


carse en el flanco elegido. Con este tipo de metodología de sincronización, no existe baipás dentro de un mismo ciclo, y la lógica de la figura 4.4 funciona correctamente. En el  **apéndice C** se analizan brevemente otras limitaciones de la sincronización —como los tiempos de preestabilización (*setup times*) y de mantenimiento (*hold times*)— así como otras metodologías.

En la arquitectura MIPS de 32 bits, casi todos estos elementos lógicos y de estado tendrán entradas y salidas de 32 bits, ya que ésta es la anchura de muchos de los datos tratados por el procesador. Se marcará de alguna forma cuando una unidad tenga una entrada o una salida con una anchura diferente. Las figuras mostrarán los buses (señales con anchura superior a un bit) mediante líneas más gruesas. Cuando se quiera combinar varios buses para formar uno de anchura superior, por ejemplo, si se quiere tener un bus de 32 bits combinando dos de 16, las etiquetas de dichos buses especificarán que hay varios buses agrupados. También se añaden como ayuda flechas para destacar la dirección del flujo de datos entre los elementos. Finalmente, el **color** indicará si se trata de una señal de control o de datos. Esta distinción se especificará mejor más adelante en este capítulo.


Verdadero o falso: ya que el banco de registros es leído y escrito en el mismo ciclo de reloj, cualquier camino de datos MIPS que utilice escrituras por flanco debe tener más de una copia del banco de registros.

Extensión: Existe una versión de 64 bits de la arquitectura MIPS y, naturalmente, la mayor parte de los componentes tienen un ancho de 64 bits. Por otra parte, se usan los términos activado y desactivado porque a veces el 1 representa el estado lógico alto y otras veces el estado lógico bajo.

4.3

Construcción de un camino de datos

Una forma razonable de empezar el diseño de un camino de datos es examinar los componentes principales necesarios para ejecutar cada tipo de instrucción del MIPS. Primero se consideran los **elementos del camino de datos** que necesita cada instrucción. Cuando se muestran los elementos del camino de datos, también se muestran sus señales de control.

La figura 4.5a muestra el primer elemento que se necesita: una unidad de memoria donde almacenar y suministrar las instrucciones a partir de una dirección. La figura 4.5b también muestra un registro, denominado **contador de programa (PC)**, que hemos visto en el capítulo 2 y se utiliza para almacenar la dirección de la instrucción actual. Finalmente se necesita un sumador encargado de incrementar el PC para que apunte a la dirección de la siguiente instrucción. Este sumador, que es combinacional, se puede construir a partir de la ALU descrita en detalle en el  **apéndice C**, haciendo simplemente que las líneas de control siempre especifiquen una operación de suma. Este sumador se representa como una ALU con la eti-

Autoevaluación

Elemento del camino de datos: unidad funcional utilizada para operar o mantener un dato dentro de un procesador. En la implementación MIPS, los elementos del camino de datos incluyen las memorias de instrucciones y datos, el banco de registros, la unidad aritmético-lógica (ALU), y sumadores.

Contador de programa (PC): registro que contiene la dirección de la instrucción del programa que está siendo ejecutada.

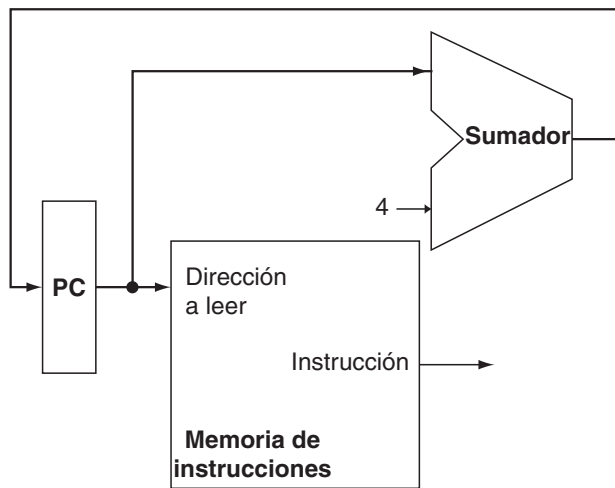


FIGURA 4.6 Parte del camino de datos utilizado para la búsqueda de las instrucciones y el incremento del contador de programa. La instrucción cargada se utiliza en otras partes del camino de datos.

presentará el valor correspondiente. Para escribir un valor se necesitan dos entradas: una para especificar el número de registro donde escribir y otra para suministrar el dato. El banco de registros siempre devuelve en sus salidas los contenidos de los registros cuyos identificadores están en las entradas de los registros a leer. Las escrituras se controlan mediante una señal de control de escritura, que debe estar activa para que una escritura se lleve a cabo en el flanco de reloj. Así, se necesitan un total de cuatro entradas (tres para los números de los registros y uno para los datos) y dos salidas (ambas de datos), como se puede ver en la figura 4.7a. Las entradas de los identificadores de registro son de cinco bits para especificar uno de los 32 ($32 = 2^5$) registros, mientras que los buses de la entrada y las dos salidas de datos son de 32 bits.

La figura 4.7b muestra la ALU, que tiene dos entradas de 32 bits y produce un resultado de 32 bits, así como una señal de 1 bit que se activa si el resultado es 0. La señal de control de cuatro bits de la ALU se describe en detalle en el [apéndice C](#); repasaremos el control de la ALU brevemente cuando necesitemos saber cómo establecerlo.

Consideremos ahora las instrucciones del MIPS de carga y almacenamiento de palabras, las cuales tienen el formato general `lw $t1, despl($t2) o sw $t1, offset($t2)`. Estas instrucciones calculan la dirección de memoria añadiendo al registro base (`$t2`), el campo de desplazamiento con signo de 16 bits contenido en la instrucción. Si la instrucción es un almacenamiento, el valor a almacenar debe leerse del registro especificado por `$t1`. En el caso de una carga, el valor leído de memoria debe escribirse en el registro especificado por `$t1` en el banco de registros. Así, se necesitarán tanto el banco de registros como la ALU que se muestran en la figura 4.7.

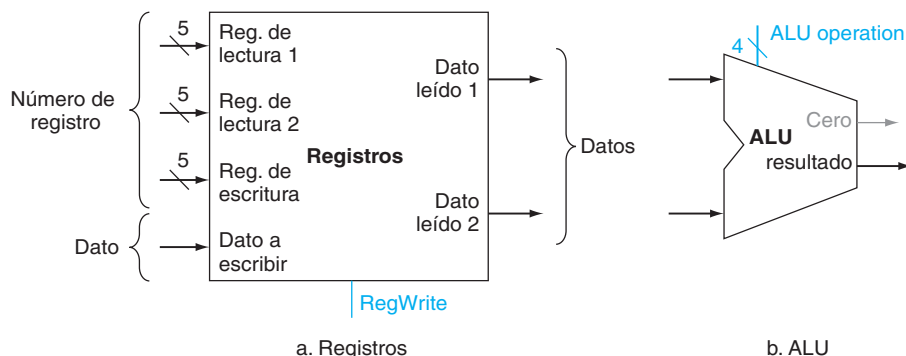


FIGURA 4.7 El banco de registros y la ALU son los dos elementos necesarios para la implementación de instrucciones de tipo R. El banco de registros contiene todos los registros y tiene dos puertos de lectura y uno de escritura. El diseño de los bancos de registros multipuerto se analiza en la sección C8 del [apéndice C](#). El banco de registros siempre devuelve a la salida el contenido de los registros correspondientes a los identificadores que se encuentran en las entradas de los registros a leer; sin ser necesaria ninguna otra entrada de control. En cambio, la escritura de un registro debe indicarse explícitamente mediante la activación de la señal de control de escritura. Recuerde que las escrituras se sincronizan por flanco, por lo que todas las señales implicadas (el valor a escribir, el número de registro y la señal de control de escritura) deben ser válidas en el flanco de reloj. Por esta razón, este diseño puede leer y escribir sin ningún problema el mismo registro en un mismo ciclo: la lectura obtiene el valor escrito en un ciclo anterior, mientras que el valor que se escribe ahora estará disponible en ciclos siguientes. Las entradas que indican el número de registro al banco son todas de 5 bits, mientras que las líneas de datos son de 32 bits. La operación que realiza la ALU se controla mediante su señal de operación, que es de 4 bits, utilizando la ALU diseñada en el [apéndice C](#). La salida de detección de Cero se utiliza para la implementación de los saltos condicionales. La salida de desbordamiento (*overflow*) no se necesitará hasta la sección 4.9, cuando se analicen las excepciones; por lo tanto se omitirá hasta entonces.

Extensión de signo: incrementar el tamaño de un dato mediante la replicación del bit de signo del dato original en los bits de orden alto del dato destino más largo.

Dirección destino de salto: dirección especificada en un salto que se convierte en la nueva dirección del contador de programas (PC) si se realiza el salto. En la arquitectura MIPS, el destino del salto viene dado por la suma del campo de desplazamiento de la instrucción y la dirección de la instrucción siguiente al salto.

Además se necesita una unidad para **extender el signo** del campo de desplazamiento de 16 bits de la instrucción a un valor con signo de 32 bits, y una unidad de memoria de datos para leer y escribir. La memoria de datos se escribe con instrucciones almacenamiento; por lo que tiene señales de control de escritura y de lectura, una entrada de dirección y una entrada de datos a escribir en memoria. La figura 4.8 muestra estos dos elementos.

La instrucción `beq` tiene tres operandos, dos registros que se comparan para comprobar su igualdad y un desplazamiento de 16 bits utilizado para calcular la **dirección destino del salto** relativa a la dirección de la instrucción. Su formato es `beq $t1, $t2, offset`. Para realizar esta instrucción se debe calcular la dirección destino del salto sumando el campo de desplazamiento con el signo extendido al PC. Hay dos detalles en la definición de las instrucciones de salto condicional (véase el capítulo 2) a los cuales se debe prestar atención:

- La arquitectura del repertorio de instrucciones especifica que es la dirección de la siguiente instrucción en orden secuencial la que se utiliza como base para el cálculo de la dirección destino. Puesto que se calcula el $PC + 4$ (la dirección de la siguiente instrucción) en el camino de datos de la carga de instrucciones, es fácil utilizar este valor como base para el cálculo de la dirección destino del salto.

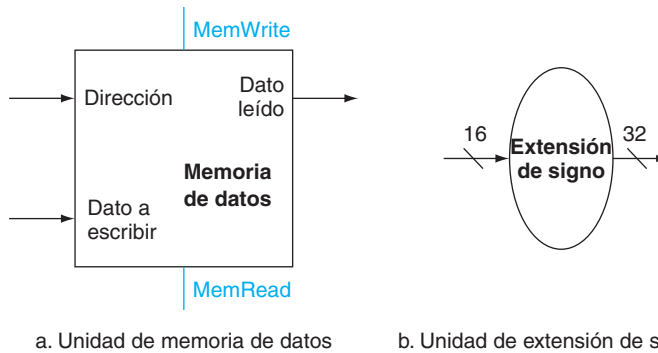


FIGURA 4.8 Las dos unidades necesarias para la implementación de cargas y almacenamientos, además del banco de registros y la ALU de la figura 4.7, son la unidad de memoria de datos y la unidad de extensión de signo. La unidad de memoria es un elemento de estado que tiene como entradas la dirección y el dato a escribir, y como única salida el valor leído. Hay controles separados de escritura y de lectura, aunque sólo uno de los dos puede estar activo en un momento determinado. La unidad de memoria requiere también una señal de lectura, ya que a diferencia del caso del banco de registros, la lectura de una dirección no válida puede causar problemas, como se verá en el capítulo 5. La unidad de extensión de signo tiene una entrada de 16 bits que se extenderá a los 32 bits que aparecen en la salida (véase capítulo 2). Se supone que la memoria de datos sólo escribe en el flanco. De hecho, los chips estándar de memoria tienen una señal de habilitación de escritura y, aunque esta señal no es con flanco, este diseño podría fácilmente adaptarse para trabajar con chips reales de memoria. Véase la sección C.8 del [apéndice C](#) para analizar más ampliamente cómo trabajan estos chips.

- La arquitectura también impone que el campo de desplazamiento se desplace hacia la izquierda 2 bits para que este desplazamiento corresponda a una palabra; de forma que se incrementa el rango efectivo de dicho campo en un factor de 4.

Para tratar la última complicación se necesita desplazar dos posiciones el campo de desplazamiento.

Además de calcular la dirección destino del salto, también se debe determinar si la siguiente instrucción a ejecutar es la que sigue secuencialmente o la situada en la dirección destino del salto. Cuando la condición se cumple (es decir, los operandos son iguales), la dirección calculada pasa a ser el nuevo PC, y se dice que el **salto condicional se ha tomado**. Si los operandos son diferentes, el PC incrementado debería reemplazar al PC actual (igual que para cualquier otra instrucción); en este caso se dice que el **salto no se ha tomado**.

Resumiendo, el camino de datos para saltos condicionales debe efectuar dos operaciones: calcular la dirección destino del salto y comparar el contenido de los registros (los saltos condicionales también requieren que se modifique la parte de carga de instrucciones del camino de datos, como se verá más adelante). La figura 4.9 muestra el camino de datos de los saltos condicionales. Para calcular la dirección destino del salto, el camino de datos incluye una unidad de extensión de signo, como en la figura 4.8, y un sumador. Para la comparación se necesita utilizar el banco de registros mostrado en la figura 4.7a a fin de obtener los dos regis-

Salto tomado: salto en el que se cumple la condición de salto y el contador de programa (PC) es cargado con la dirección destino. Todos los saltos incondicionales son saltos tomados.

Salto no tomado: salto donde la condición de salto es falsa y el contador de programa (PC) se carga con la dirección de la instrucción siguiente al salto.

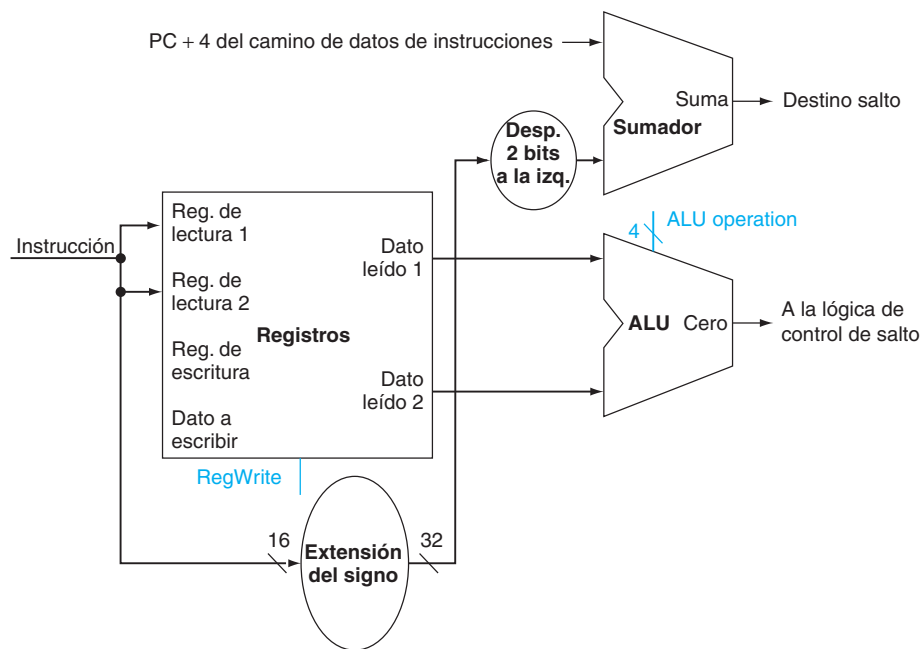


FIGURA 4.9 El camino de datos para un salto condicional utiliza la ALU para evaluar la condición de salto y un sumador aparte para calcular la dirección destino del salto como la suma del PC incrementado y los 16 bits de menor peso de la instrucción con el signo extendido (el desplazamiento de salto) y desplazado dos bits a la izquierda. La unidad etiquetada como «desp. 2 bits a la izq.» es simplemente un encaminamiento de las señales entre la entrada y la salida que añade 00_{dos} en la zona baja del campo de desplazamiento. No se necesita circuitería para el desplazamiento, ya que la cantidad a desplazar es constante. Como se sabe que el desplazamiento se extiende desde 16 bits, esta operación sólo desperdicia bits de signo. La lógica de control se utiliza para decidir si el PC incrementado o el destino del salto deberían reemplazar al PC, basándose en la salida Cero de la ALU.

tros de operando (aunque no será necesario escribir en él) y la ALU (diseñada en el [apéndice C](#)) para realizar dicha operación. Ya que esta ALU proporciona una señal de salida que indica si el resultado era 0, se le pueden enviar los dos operandos con la señal de control activada de forma que efectúe una resta. Si la señal Cero a la salida de la ALU está activa, entonces los dos valores son iguales. Aunque la salida Cero siempre indica si el resultado es 0, sólo se utiliza para realizar el test de igualdad en saltos condicionales. Más tarde se estudiará de forma más exacta cómo se conectan las señales de control de la ALU para utilizarla en el camino de datos.

La instrucción de salto incondicional reemplaza los 28 bits de menor peso del PC con los 26 bits de menor peso de la instrucción desplazados 2 bits hacia la izquierda. Este desplazamiento se realiza simplemente concatenando 00 a estos 26 bits (como se describe en el capítulo 2).

Extensión: En el repertorio de instrucciones del MIPS, los **saltos condicionales se retardan**, lo cual significa que la siguiente instrucción inmediatamente después del salto se ejecuta siempre, independientemente de si la condición de salto se cumple o no. Cuando la condición es falsa, la ejecución se comporta como un salto normal. Cuando es cierta, un salto retardado primero ejecuta la instrucción inmediatamente posterior al salto y posteriormente salta a la dirección destino. El motivo de los saltos retardados surge por el modo en que les afecta la segmentación (véase sección 4.8). Para simplificar, se ignorarán los saltos retardados en este capítulo y se realizará una instrucción `beq` no retardada.

Salto retardado: tipo de salto en el que la instrucción inmediatamente siguiente al salto se ejecuta siempre, independientemente de si la condición del salto es verdadera o falsa.

Implementación de un camino de datos sencillo

Una vez que se han descrito los componentes necesarios para los distintos tipos de instrucciones, éstos pueden combinarse en un camino de datos sencillo y añadir el control para completar la implementación. El más sencillo de los diseños intentará ejecutar todas las instrucciones en un solo ciclo. Esto significa que ningún elemento del camino de datos puede utilizarse más de una vez por instrucción, de forma que cualquier recurso que se necesite más de una vez deberá estar replicado. Por tanto, la memoria de instrucciones ha de estar separada de la memoria de datos. Aunque se necesite duplicar algunas de las unidades funcionales, muchos de estos elementos pueden compartirse en los diferentes flujos de instrucciones.

Para compartir un elemento del camino de datos entre dos tipos de instrucciones diferentes, se requiere que dicho elemento disponga de múltiples conexiones a la entrada de un elemento utilizando un multiplexor, así como de una señal de control para seleccionar entre las múltiples entradas.

Construcción de un camino de datos

El camino de datos de las instrucciones aritmético-lógicas (o tipo R), así como el de las instrucciones de referencia a memoria son muy parecidos, siendo las principales diferencias las siguientes:

- Las instrucciones aritmético-lógicas utilizan como entradas de la ALU los valores procedentes de dos registros. Las instrucciones de referencia a memoria pueden utilizar la ALU para realizar el cálculo de la dirección, aunque la segunda entrada es el campo de desplazamiento de 16 bits procedente de la instrucción con signo extendido.
- El valor guardado en el registro destino, o bien proviene de la ALU (para instrucciones tipo R) o de memoria (en caso de una carga).

Determine cómo construir un camino de datos para la parte operacional de las instrucciones de referencia a memoria y aritmético-lógicas que utilice un único banco de registros y una sola ALU para soportar ambos tipos de instrucciones, añadiendo los multiplexores necesarios.

