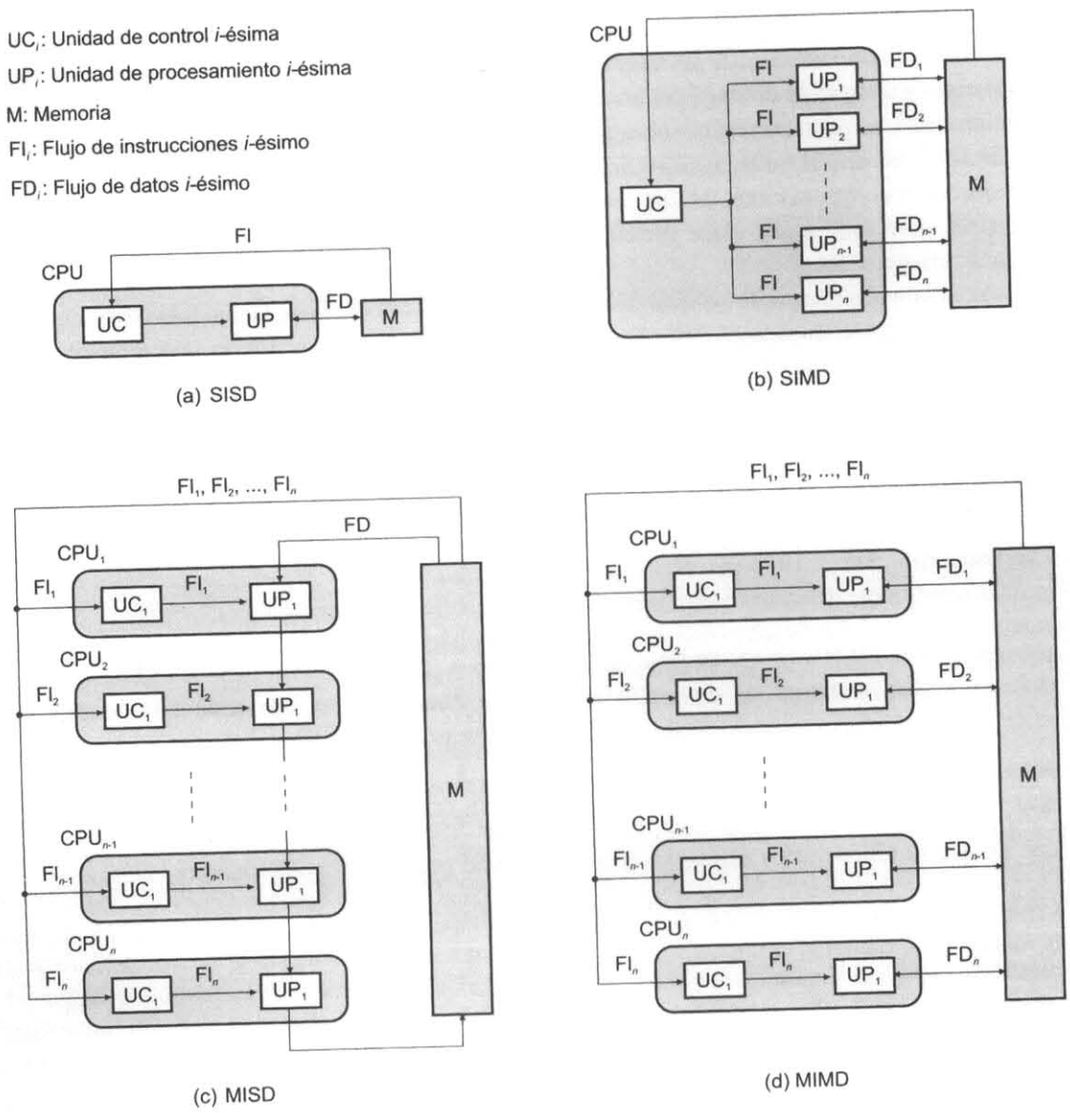


Figura 1.1: Clases de computadores que establece la taxonomía de Flynn.

other parts of the s
operational failures
handling and recov
handling exception

Generally, exceptio
ones. For example,
by Tracey et al. [1
automatically gene
exceptions in safet
technique to accele
to occur.

According to our
handling in databa
covers exceptions
benefits from ide
research work rep
optimization searc
exceptions that re
proposed researcl
Too_Many_Rows

This paper is orga
Then, in Section
shows the condu
Sections 5. Finall

2. Background

The most comr
"exceptional case
an exception stop
are usually divid
predefined excep
language rules a
defined exceptio

```
for all UPi ( i=1; i<=4; i++)
    C[i] := A[i] + B[i];
    F[i] := D[i] - E[i];
    G[i] := K[i] * H[i];
end for;
```

Figura 1.3: Fragmento de pseudocódigo del ejemplo correspondiente a un computador SIMD con cuatro unidades de procesamiento.

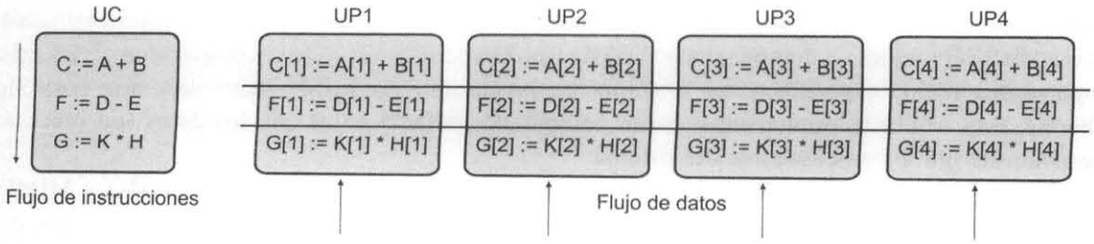


Figura 1.4: Esquema de procesamiento de tres instrucciones en un computador SIMD con cuatro unidades de procesamiento.

posibilidades de aprovechamiento del paralelismo. En la Figura 1.5 se consideran tres procesadores, y cada uno ejecuta una operación distinta sobre cuatro componentes de los vectores. En este caso, el coste temporal sería de cuatro intervalos de tiempo, es decir, tres veces menos que en la implementación secuencial.

También se puede utilizar un MIMD con cuatro procesadores, en el que cada uno de ellos realiza las tres operaciones con una de las cuatro componentes de los vectores. En este caso, el tiempo se reduciría a tres intervalos, cuatro veces menos que en el caso de un SISD. En esta situación se dice que el computador MIMD utiliza una implementación SPMD (*Simple Program Multiple Data*) del problema.

La taxonomía de Flynn pone de manifiesto dos tipos de paralelismo que pueden aprovecharse según

```
Procesador 1                Procesador 2                Procesador 3
for  (i=1; i<=4; i++)       for  (i=1; i<=4; i++)       for  (i=1; i<=4; i++)
    C[i] := A[i] + B[i];        F[i] := D[i] - E[i];        G[i] := K[i] * H[i];
end for;                     end for;                    end for;
```

Figura 1.5: Fragmento de código del ejemplo cuando se paraleliza en tres procesadores.

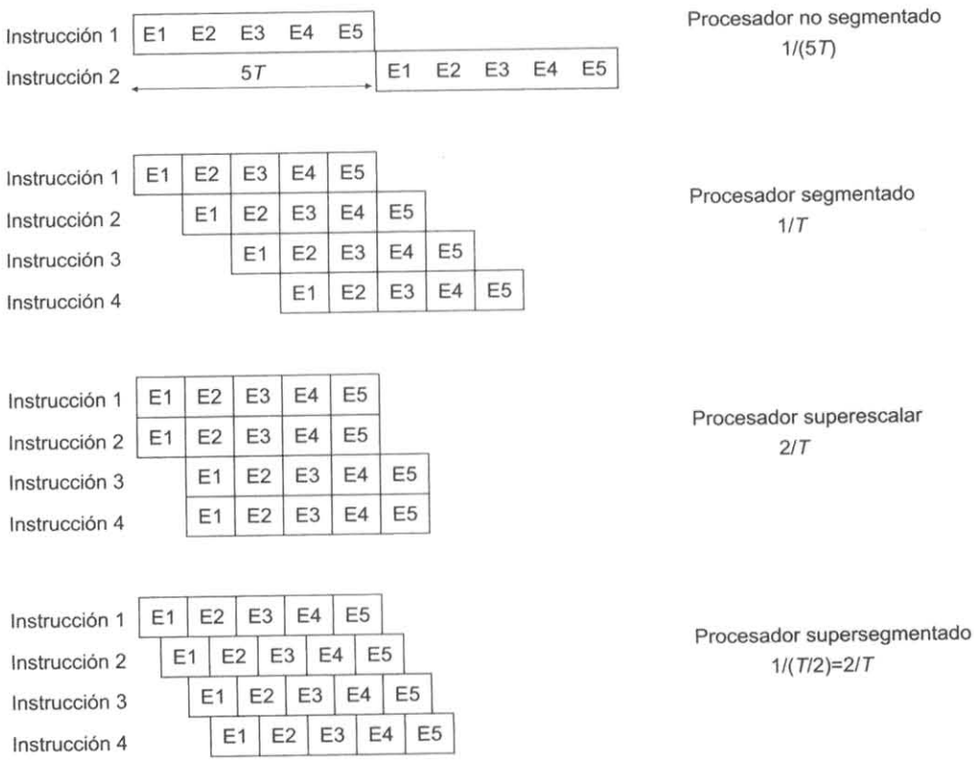


Figura 1.6: Alternativas para el aprovechamiento del paralelismo entre instrucciones (ILP).

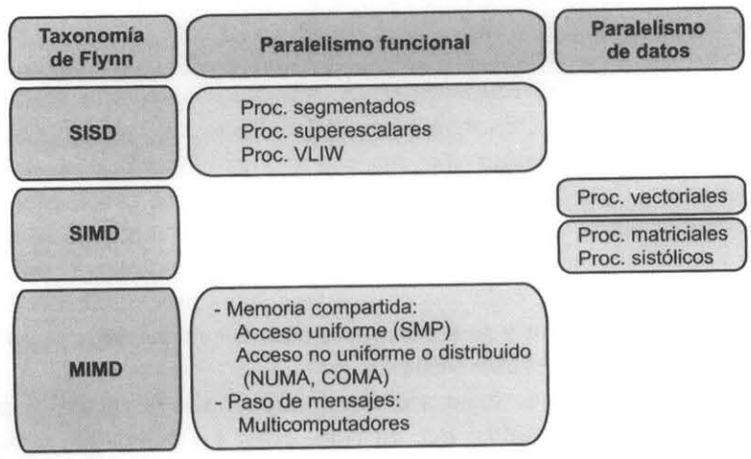


Figura 1.7: Arquitecturas según el tipo de paralelismo que aprovechan y su ubicación en la taxonomía de Flynn.

medio de operaciones que puede codificar una instrucción ($Op_{instrucción}$), esto es

$$T_{CPU} = \left(\frac{N_{operaciones}}{Op_{instrucción}}\right) \cdot CPI \cdot T_{ciclo}$$

Así, si se considera el ejemplo de procesamiento SIMD que se muestra en la Figura 1.4, el número de operaciones que se ejecutan es $N_{operaciones} = 12$. Como existen cuatro elementos de procesamiento que trabajan simultáneamente, cada instrucción codifica cuatro operaciones ($Op_{instrucción} = 4$) y el programa a ejecutar consta de tres instrucciones del repertorio del procesador SIMD ($NI = 3$). En las arquitecturas VLIW cada instrucción puede codificar varias operaciones, como muestra la Figura 1.8. Como en este ejemplo cada instrucción codifica dos operaciones, en principio el rendimiento sería similar al del procesador superescalar que emite dos instrucciones por ciclo codificando en cada instrucción una única operación.

