada la instrucción de regreso de interrupción, *rti*.

Cuando la instrucción *rti* es decodificada, la unidad de control genera dos señales muy importantes: SelI que selecciona la dirección de regreso de la interrupción, y EnaINT que habilitará nuevamente a la unidad de interrupciones para la atención de nuevas interrupciones. No olvide que la unidad de interrupciones es deshabilitada cada vez que se inicia la atención a una interrupción, de esta manera, es imposible aceptar nuevas interrupciones si hay una interrupción en proceso.

La figura 7.74 muestra los eventos que acontecen cuando se ejecuta una instrucción de regreso de interrupción. Observe que el regreso de la interrupción ocurre hasta que la instrucción *rti* está en la última etapa de la segmentación, es decir, hasta el ciclo de reloj CC4. En ese momento, la unidad de interrupciones vuelve a habilitarse, y mientras, la arquitectura ejecuta un salto incondicional hacia la dirección de regreso de la interrupción. De manera que en el ciclo de reloj CC4, el PC se carga con la dirección 0x0403 para comenzar la búsqueda en memoria de la instrucción *abx*.

Suponiendo que en el ciclo de reloj CC4 ocurriera otra interrupción, ésta no podría ser atendida, ya que en el mismo instante, la instrucción *rti* estaría ejecutando un salto. No está de sobra recordar que la unidad de interrupciones le otorga mayor prioridad a los saltos que a las interrupciones. Por último, las instrucciones que se ejecutaban después de la instrucción *rti*, *instrucción_x1* e *instrucción_x2*, son descartadas de la segmentación gracias a la unidad de detenciones y al salto incondicional que ejecuta la instrucción *rti*.

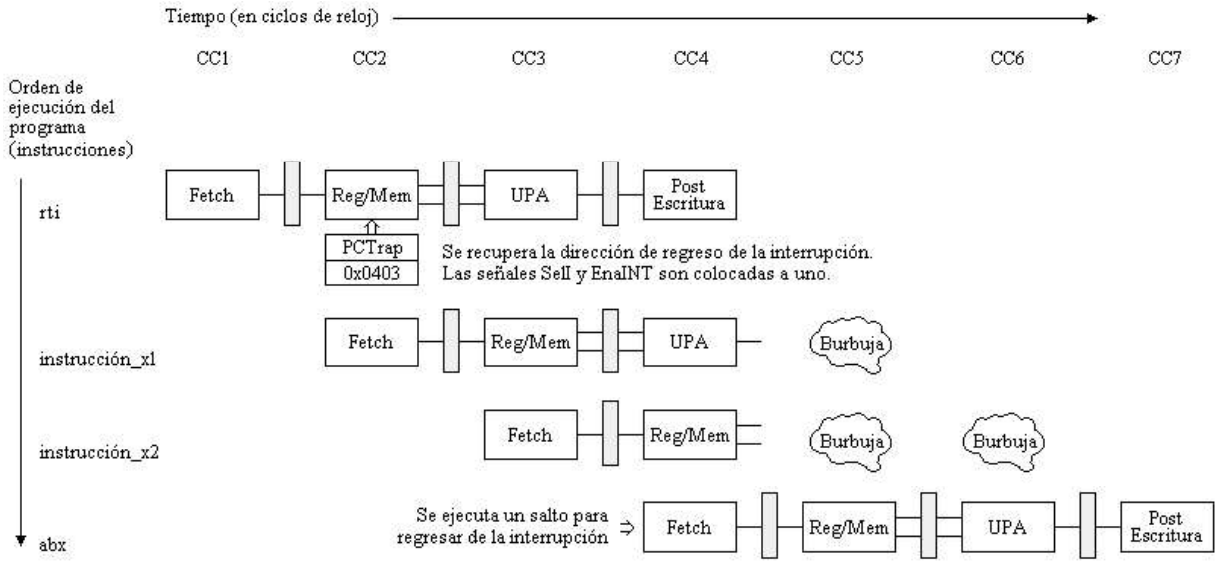


Figura 7.74. Regreso de interrupción.