EJEMPLO

RESPUESTA

Para combinar ambos caminos de datos y usar un único banco de registros y una sola ALU, la segunda entrada de ésta ha de soportar dos tipos de datos diferentes, además de dos posibles caminos para el dato a almacenar en el banco de registros. De esta manera, se coloca un multiplexor en la entrada de la ALU y un segundo en la entrada de datos del banco de registros. La figura 4.10 muestra este nuevo camino de datos.

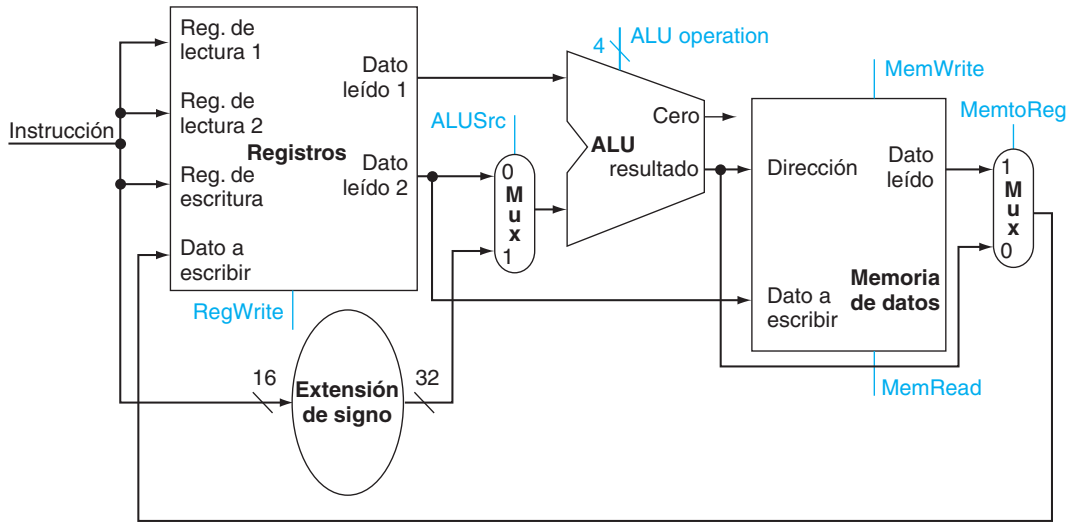


FIGURA 4.10 El camino de datos para las instrucciones de memoria y tipo R. Este ejemplo muestra cómo puede obtenerse un camino uniendo diferentes piezas (figuras 4.7 y 4.8) y añadiendo multiplexores. Se requieren dos multiplexores tal como se describe en el ejemplo.

Ahora se pueden combinar el resto de las piezas para obtener un camino de datos único para la arquitectura MIPS, añadiendo la parte encargada de la búsqueda de las instrucciones (figura 4.6), al camino de datos de las instrucciones tipo R y de referencia a memoria (figura 4.10), y el camino de datos para los saltos (figura 4.9). La figura 4.11 muestra este camino de datos obtenido al ensamblar las diferentes piezas. Las instrucciones de salto utilizan la ALU para la comparación de los registros fuente, de forma que debe ponerse el sumador de la figura 4.9 para calcular la dirección de destino del salto. También es necesario un nuevo multiplexor para escoger entre seguir en secuencia ($PC + 4$) y la dirección destino del salto para escribir esta nueva dirección en el PC.

Una vez completado este sencillo camino de datos, se puede añadir la unidad de control. Ésta debe ser capaz de generar las señales de escritura para cada elemento de estado y las de control para la ALU y para cada multiplexor a partir de las entradas. Debido a que el control de la ALU es diferente del resto en mayor o menor medida, resultaría útil diseñarlo como paso previo al diseño de la unidad de control.

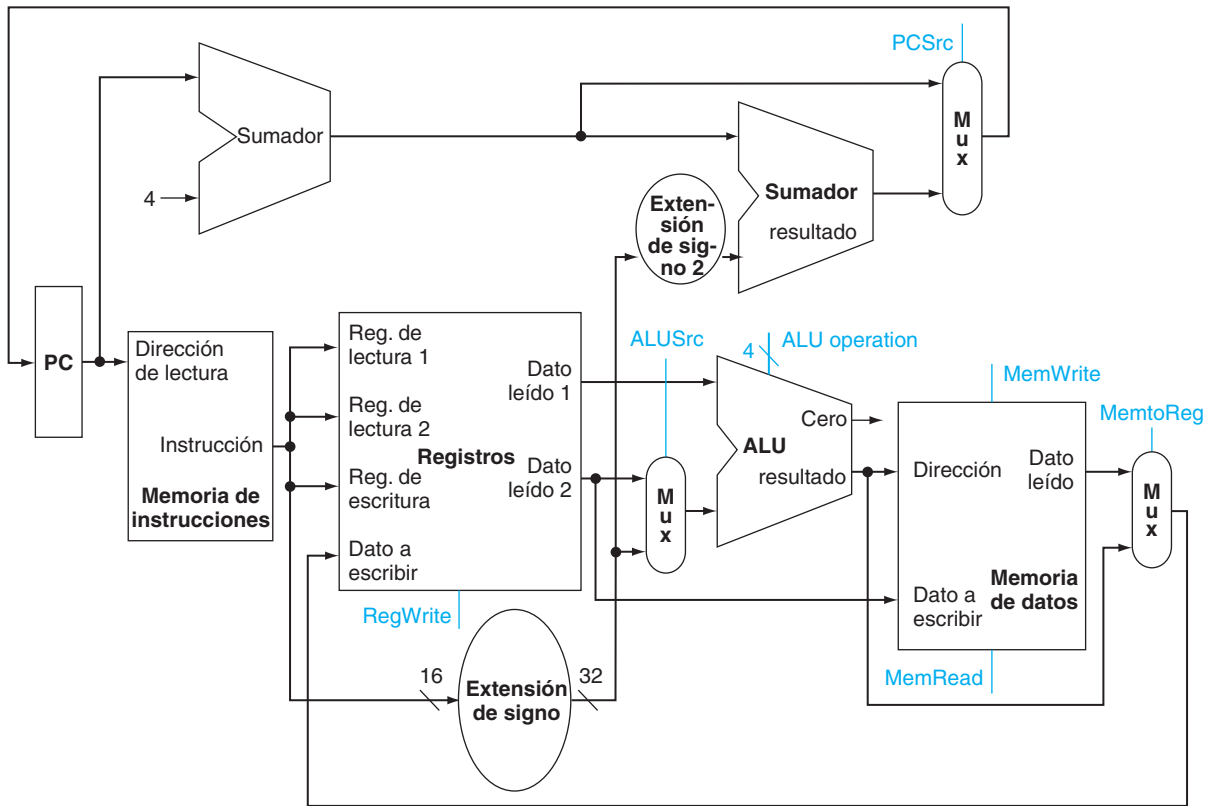


FIGURA 4.11 Un camino de datos sencillo para la arquitectura MIPS que combina los elementos necesarios para los diferentes tipos de instrucciones. Este camino de datos puede ejecutar las instrucciones básicas (carga/almacenamiento de palabras, operaciones de la ALU y saltos) en un solo ciclo de reloj. Es necesario un nuevo multiplexor para integrar saltos condicionales. La lógica necesaria para ejecutar instrucciones de salto incondicional (*jump*) se añadirá más tarde.

I. ¿Cuáles de las siguientes afirmaciones son correctas para una instrucción de carga? Utilizar la figura 4.10.

Autoevaluación

- MemtoReg debe ser activada para provocar que el dato procedente de memoria sea enviado al banco de registros.
- MemtoReg debe se activada para provocar que el registro destino correcto sea enviado al banco de registros.
- La activación de MemtoReg no es relevante.

II. El camino de datos de ciclo único descrito conceptualmente en esta sección debe tener memorias de datos e instrucciones separadas, porque

- En MIPS los formatos de datos e instrucciones son diferentes, y por lo tanto, se necesitan memorias diferentes.


- b. Tener memorias separadas es más barato.
- c. El procesador opera en un solo ciclo y no puede usar una memoria con un único puerto para dos accesos diferentes en el mismo ciclo.

4.4

Esquema de una implementación simple

En esta sección veremos la que podría considerarse como la implementación más sencilla posible de nuestro subconjunto de instrucciones MIPS. Se construirá esta sencilla implementación utilizando el camino de datos de la sección anterior y añadiendo una función de control simple. Esta implementación simple será capaz de ejecutar instrucciones de almacenamiento y carga de palabras de memoria (`lw` y `sw`), saltar si igual (`beq`), instrucciones aritméticas (`add`, `sub`, `and`, `or` y `slt`) y activar si es menor que (`set on less than`). Posteriormente se mejorará dicho diseño incluyendo la instrucción de salto incondicional (`j`).

El control de la ALU

La ALU MIPS del  [apéndice C](#) define las siguientes 6 combinaciones de 4 entradas de control:

Líneas de control de la ALU	Función
0000	Y-lógico (AND)
0001	O-lógico (OR)
0010	sumar
0110	restar
0111	iniciar si menor que
1100	NOR

Dependiendo del tipo de instrucción a ejecutar, la ALU debe realizar una de las cinco primeras operaciones (NOR es necesaria para otras partes del repertorio de instrucciones MIPS). Las instrucciones de referencia a memoria utilizan la ALU para calcular la dirección de memoria por medio de una suma. Para las instrucciones tipo R, la ALU debe ejecutar una de las cinco operaciones (`and`, `or`, `add`, `sub` y `slt`) en función del valor de los 6 bits de menor peso de la instrucción, los cuales componen el código de función (véase el capítulo 2), mientras que para las instrucciones de saltar si igual, la ALU debe realizar una resta.

Mediante una pequeña unidad de control que tiene como entradas el código de función de la instrucción y 2 bits de control adicionales que reciben el nombre de ALUOp, se pueden generar los 4 bits que conforman las señales de control de la ALU. Estos bits (ALUOp) indican si la operación a realizar debería ser o bien una suma (00) para accesos a memoria, o bien una resta (01) para saltos condicionales, o bien si la operación a realizar está codificada en el código de función (10). La

salida de la unidad de control de la ALU es una señal de 4 bits que controla la ALU codificando una de las combinaciones de 4 bits mostradas anteriormente.

En la figura 4.12 se observa el conjunto de combinaciones de las señales de entrada formadas por los 2 bits que conforman la señal de ALUOp y los 6 bits del código de función. Posteriormente, en este capítulo, se verá cómo genera la unidad de control principal los bits de ALUOp.

Código de operación	ALUOp	Operación	Campo de la función	Acción deseada de la ALU	Entrada del control de la ALU
LW	00	cargar palabra	XXXXXX	sumar	0010
SW	00	almacenar palabra	XXXXXX	sumar	0010
Branch equal	01	saltar si igual	XXXXXX	restar	0110
R-type	10	sumar	100000	sumar	0010
R-type	10	restar	100010	restar	0110
R-type	10	AND	100100	Y lógica	0000
R-type	10	OR	100101	O lógica	0001
R-type	10	activar si es menor que	101010	activar si es menor que	0111

FIGURA 4.12 Cálculo de los bits de control de la ALU en función de los bits de ALUOp y los diferentes códigos de función de las instrucciones tipo R. El código de operación, que aparece en la primera columna, determina los valores de los bits del campo ALUOp. Todas las codificaciones se muestran en binario. Obsérvese que cuando ALUOp tiene como valor 00 ó 01, la salida no depende del código de función, y se dice que su valor es no-determinado y codificado como XXXXXX. En el caso de que ALUOp valga 10, el código de la función se utiliza para calcular la señal de control de la ALU. Véase el [apéndice C](#).

Esta técnica basada en el uso de múltiples niveles de descodificación (o decodificación), es decir, la unidad de control principal genera los bits de ALUOp que se utilizarán como entrada de la unidad de control encargada de generar las señales de la ALU, es muy común. El hecho de usar múltiples niveles puede reducir el tamaño de la unidad principal. De igual manera, el uso de varias unidades de control más pequeñas puede incluso incrementar la velocidad de dicha unidad de control. Todas estas optimizaciones son importantes ya que, a menudo, la unidad de control se encuentra en el camino crítico.

Existen diferentes formas de establecer la correspondencia entre los 2 bits del campo de ALUOp y los 6 del código de función con los 3 bits que conforman la operación a realizar por la ALU. Debido a que sólo una pequeña parte de los 64 valores posibles del código de función son importantes y que dichos bits únicamente se utilizan cuando ALUOp vale 10, podría utilizarse una pequeña parte de lógica que reconociera dicho subconjunto de valores posibles y diera como resultado los valores correctos de los bits de control de la ALU.

Como paso previo al diseño de la lógica combinacional, es útil construir una tabla de verdad para aquellas combinaciones de interés de los códigos de función y de los bits de ALUOp, tal como se ha realizado en la figura 4.13. Esta tabla muestra cómo dependen de ambos campos las señales de control de la ALU. Debido a que la [tabla de verdad](#) es muy grande ($2^8 = 256$ entradas), y teniendo en cuenta que para muchas de dichas combinaciones los valores de la salida no tienen importancia, únicamente

Tabla de verdad: desde el punto de vista lógico, es una representación de una operación lógica que muestra todos los valores de las entradas y el valor de las salidas para cada caso.

ALUOp		Campo de la función						Operación
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURA 4.13 Tabla de verdad de los 3 bits de control de la ALU (también llamados operación). Las entradas son la ALUOp y el código de función. Únicamente se muestran aquellas entradas para las cuales la señal de control de la ALU tiene sentido. También se han añadido algunas entradas cuyo valor es indeterminado. Por ejemplo, el campo ALUOp no utiliza la codificación 11, de forma que la tabla de verdad puede contener las entradas 1X y X1 en vez de 10 y 01. También, cuando se utiliza el código de función, los 2 primeros bits (F5 y F4) de dichas instrucciones son siempre 10, de forma que también se consideran no-determinados y se reemplazan por XX en la tabla de verdad.

se dan los valores de las salidas para aquellas entradas de la tabla donde el control de la ALU debe tener un valor específico. Las diferentes tablas de verdad que se irán viendo a lo largo de este capítulo contendrán únicamente aquellos subconjuntos de entradas que deban estar activadas, eliminando aquellos cuyos valores de salida sean indeterminados. (Este método tiene un inconveniente que se analizará en la sección D.2 del [apéndice D](#)).

Términos indeterminados: elementos de una función lógica cuya salida no depende de los valores de todas las entradas. Los términos no-determinados pueden especificarse de distintas maneras.

Debido a que en muchos casos los valores de algunas entradas no tienen importancia, para mantener la tabla compacta incluimos **términos indeterminados**. Un término de este tipo (representado en la tabla mediante una X en la columna de entrada correspondiente) indica que la salida es independiente del valor de dicha entrada. Por ejemplo, cuando el campo ALUOp vale 00, caso de la primera fila de la tabla de la figura 4.13, la señal de control de la ALU siempre será 0010, independientemente del código de función. Es decir, en este caso, el código de la función se considera indeterminado en esta fila de la tabla de verdad. Más adelante se verán ejemplos de otro tipo de términos indeterminados. Si no se está familiarizado con este tipo de términos, véase el [apéndice C](#) para mayor información.

Una vez que se ha construido la tabla de verdad, ésta puede optimizarse y entonces pasar a su implementación mediante puertas lógicas. Este proceso es completamente mecánico. Así, en lugar de mostrar aquí los pasos finales, este proceso, así como sus resultados, se describe en la sección D.2 del [apéndice D](#).

Diseño de la unidad de control principal

Una vez que ya se ha descrito como diseñar una ALU que utiliza el código de función y una señal de dos bits como entradas de control, podemos centrarnos en el resto del control. Para iniciar este proceso se deben identificar los campos de las instrucciones y las líneas de control necesarias para la construcción del camino de datos mostrado en la figura 4.11. Para entender mejor cómo se conectan los diferentes campos de las instrucciones en el camino de datos, sería útil revisar los formatos de los tres tipos de instrucciones: el tipo R (aritmético-lógica), los saltos y las operaciones de carga y almacenamiento. Estos formatos se muestran en la figura 4.14.

Campo	0	rs	rt	rd	shamt	funct
Posición de los bits	31:26	25:21	20:16	15:11	10:6	5:0

a. Instrucción tipo R

Campo	35 o 43	rs	rt	dirección
Posición de los bits	31:26	25:21	20:16	15:0

b. Instrucción de carga o almacenamiento

Campo	4	rs	rt	dirección
Posición de los bits	31:26	25:21	20:16	15:0

c. Instrucción de salto condicional

FIGURA 4.14 Los tres tipos de instrucciones (tipo R, referencia a memoria y salto condicional) utilizan dos formatos de instrucción diferentes. La instrucción de salto incondicional hace uso de otro formato, que se explicará más adelante. (a) Formato para las instrucciones tipo R, que tiene el campo de tipo de operación a 0. Estas instrucciones disponen de tres operandos registros: rs, rt y rd, donde rs y rt son registros fuente y rd es el registro destino. La función a realizar en la ALU se encuentra en el código de función (campo *funct*) y es descodificada por la unidad de control de la ALU, tal como se ha visto en la sección anterior. Pertenecen a este tipo las instrucciones *add*, *sub*, *and*, *or* y *sllt*. El campo de desplazamiento (*shamt*) sólo se utiliza para desplazar y se ignora en este capítulo. (b) Formato de las instrucciones carga (código de operación 35_{diez}) y almacenamiento (código de operación 43_{diez}). El registro rs se utiliza como registro base al cual se le añade el campo de dirección de 16 bits para obtener la dirección de memoria. En el caso de las instrucciones de carga, rt es el registro destino del valor cargado de memoria, mientras que en los almacenamientos, rt es el registro fuente del valor que se debe almacenar en memoria. (c) Formato de las instrucciones de saltar si igual (código de operación 4_{diez}). Los registros rs y rt son los registros fuente que se compararán. Al campo de 16 bits de la dirección se le extiende el signo, se desplaza y se suma al PC para calcular la dirección de destino de salto.

Existen varias observaciones importantes a realizar sobre este formato de las instrucciones y que siempre se supondrán ciertas.

- El campo de código de operación (**opcode**) estará siempre contenido en los bits 31-26. Se referenciará dicho campo como Op[5-0].
- Los dos registros de lectura están siempre especificados en los campos rs y rt en las posiciones 25-21 y 20-16. Este hecho se cumple para las instrucciones tipo R, *beq* y *almacenamiento*.
- El registro base para las instrucciones de acceso a memoria se encuentra siempre en rs (bits 25-21).
- El desplazamiento relativo de 16 bits para saltar si igual (*beq*), *cargas* y *almacenamientos* está siempre en las posiciones 15-0.
- El registro destino puede estar en dos lugares. Para instrucciones de carga, su posición es 20-16 (rt), mientras que en las instrucciones aritmético-lógicas está en los bits 15-11 (rd). Esto implica tener que añadir un multiplexor para seleccionar cuál de estos dos campos de la instrucción va a utilizarse en el momento de indicar el registro en el que se va a escribir.

Opcode: campo que denota la operación y formato de una instrucción.

El primer principio de diseño del capítulo 2, *la sencillez favorece la regularidad*, vale para la especificación del control.

Mediante esta información pueden añadirse las etiquetas de las diferentes partes de las instrucciones y el multiplexor adicional (para el registro destino) a nuestro sencillo camino de datos. La figura 4.15 muestra estos añadidos, que incluyen la unidad de control de la ALU, las señales de escritura de los elementos de estado, la señal de lectura de la memoria de datos y las señales de control de los multiplexores. Debido a que estos últimos únicamente tienen dos entradas, sólo requieren una única línea de control cada uno.

La figura 4.15 muestra las siete señales de control de 1 bit a las que hay que añadir la señal de ALUOp (de 2 bits). El funcionamiento de dicha señal ya se ha explicado y ahora sería útil definirlo informalmente para el resto de las señales antes de determinar cómo van a trabajar durante la ejecución de las instrucciones. La figura 4.16 describe la funcionalidad de dichas líneas de control.