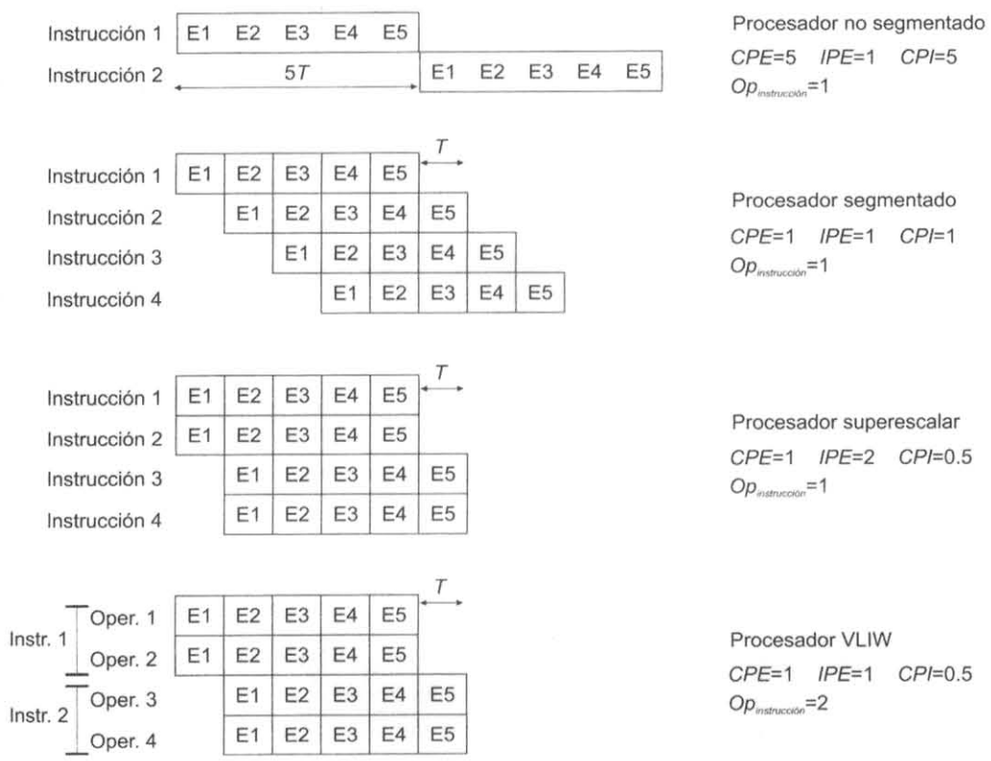


Figura 1.8: Ejemplos de valores característicos de *CPE*, *IPE* y *CPI* según las microarquitecturas segmentadas, superescalares y VLIW.

Figura 1.9: Tarea procesada de forma totalmente secuencial.

para llevar a cabo varias de las subtareas y, por ello, esas subtareas no podrán superponerse en el tiempo.

Si se emplea un procesador segmentado para procesar esa misma tarea, basta que se haya terminado la primera subtarea para poder empezar a procesar una nueva subtarea. En la Figura 1.10 puede verse el flujo continuo de tareas que se van procesando a través de las n etapas encargadas de procesar cada una de las subtareas. Puede observarse que el tiempo total de procesamiento de una tarea completa puede ser el mismo que el tiempo empleado para el procesamiento secuencial de la misma tarea mostrado en la Figura 1.9, aunque frecuentemente será mayor. Esto, sin embargo, carece de relevancia ya que lo verdaderamente importante es el ritmo al que las tareas van saliendo del procesador (*velocidad de emisión de tareas*). Al número de etapas del procesador, n, se le denomina en muchas ocasiones *profundidad de la segmentación*.

Para que el tiempo de latencia del procesador segmentado sea el mínimo posible, es necesario que el procesador esté *equilibrado*, es decir, que todas las subtareas en que se haya dividido la tarea total tarden en procesarse el mismo tiempo. Esto es debido a que las tareas no podrán avanzar a la etapa siguiente hasta que no se haya terminado la subtarea más lenta. Por ello, si el procesador no está equilibrado, las etapas más rápidas estarán cierto tiempo sin hacer trabajo alguno, lo que disminuirá el rendimiento total del procesador.

La *relación de precedencia* de un conjunto de subtareas $T_1, \ldots, T_n$ que componen cierta tarea T, especifica para cada subtarea T_j que no puede comenzarse hasta que hayan terminado ciertas subtareas

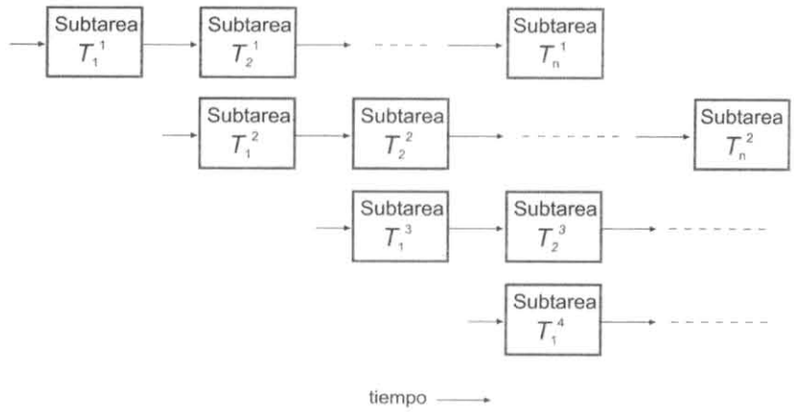


Figura 1.10: Conjunto de tareas ejecutadas en un procesador segmentado.

Por tanto, no es necesario ningún acceso a memoria cuando se ejecuta una instrucción ALU, con lo que la escritura en el registro destino puede realizarse, en teoría, inmediatamente después de la etapa EX. Sin embargo, para homogeneizar la escritura en las instrucciones ALU con la escritura en las instrucciones de carga se retrasa la escritura a la última etapa (WB). Esto provoca un ciclo de máquina libre para las instrucciones ALU en la etapa MEM de la segmentación.

Para las instrucciones de salto condicional es necesario determinar si la condición de salto se cumple antes de actualizar el contador de programa. Como la ALU se utiliza para realizar el cálculo de las direcciones efectivas, no se puede utilizar para efectuar la evaluación de la condición de salto. Si la evaluación de la condición de salto únicamente implica comprobar un registro para determinar si es igual a cero, o si es positivo o negativo, solo será necesario añadir un comparador sencillo. Este comparador se puede introducir, como muy pronto, en la etapa EX de la segmentación, es decir, después de leer el registro de referencia en la etapa ID. Por tanto, la primera etapa de la segmentación donde se puede actualizar el contador del programa con la dirección de destino del salto, suponiendo que el salto condicional sea efectivo, es durante la etapa MEM, es decir, una vez calculada la dirección de destino y determinada la condición de salto en la etapa EX.

Las instrucciones de almacenamiento y salto no necesitan escribir en el fichero de registros y se encuentran liberadas durante la etapa WB. Además, las instrucciones de carga utilizan las cinco etapas de la segmentación, mientras que los otros tres tipos de instrucciones utilizan cuatro de las cinco etapas, véase la Figura 1.13.

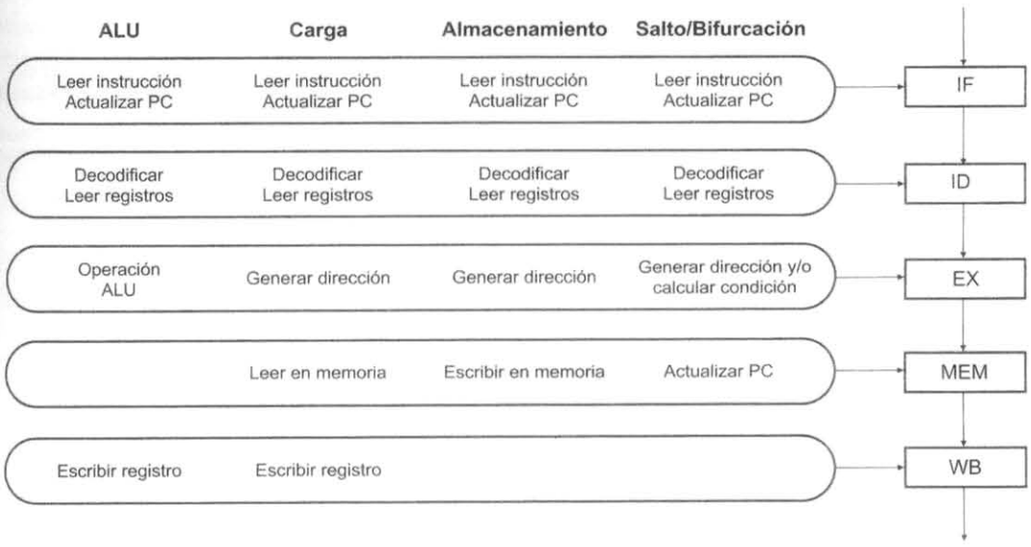


Figura 1.13: Operaciones que cada tipo de instrucción (ALU, carga, almacenamiento y salto) realiza en las etapas de que consta la segmentación ASG.

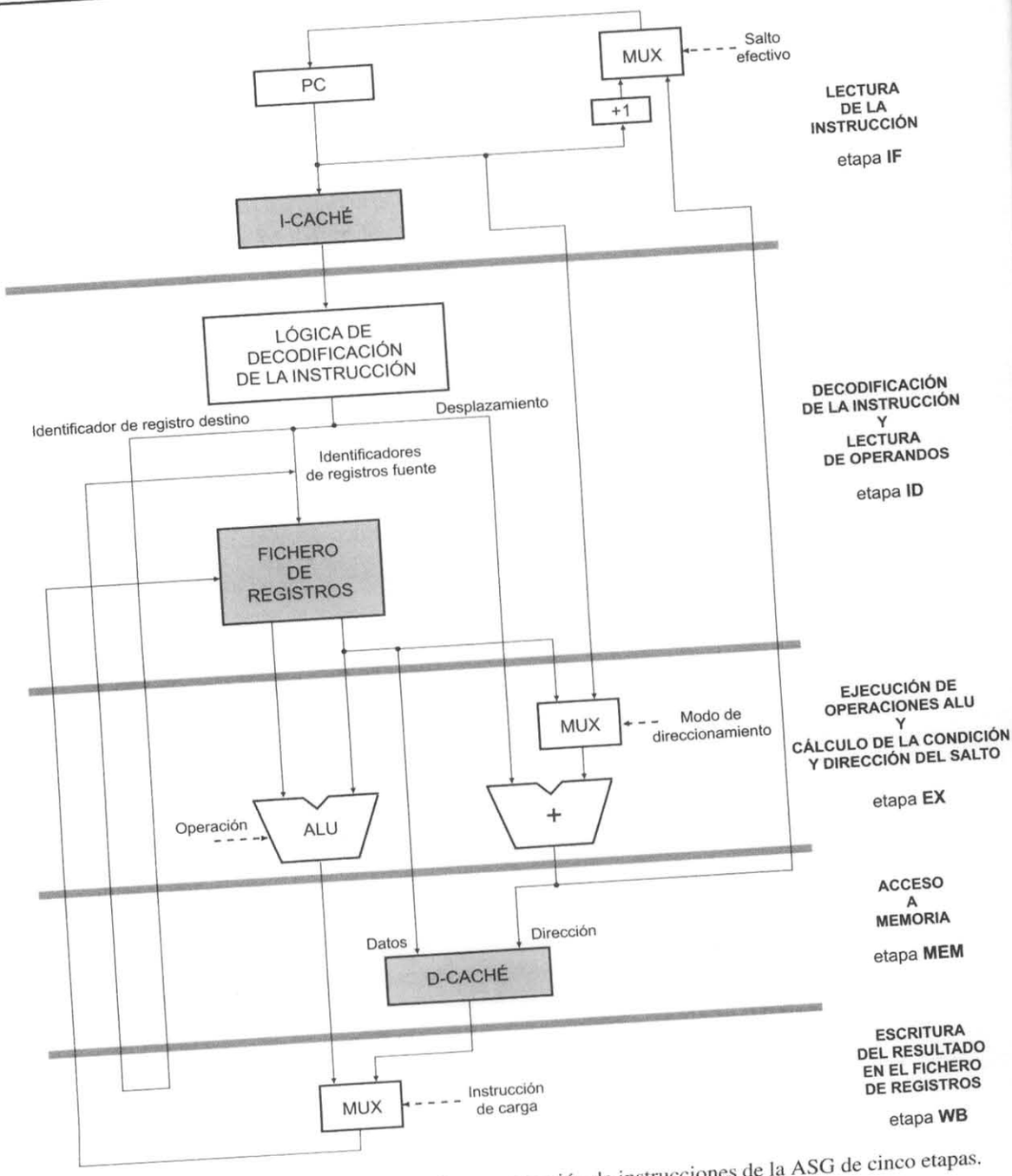


Figura 1.14: Organización lógica de la segmentación de instrucciones de la ASG de cinco etapas.