Como se habrá dado cuenta, el manejo y control de las interrupciones en la segmentación no son tareas triviales, pues hay muchas condiciones que se deben considerar para poder realizar las decisiones correctas. Aún así, el manejo de las interrupciones externas son mucho más flexibles de implantar en hardware que las interrupciones internas, las cuales requieren de rigurosos métodos de control para saber exactamente qué instrucción causa la excepción.

PROBLEMAS

1. Utilice la arquitectura de la figura 7.12 para mostrar el comportamiento de las siguientes instrucciones en cada una de las etapas del "pipeline". Emplee diagramas de un sólo ciclo de reloj para cada una de las etapas y explique los eventos que ocurren en cada una de ellas.

a.	0x00	0x89	0xjj	0xkk	Instrucción adca (acceso inmediato)
b.	0x00	0x08	0x00	0x00	Instrucción inx (acceso inherente)
c.	0x00	0x2B	0x00	0xrr	Instrucción bmi (acceso relativo)
d.	0x00	0x7E	0xhh	0xll	Instrucción jmp (acceso extendido)

2. Construya una tabla con las señales de control para las siguientes instrucciones.

Instrucciones		*Señales de Control*		
		Etapa 2	Etapa 3	Etapa 4
abx (INH)	003A			
adca (IMM)	0089			
asla (INH)	0048			
bmi (REL)	002B			
clc (INH)	000C			
inx (INH)	0008			
jmp (EXT)	007E			
rol (EXT)	0079			
staa (EXT)	00B7			

3. Construya una tabla con las señales de control para las siguientes instrucciones.

Instrucciones		*Señales de Control*		
		Etapa 2	Etapa 3	Etapa 4
bcc (REL)	0024			
deca (INH)	004A			
eora (IMM)	0088			
inc (EXT)	007C			
iny (INH)	1808			
lslb (INH)	0058			
oraa (DIR)	009A			
rola (INH)	0049			
tab (INH)	0016			

4. Demuestre que las instrucciones *psha* y *pula* no pueden implantarse de manera directa en la arquitectura segmentada de la figura 7.12. Utilice diagramas de un sólo ciclo de reloj para mostrar sus conclusiones, y explique qué ocurre en cada uno ellos.

   Recuerde que la instrucción *psha* inserta el contenido del registro ACCA en la pila, mientras que la instrucción *pula* extrae un dato de la pila y lo guarda en el registro ACCA. Suponga que el área de memoria de la pila se encuentra en la parte alta de la memoria de datos, la cual es direccionada por medio del registro SP (stack pointer, apuntador de pila).

5. Conteste las siguientes preguntas con base en las conclusiones del problema anterior.

   a) ¿Las instrucciones *pshb* y *pulb* puede implantarse de manera directa?, ¿ocurre lo mismo para cualquier instrucción push ó pull que se intentara implantar?
   b) Sugiera una alternativa para poder implantar cualquier instrucción push y pull en la arquitectura segmentada de la figura 7.12. Demuestre que su alternativa funciona.

6. Utilice el diagrama de la figura 7.12 para ejecutar la siguiente secuencia de instrucciones. Incluya diagramas de un sólo ciclo de reloj para mostrar los eventos que ocurren en cada etapa de la segmentación, y coloque etiquetas sobre los buses de datos y señales de control indicando el valor que toman en ese instante.

   ```
   0x0400   staa 1000
   0x0401   ldaa #0080
   ```

7. Utilice el diagrama de la figura 7.61 para ejecutar la siguiente secuencia de instrucciones. Incluya diagramas de un sólo ciclo de reloj para mostrar los eventos que ocurren en cada etapa de la segmentación, y coloque etiquetas sobre los buses de datos y señales de control indicando el valor que toman en ese instante.

   ```
   0x0400   ldaa #0080
   0x0401   staa 1000
   ```