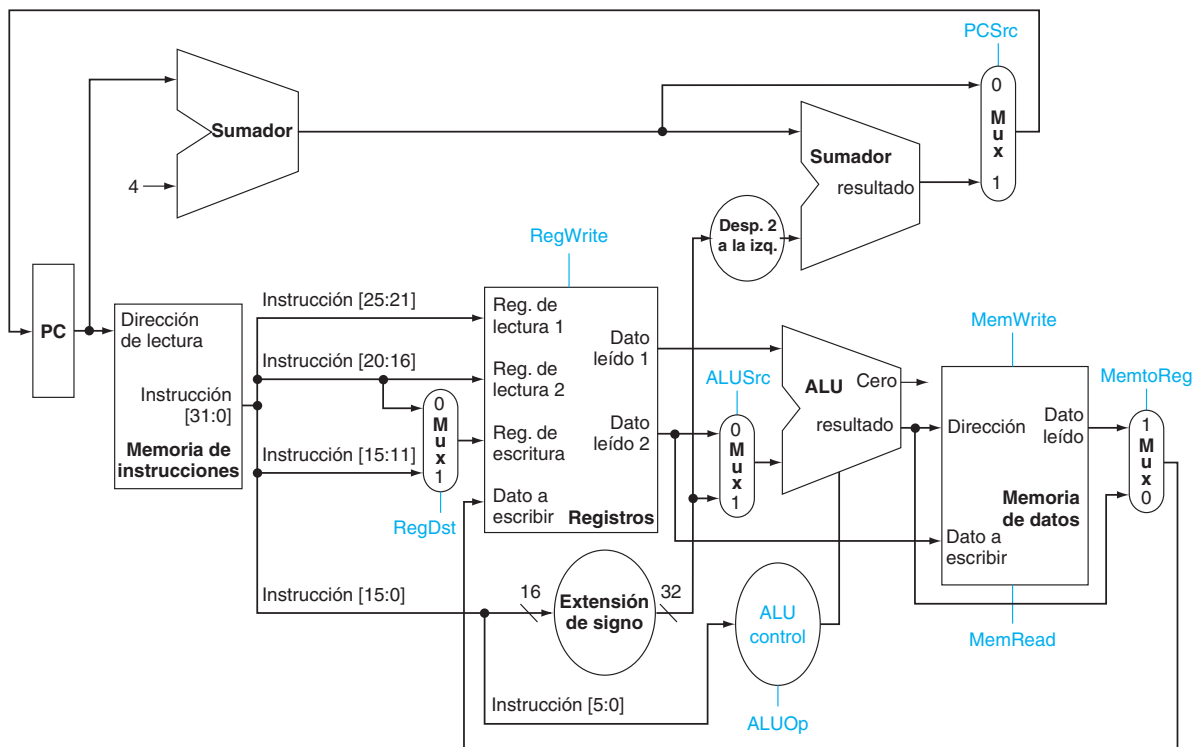


FIGURA 4.15 El camino de datos de la figura 4.12 con todos los multiplexores necesarios y todas las líneas de control identificadas. Las líneas de control se muestran en color. También se ha añadido el bloque encargado de controlar la ALU. El registro PC no necesita un control de escritura ya que sólo se escribe una vez al final de cada ciclo de reloj. La lógica de control del salto es la que determina si el nuevo valor será el valor anterior incrementado o la dirección destino del salto.

Señal de control	Efecto cuando no está activa	Efecto cuando está activa
RegDst	El identificador del registro destino viene determinado por el campo rt (bits 20-16).	El identificador del registro destino viene determinado por el campo rd (bits 15-11).
RegWrite	Ninguno.	El registro destino se actualiza con el valor a escribir.
ALUSrc	El segundo operando de la ALU proviene del segundo registro leído del banco de registros.	El segundo operando de la ALU son los 16 bits de menor peso de la instrucción con el signo extendido.
PCSrc	El PC es reemplazado por su valor anterior más 4 ($PC + 4$).	El PC es reemplazado por la salida del sumador que calcula la dirección destino del salto.
MemRead	Ninguno.	El valor de la posición de memoria designada por la dirección se coloca en la salida de lectura.
MemWrite	Ninguno.	El valor de la posición de memoria designada por la dirección se reemplaza por el valor de la entrada de datos.
MemtoReg	El valor de entrada del banco de registros proviene de la ALU.	El valor de entrada del banco de registros proviene de la memoria.

FIGURA 4.16 El efecto de cada una de las siete señales de control. Cuando el bit de control encargado de controlar un multiplexor de dos vías está a 1, dicho multiplexor seleccionará la entrada etiquetada como 1. De lo contrario, si el control está desactivado, se tomará la entrada etiquetada como 0. Recuerde que todos los elementos de estado tienen un reloj como señal de entrada implícita y cuya función es controlar las escrituras. Nunca se hace atravesar puertas al reloj, ya que puede crear problemas. (Véase en [apéndice C](#) una discusión más detallada de este problema.)

Una vez definidas las funciones de cada una de las señales de control, se puede pasar a ver cómo activarlas. La unidad de control puede activar todas las señales en función de los códigos de operación de las instrucciones exceptuando una de ellas (la señal PCSrc). Esta línea de control debe activarse si la instrucción es de tipo salto condicional (decisión que puede tomar la unidad de control) y la salida Cero de la ALU, que se utiliza en las comparaciones, está a 1. Para generar la señal de PCSrc se necesita aplicar una AND a una señal llamada *Branch*, que viene de la unidad de control, con la señal Cero procedente de la ALU.

Estas nueve señales de control (las siete de la figura 4.16, más las dos de la señal ALUOp), pueden activarse en función de las seis señales de entrada de la unidad de control, las cuales pertenecen al código de operación, bits 31 a 26. El camino de datos con la unidad y las señales de control se muestra en la figura 4.17.

Antes de intentar escribir el conjunto de ecuaciones de la tabla de verdad de la unidad de control, sería útil definirla informalmente. Debido a que la activación de las líneas de control únicamente depende del código de operación, se define si cada una de las señales de control debería valer 0, 1 o bien indeterminada (X), para cada uno de los códigos de operación. La figura 4.18 define cómo deben activarse las señales de control para cada código de operación; información que proviene directamente de las figuras 4.12, 4.16 y 4.17.

Funcionamiento del camino de datos

Con la información contenida en las figuras 4.16 y 4.18, se puede diseñar la lógica de la unidad de control, pero antes de esto, se verá cómo cada instrucción hace uso del

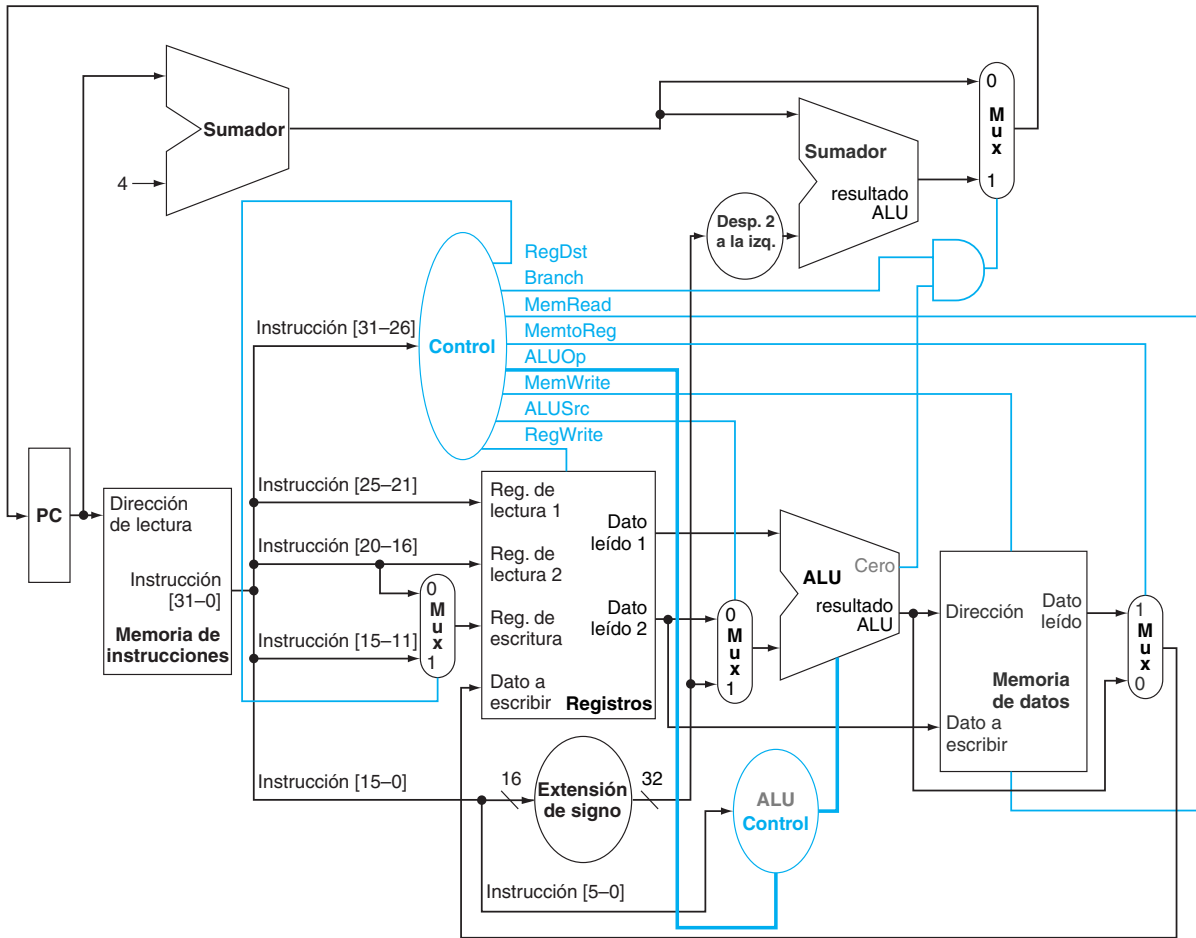


FIGURA 4.17 Un sencillo camino de datos con la unidad de control. La entrada de la unidad de control está compuesta por los seis bits pertenecientes al campo del código de operación de la instrucción. Las salidas de dicha unidad consisten en tres señales de un bit que se utilizan para controlar los multiplexores (RegDst, ALUSrc, y MemtoReg), tres señales para controlar lecturas y escrituras en el banco de registros y en la memoria (RegWrite, MemRead, y MemWrite), una señal de 1 bit para determinar si es posible saltar (*branch*), y una señal de control de 2 bits para la ALU (ALUOp). Se utiliza una puerta AND para combinar la señal de control *branch* de la unidad de control y la salida Cero de la ALU. Dicha puerta será la que controle la selección del próximo PC. Obsérvese que ahora PCSrc es una señal derivada aunque una de sus partes proviene directamente de la unidad de control. En adelante se suprimirá este nombre.

camino de datos. En las siguientes figuras se muestra el flujo a través del camino de datos de las tres clases diferentes de instrucciones. Las señales de control activadas, así como los elementos activos del camino de datos, están resaltados en cada caso. Obsérvese que un multiplexor cuyo control está a 0 es una acción definida, a pesar de que su línea de control no esté destacada. Las señales de control formadas por múltiples bits están destacadas si cualquiera de los que la constituyen está activa.

Instrucción	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
Formato R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

FIGURA 4.18 La activación de las líneas de control está completamente determinada por el código de operación de las instrucciones. La primera fila de la tabla corresponde a las instrucciones tipo R (add, sub, and, or y slt). En todas estas instrucciones, los registros fuente se encuentran en rs y rt, y el registro destino en rd, de forma que las señales ALUSrc y RegDst deben estar activas. Además, este tipo de instrucciones siempre escribe en un registro (RegWrite = 1) y en ningún caso accede a la memoria de datos. Cuando la señal Branch está a 0, el PC se reemplaza de forma incondicional por PC + 4; en otro caso, el registro de PC se reemplaza por la dirección destino del salto si la salida Cero de la ALU también está en nivel alto. El campo ALUOp para las instrucciones tipo R siempre tiene el valor de 10 para indicar que las señales de control de la ALU deben generarse con ayuda del campo de función. La segunda y tercera fila de la tabla contienen el valor de las señales de control para las instrucciones lw y sw. En ambos casos, ALUSrc y ALUOp están activas para poder llevar a cabo el cálculo de direcciones. Las señales MemRead y MemWrite estarán a 1 según el tipo de acceso. Finalmente, cabe destacar que RegDst y RegWrite deben tener los valores 0 y 1, respectivamente, en caso de una carga, para que el valor se almacene en el registro rt. El control de las instrucciones de salto es similar a las tipo R ya que también envía los registros rs y rt a la ALU. La señal ALUOp, en caso de instrucción beq, toma el valor 01, para que la ALU realice una operación de resta y compruebe así la igualdad. Observe que el valor de la señal MemtoReg es irrelevante cuando RegWrite vale 0 (como el registro no va a ser escrito, el valor del dato existente en el puerto de escritura del banco de registros no va a ser utilizado). De esta manera, en las dos últimas filas de la tabla, los valores de la señal MemtoReg se han reemplazado por X, que indica términos indeterminados. De igual forma, este tipo de términos pueden añadirse a RegDst cuando RegWrite vale 0. Este tipo de términos los debe añadir el diseñador, pues dependen del conocimiento que se tenga sobre el funcionamiento del camino de datos.

La figura 4.19 muestra el funcionamiento de una instrucción tipo R, tal como add \$t1, \$t2, \$t3. Aunque todo sucede en un único ciclo de reloj, la ejecución de una instrucción se puede dividir en cuatro pasos; estos pasos se pueden ordenar según el flujo de información:

1. Se carga una instrucción de la memoria de instrucciones y se incrementa el PC.
2. Se leen los registros \$t2 y \$t3 del banco de registros. Adicionalmente, la unidad de control principal se encarga de calcular la activación de las señales de control durante este paso.
3. La ALU se encarga de realizar la operación adecuada con los datos procedentes del banco de registros, utilizando para ello el campo de función (bits 5-0) para obtener la función de la ALU.
4. El resultado calculado en la ALU se escribe en el banco de registros utilizando los bits 15-11 de la instrucción para seleccionar el registro destino (\$t1).

De forma similar, se puede ilustrar la ejecución de una instrucción de carga, tal como

```
lw $t1, offset($t2)
```

de un modo parecido a la figura 4.19. La figura 4.20 muestra las unidades funcionales y las líneas de control activadas para esta instrucción. Puede verse una carga como una instrucción que se ejecuta en 5 pasos (similares a los de las instrucciones tipo R, que se ejecutaban en 4).

1. Se carga una instrucción de la memoria de instrucciones y se incrementa el PC.
2. Se lee el valor del registro \$t2 del banco de registros.

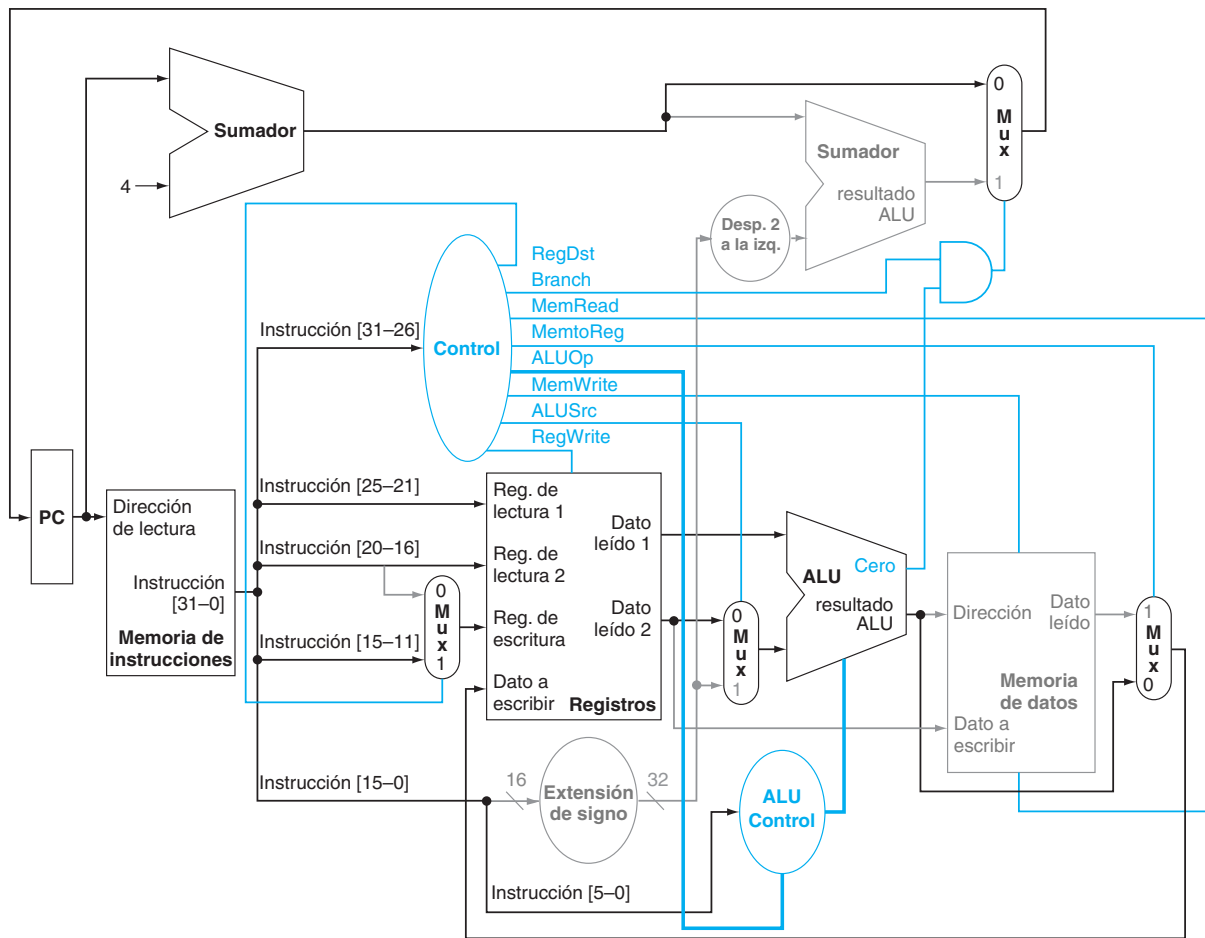


FIGURA 4.19 Funcionamiento del camino de datos para una instrucción R-type tal como `add $t1, $t2, $t3`. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas.

3. La ALU calcula la suma del valor leído del banco de registros y los 16 bits de menor peso de la instrucción, con el signo extendido (*offset*).
4. Dicha suma se utiliza como dirección para acceder a la memoria de datos.
5. El dato procedente de la memoria se escribe en el banco de registros. El registro destino viene determinado por los bits 20-16 de la instrucción (`$t1`).