Resumiendo, la segmentación de la ASG da como patrón de ejecución el mostrado en la Figura 1.15. Se puede observar que aunque cada instrucción necesita cinco ciclos de reloj para ser ejecutada, durante cada ciclo de reloj se está ejecutando alguna de las etapas pertenecientes a cinco instrucciones diferentes.

La gran ventaja que proporciona la segmentación radica en que es posible comenzar la ejecución de una nueva instrucción en cada ciclo de reloj. Así, aunque la instrucción consume sus cinco ciclos (un ciclo por etapa), el resultado real es que en cada ciclo de reloj se concluye la ejecución de una instrucción. De esta forma y sin considerar los riesgos de la segmentación (estructurales, datos y control), el rendimiento obtenido es de cinco con respecto a la misma máquina no segmentada para el ejemplo de la Figura 1.15.

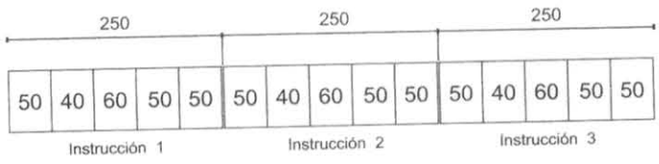
Número de instrucción	Ciclo de reloj								
	1	2	3	4	5	6	7	8	9
Instrucción *i*	IF	ID	EX	MEM	WB				
Instrucción *i*+1		IF	ID	EX	MEM	WB			
Instrucción *i*+2			IF	ID	EX	MEM	WB		
Instrucción *i*+3				IF	ID	EX	MEM	WB	
Instrucción *i*+4					IF	ID	EX	MEM	WB

Figura 1.15: Patrón de ejecución obtenido en la segmentación de la ASG al procesar cinco instrucciones.

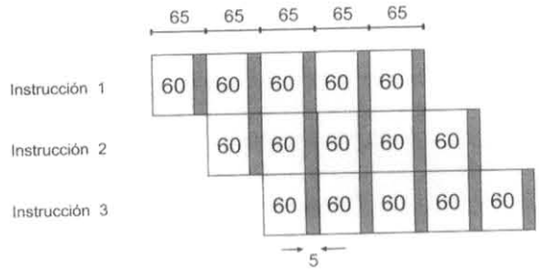
Sin embargo, aunque la segmentación incrementa la productividad de la CPU, es decir, el número de instrucciones completadas por unidad de tiempo, no reduce el tiempo de ejecución de una instrucción individual. Por el contrario, el tiempo total de ejecución de la instrucción segmentada es ligeramente superior al de su equivalente no segmentada debido al tiempo que se consume en el control de la segmentación. Este tiempo viene determinado por varios factores:

- Los cerrojos o buffers de contención que hay que colocar en el camino de datos del procesador con el objeto de aislar la información entre etapas.
- La duración de todas las etapas de la segmentación es similar y viene determinada por la duración de la etapa más lenta. Ello es debido a que el paso de información de etapa a etapa se debe realizar de forma síncrona por lo que todas las etapas deben tener la misma duración.
- Los riesgos que se producen en la segmentación y que introducen detenciones en el cauce.

La Figura 1.16 muestra el patrón de ejecución de tres instrucciones para las versiones segmentada y no segmentada de ASG. La duración de las etapas de la segmentación oscila entre 40 y 60 nseg., con lo que la instrucción no segmentada consume un total de 250 nseg., mientras que su equivalente segmentada emplea 325 nseg. debido a que todas las etapas duran lo que la más lenta, 60 nseg., y hay que sumar los retardos producidos por los cerrojos en cada etapa, 5 nseg.



(a) Ejecución no segmentada



(b) Ejecución segmentada

Figura 1.16: Patrón de ejecución segmentada y no segmentada de tres instrucciones.

1.8. Riesgos en la segmentación

Una vez unificados todos los tipos de instrucción en una segmentación y definida la funcionalidad de todas las etapas de la segmentación, es posible realizar un análisis de la segmentación para identificar todos los riesgos que se pueden producir en la segmentación. Los riesgos de segmentación son consecuencia tanto de la organización de la segmentación como de las dependencias entre las instrucciones. La dependencia entre instrucciones provoca que la instrucción que sucede a aquella con la cual posee dependencias no pueda ejecutarse en el ciclo de reloj que le corresponde, ya que ha de esperar algún resultado para poder efectuar su ejecución. Se denomina *riesgo* a la situación que impide a una instrucción acceder a la ejecución de sus etapas al depender de otra anterior. Los riesgos se traducen en una parada en el flujo de las instrucciones dentro de las etapas de la segmentación a la espera de que la dependencia causante se resuelva.

Las causas de los riesgos pueden ser varias. A continuación, se consideran los tres posibles tipos de riesgos:

- *Riesgos estructurales*: Surgen de conflictos por los recursos, es decir, por insuficiencia del hardware debido a que una instrucción ubicada en una etapa no puede avanzar a la siguiente porque el hardware que necesita está siendo utilizado por otra.
- *Riesgos por dependencia de datos*: Surgen cuando una instrucción necesita los resultados de otra anterior, que todavía no los tiene disponibles por no haberse terminado de ejecutar completamente.

- *Riesgos de control*: Se originan a partir de las instrucciones de control de flujo (saltos y bifurcaciones) debido a que no se puede leer la instrucción siguiente hasta que no se conozca su dirección, que precisamente es calculada por la propia instrucción de control de flujo.

En las siguientes secciones se analizan estos riesgos con más detalle.

1.8.1. Riesgos estructurales

Cuando se segmenta una máquina, la ejecución solapada de las instrucciones requiere la segmentación de las unidades funcionales y la duplicación de ciertos recursos para permitir todas las posibles combinaciones de instrucciones en el cauce. Si alguna combinación de instrucciones no se puede realizar debido a conflictos por los recursos se dice que la máquina tiene un *riesgo estructural*.

Como ejemplo considérese una máquina segmentada que comparte un único puerto de memoria para datos e instrucciones. En este caso, cuando una instrucción contenga una referencia a la caché de datos,

Acceso a memoria para buscar un dato

Conflicto al acceder a memoria

	Ciclo de reloj									
Instrucción	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción de carga	IF	ID	EX	MEM	WB					
Instrucción *i*+1		IF	ID	EX	MEM	WB				
Instrucción *i*+2			IF	ID	EX	MEM	WB			
Instrucción *i*+3				IF	ID	EX	MEM	WB		
Instrucción *i*+4					IF	ID	EX	MEM	WB	

Acceso a memoria para buscar una instrucción

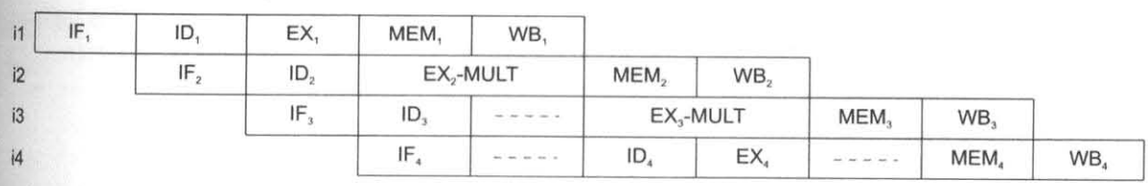
Solución al conflicto al acceder a memoria

	Ciclo de reloj									
Instrucción	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Instrucción de carga	IF	ID	EX	MEM	WB					
Instrucción *i*+1		IF	ID	EX	MEM	WB				
Instrucción *i*+2			IF	ID	EX	MEM	WB			
Instrucción *i*+3				detención	IF	ID	EX	MEM	WB	
Instrucción *i*+4						IF	ID	EX	MEM	WB

Figura 1.17: Detención durante un riesgo estructural en una máquina con un único puerto de memoria cuando se ejecuta una instrucción de carga seguida de cuatro instrucciones aritmético-lógicas.

```
i1: ADD   R3, R6, R7
i2: MULT  R4, R1, R2
i3: MULT  R5, R8, R9
i4: SD    7(R11), R10
```

(a)



(b)

Figura 1.18: Ejemplo de riesgo estructural cuando hay instrucciones más complejas que otras.

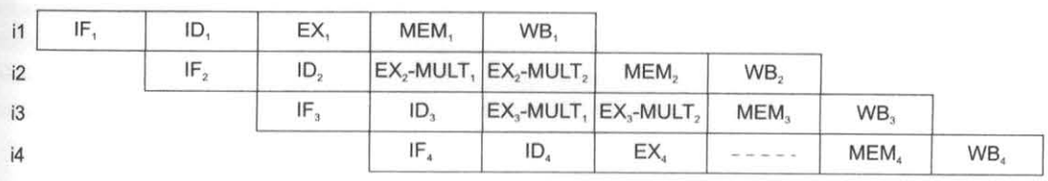


Figura 1.19: Duplicación de la unidad funcional aritmética. Se tiene un ciclo de espera en la ejecución de estas 4 instrucciones.

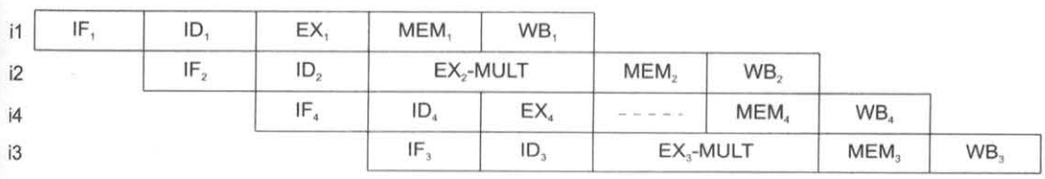


Figura 1.20: Planificación de código del tal manera que no se solapen dos etapas EX de dos instrucciones en el mismo ciclo de reloj. Se tiene un ciclo de espera en la ejecución de estas 4 instrucciones.

En este caso se pueden plantear dos alternativas para evitar las detenciones en las etapas de la segmentación. Una primera solución sería segmentar o duplicar la unidad funcional aritmética de tal forma que se puedan solapar dos etapas EX de dos instrucciones en el mismo ciclo de reloj (Figura 1.19). Otra solución sería separar, si se puede, las dos instrucciones de multiplicar (siendo ahora el orden de ejecución de las instrucciones `i1-i2-i4-i3`) como se muestra en la Figura 1.20, lo que se denomina *planificación de código*. Con estas soluciones se tiene únicamente un ciclo de detención en la segmentación a diferencia de los dos ciclos que se tenían en el caso inicial (Figura 1.18.b). Es importante

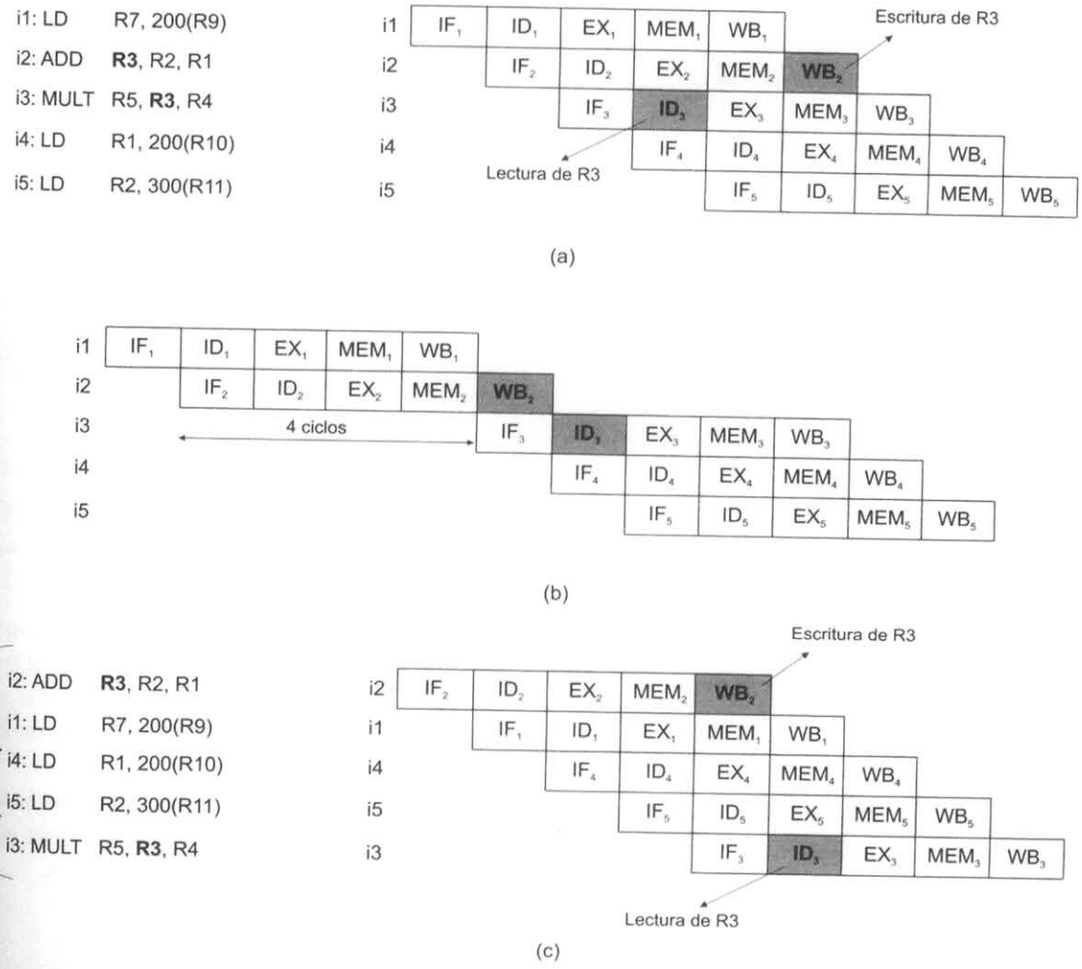


Figura 1.24: Ejemplo de riesgo por dependencia de datos RAW con reorganización de código.

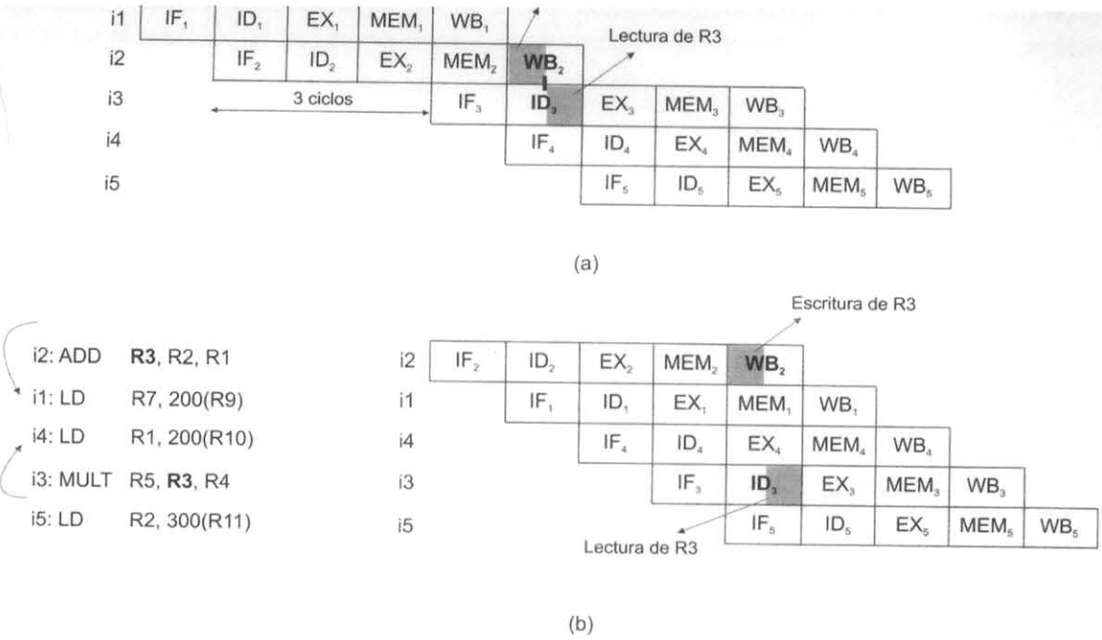


Figura 1.25: Ejemplo de riesgo por dependencia de datos RAW sin y con reorganización de código cuando la escritura se realiza en el flanco de subida y la lectura en el flanco de bajada.

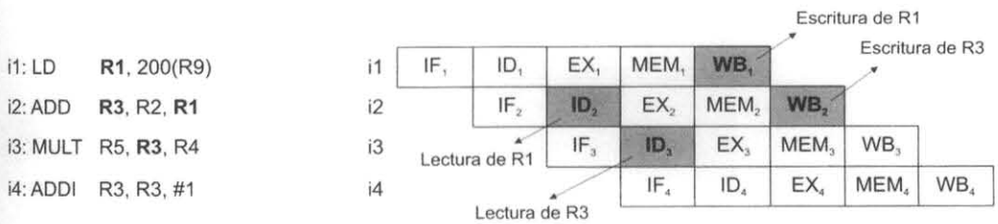
Considérese el fragmento de código de la Figura 1.26.a en el que existen dependencias de tipo RAW entre las instrucciones `i1`, `i2` e `i3` a causa de los registros R1 y R3. En este caso se observa que las instrucciones no se pueden reordenar sin alterar la lógica del programa, por lo que el compilador inserta las instrucciones `NOP` necesarias para evitar la ejecución errónea de instrucciones por dependencia de datos como se muestra en la Figura 1.26.b.

La ventaja de la solución de la reorganización de código es que no se requiere un hardware adicional pero se necesita un compilador más complejo y una pérdida de tiempo si es necesario insertar instrucciones `NOP` cuando no se puede reordenar el programa para insertar instrucciones útiles.

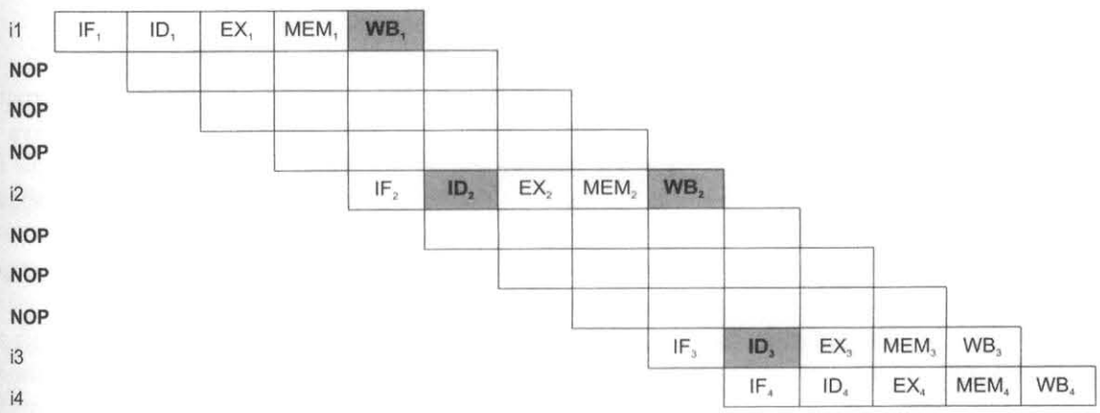
1.8.2.2. El interbloqueo entre etapas

En este método se introducen elementos hardware en el cauce para detectar la existencia de dependencias. En el caso de que se detecte una, la instrucción que debe leer el resultado proporcionado por la primera se detiene el número de ciclos necesario. Mediante esta técnica se consigue que el programa termine correctamente (igual que cuando se introduce una instrucción `NOP`), pero se siguen perdiendo ciclos sin que se terminen instrucciones, no evitándose una disminución del rendimiento.

La Figura 1.27 muestra un ejemplo de esta técnica. La instrucción `i3` queda detenida en su etapa IF_3



(a)



(b)

Figura 1.26: Ejemplo de riesgo por dependencia de datos RAW insertando instrucciones NOP.

hasta que la instrucción `i2` escribe en R3 en su etapa WB_2. En ese momento la instrucción `i3` ya puede leer el contenido de R3 en su etapa ID_3. La instrucción `i4` está detenida durante tres ciclos porque la `i3` todavía no ha abandonado la etapa IF de la segmentación.

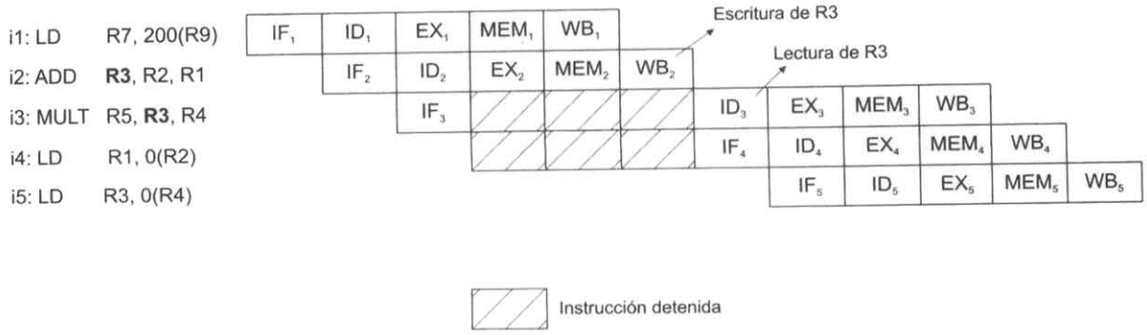


Figura 1.27: Ejemplo de riesgo por dependencia de datos RAW con interbloqueo entre etapas.

1.8.2.3. El adelantamiento (*caminos de bypass* o *forwarding*)

Mediante esta técnica se pueden aprovechar los elementos que en la técnica de interbloqueo permiten detectar la existencia de dependencias entre instrucciones. Sin embargo, esta información ahora se aprovecha para habilitar una serie de caminos (buses) que se añaden al cauce para permitir que los resultados de una etapa pasen como entradas a la etapa donde son necesarios en caso de dependencias RAW, al mismo tiempo que siguen su camino para almacenarse en el fichero de registros.

En la Figura 1.28 se muestra un esquema de las modificaciones que hay que introducir en las etapas ID y EX de la segmentación de la Figura 1.14 para implementar esta técnica.