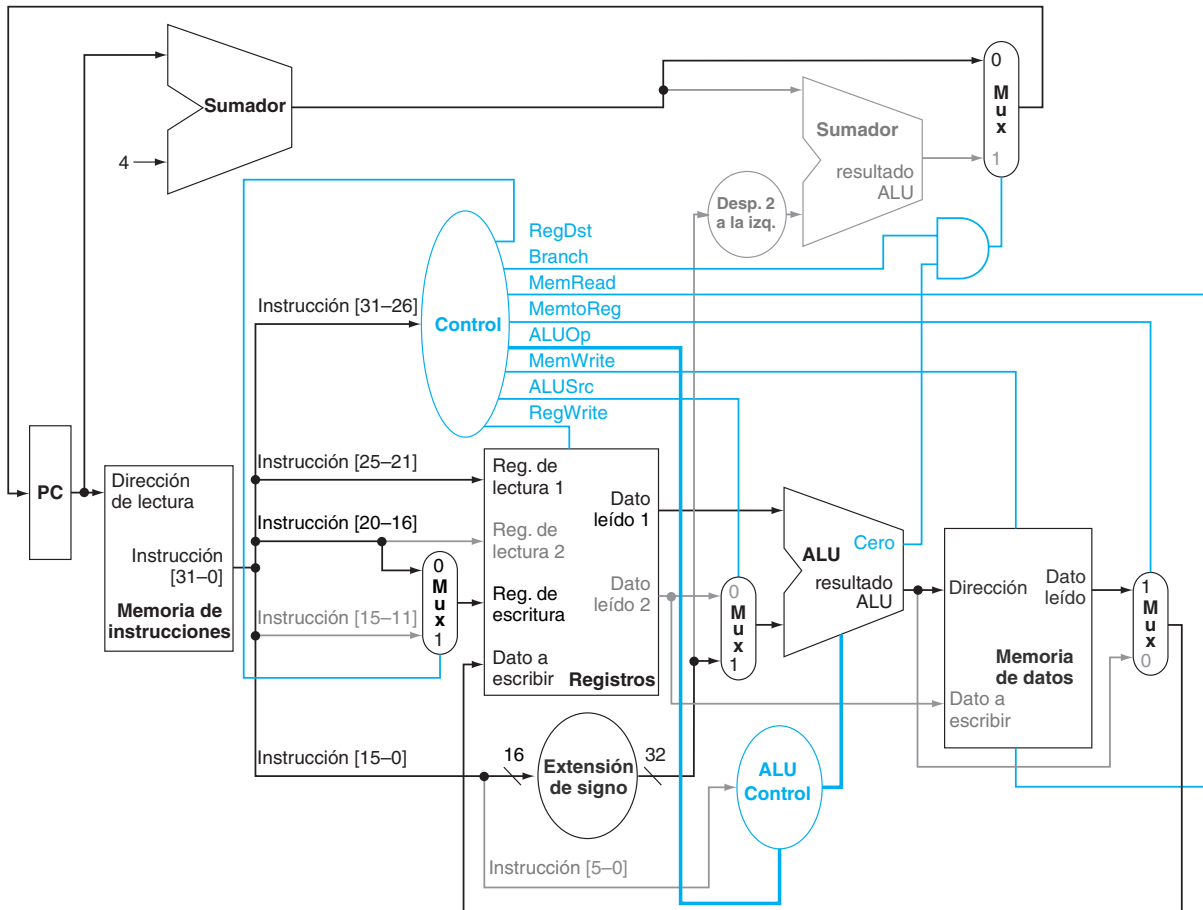


FIGURA 4.20 Funcionamiento del camino de datos para una instrucción de carga. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas. Una instrucción de almacenamiento opera de forma similar. La diferencia principal es que el control de la memoria indicará una escritura en lugar de una lectura, el valor del segundo registro contendría el dato a almacenar, y la operación final de escribir el valor de memoria en el banco de registros no se realizaría.

Finalmente, se puede ver de la misma manera la operación de saltar si igual, por ejemplo, `beq $t1, $t2, offset`. Su ejecución es similar a las instrucciones tipo R, pero la salida de la ALU se utiliza para determinar si el PC se actualizará con $PC + 4$ o con la dirección destino del salto. La figura 4.21 muestra los cuatro pasos de la ejecución.

1. Se carga una instrucción de la memoria de instrucciones y se incrementa el PC.
2. Se leen los registros `$t1` y `$t2` del banco de registros.

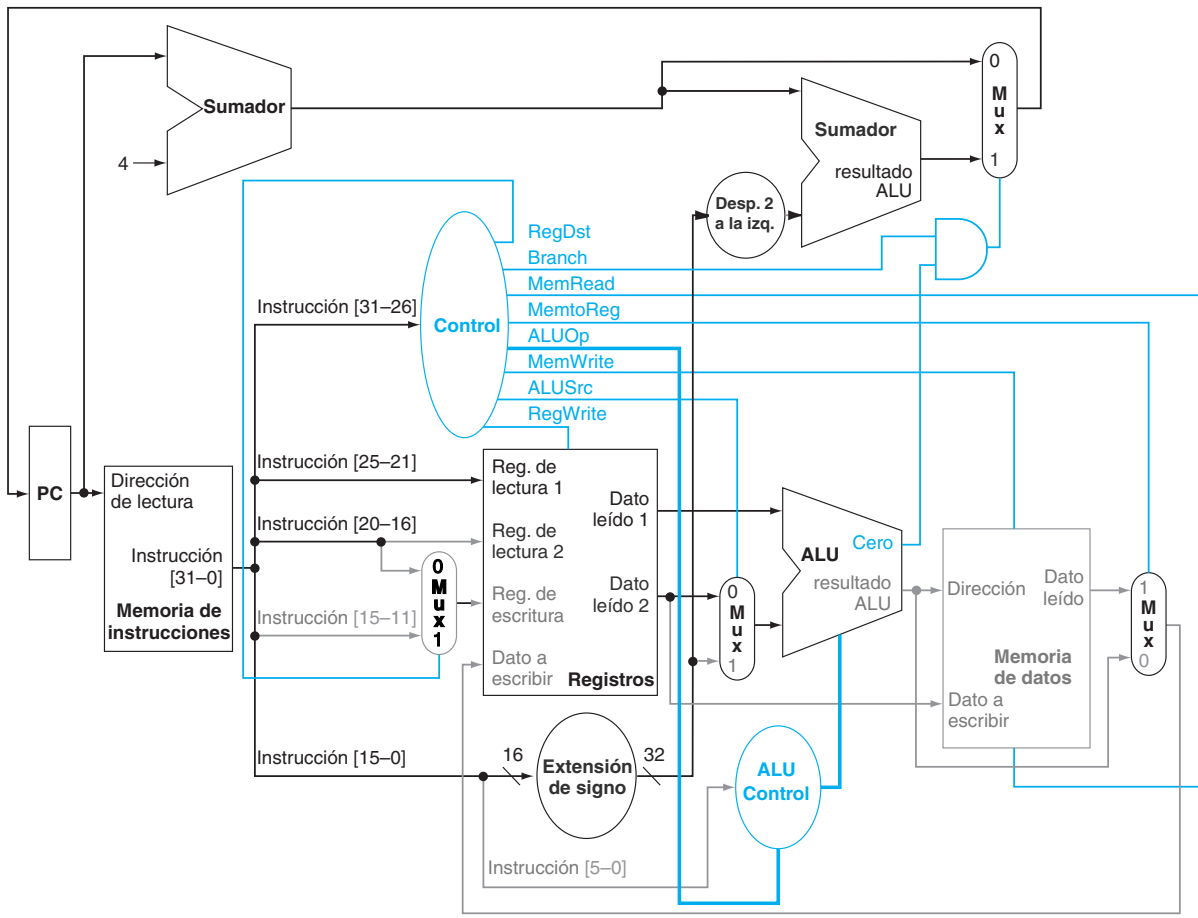


FIGURA 4.21 Funcionamiento del camino de datos para una instrucción de salto si igual. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas. Después de usar el banco de registros y la ALU para realizar la comparación, la salida Cero se utiliza para seleccionar el siguiente valor del contador de programas entre los dos valores candidatos.

3. La ALU realiza una resta de los operandos leídos del banco de registros. Se suma el valor de $PC + 4$ a los 16 bits de menor peso (con el signo extendido) de la instrucción (offset) desplazados 2 bits. El resultado es la dirección destino del salto.
4. Se utiliza la señal Cero de la ALU para decidir qué valor se almacena en el PC.

Finalización del control

Una vez que se ha visto los pasos en la ejecución de las instrucciones continuaremos con la implementación del control. La función de control puede definirse de forma precisa utilizando los contenidos de la figura 4.18. Las salidas son las líneas de control y las entradas los 6 bits que conforman el campo del código de operación (Op[5-0]). De esta manera se puede crear una tabla de verdad para cada una de las salidas basada en la codificación binaria de los códigos de operación.

La figura 4.22 muestra la lógica de la unidad de control como una gran tabla de verdad que combina todas las salidas y utiliza los bits del código de operación como entradas. Ésta especifica de forma completa la función de control y puede implementarse mediante puertas lógicas de forma automática. Este paso final se muestra en la sección D.2 en el [apéndice D](#).

Ahora que ya tenemos una [implementación de ciclo de reloj único](#) para la mayor parte del núcleo del repertorio de instrucciones MIPS, se va a añadir la instrucción de salto incondicional (*jump*) y se verá cómo puede extenderse el camino de datos y su control para poder ejecutar otro tipo de instrucciones del repertorio.

Implementación de ciclo único (implementación de ciclo de reloj único): implementación en la que una instrucción se ejecuta en un único ciclo de reloj.

Entrada o salida	Nombre de la señal	Formato R	lw	sw	beq
Entradas	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Salidas	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURA 4.22 La función de control para una implementación de ciclo único está completamente especificada en esta tabla de verdad. La mitad superior de la tabla muestra las posibles combinaciones de las señales a la entrada, que corresponden a los cuatro posibles códigos de operación y que determinan la activación de las señales de control. Recuerde que Op[5-0] se corresponde con los bits 31-26 de la instrucción (el código de operación). La parte inferior muestra las salidas. Así, la señal de salida RegWrite está activada para dos combinaciones diferentes de entradas. Si únicamente se consideran estos cuatro posibles códigos de operación, esta tabla puede simplificarse utilizando términos indeterminados en las señales de entrada. Por ejemplo, puede detectarse una instrucción de tipo aritmético-lógico con la expresión $Op5 \cdot Op2$, ya que es suficiente para distinguirlas del resto. De todas formas, no se utilizará esta simplificación, pues el resto de los códigos de operación del MIPS se usarán en la implementación completa.

EJEMPLO

RESPUESTA

Implementación de saltos incondicionales (*jump*)

La figura 4.17 muestra la implementación de muchas de las instrucciones vistas en el capítulo 2. Un tipo de instrucción ausente es el salto incondicional. Extienda el camino de datos de la figura 4.17, así como su control, para incluir dicho tipo de instrucciones. Describa cómo se ha de activar cualquier línea de control nueva.

La instrucción *jump*, mostrada en la figura 4.23, tiene un cierto parecido a la instrucción de salto condicional, pero calcula la dirección de destino de forma diferente y, además, es incondicional. Como en los saltos condicionales, los 2 bits de menor peso de la dirección de salto son siempre 00_{dos}. Los siguientes 26 bits de menor peso de la dirección están en el campo inmediato de la instrucción. Los 4 bits de mayor peso de la dirección que debería reemplazar al PC vienen del PC de la instrucción al cual se le ha sumado 4. Así, podría realizarse un salto incondicional almacenando en el registro de PC la concatenación de:

- Los 4 bits de mayor peso del actual PC + 4 (son los bits 31-28 de la dirección de la siguiente instrucción en orden secuencial).
- Los 26 bits correspondientes al campo inmediato de la instrucción *jump*.
- Los bits 00_{dos}.

La figura 4.24 muestra las partes añadidas al control para este tipo de instrucciones respecto a la figura 4.17. Se necesita un nuevo multiplexor para seleccionar el origen del nuevo PC, la dirección de la siguiente instrucción en orden secuencial (PC + 4), la dirección de salto condicional o la de una instrucción de salto incondicional. También se necesita una nueva señal de control para este multiplexor. Esta señal, llamada *Jump*, únicamente se activa cuando la instrucción es un salto incondicional, es decir, cuando el código de operación es 2.

Campo	000010	dirección
Posición de los bits	31-26	25-0

FIGURA 4.23 Formato de la instrucción *jump* (código de operación = 2). La dirección destino de este tipo de instrucciones se forma mediante la concatenación de los 4 bits de mayor peso de PC + 4 y los 26 bits de campo de dirección de la instrucción añadiendo 00 como los 2 bits de menor peso.

Por qué no se utiliza una implementación de ciclo único hoy en día

Aunque este tipo de implementaciones funciona correctamente, no se utiliza en los diseños actuales porque es ineficiente. Para ver por qué ocurre esto, debe saberse que el ciclo de reloj debe tener la misma longitud para todos los tipos de instrucciones. Por supuesto, el ciclo de reloj viene determinado por el mayor

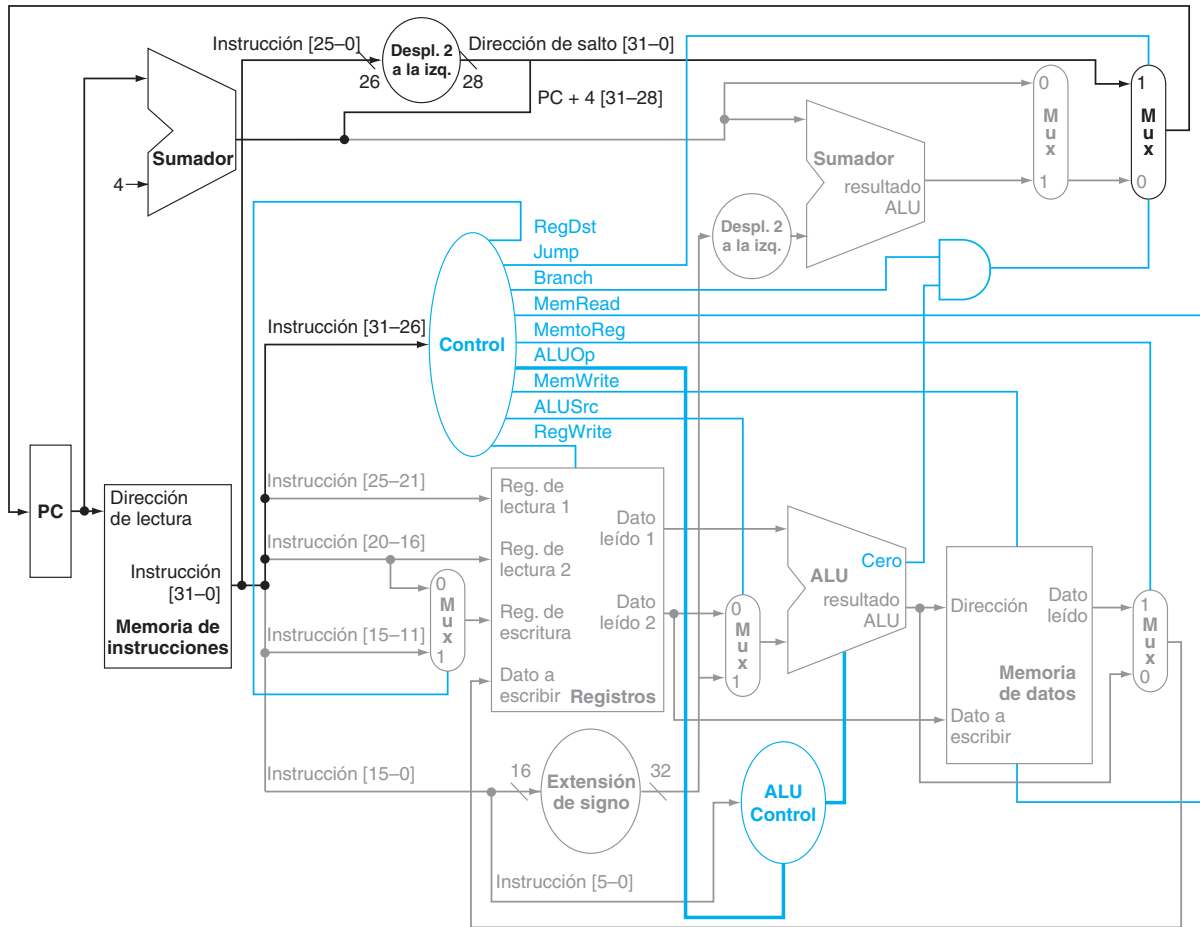


FIGURA 4.24 Un camino de datos sencillo y su control extendido para ejecutar instrucciones de salto incondicional. Se utiliza un multiplexor adicional (en la esquina superior derecha) para elegir entre el destino de la instrucción de salto incondicional y la dirección de salto secuencial. Este multiplexor se controla mediante la señal de salto incondicional. La dirección de destino de la instrucción de salto incondicional se obtiene desplazando los 26 bits de menor peso 2 bits a la izquierda (se añaden 00 como bits de menor peso), concatenando los 4 bits de mayor peso de PC + 4 como bits de mayor peso, y obteniendo de esta manera una dirección de 32 bits.

tiempo de ejecución posible en el procesador. Esta instrucción en la mayoría de los casos es la instrucción *load*, que usa 5 unidades funcionales en serie: la memoria de instrucciones, el banco de registros, la ALU, la memoria de datos y el banco de registros nuevamente. Aunque el CPI es 1 (véase el capítulo 1), las prestaciones generales de la implementación de ciclo único probablemente no es muy bueno, porque el ciclo de reloj es demasiado largo.

La penalización por utilizar un diseño de ciclo único con un ciclo de reloj fijo es importante, pero podría considerarse aceptable para un repertorio con pocas instrucciones. Históricamente, los primeros computadores con un repertorio de instrucciones muy simple usaban esta técnica de implementación. Sin embargo, si

se intenta implementar una unidad de punto flotante o un repertorio de instrucciones con instrucciones más complejas, el diseño de ciclo único podría no funcionar correctamente.

Es inútil intentar implementar técnicas que reduzcan el retraso del caso más habitual pero que no mejoren el tiempo de ciclo del peor caso, porque el ciclo de la señal de reloj debe ser igual al retraso del peor caso de todas las instrucciones. Así, la implementación de ciclo único viola el principio de diseño clave del capítulo 2, de hacer rápido el caso más habitual.

En la siguiente sección analizaremos otra técnica de implementación, llamada segmentación, que utiliza un camino de datos muy similar a la de ciclo único pero mucho más eficiente, con una productividad más elevada. La segmentación mejora la eficiencia mediante la ejecución de varias instrucciones simultáneamente.

Autoevaluación

Fíjese en el control de la figura 4.22. ¿Se pueden combinar algunas señales de control? ¿Se puede reemplazar alguna señal de control de salida por la inversa de otra? (ténganse en cuenta las indeterminaciones). Si es así, ¿se puede reemplazar una señal por otra sin añadir inversores?

No malgastes el tiempo.
Proverbio Americano

Segmentación (*pipelining*): técnica de implementación que solapa la ejecución de múltiples instrucciones, de forma muy similar a una línea de ensamblaje.

4.5

Descripción general de la segmentación

Segmentación (*pipelining*) es una técnica de implementación que consiste en solapar la ejecución de múltiples instrucciones. Hoy en día, la segmentación es universal.

En esta sección utilizaremos una analogía para describir los términos básicos y las características principales de la segmentación. Si el lector sólo está interesado en la idea general, debe centrarse en esta sección y después saltar a las secciones 4.10 y 4.11 para leer una introducción a las técnicas avanzadas de segmentación que usan procesadores recientes como el AMD Opteron X4 (Barcelona) o Intel Core. Pero si está interesado en explorar la anatomía de un computador segmentado, entonces esta sección es una buena introducción a las secciones de la 4.6 a la 4.9.

Cualquiera que haya tenido que lavar grandes cantidades de ropa ha usado de forma intuitiva la estrategia de la segmentación. El enfoque no segmentado de hacer la colada sería

1. Introducir ropa sucia en la lavadora.
2. Cuando finaliza el lavado, introducir la ropa mojada en la secadora.
3. Cuando finaliza el secado, poner la ropa seca en una mesa para ordenarla y doblarla.
4. Una vez que toda la ropa está doblada, pedir al compañero de piso que guarde la ropa.

Cuando el compañero ha finalizado, entonces se vuelve a comenzar con la siguiente colada.

El enfoque segmentado de lavado requiere mucho menos tiempo, tal y como muestra la figura 4.25. Tan pronto como la lavadora termina con la primera colada y ésta es colocada en la secadora, se vuelve a cargar la lavadora con una

Comprender las prestaciones de los programas

Aparte del sistema de memoria, el funcionamiento efectivo del pipeline es generalmente el factor más importante para determinar el CPI del procesador, y por tanto sus prestaciones. Tal como veremos en la sección 4.10, comprender las prestaciones de un procesador segmentado actual con ejecución múltiple de instrucciones es complejo y requiere comprender muchas más cosas de las que surgen del análisis de un procesador segmentado sencillo. De todos modos, los riesgos estructurales, de datos y de control mantienen su importancia tanto en los pipelines sencillos como en los más sofisticados.