Como se puede observar en la Figura 1.28, para implementar esta técnica se han introducido un par de multiplexores en cada una de las entradas de la ALU y una lógica de comparación para posibilitar el *bypass*. Las entradas de estos multiplexores son los identificadores de los registros de los operandos fuente y el identificador del registro destino en el que la ALU escribirá su resultado. Los buses que conectan el registro `Resultado ALU` con la entrada correspondiente en los multiplexores se muestran en la figura con un trazo más grueso, y son los que forman el *atajo* o *camino de bypass* que permite adelantar el resultado obtenido al final de la etapa EX, en el caso de que hubiera una dependencia de datos de tipo RAW entre dos instrucciones consecutivas. Para determinar la existencia de ese riesgo es para lo que se utiliza la *lógica de bypass* o *lógica de atajo*. Este circuito comprueba si hay coincidencia entre el identificador del registro de destino `Rd` de la instrucción que acaba su etapa EX (entrada 3 a la lógica de bypass) y los identificadores de los registros fuente de los operandos (`Rf1` y `Rf2`) de la siguiente instrucción, que va a iniciar su etapa EX (entradas 1 y 2 a la lógica de bypass). Si existe coincidencia con `Rf1` se activa la entrada 4 del multiplexor para conectar el registro `Resultado ALU` con la entrada de la ALU en lugar de que sea el registro `Operando 1` el utilizado. Se procede de igual forma si la

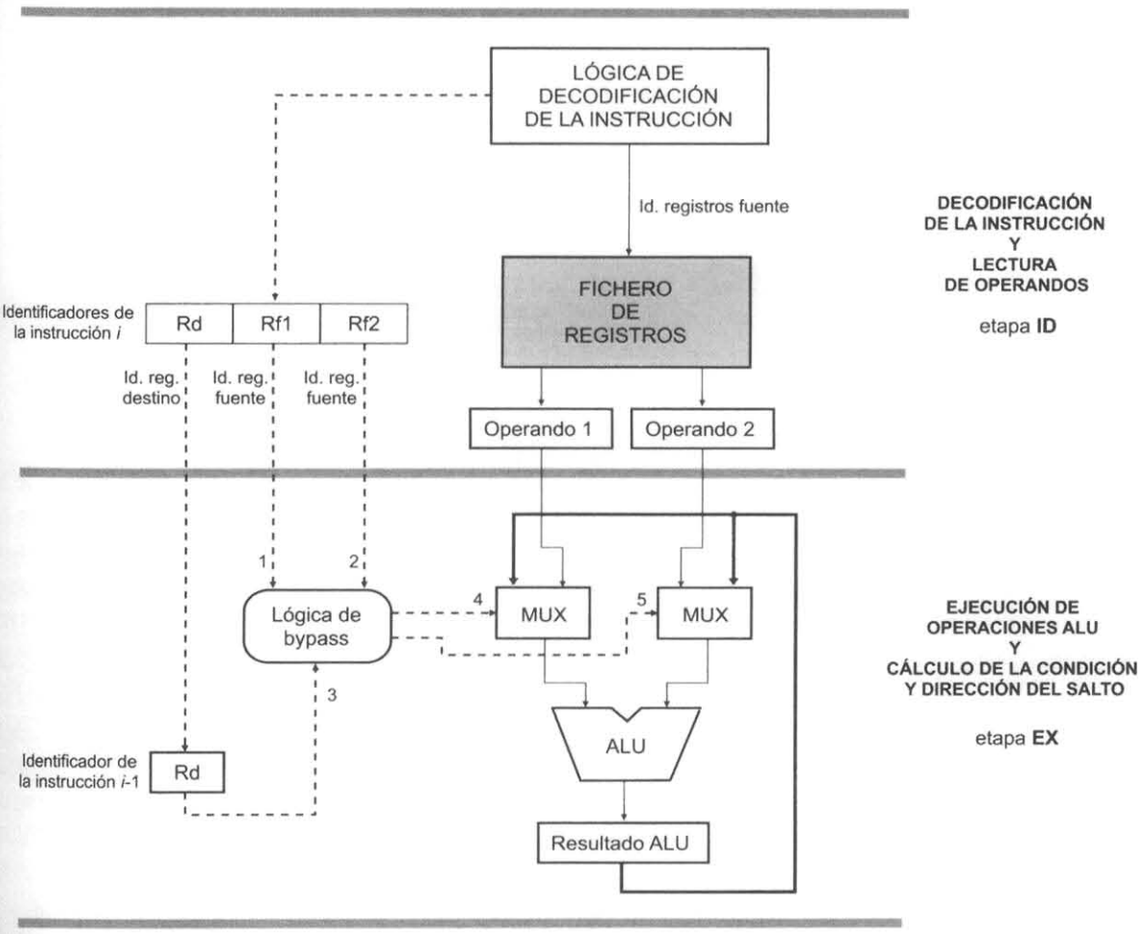


Figura 1.28: Solución con bypass o atajo para riesgos por dependencia de datos RAW.

coincidencia es entre Rd y Rf2. En este caso se activa la entrada 5 del otro multiplexor para que se utilice como entrada a la ALU el registro `Resultado ALU` en lugar del registro `Operando 2`.

En la Figura 1.29 se muestra un ejemplo de ejecución de dos instrucciones con dependencia RAW que ilustra el funcionamiento de la técnica de atajo o bypass, indicando la sucesión temporal de los eventos asociados a la gestión del riesgo.

1.8.3. Riesgos de control

Cuando se ejecuta un salto condicional, el valor del contador del programa puede incrementarse automáticamente o cambiar su valor en función de que el salto sea efectivo o no efectivo. En la ASG, si

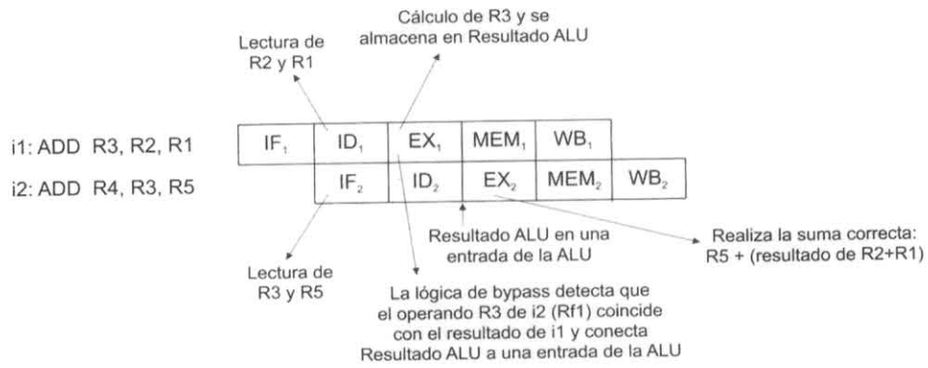
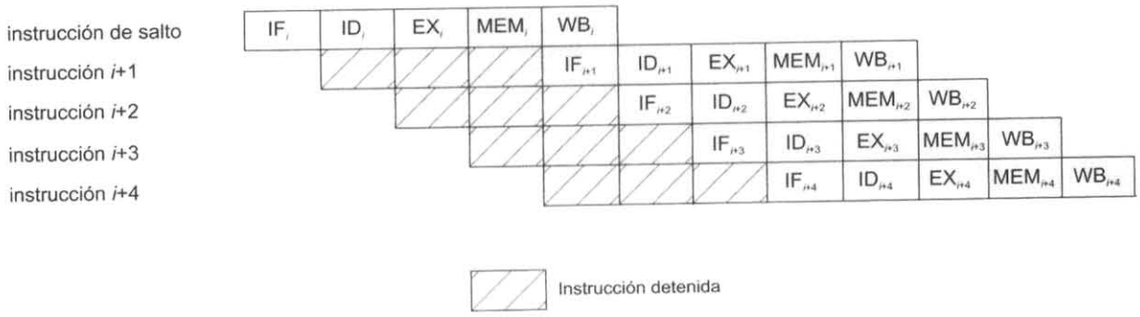


Figura 1.29: Ejemplo del funcionamiento de la técnica de atajo o bypass.

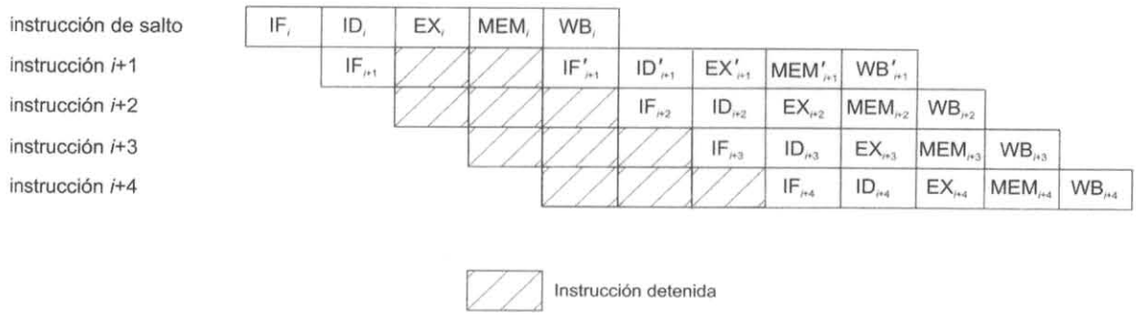
la instrucción `i` es un salto efectivo entonces el PC no se actualiza hasta el final de la etapa MEM de la segmentación, una vez que se haya verificado la condición y calculado la nueva dirección del PC en la etapa EX. Esto significa que la detención en el cauce es de tres ciclos de reloj, al final de los cuales el nuevo PC es conocido y se puede buscar la instrucción de destino del salto. Esta situación se denomina *riesgo de control* o *riesgo de salto*. La Figura 1.30 muestra una detención de tres ciclos ocasionada por un riesgo de control debido a que en ASG la instrucción de destino de salto no se conoce hast el final de la etapa MEM. La Figura 1.31 soluciona este inconveniente debido a que se busca la siguiente instrucción en la secuencia, la `i+1`, aunque se ignora y se vuelve a comenzar la búsqueda de la instrucción correcta una vez que se conoce el destino del salto.



La instrucción *i*+1 puede corresponderse con la siguiente en la secuencia del programa o estar alejada cuando el salto sea efectivo.

Figura 1.30: Detención ideal de la ASG debido a un riesgo de control.

En la Figura 1.32 se muestra un esquema de las modificaciones que hay que introducir en las etapas IF, ID y EX de la segmentación de la Figura 1.14 para procesar las instrucciones de salto condicional. En la figura aparece explícitamente el PC, que en cada ciclo se incrementa automáticamente para apuntar a la siguiente instrucción a buscar (en condiciones normales se incrementa en 4 bytes si la memoria



La instrucción *i*+1 puede corresponderse con la siguiente en la secuencia del programa o estar alejada cuando el salto sea efectivo. Si el salto no es efectivo, la etapa IF'$_{i+1}$ es redundante para la instrucción *i*+1.

Figura 1.31: Detención real de la ASG después de un riesgo de control.

se direcciona por bytes o en 1 si se direcciona por palabras de 4 bytes). Como en las instrucciones de salto condicional el PC debe cargarse con la dirección de destino del salto (si éste es efectivo), existe un multiplexor que permite controlar el valor de carga del PC: el valor actual incrementado (si el salto no es efectivo) o el valor correspondiente a la dirección de destino (si el salto es efectivo) que se ha calculado en la etapa EX. La señal que controla el camino habilitado por el multiplexor se genera mediante el módulo denominado *Lógica de condición*, y tiene en cuenta los bits de condición de la instrucción (si igual a cero, si distinto a cero, si mayor, etc.), el tipo de salto que codifica el código de la instrucción de salto cargada (salto condicional o bifurcación) y el registro fuente con el que se comprueba la condición (`Operando 1`). Si se verifica la condición de salto, se genera la señal de entrada del multiplexor con el resultado de la suma de PC y el registro `Operando 2` (almacena el valor que hay que sumar al PC para obtener la dirección del salto) para que se cargue en el PC.

Aunque existen varios métodos para tratar los riesgos de control, a continuación se comentan únicamente los esquemas que se aplican en tiempo de compilación. En estos esquemas, las predicciones sobre si el salto es efectivo o no son estáticas ya que son fijas para cada salto y no tienen en cuenta el historial del salto. En el Capítulo 2 se estudiarán detalladamente las técnicas de predicción dinámicas.

El esquema más fácil de implementar en ASG es detener la segmentación hasta que se conoce el resultado del salto, introduciendo tres instrucciones de no operación (`NOP`) después de cada instrucción de salto condicional. En este caso se reduce el rendimiento de la segmentación ya que no se termina ninguna instrucción durante tres ciclos. La Figura 1.33 muestra con un ejemplo la secuencia de eventos que se producen cuando se introducen tres instrucciones de no operación después de un salto condicional.

Otra alternativa sería dejar que las instrucciones que se han captado prosigan su ejecución en el cauce. En este caso, el compilador debería introducir en esos huecos instrucciones que se tengan que ejecutar antes de la instrucción destino de salto de forma que su efecto sea independiente de que el salto sea efectivo o no. En la Figura 1.34 se muestra un ejemplo de esta situación. Las instrucciones `i1`, `i2` e `i3` deben ejecutarse antes de que se produzca el salto `i4`. Como son independientes de la instrucción de salto

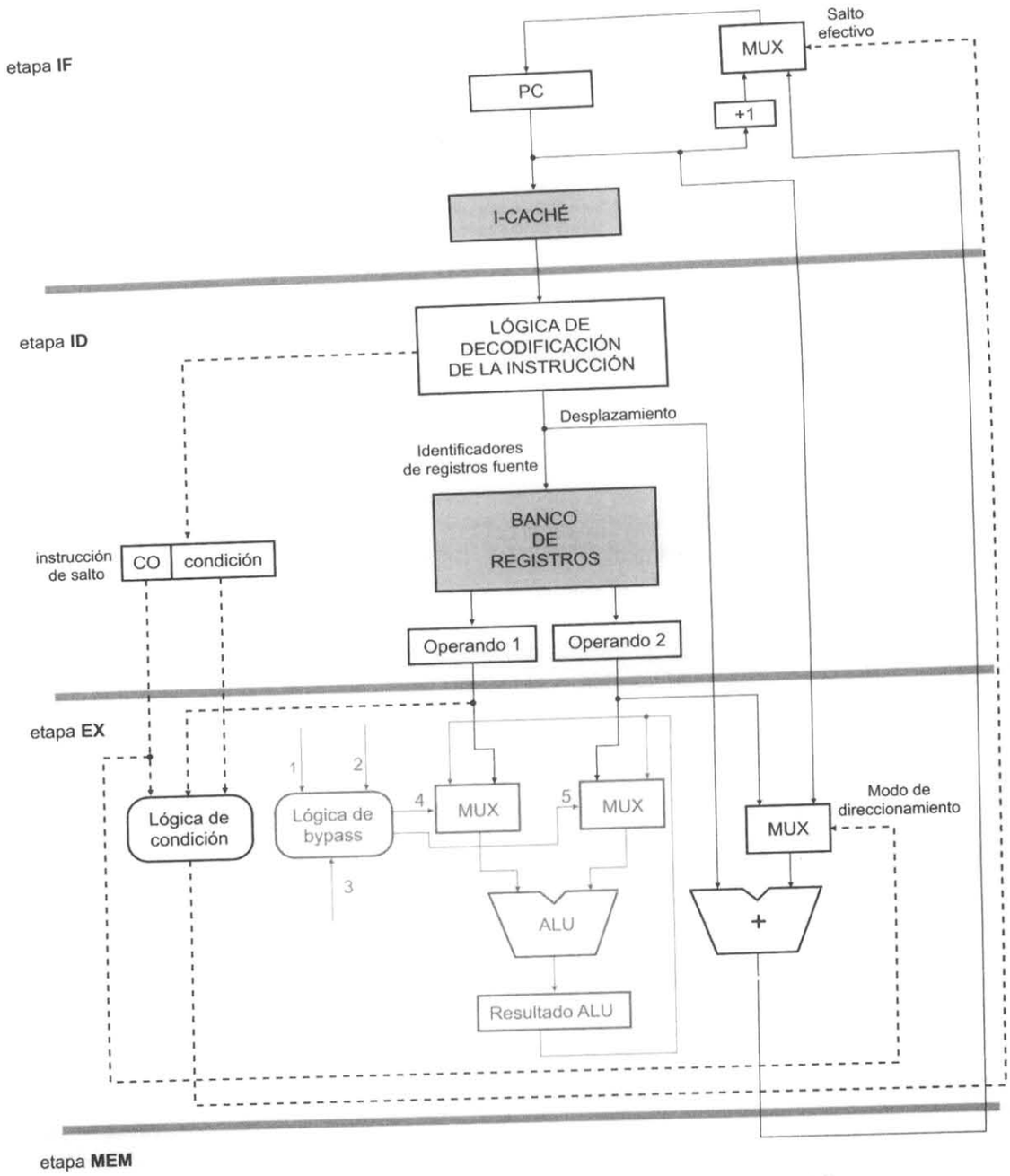


Figura 1.32: Implementación de los saltos condicionales en la ASG.

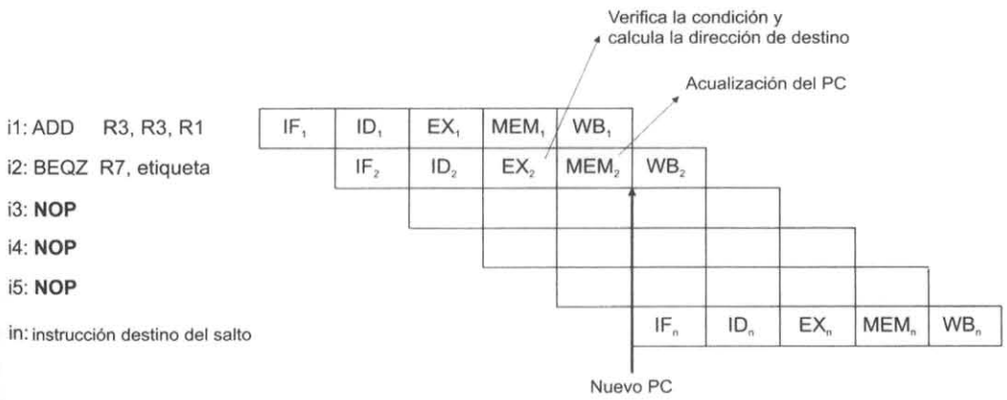
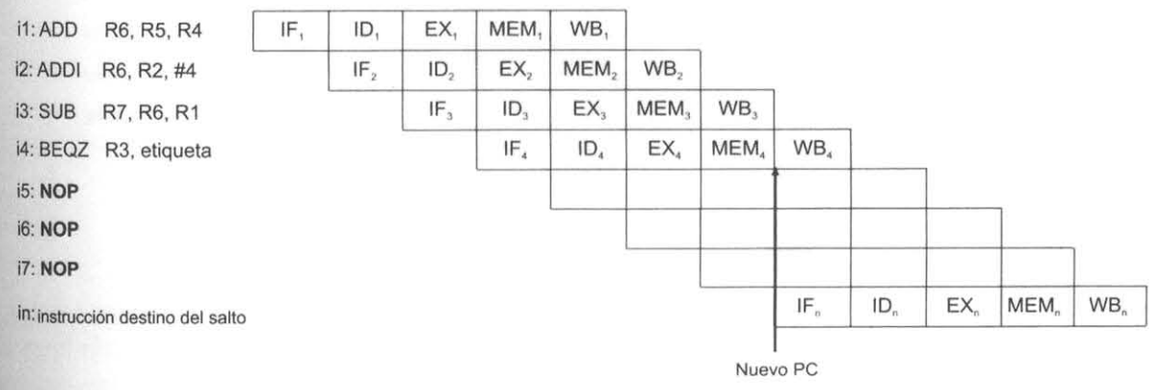
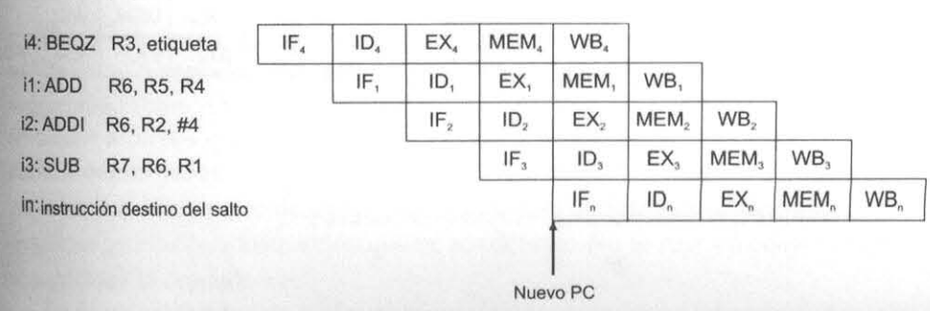


Figura 1.33: Uso de instrucciones NOP en la ejecución de una instrucción de salto condicional para no abortar instrucciones captadas erróneamente.





debido a que no hay dependencias de datos se podrían ubicar justo después de la instrucción de salto. De esta forma se ejecutarían las instrucciones `i1`, `i2` e `i3` y después la instrucción que debe seguir a la instrucción de salto, sea éste efectivo o no, tal como debía ocurrir en la situación inicial. Gracias a esta técnica, el cauce puede continuar terminando una instrucción por ciclo, mejorando así su rendimiento. Esta técnica se conoce con el nombre de *salto retardado* (*delayed branch*).

Un esquema mejor y solo ligeramente más complejo es predecir el salto como no efectivo, permitiendo que el hardware continúe procesando la siguiente instrucción en la secuencia del programa como si el salto fuese efectivo. En esta técnica, denominada *ejecución especulativa*, hay que tener cuidado de no cambiar el estado de la máquina[7] hasta que no se conozca definitivamente el resultado del salto ya que se produce la ejecución de instrucciones que, dependiendo del resultado del salto, podrían no haberse tenido que ejecutar. En el caso de que el salto sea efectivo será necesario detener la segmentación y recomenzar la búsqueda de la instrucción destino del salto. Por tanto, las instrucciones que se estaban

[7]El estado de la máquina viene definido por el contador de programa, el fichero de registros y el espacio de memoria asignado al programa que ocupa el procesador en un instante dado.

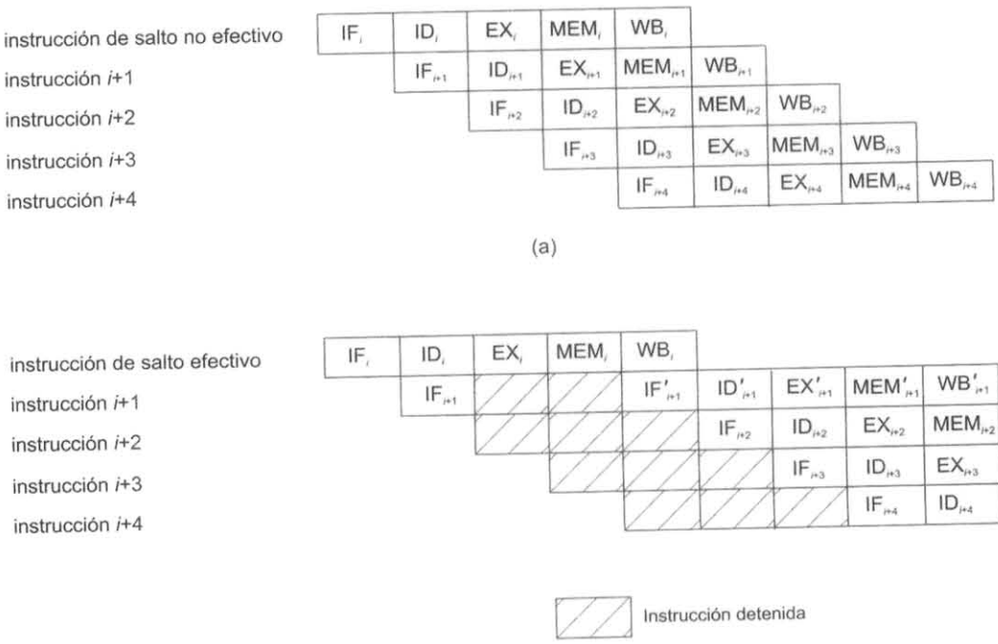


Figura 1.35: Esquema de predecir un salto como no efectivo y la secuencia de la segmentación cuando no es efectivo (a) y cuando es efectivo (b).

ejecutando por detrás de la instrucción de salto se eliminan. Esta situación provoca una pérdida de tres ciclos de reloj. En el caso de que el salto no sea efectivo no penaliza con pérdidas de ciclo de reloj. La Figura 1.35 muestra ambas situaciones.