En los pipelines actuales, los riesgos estructurales generalmente involucran a la unidad de punto flotante, que no puede ser completamente segmentada, mientras que los riesgos de control son generalmente un problema grande en los programas enteros, que tienden a tener una alta frecuencia de instrucciones de salto además de saltos menos predecibles. Los riesgos de datos pueden llegar a ser cuellos de botella para las prestaciones tanto en programas enteros como de punto flotante. Con frecuencia es más fácil tratar con los riesgos de datos en los programas de punto flotante, porque la menor frecuencia de saltos y un patrón de accesos a memoria más regular facilitan que el compilador pueda planificar la ejecución de las instrucciones para evitar los riesgos. Es más difícil realizar esas optimizaciones con programas enteros que tienen patrones de accesos a memoria menos regulares y que involucran el uso de apuntadores en más ocasiones. Tal como se verá en la sección 4.10, existen técnicas más ambiciosas, tanto por parte del compilador como del hardware, para reducir las dependencias de datos mediante la planificación de la ejecución de las instrucciones.

IDEA clave

Latencia (del pipeline): número de etapas en un pipeline, o el número de etapas entre dos instrucciones durante la ejecución.

La segmentación incrementa el número de instrucciones que se están ejecutando a la vez y la rapidez con que las instrucciones empiezan y acaban. La segmentación no reduce el tiempo que se tarda en completar una instrucción individual, también denominada la **latencia (latency)**. Por ejemplo, el pipeline de cinco etapas todavía necesita que las instrucciones tarden 5 ciclos para ser completadas. Según los términos usados en el capítulo 4, la segmentación mejora la productividad (*throughput*) de instrucciones en vez del *tiempo de ejecución* o *latencia* de cada instrucción.

Los repertorios de instrucciones pueden tanto simplificar como dificultar la tarea de los diseñadores de procesadores segmentados, los cuales tienen ya que hacer frente a los riesgos estructurales, de control y de datos. La predicción de saltos, la anticipación de resultados y los bloqueos ayudan a hacer un computador más rápido y que aún siga produciendo las respuestas correctas.

Aquí hay menos de lo que el ojo puede ver.

Tallulah Bankhead, observación a Alexander Wollcott, 1922

4.6

Camino de datos segmentados y control de la segmentación

La figura 4.33 muestra el camino de datos monociclo de la sección 4.4 con las etapas de segmentación identificadas. La división de una instrucción en cinco pasos implica

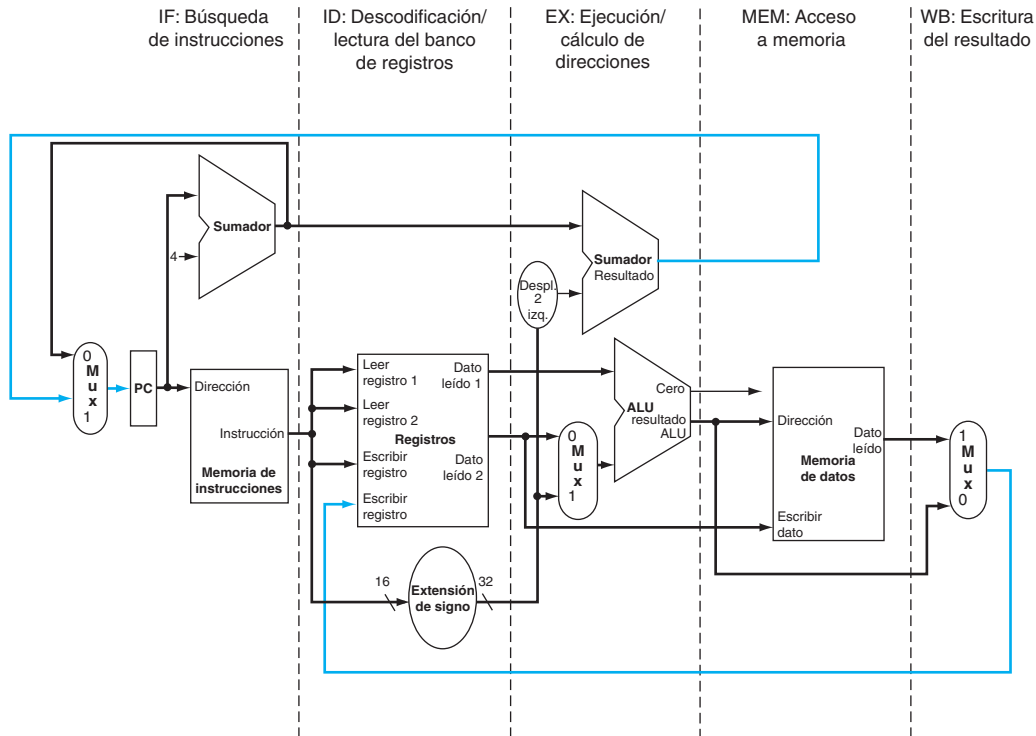


FIGURA 4.33 Camino de datos monociclo extraído de la sección 4.4 (similar al de la figura 4.17). Cada paso de la instrucción se puede situar en el camino de datos de izquierda a derecha. Las únicas excepciones a esta regla son la actualización del registro PC y el paso final de escritura de resultados, mostrados en color. En este último caso, el resultado de la ALU o de la memoria de datos se envía hacia la izquierda para ser escritos en el banco de registros. (Aunque normalmente se utilizan las líneas de color para representar líneas de control, en este caso representan líneas de datos.)

una segmentación de cinco etapas, lo que a su vez significa que durante un ciclo de reloj se estarán ejecutando hasta cinco instrucciones. Por tanto el camino de datos se debe dividir en cinco partes, y cada una de ellas se nombrará haciéndola corresponder con un paso de la ejecución de la instrucción:

1. IF: Búsqueda de instrucciones
2. ID: Descodificación de instrucciones y lectura del banco de registros
3. EX: Ejecución de la instrucción o cálculo de dirección
4. MEM: Acceso a la memoria de datos
5. WB: Escritura del resultado (*write back*)

En la figura 4.33 estas cinco etapas quedan identificadas aproximadamente por la forma como se dibuja el camino de datos. En general, a medida que se va completando la ejecución, las instrucciones y los datos se mueven de izquierda a derecha a través de las cinco etapas. Volviendo a la analogía de la lavandería, la ropa se lava, se seca y se dobla mientras se mueve a través de la línea de segmentación, y nunca vuelve hacia atrás.

Sin embargo, hay dos excepciones a este movimiento de izquierda a derecha de las instrucciones:

- La etapa de escritura de resultados, que pone el resultado en el banco de registros que está situado más atrás, a mitad del camino de datos.
- La selección del siguiente valor del PC, que se elige entre el PC incrementado y la dirección de salto obtenida al final de la etapa MEM.

El flujo de datos de derecha a izquierda no afecta a la instrucción actual. Estos movimientos de datos hacia atrás sólo influyen a las instrucciones que entran al pipeline después de la instrucción en curso. Obsérvese que la primera flecha de derecha a izquierda puede dar lugar a riesgos de datos y la segunda flecha puede dar lugar a riesgos de control.

Una manera de mostrar lo que ocurre en la ejecución segmentada es suponer que cada instrucción tiene su propio camino de datos, y entonces colocar estas rutas de datos en una misma línea de tiempo para mostrar su relación. La figura 4.34 muestra la ejecución de las instrucciones de la figura 4.27 representando en una línea de tiempo común sus rutas de datos particulares. Para mostrar estas relaciones, la figura 4.33 usa una versión estilizada del camino de datos que se había mostrado en la figura 4.34.

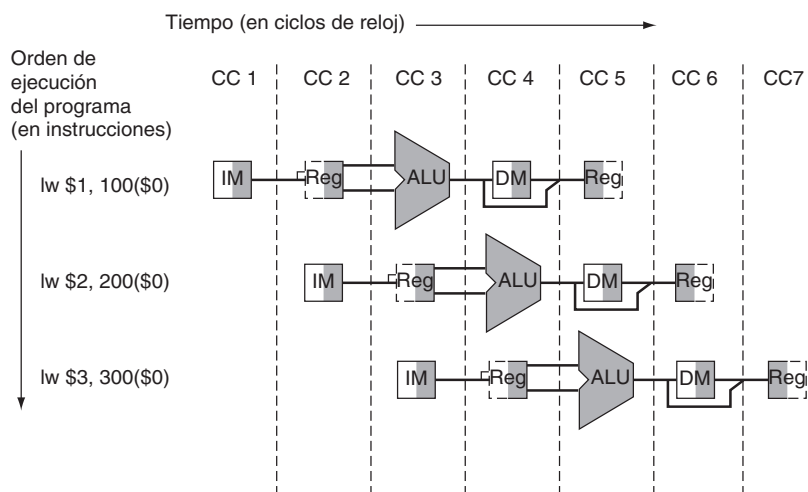


FIGURA 4.34 Ejecución de instrucciones usando el camino de datos monociclo de la figura 4.33, suponiendo que la ejecución está segmentada. De un modo similar a como se hace en las figuras 4.28 a 4.30, esta figura supone que cada instrucción tiene su propio camino de datos, y cada parte del camino de datos está sombreado según su uso. A diferencia de las otras figuras, ahora cada etapa está etiquetada con el recurso físico usado en esa etapa, y que corresponde con la parte del camino de datos mostrado en la figura 4.33. *IM* representa tanto a la memoria de instrucciones como al registro PC de la etapa de búsqueda de instrucciones, *Reg* representa el banco de registros y la unidad de extensión de signo en la etapa de decodificación de instrucciones/lectura del banco de registros (ID), y así todas las demás etapas. Para mantener correctamente el orden temporal, este camino de datos estilizado divide el banco de registros en dos partes lógicas: la lectura de registros durante la etapa ID y la escritura en registro durante la etapa WB. Para representar este doble uso, en la etapa ID se dibuja la mitad izquierda del banco de registros sin sombreado y con líneas discontinuas, ya que no está siendo escrito, mientras que en la etapa WB es la mitad derecha del banco de registros la que se dibuja sin sombreado y con líneas discontinuas, ya que no está siendo leído. Igual que se ha hecho con anterioridad, se supone que la escritura en el banco de registros se hace en la primera mitad del ciclo y la lectura durante la segunda mitad.

La figura 4.34 puede parecer sugerir que tres instrucciones necesitan tres rutas de datos. Así, se añadieron registros para almacenar datos intermedios y permitir así compartir partes del camino de datos durante la ejecución de las instrucciones.

Por ejemplo, tal como muestra la figura 4.34, la memoria de instrucciones sólo se usa durante una de las cinco etapas de una instrucción, permitiendo que sea compartida con otras instrucciones durante las cuatro etapas restantes. Para poder conservar el valor de cada instrucción individual durante las cuatro últimas etapas del pipeline, el valor leído de la memoria de instrucciones se debe guardar en un registro. Aplicando argumentos similares a cada una de las etapas del pipeline, se puede razonar la necesidad de colocar registros en todas las líneas entre etapas de segmentación que se muestran en la figura 4.33. Volviendo a la analogía de la lavandería, se debería tener una cesta entre cada una de las etapas para mantener la ropa producida por una etapa y que debe ser usada por la siguiente etapa.

La figura 4.35 muestra el camino de datos segmentado en la que se han resaltado los registros de segmentación. Durante cada ciclo de reloj todas las instrucciones avanzan de un registro de segmentación al siguiente. Los registros tienen el nombre de las dos etapas que separan. Por ejemplo, el registro de segmentación entre las etapas IF e ID se llama IF/ID.

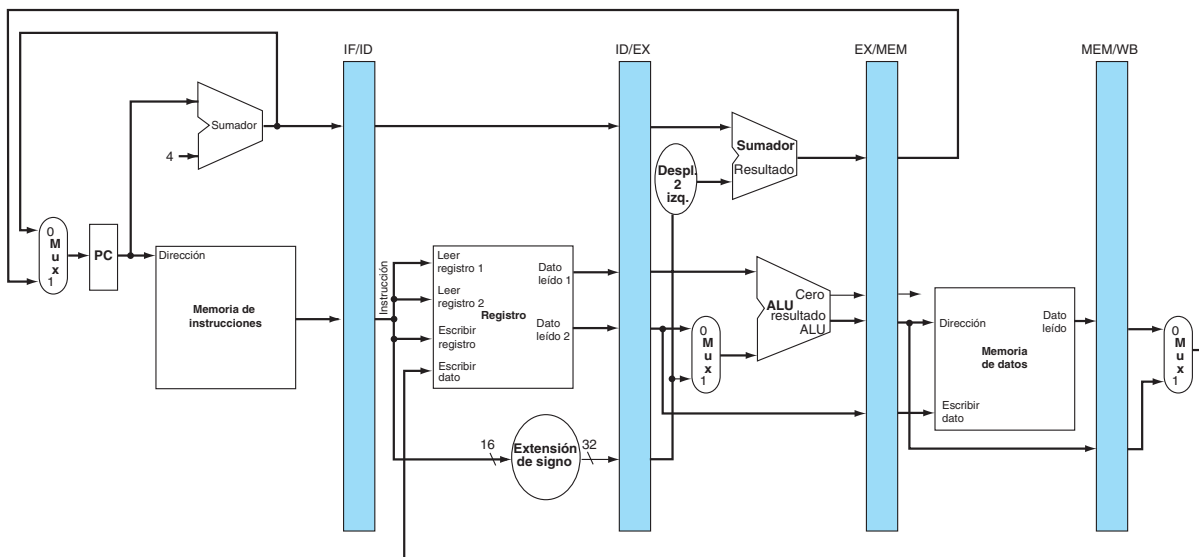


FIGURA 4.35 Versión segmentada del camino de datos de la figura 4.33. Los registros de segmentación, en color, separan cada una de las etapas. Están etiquetados con las etapas que separan; por ejemplo, el primero está etiquetado como IF/ID ya que separa las etapas de búsqueda de instrucciones y de decodificación. Los registros han de ser suficientemente anchos para almacenar todos los datos que corresponden con las líneas que pasan a través de ellos. Por ejemplo, el registro IF/ID debe ser de 64 bits de ancho, ya que debe guardar tanto los 32 bits de la instrucción leída de memoria como los 32 de la dirección obtenida del PC e incrementada. Aunque a lo largo de este capítulo estos registros se ampliarán, de momento supondremos que los otros tres registros de segmentación contienen 128, 97 y 64 bits, respectivamente.

Obsérvese que no hay registro de segmentación al final de la etapa de escritura de resultado. Todas las instrucciones deben actualizar algún estado de la máquina —el banco de registros, la memoria o el registro PC— por lo que sería redundante usar un

registro de segmentación específico para un estado que ya se está actualizando. Por ejemplo, una instrucción de carga guarda su resultado en uno de los 32 registros generales, y cualquier instrucción posterior que necesite el dato puede leerlo directamente de ese registro concreto.

Todas las instrucciones actualizan el registro PC, bien sea incrementando su valor o bien modificando su valor con la dirección destino de un salto. El registro PC se puede considerar un registro de segmentación: el que se utiliza para alimentar la etapa IF del pipeline. Sin embargo, a diferencia de los registros de segmentación mostrados de forma sombreada en la figura 4.35, el registro PC forma parte del estado visible de la arquitectura; es decir, su contenido debe ser guardado cuando ocurre una excepción, mientras que el contenido del resto de registros de segmentación puede ser descartado. En la analogía de la lavandería, se puede considerar que el PC corresponde con la cesta que contiene la carga inicial de ropa sucia antes de la etapa de lavado.

Para mostrar el funcionamiento de la segmentación, durante este capítulo se usarán secuencias de figuras que ilustran la operación sobre el pipeline a lo largo del tiempo. Quizás parezca que se requiere mucho tiempo para comprender estas páginas adicionales. No se debe tener ningún temor: sólo se trata de comparar las secuencias entre sí para identificar los cambios que ocurren en cada ciclo de reloj y ello requiere menos esfuerzo de comprensión del que podría parecer. La sección 4.7 describe lo que ocurre cuando se producen riesgos de datos entre las instrucciones segmentadas, así que de momento pueden ser ignorados.

Las figuras 4.36 a 4.38, que suponen la primera secuencia, muestran resaltadas las partes activas del pipeline a medida que una instrucción de carga avanza a través de las cinco etapas de la ejecución segmentada. Se muestra en primer lugar una instrucción de carga porque está activa en cada una de las cinco etapas. Igual que en las figuras 4.28 a 4.30, se resalta la *mitad derecha* del banco de registros o de la memoria cuando están siendo *leídos* y se resalta la *mitad izquierda* cuando están siendo *escritos*.

En cada figura se muestra la abreviación de la instrucción, $\downarrow w$, junto con el nombre de la etapa que está activa. Las cinco etapas son las siguientes:

1. *Búsqueda de instrucción*: La parte superior de la figura 4.36 muestra cómo se lee la instrucción de memoria usando la dirección del PC y después se coloca en el registro de segmentación IF/ID. La dirección del PC se incrementa en 4 y se escribe de nuevo en el PC para prepararse para el siguiente ciclo de reloj. Esta dirección incrementada se guarda también en el registro IF/ID por si alguna instrucción, como por ejemplo *beq*, la necesita con posterioridad. El computador no puede conocer el tipo de instrucción que se está buscando hasta que ésta es descodificada, por lo que debe estar preparado ante cualquier posible instrucción, pasando la información que sea potencialmente necesaria a lo largo del pipeline.
2. *Descodificación de instrucción y lectura del banco de registros*: La parte inferior de la figura 4.36 muestra la parte del registro de segmentación IF/ID donde está guardada la instrucción. Este registro proporciona el campo inmediato de 16 bits, que es extendido a 32 bits con signo, y los dos identificadores de los registros que se deben leer. Los tres valores se guardan, junto con la dirección del PC incrementada, en el registro ID/EX. Una vez más se transfiere todo lo que pueda necesitar cualquier instrucción durante los ciclos posteriores.