Finalmente, también se contribuiría a mejorar considerablemente el rendimiento del procesador frente a los saltos si se adelantara el cálculo de la dirección y de la condición de la instrucción de salto a las primeras etapas del cauce. Por ejemplo, si se organiza convenientemente el hardware de la etapa ID para que en dicha etapa se realice la determinación de que la instrucción es de salto, el acceso al registro en el que se evalúa la condición, así como la obtención de la dirección del destino del salto (se requiere un sumador que pueda sumar en la etapa ID), solo habrá una detención de un ciclo de reloj en los saltos frente a las tres que se tenían en el caso de no añadir este hardware adicional (Figura 1.31). La Figura 1.36 muestra la segmentación cuando hay una instrucción de salto y se tiene el hardware adicional necesario para tratar el salto en la etapa ID.

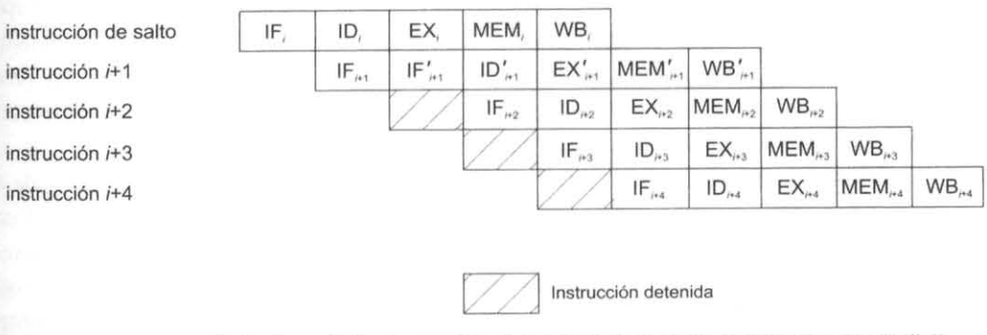


Figura 1.36: Detención real de la ASG después de un riesgo de control cuando se tiene hardware adicional para tratar el salto en la etapa ID.

1.9. Planificación dinámica: Algoritmo de Tomasulo

Hasta este momento se ha considerado la planificación estática como la única técnica que se utiliza en un procesador segmentado para realizar el procesamiento de las instrucciones. Un procesador segmentado como ASG con planificación estática busca una instrucción y la envía a la unidad funcional (*emisión*) a menos que haya una dependencia verdadera entre esa instrucción y otra que ya esté en el cauce y que no pueda ser evitada mediante adelantamiento. Si hay dependencias tipo RAW que no pueden ser resueltas entonces se detienen, empezando con la instrucción dependiente que usa el resultado que tiene que generar otra instrucción que va por delante. No se busca o emite ninguna instrucción hasta que se solucione la dependencia.

La principal limitación de las técnicas de segmentación estática es que emiten las instrucciones en

La decodificación de instrucciones (ID) precede a la etapa de emisión. La ejecución (etapa EX) es la etapa siguiente a la emisión (II). Como el cauce permite tener múltiples instrucciones en ejecución al mismo tiempo, será necesario disponer de múltiples unidades funcionales (hipótesis que se asume).

Para mostrar las diferencias en la ejecución entre un código no planificado y el mismo código usando planificación dinámica, se supone una configuración en la que la latencia de la unidad de multiplicación es de cinco ciclos y la de la unidad de suma de dos ciclos permitiéndose el adelantamiento. Se parte del fragmento de código que se muestra en la Figura 1.37.a. En este código existe una dependencia de datos entre la primera y la segunda instrucción, que será la que resalte las diferencias entre la ejecución sin planificación y con planificación dinámica. Esta dependencia de datos influirá en ambas ejecuciones, tal y como se ve en la Figura 1.37, ya que en ambos casos detiene la ejecución de la segunda instrucción. Lo que varía es el tratamiento de aquellas instrucciones que son posteriores a la que tiene el riesgo, pero que no tienen ninguna dependencia y, por lo tanto, podrían ejecutarse correctamente.

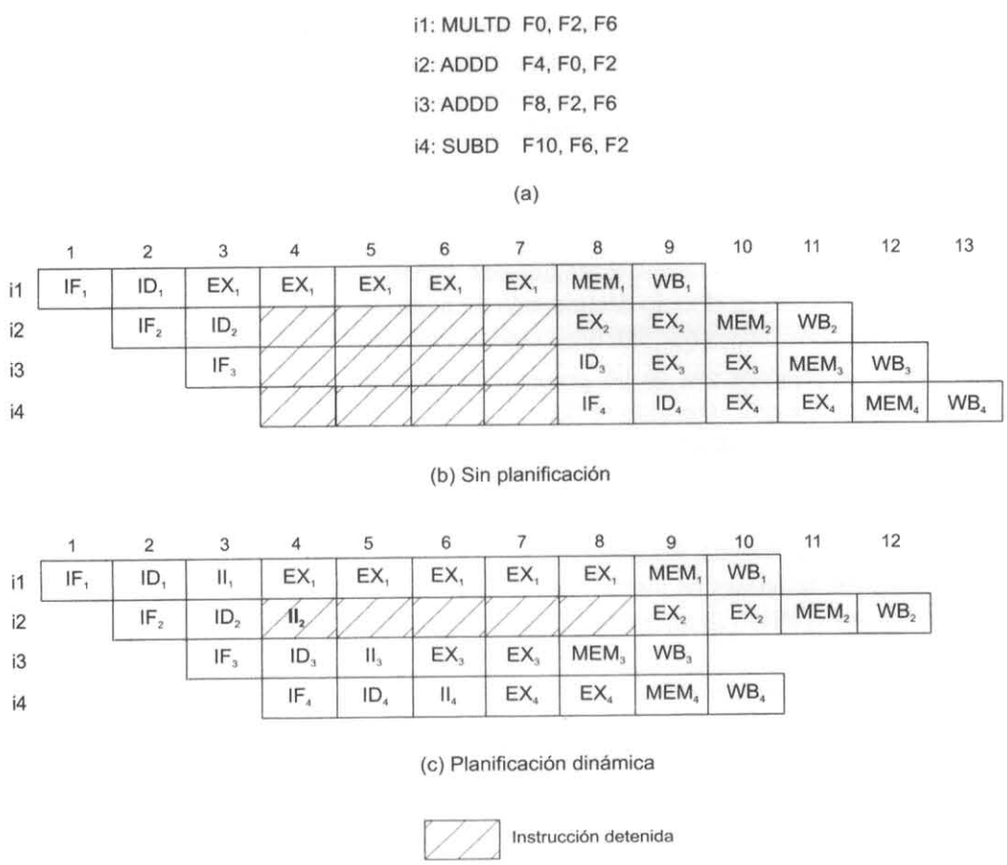


Figura 1.37: Ejemplo de la ejecución de un fragmento de código sin planificación y con planificación dinámica.

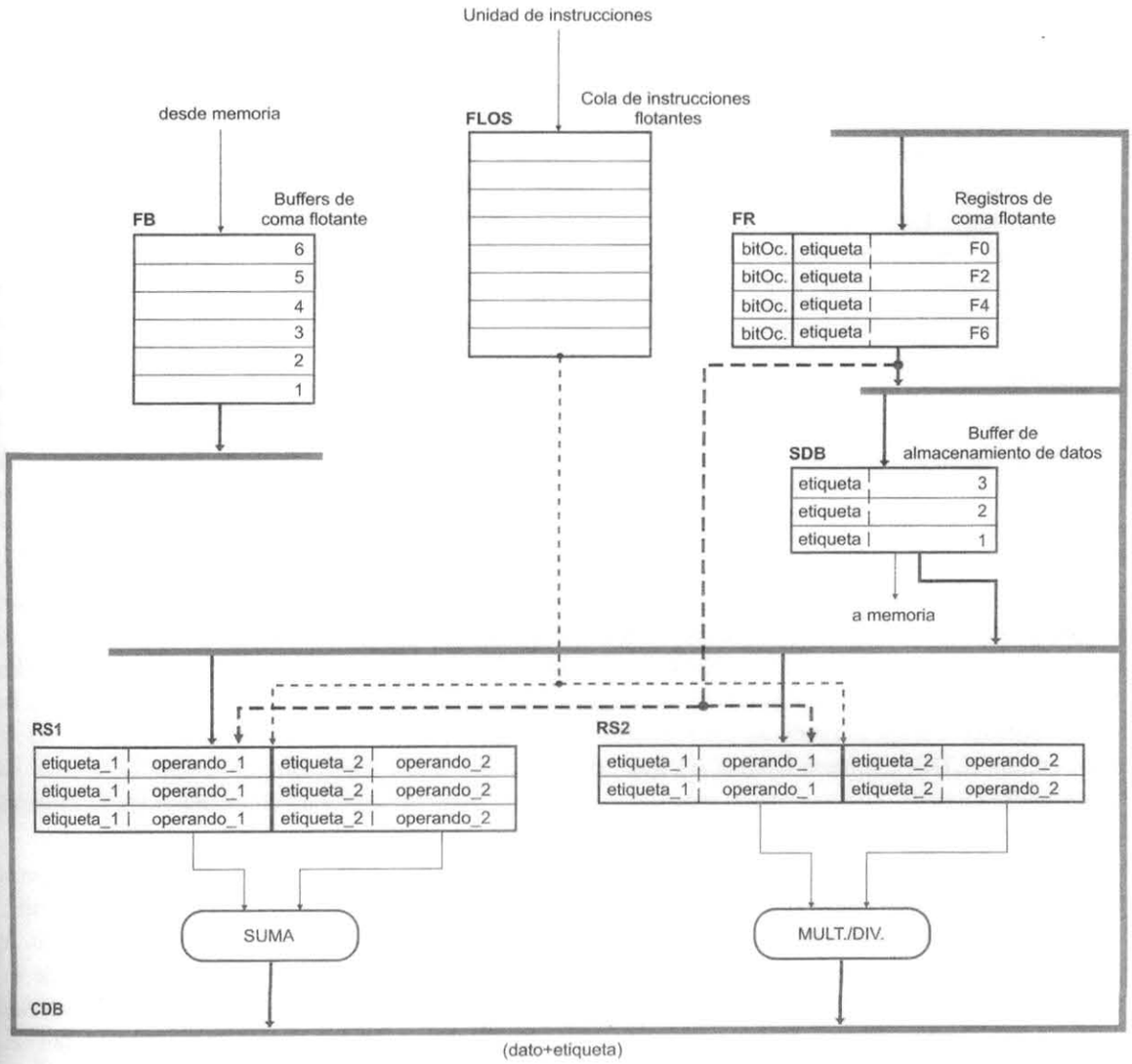


Figura 1.38: Diseño modificado de la unidad de coma flotante del IBM 360/91 con el algoritmo de Tomasulo.