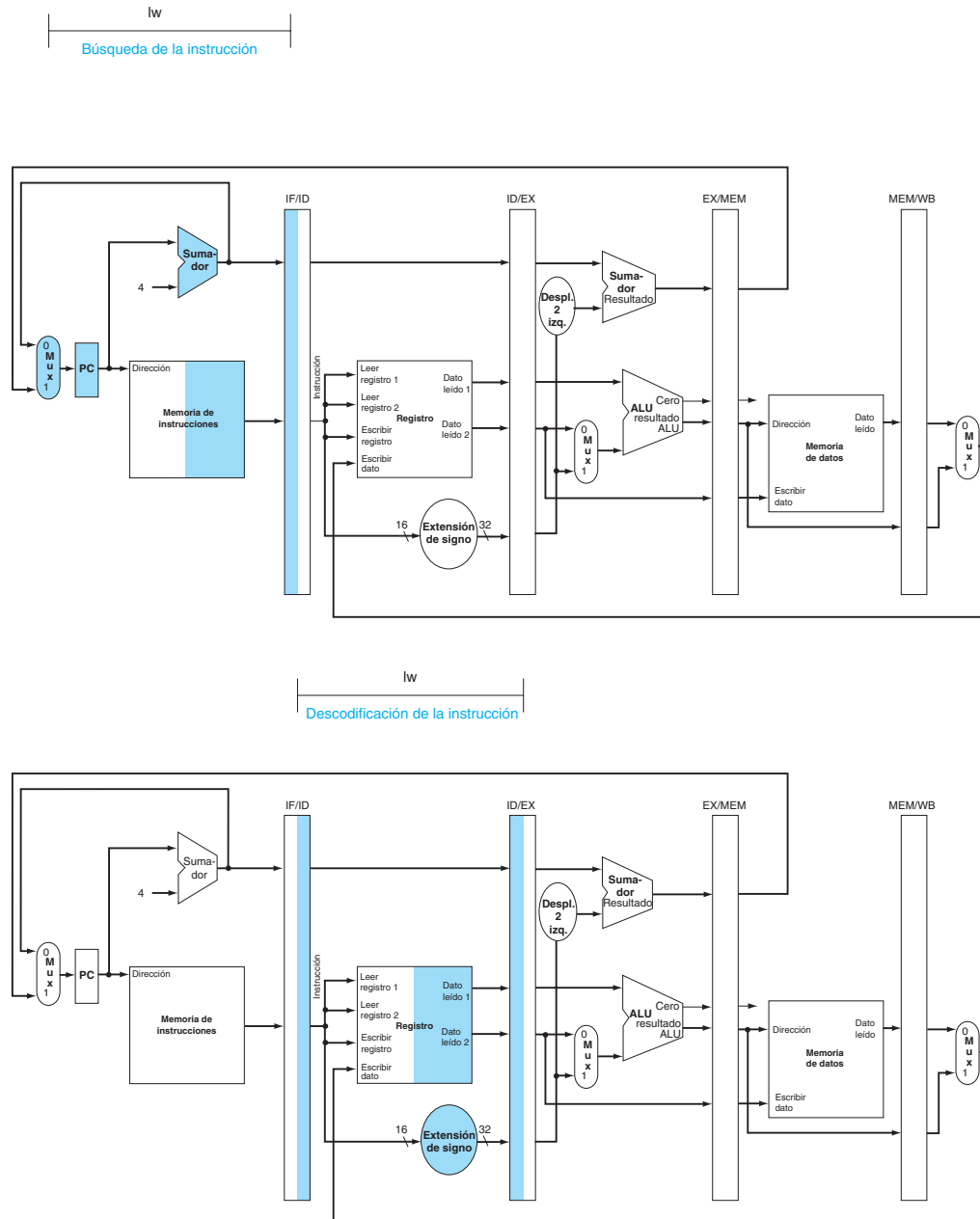


FIGURA 4.36 IF e ID: primera y segunda etapa de segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en estas dos etapas. La convención para resaltar los elementos del camino de datos es la misma que se usó en la figura 4.28. Igual que en la sección 4.2, no hay confusión al leer y escribir registros porque el contenido de éstos cambia sólo con la transición de la señal de reloj. Aunque la instrucción de carga en la etapa 2 sólo necesita el registro de arriba, el procesador no sabe qué instrucción se está descodificando, así que extiende el signo de la constante de 16 bits obtenida de la instrucción y lee ambos registros de entrada y los tres valores se almacenan sobre el registro de segmentación ID/EX. Seguro que no se necesitan los tres operandos, pero disponer de los tres simplifica el control.

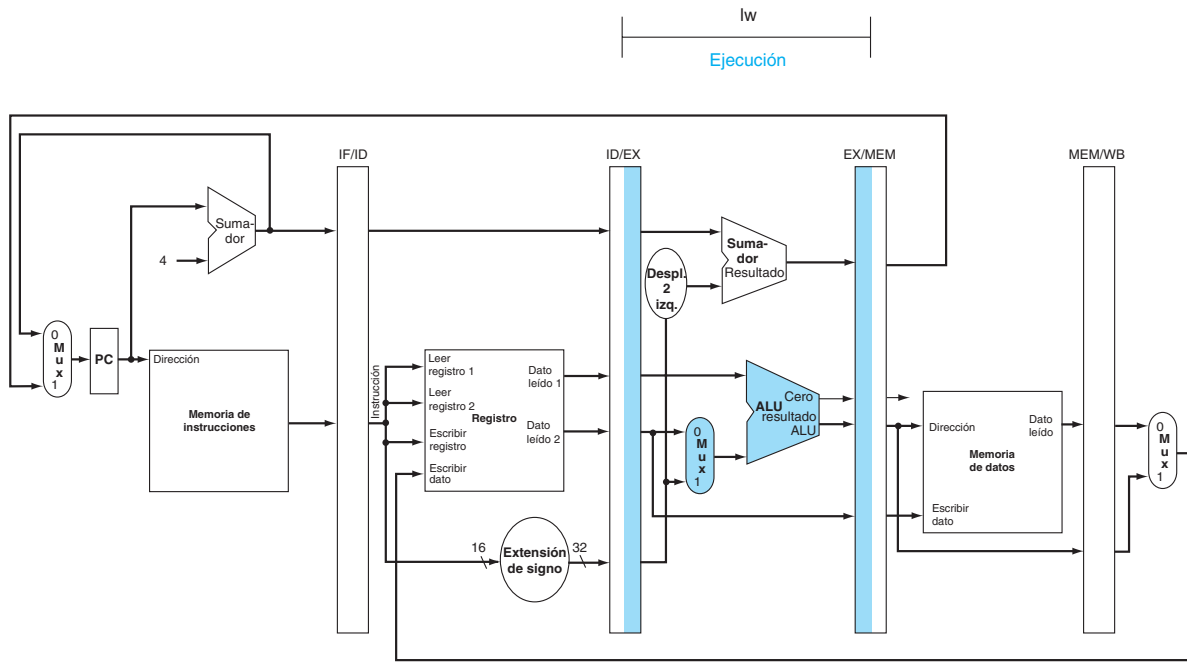


FIGURA 4.37 EX: tercera etapa de la segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en esta etapa. El registro se suma al valor inmediato con el signo extendido, y el resultado de la suma se coloca en el registro de segmentación EX/MEM.

3. *Ejecución o cálculo de dirección:* La figura 4.38 muestra que la instrucción de carga lee del registro IF/ID el contenido del registro 1 y el valor inmediato con el signo extendido, y los suma usando la ALU. El resultado de esta suma se coloca en el registro de segmentación EX/MEM.
4. *Acceso a memoria:* La parte superior de la figura 4.38 muestra la instrucción de carga cuando usa la dirección obtenida del registro EX/MEM para leer un dato de memoria y después guardar el dato leído en el registro de segmentación MEM/WB.
5. *Escritura de resultado:* La parte inferior de la figura 4.38 muestra el paso final: la lectura del resultado guardado en el registro MEM/WB y la escritura de este resultado en el banco de registros mostrado en el centro de la figura.

Este recorrido de la instrucción de carga indica que toda la información que se pueda necesitar en etapas posteriores del pipeline se debe pasar a cada etapa mediante los registros de segmentación. El recorrido de una instrucción de almacenamiento es similar en la manera de ejecutarse y en la manera de pasar la información a las etapas posteriores del pipeline. A continuación se muestran las cinco etapas de un almacenamiento:

1. *Búsqueda de la instrucción:* Se lee la instrucción de la memoria usando la dirección del PC y se guarda en el registro IF/ID. Esta etapa ocurre antes de

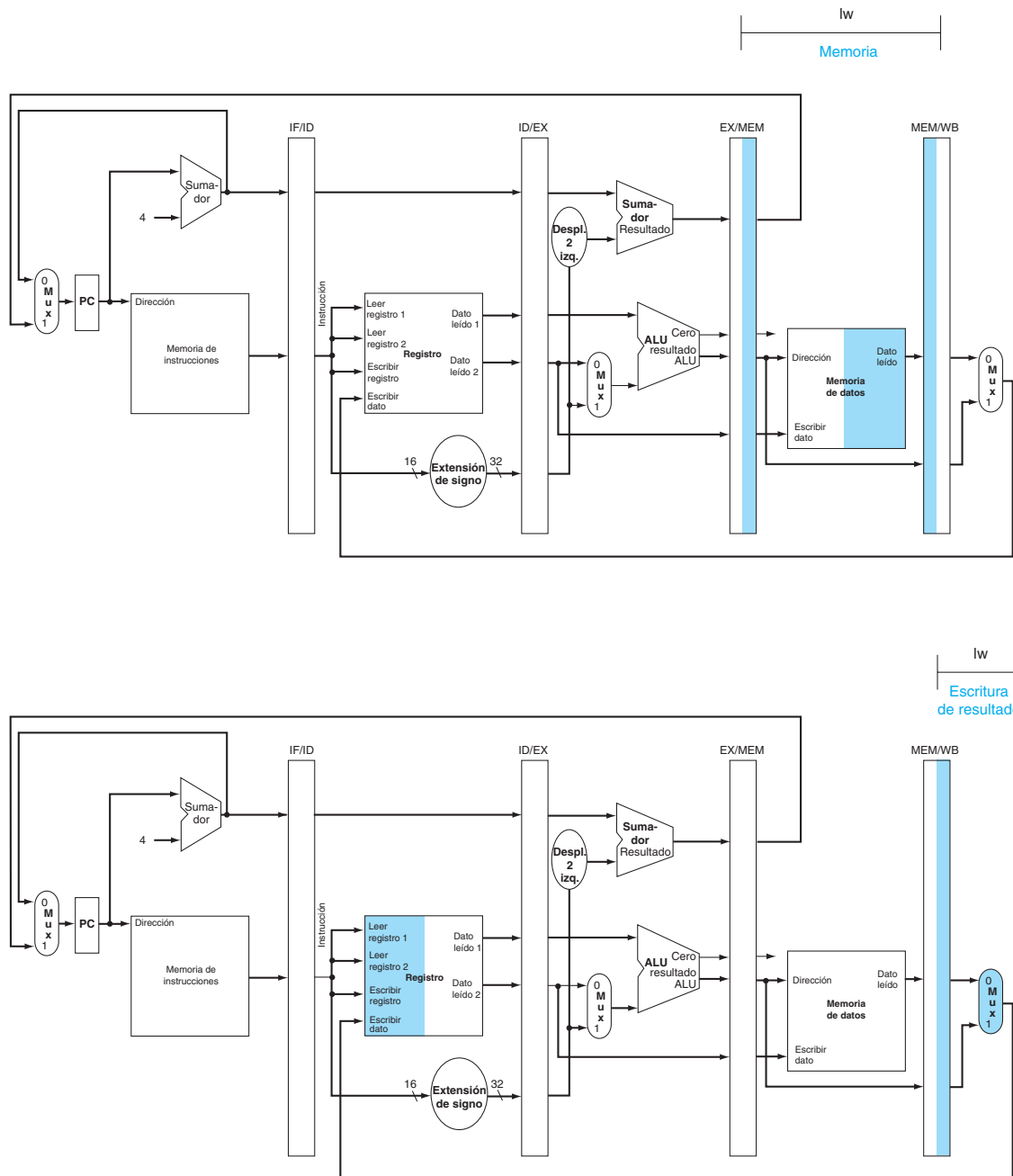


FIGURA 4.38 MEM y WB: cuarta y quinta etapas de la segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en esta etapa. Se lee la memoria de datos usando la dirección contenida en el registro EX/MEM, y el dato leído se guarda en el registro de segmentación MEM/WB. A continuación, este dato, guardado en el registro de segmentación MEM/WB, se escribe en el banco de registros, que se encuentra en el medio del camino de datos. Nota: hay un error en este diseño que se corrige en la figura 4.41.

que se identifique la instrucción, por lo que la parte superior de la figura 4.36 sirve igual tanto para cargas como para almacenamientos.

2. *Descodificación de instrucción y lectura del banco de registros*: La instrucción guardada en el registro IF/ID proporciona los identificadores de los dos registros que deben ser leídos y proporciona el valor inmediato de 16 bits cuyo signo ha de ser extendido. Estos tres valores de 32 bits se guardan en el registro de segmentación ID/EX. La parte inferior de la figura 4.36 para las instrucciones de carga también muestra las operaciones a realizar en la segunda etapa de las instrucciones de almacenamiento. En realidad, estos dos primeros pasos se ejecutan siempre igual para todas las instrucciones, ya que es demasiado pronto para que se conozca el tipo de la instrucción.
3. *Ejecución o cálculo de dirección*: La figura 4.39 muestra el tercer paso; la dirección efectiva se coloca en el registro de segmentación EX/MEM.
4. *Acceso a memoria*: La parte superior de la figura 4.40 muestra el dato que se está escribiendo en memoria. Observe que el registro que contiene el dato que se tiene que guardar en memoria fue leído en una etapa anterior y guardado en el registro ID/EX. La única manera de hacer que el dato esté disponible durante la etapa MEM es que se coloque en el registro de segmentación EX/MEM durante la etapa EX, de la misma manera que también se ha guardado la dirección efectiva.
5. *Escritura de resultado*: La parte inferior de la figura 4.40 muestra el paso final del almacenamiento. En esta etapa no ocurre nada para esta instrucción. Puesto que todas las instrucciones posteriores al almacenamiento ya están en progreso, no hay manera de acelerarlas aprovechando que la instrucción de almacenamiento no tiene nada que hacer. En general, las instrucciones pasan a través de todas las etapas aunque en ellas no tengan nada que hacer, ya que las instrucciones posteriores ya están progresando a la máxima velocidad.

La instrucción de almacenamiento ilustra una vez más que, para pasar datos de una etapa del pipeline a otra posterior, la información se debe colocar en un registro de segmentación; si no se hiciera así, la información se perdería cada vez que llegase la siguiente instrucción. Para ejecutar el almacenamiento se ha necesitado pasar uno de los registros leídos en la etapa ID a la etapa MEM, donde se guarda en memoria. El dato se ha tenido que guardar primero en el registro de segmentación ID/EX y después se ha tenido que pasar al registro de segmentación EX/MEM.

Las cargas y almacenamientos ilustran un segundo punto importante: cada componente lógico del camino de datos —como la memoria de instrucciones, los puertos de lectura del banco de registros, la ALU, la memoria de datos y el puerto de escritura en el banco de registros— pueden usarse solamente dentro de una única etapa de la segmentación. De otra forma se tendría un *riesgo estructural* (véase la página 335). Por lo tanto, estos componentes y su control pueden asociarse a una sola etapa de la segmentación.

En este momento ya se puede destapar un error en el diseño de la instrucción de carga. ¿Lo ha descubierto? ¿Qué registro se modifica en la última etapa de la carga? Más específicamente, ¿qué instrucción proporciona el identificador del registro de escritura? La instrucción guardada en el registro de segmentación

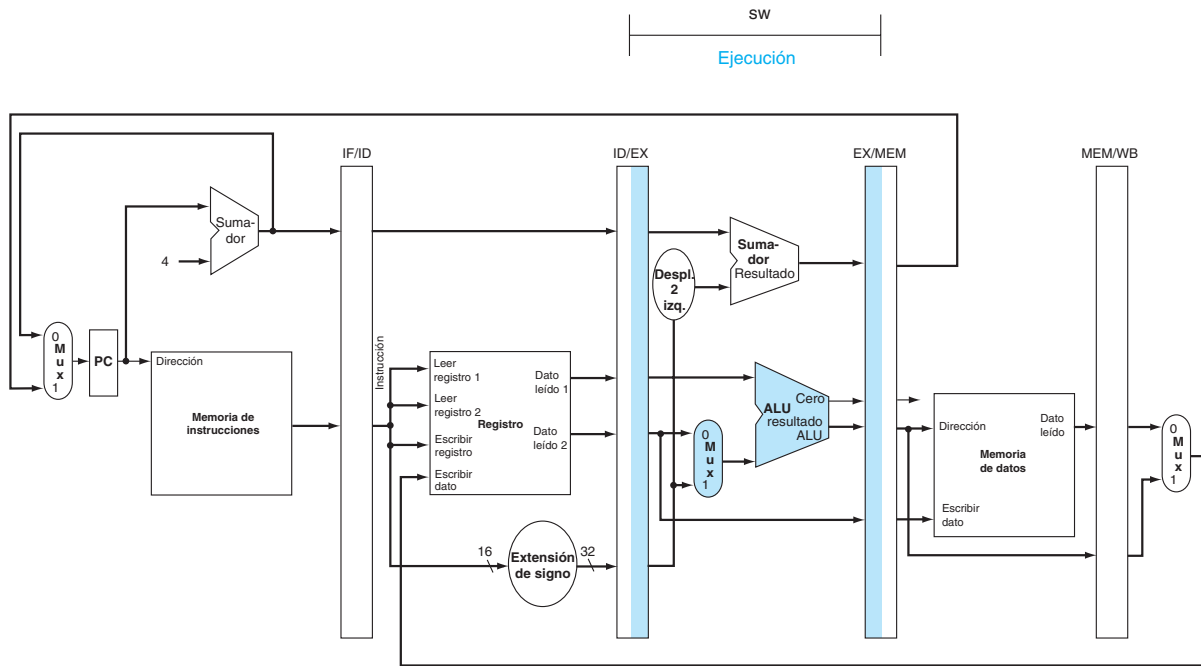


FIGURA 4.39 EX: tercera etapa de la segmentación de una instrucción de almacenamiento. A diferencia de la tercera etapa de la instrucción de carga en la figura 4.37, el valor del segundo registro es cargado en el registro de segmentación EX/MEM para ser usado en la siguiente etapa. Aunque no importaría mucho si siempre se escribiera este segundo registro en el registro de segmentación EX/MEM, para hacer el pipeline más fácil de entender sólo se escribirá en el caso de las instrucciones store.

IF/ID proporciona el número del registro sobre el que se va a escribir, ¡pero en el orden de ejecución, esta instrucción es bastante posterior a la carga que debe hacer la escritura!

Por lo tanto, para la instrucción de carga es necesario conservar el identificador del registro destino. Así como el almacenamiento ha pasado el contenido del registro que se tenía que guardar en memoria desde el registro de segmentación ID/EX al registro EX/MEM para poder ser usado posteriormente en la etapa MEM, también la carga debe pasar el identificador de registro destino desde ID/EX a través de EX/MEM y hasta MEM/WB para que pueda ser usado correctamente en la etapa WB. Otra manera de interpretar el paso del número del registro es que, para poder compartir el camino de datos segmentado, es necesario preservar la información de la instrucción que se ha leído en la etapa IF, de modo que cada registro de segmentación contiene las partes de la instrucción necesarias tanto para esa etapa como para las siguientes.

La figura 4.41 muestra la versión correcta del camino de datos, en la que se pasa el número del registro de escritura primero al registro ID/EX, después al EX/MEM y finalmente al MEM/WB. Este identificador del registro se usa durante la etapa WB para especificar el registro sobre el que se debe escribir. La figura 4.42 representa el camino de datos correcto en un solo dibujo, resaltando el hardware que se usa en cada una de las cinco etapas de la carga de las figuras 4.36 a 4.38. La explicación de cómo lograr que la instrucción de salto funcione tal y como se espera se dará en la sección 4.8.