i1: ADDD F4, F0, F6

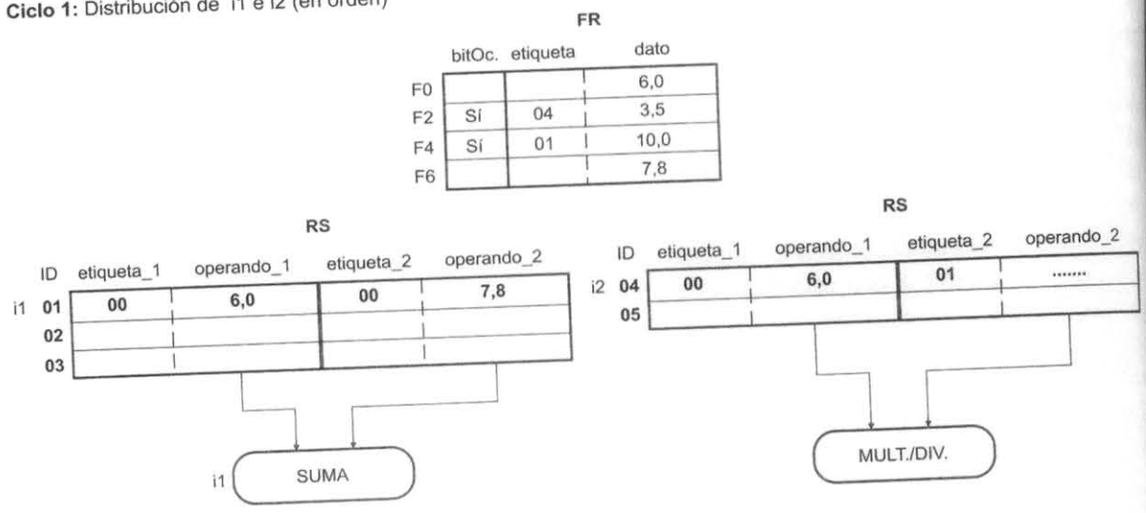
i2: MULTD F2, F0, F4

i3: ADDD F4, F4, F6

i4: MULTD F6, F4, F2

(a) Secuencia de instrucciones

Ciclo 1: Distribución de i1 e i2 (en orden)



Ciclo 2: Distribución de i3 e i4 (en orden)

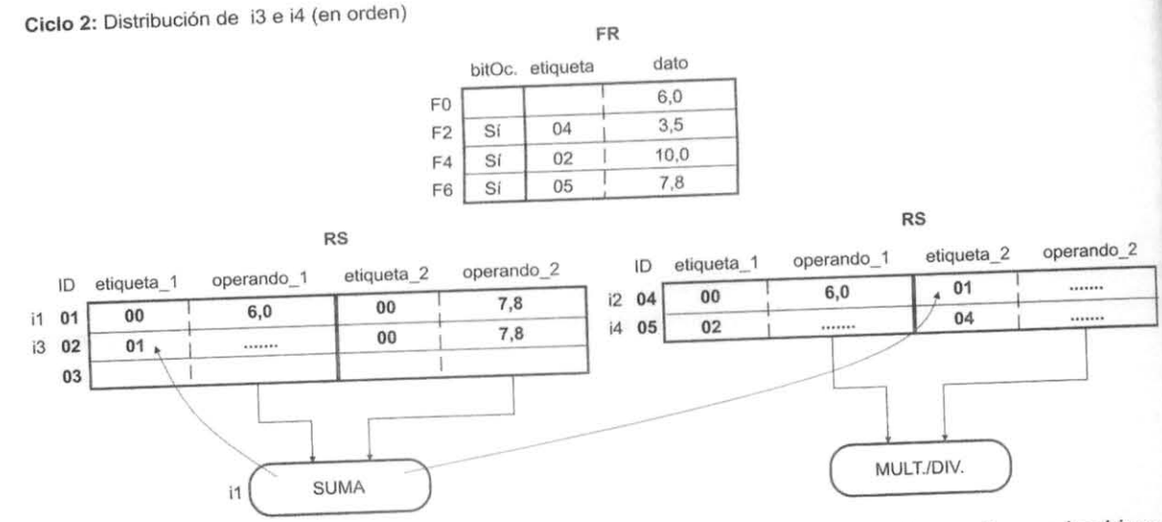
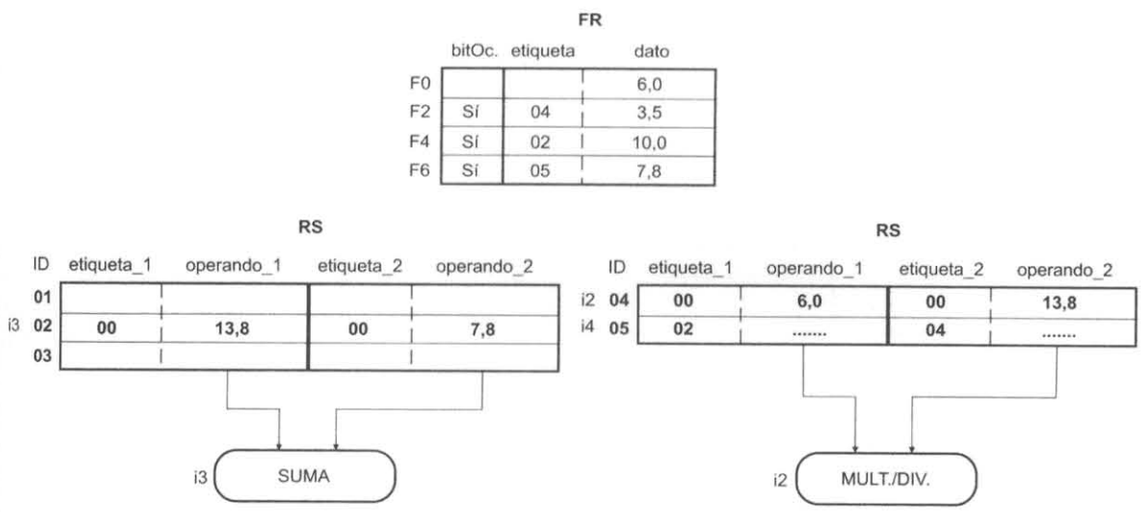


Figura 1.39: Ilustración del algoritmo de Tomasulo para un fragmento de código con instrucciones aritméticas en coma flotante (continúa).

Ciclo 3:



Ciclo 4:

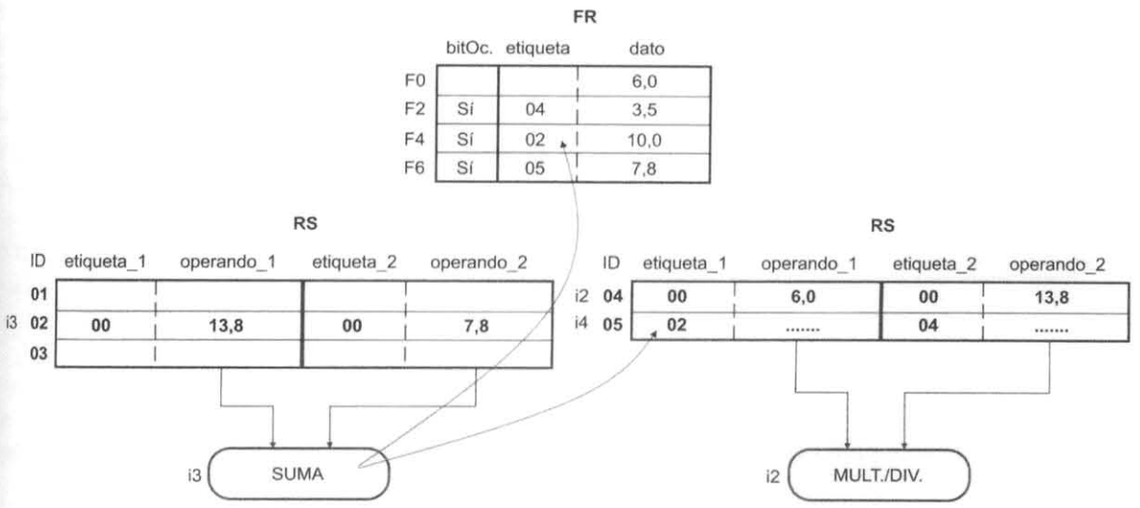
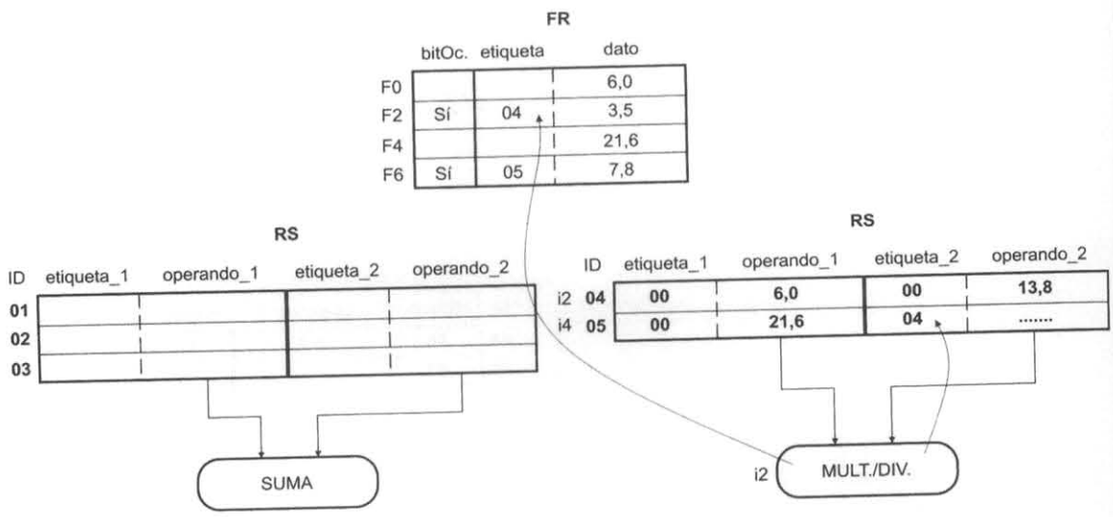


Figura 1.39: [Continuación] Ilustración del algoritmo de Tomasulo para un fragmento de código con instrucciones aritméticas en coma flotante (continúa).

Ciclo 5:



Ciclo 6:

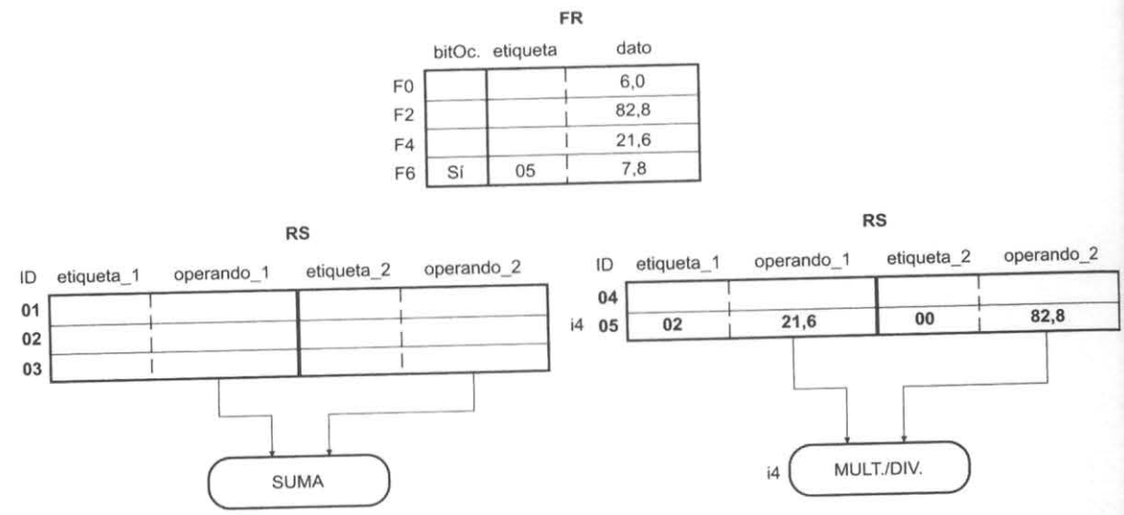


Figura 1.39: [Continuación] Ilustración del algoritmo de Tomasulo para un fragmento de código con instrucciones aritméticas en coma flotante.