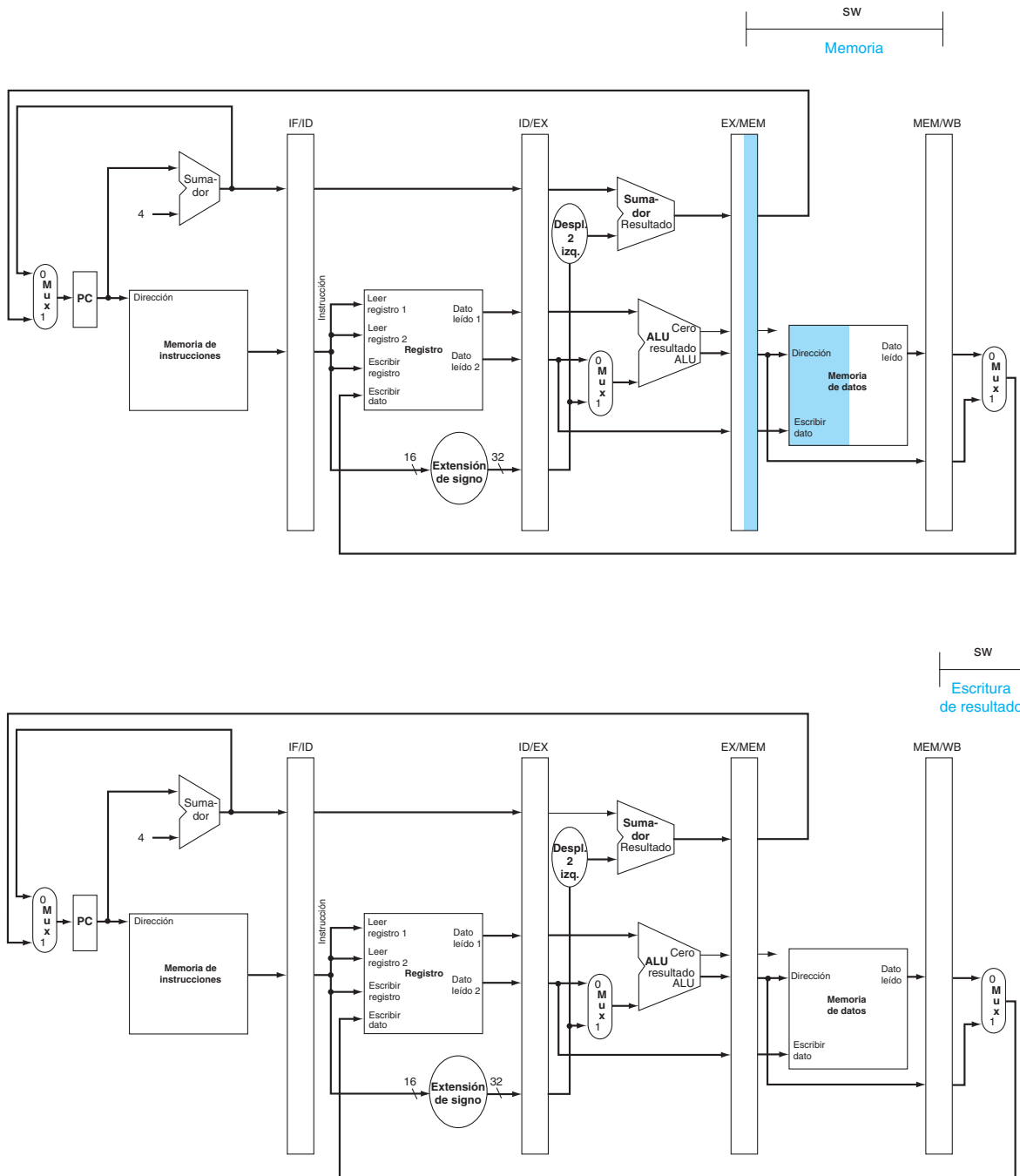


FIGURA 4.40 MEM y WB: cuarta y quinta etapas en la ejecución de un almacenamiento. En la cuarta etapa, el dato se escribe en la memoria de datos. Observe que el dato viene del registro de segmentación EX/MEM y que no se cambia nada en el registro de segmentación MEM/WB. Una vez escrito el dato en memoria, no hay nada más que hacer para el almacenamiento, por lo que en la etapa 5 no ocurre nada.

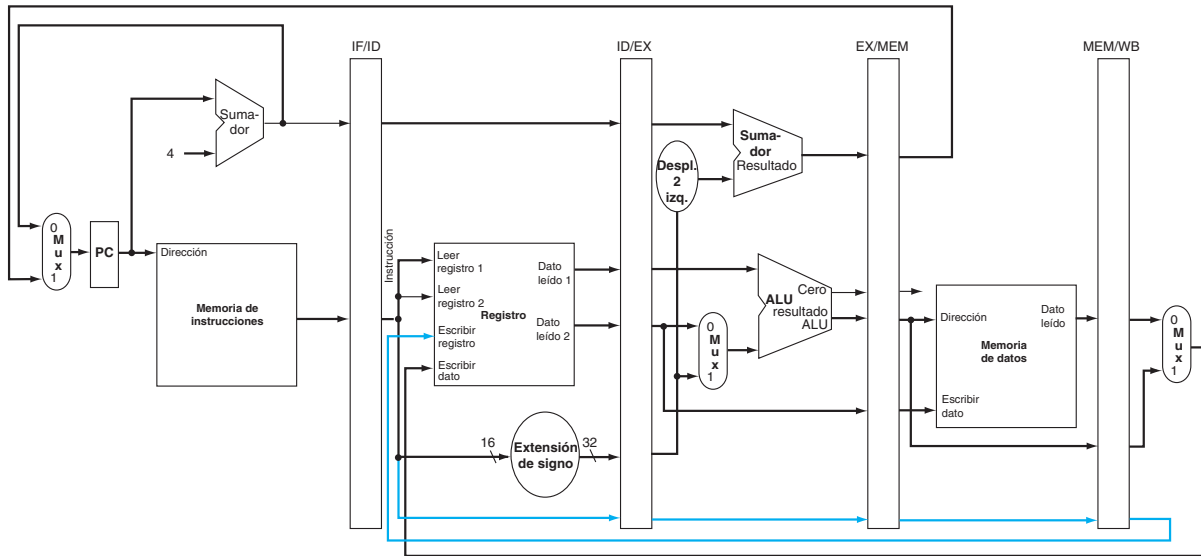


FIGURA 4.41 Camino de datos correctamente modificado para gestionar debidamente la instrucción de carga. Ahora el identificador del registro de escritura viene, junto con el dato a escribir, del registro segmentado MEM/WB. Este identificador se pasa desde la etapa ID hasta que llega al registro de segmentación MEM/WB, añadiendo 5 bits más a los tres últimos registros de segmentación. Este nuevo camino se muestra coloreado.

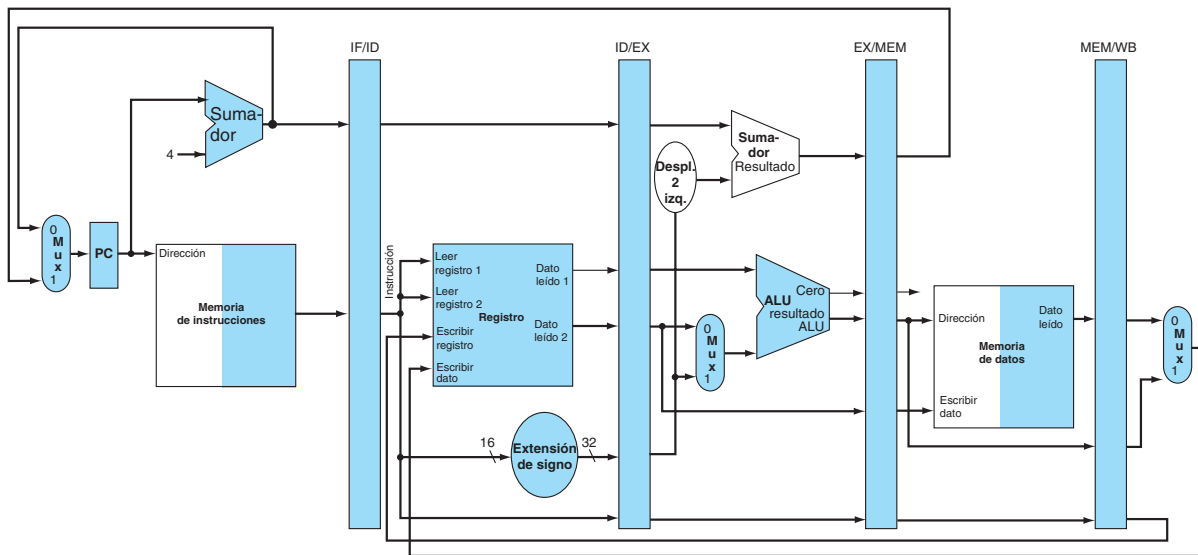


FIGURA 4.42 Porción del camino de datos de la figura 4.41 que es usada por las cinco etapas de la instrucción de carga.

Representación gráfica de la segmentación

La segmentación puede ser difícil de entender, ya que en cada ciclo de reloj hay varias instrucciones ejecutándose simultáneamente en el mismo camino de datos. Para ayudar a entenderla, el pipeline se dibuja usando dos estilos básicos: *diagramas multiciclo de segmentación*, como el de la figura 4.34 de la página 346, y *diagramas monociclo de segmentación*, como los de las figuras 4.36 a 4.40. Los diagramas multiciclo son más simples, pero no contienen todos los detalles. Por ejemplo, consideremos la siguiente secuencia de cinco instrucciones:

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

La figura 4.43 muestra el diagrama multiciclo de segmentación para estas instrucciones. El tiempo avanza de izquierda a derecha a lo largo de la página y las instrucciones avanzan desde la parte superior a la parte inferior de la página, de una forma parecida a como se representaba la segmentación de la lavandería de la figura 4.25. En cada fila del eje de las instrucciones, a lo largo de él, se coloca una representación de las etapas del pipeline, ocupando los ciclos que sean necesarios. Estas rutas de datos estilizadas representan las cinco etapas de nuestro pipeline, pero un rectángulo con el nombre de cada etapa funciona igual de bien. La figura 4.44 muestra la versión más tradicional del diagrama multiciclo de la segmentación. Debe notarse que la figura 4.43 muestra los recursos físicos usados en cada etapa, mientras que la figura 4.44 emplea el nombre de cada etapa.

Los diagramas monociclo muestran el estado del camino de datos completo durante un ciclo de reloj, y las cinco instrucciones que se encuentran en las cinco etapas diferentes del pipeline normalmente se identifican con etiquetas encima de las respectivas etapas. Se usa este tipo de figura para mostrar con más detalle lo que está ocurriendo durante cada ciclo de reloj dentro del pipeline. Habitualmente los diagramas se agrupan para mostrar las operaciones de la segmentación a lo largo de una secuencia de ciclos. Usaremos los diagramas multiciclo para dar visiones generales de las distintas circunstancias de la segmentación. (Si se quieren ver más detalles de la figura 4.43, en la [sección 4.12](#) se pueden encontrar más diagramas monociclo). Un diagrama monociclo representa un corte vertical de un diagrama multiciclo, mostrando la utilización del camino de datos por cada una de las instrucciones que se encuentran en el pipeline durante un determinado ciclo de reloj. Por ejemplo, la figura 4.45 muestra el diagrama monociclo que corresponde al quinto ciclo de las figuras 4.43 y 4.44. Obviamente, los diagramas monociclo incluyen más detalles y requieren un espacio significativamente mayor para mostrar lo que ocurre durante un cierto número de ciclos. Los ejercicios le pedirán que cree estos diagramas para otras secuencias de código.

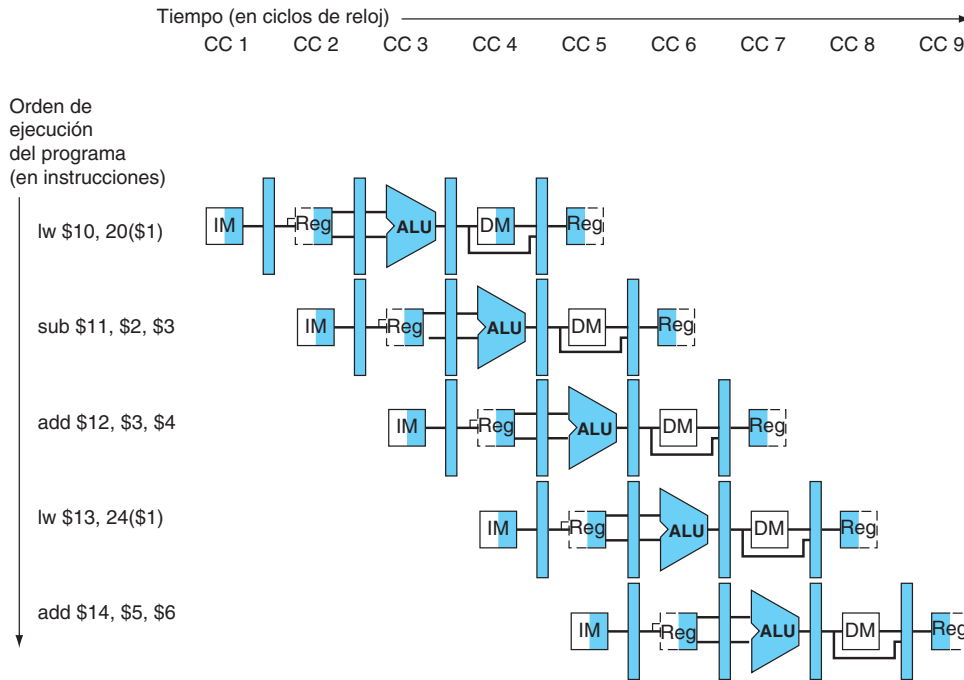


FIGURA 4.43 Diagrama multiciclo de la segmentación de cinco instrucciones. Este estilo de representación del pipeline muestra la ejecución completa de las instrucciones en una sola figura. La relación de instrucciones se hace en orden de ejecución desde la parte superior a la inferior, y los ciclos de reloj avanzan de izquierda a derecha. Al contrario de la figura 4.28, aquí se muestran los registros de segmentación entre cada etapa. La figura 4.44 muestra la manera tradicional de dibujar este diagrama.

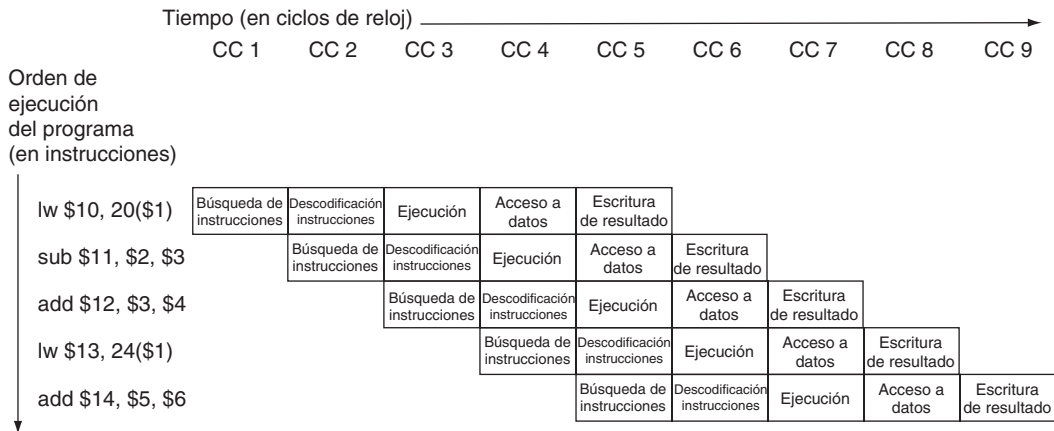


FIGURA 4.44 Versión tradicional del diagrama multiciclo de la segmentación de cinco instrucciones que se muestra en la figura 4.43.

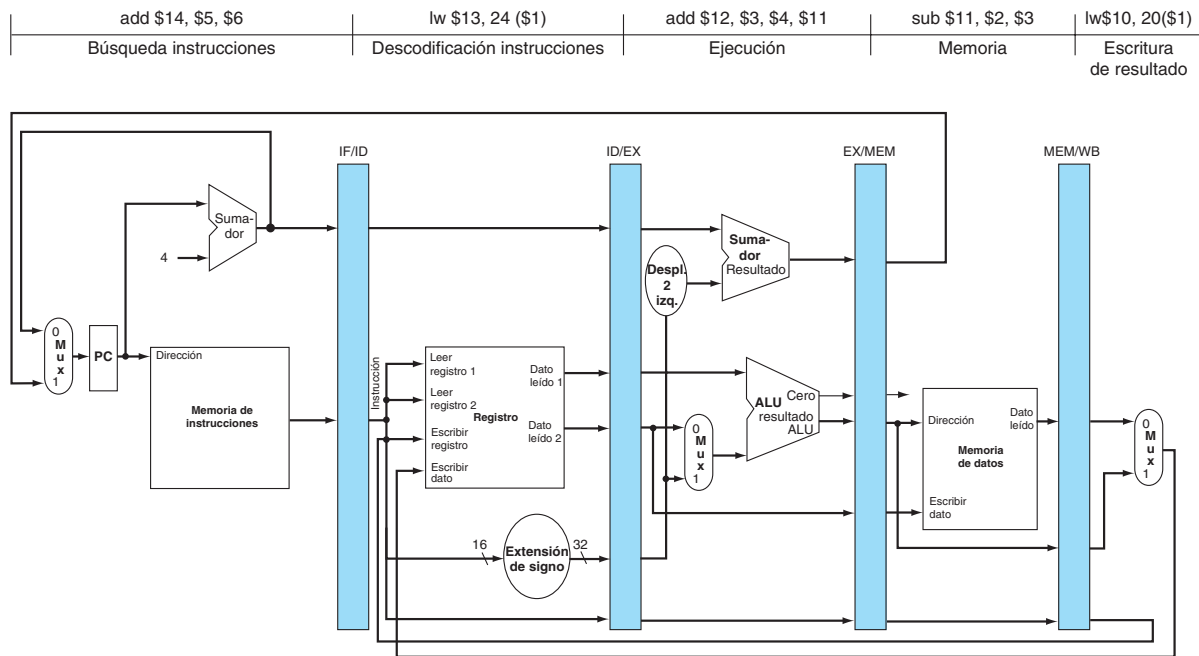


FIGURA 4.45 Diagrama monociclo correspondiente al ciclo 5 del pipeline de las figuras 4.43 y 4.44. Como puede verse, la figura monociclo es una porción vertical del diagrama multiciclo.

Autoevaluación

Un grupo de estudiantes estaba debatiendo sobre la eficiencia de un pipeline de cinco etapas, cuando uno de ellos se dio cuenta de que no todas las instrucciones están activas en cada una de las etapas del pipeline. Después de decidir ignorar el efecto de los riesgos, los estudiantes hicieron las siguientes afirmaciones. ¿Cuáles son correctas?

1. Permitiendo que las instrucciones de salto condicional e incondicional y que las instrucciones que usan la ALU tarden menos ciclos que los cinco requeridos por la instrucción de carga incrementará las prestaciones en todos los casos.
2. Intentar que algunas instrucciones tarden menos ciclos en el pipeline no ayuda, ya que la productividad viene determinada por el ciclo de reloj; el número de etapas del pipeline que requiere cada instrucción afecta a la latencia y no a la productividad.
3. No se puede hacer que las instrucciones que usan la ALU tarden menos ciclos debido a la escritura del resultado final, pero las instrucciones de salto condicional e incondicional sí que pueden tardar menos ciclos, y por tanto hay alguna oportunidad de mejorar.
4. En lugar de tratar que las instrucciones tarden menos ciclos, deberíamos explorar la posibilidad de hacer que el pipeline fuera más largo, de forma que las instrucciones tardaran más ciclos, pero que los ciclos fueran más cortos. Esto podría mejorar las prestaciones.

Control de la segmentación

Del mismo modo que en la sección 4.3 añadimos el control a un camino de datos de ciclo único, ahora añadiremos el control a un camino de datos segmentado. Comenzaremos con un diseño simple en el que el problema se verá a través de una gafas con cristales de color rosa. En las secciones 4.7 a 4.9, se prescindirá de estas gafas para desvelar así los riesgos presentes en el mundo real.

El primer paso consiste en etiquetar las líneas de control en el nuevo camino de datos. La figura 4.46 muestra estas líneas. El control del camino de datos sencillo de la figura 4.17 se ha reutilizado lo máximo posible. En concreto, se usa la misma lógica de control para la ALU, la misma lógica de control para los saltos, el mismo multiplexor para los identificadores de registro destino, y las mismas líneas de control. Estas funciones se definieron en las figuras 4.12, 4.16 y 4.18. Para que el texto que sigue sea más fácil de seguir, las figuras 4.47 a 4.48 reproducen la misma información clave.

En el Computador 6600, quizás aún más que en cualquier computador anterior, es el sistema de control el que marca la diferencia.

James Thornton, *Design of a Computer: The Control Data 6600*, 1970

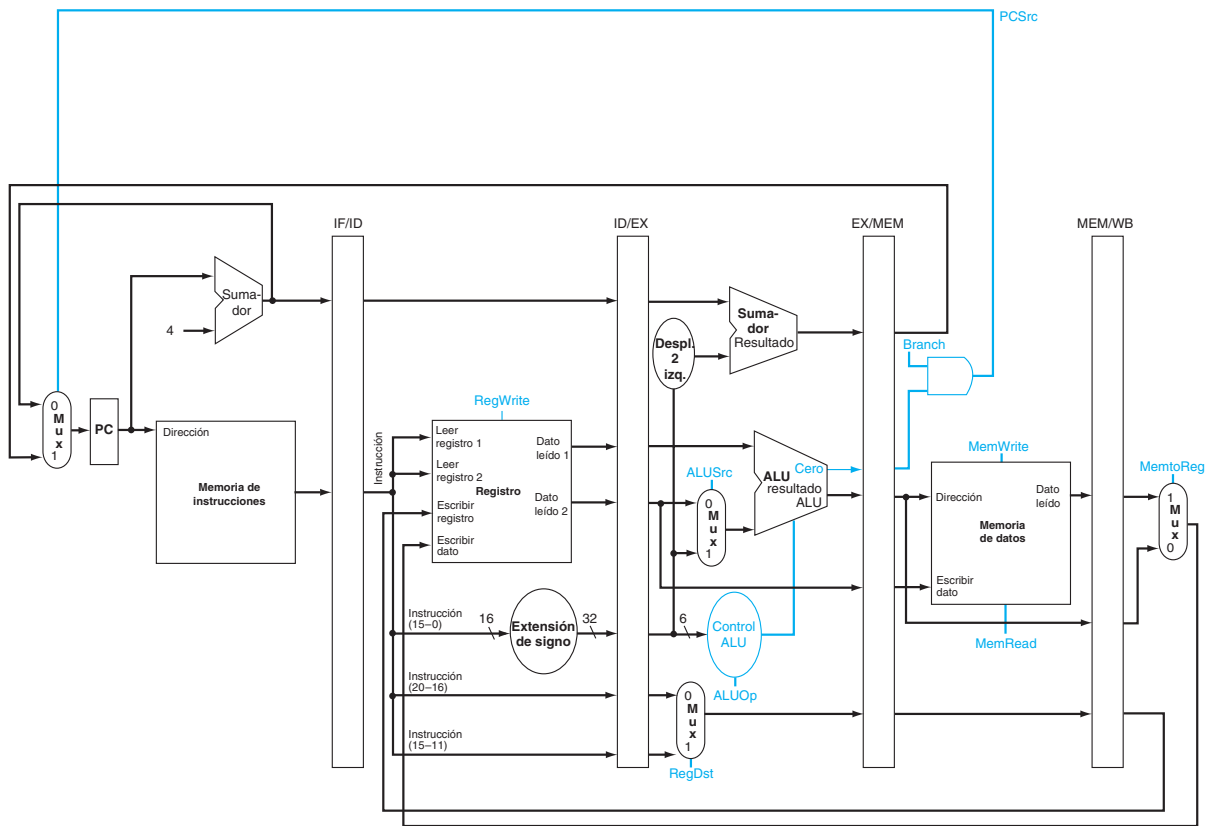


FIGURA 4.46 El camino de datos segmentado de la figura 4.41 en la que se identifican las señales de control. Este camino de datos toma prestada la lógica de control de la sección 4.4 para el PC, el identificador de registro fuente y destino, y el control de la ALU. Debe notarse que en la etapa EX, como entrada de control de la ALU, son ahora necesarios los 6 bits del campo funct (código de función) de la instrucción, y por tanto estos bits también deben ser incluidos en el registro de segmentación ID/EX. Recuerde que ya que estos 6 bits también pueden representar los 6 bits menos significativos del campo inmediato de la instrucción, el registro de segmentación ID/EX los proporciona como parte del campo inmediato, pues la extensión del signo mantiene el valor de los bits.

Código de operación	ALUOp	Operación	Código Función	Acción deseada en ALU	Entrada Control ALU
LW	00	cargar palabra	XXXXXX	sumar	0010
SW	00	almacenar palabra	XXXXXX	sumar	0010
Saltar si igual	01	saltar si igual	XXXXXX	restar	0110
tipo R	10	sumar	100000	sumar	0010
tipo R	10	restar	100010	restar	0110
tipo R	10	AND	100100	Y-lógica	0000
tipo R	10	OR	100101	O-lógica	0001
tipo R	10	iniciar si menor que	101010	iniciar si menor que	0111

FIGURA 4.47 Copia de la figura 4.12. Esta figura muestra cómo activar los bits de control de la ALU dependiendo de los bits de control de ALUOp y de los diferentes códigos de función de las instrucciones de tipo R.

Nombre de señal	Efecto cuando desactiva (0)	Efecto cuando activa (1)
RegDst	El identificador del registro destino para la escritura a registro viene del campo rt (bits 20:16).	El identificador del registro destino para la escritura a registro viene del campo rd (bits 15:11).
RegWrite	Ninguno.	El registro se escribe con el valor de escritura.
ALUSrc	El segundo operando de la ALU proviene del segundo registro leído del banco de registros.	El segundo operando de la ALU son los 16 bits de menor peso de la instrucción con el signo extendido.
PCSrc	El PC es reemplazado por su valor anterior más 4 ($PC + 4$).	El PC es reemplazado por la salida del sumador que calcula la dirección destino del salto.
MemRead	Ninguno.	El valor de la posición de memoria designada por la dirección se coloca en la salida de lectura.
MemWrite	Ninguno.	El valor de la posición de memoria designada por la dirección se reemplaza por el valor de la entrada de datos.
MementoReg	El valor de entrada del banco de registros proviene de la ALU.	El valor de entrada del banco de registros proviene de la memoria.

FIGURA 4.48 Copia de la figura 4.16. Se define la función de cada una de las siete señales de control. Las líneas de control de la ALU (ALUOp) se definen en la segunda columna de la figura 4.47. Cuando se activa el bit de control de un multiplexor de dos entradas, éste selecciona la entrada correspondiente a 1. En caso contrario, si el control está desactivado, el multiplexor selecciona la entrada 0. Obsérvese que en la figura 4.46, PCSrc se controla mediante una puerta AND. Si la señal Branch y la señal Cero de la ALU están activadas, entonces la señal PCSrc es 1; en caso contrario es 0. El control activa la señal Branch sólo para una instrucción beq; en otro caso PCSrc se pone a 0.

Instrucción	Ejecución / cálculo de dirección líneas de control				Acceso a memoria líneas de control			Escritura de resultado líneas de control	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
Formato R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

FIGURA 4.49 Los valores de las líneas de control son los mismos que en la figura 4.18, pero han sido distribuidas en tres grupos que corresponden a las tres últimas etapas de la segmentación.

Igual que para la implementación monociclo, se supondrá que el PC se escribe en cada ciclo y que no es necesaria una línea separada para controlar la escritura en el PC. Por el mismo motivo, no existen señales de escritura separadas para los registros de segmentación (IF/ID, ID/EX, EX/MEM y MEM/WB), en los cuales también se escribe en cada ciclo de reloj.

Para especificar el control en el pipeline sólo se necesita activar los valores de control durante cada etapa de la segmentación. Puesto que cada línea de control se asocia con un componente activo en una única etapa, las líneas de control se pueden dividir en cinco grupos según las etapas de la segmentación:

1. *Búsqueda de instrucción*: Las señales de control para leer de la memoria de instrucciones y para escribir el PC están siempre activadas, por lo que el control en esta etapa no tiene nada de especial.
2. *Descodificación de instrucción y lectura del banco de registros*: Aquí pasa lo mismo que en la etapa anterior, por lo que no hay líneas de control opcionales que activar.
3. *Ejecución / cálculo de dirección*: Las señales a activar son RegDst, ALUOp y ALUSrc (véanse las figuras 4.47 y 4.48). Estas señales seleccionan el registro de resultado, la operación de la ALU, y seleccionan como entrada de la ALU o bien el dato leído del segundo registro o bien el valor inmediato con signo extendido.

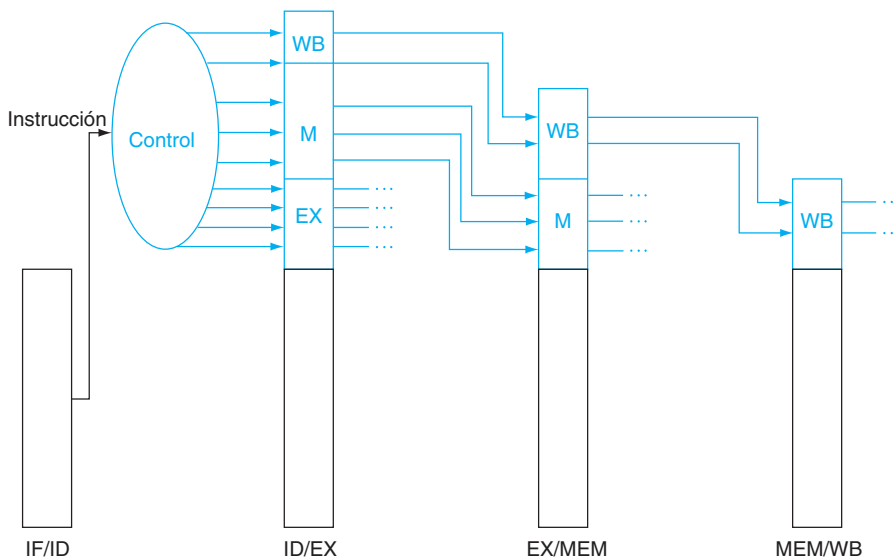


FIGURA 4.50 Líneas de control para las tres etapas finales. Observe que cuatro de las nueve líneas de control se usan en la etapa EX, mientras que las cinco restantes pasan al registro de segmentación EX/MEM, que ha sido extendido para poder almacenar las líneas de control; tres se usan durante la etapa MEM, y las dos últimas se pasan al registro MEM/WB para ser usadas en la etapa WB.

4. *Acceso a memoria:* Las líneas de control que se activan en esta etapa son Branch, MemRead y MemWrite. Estas señales se activan para las instrucciones beq, load, y store respectivamente. Recuerde que PCSrc en la figura 4.48 selecciona la dirección siguiente en orden secuencial a no ser que la lógica de control active la señal Branch y el resultado de la ALU sea cero.
5. *Escritura de resultado:* Las dos líneas de control son MemtoReg, la cual decide entre escribir en el banco de registros o bien el resultado de la ALU o bien el valor leído de memoria, y RegWrite, que escribe el valor escogido.

Ya que al segmentar el camino de datos no cambia el significado de las líneas de control, se pueden emplear los mismos valores para el control que antes. La figura 4.49 presenta los mismos valores que en la sección 4.4, pero ahora las nueve líneas de control se agrupan por etapas de segmentación.

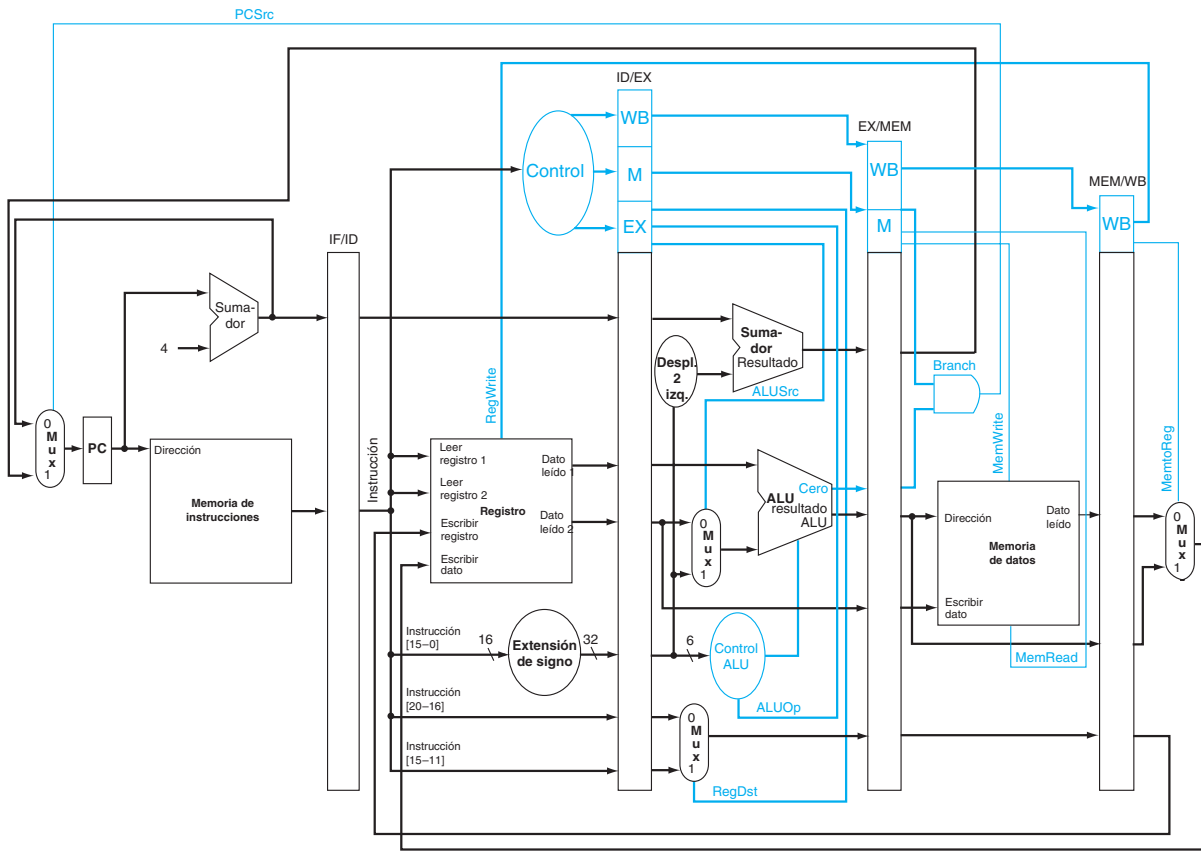


FIGURA 4.51 Camino de datos segmentado de la figura 4.40, con las señales de control conectadas a la parte de control de los registros de segmentación. Los valores de control de las tres últimas etapas se crean durante la descodificación de la instrucción y son escritos en el registro de segmentación ID/EX. En cada etapa de segmentación se usan ciertas líneas de control, y las líneas restantes se pasan a la etapa siguiente.

Realizar el control significa activar las nueve líneas de control a estos valores en cada etapa para cada instrucción. La manera más simple de hacerlo es extendiendo los registros de segmentación para incluir la información de control.

Ya que las líneas de control empiezan en la etapa EX, se puede crear la información de control durante la decodificación de la instrucción. La figura 4.50 muestra que estas señales de control se usan en la etapa de segmentación adecuada mientras la instrucción avanza por el pipeline, tal y como avanza el identificador de registro destino de las cargas en la figura 4.41. La figura 4.51 muestra el camino de datos completo con los registros de segmentación extendidos y con las líneas de control conectadas a la etapa correcta. (Si se quieren más detalles, la [sección 4.12](#) tiene más diagramas monociclo con ejemplos de ejecución de códigos MIPS en el pipeline).

4.7

Riesgos de datos: anticipación frente a bloqueos

Los ejemplos de la sección anterior muestran la potencia de la ejecución segmentada y cómo el hardware realiza esta tarea. Ahora es el momento de quitarse las gafas con cristales de color rosa y mirar qué pasa en programas reales. Las instrucciones de las figuras 4.43 a 4.45 eran independientes; ninguna de ellas usaba los resultados calculados por alguna de las anteriores. En cambio, en la sección 4.5 se vio que los riesgos de datos suponían un obstáculo para la ejecución segmentada.

Vamos a ver una secuencia de instrucciones con varias dependencias, mostradas en color.

```
sub    $2, $1, $3    # sub escribe en registro $2
and    $12, $2, $5    # 1er operando($2) depende de sub
or     $13, $6, $2    # 2º operando($2) depende de sub
add    $14, $2, $2    # 1er($2) y 2º($2) depende de sub
sw     $15, 100($2)  # Base ($2) depende de sub
```

Las últimas cuatro instrucciones dependen todas del resultado de la primera instrucción, guardado en el registro \$2. Si este registro tenía el valor 10 antes de la instrucción de resta y el valor -20 después, la intención del programador es que el valor -20 sea usado por las instrucciones posteriores que referencian al registro \$2.

¿Cómo se ejecutaría esta secuencia de instrucciones en el procesador segmentado? La figura 4.52 ilustra la ejecución de estas instrucciones usando una representación multiciclo. Para mostrar la ejecución de esta secuencia de instrucciones en el pipeline, la parte superior de la figura 4.52 muestra el valor del registro \$2, que cambia en la mitad del quinto ciclo, cuando la instrucción `sub` escribe su resultado.

Uno de los riesgos potenciales se puede resolver con el propio diseño del hardware del banco de registros: ¿qué ocurre cuando un registro se lee y se escribe en el mismo ciclo de reloj? Se supone que la escritura se hace en la primera mitad del ciclo y la lectura se hace en la segunda mitad, por lo que la lectura proporciona el valor que acaba de ser escrito. Como esto ya lo hacen muchas implementaciones de bancos de registros, en este caso entenderemos que no hay riesgo de datos.

*¿Qué quieres decir,
por qué se debería
construir? Es un baipás.
Debes construir
realimentaciones.*

Douglas Adams,
*Hitchhikers Guide
to the Galaxy*, 1979

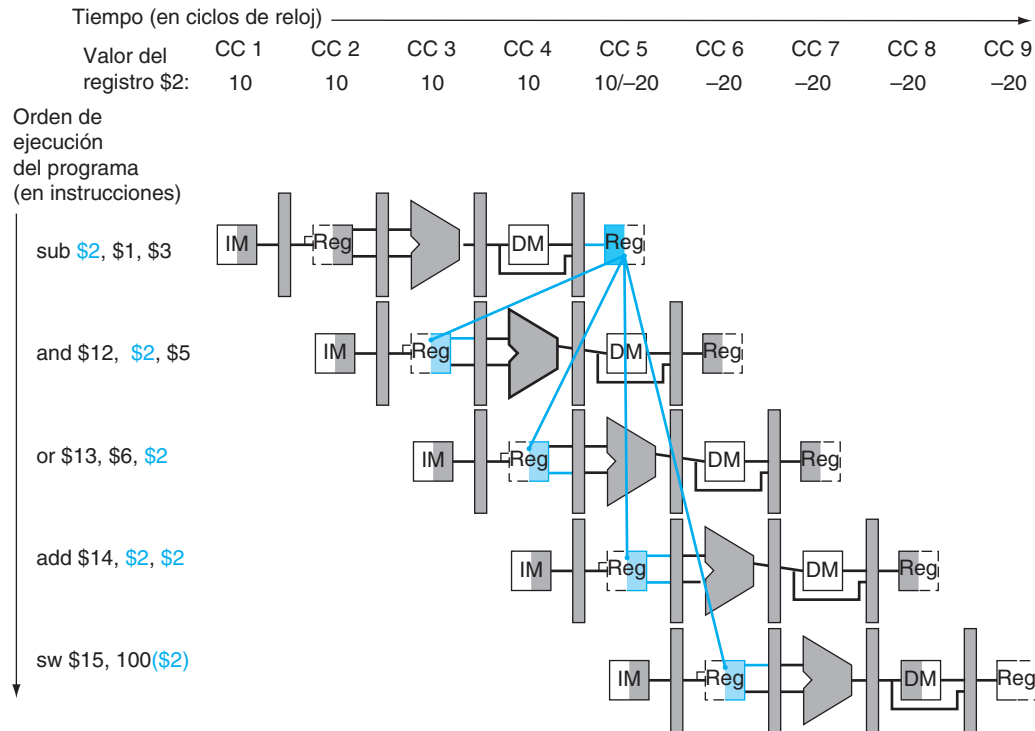


FIGURA 4.52 Dependencias en la segmentación de la ejecución de la secuencia de cinco instrucciones usando caminos de datos simplificados para mostrar las dependencias. Todas las acciones dependientes se muestran en color y “CC *i*” en la parte superior de la figura representa el ciclo de reloj *i*. La primera instrucción escribe en \$2, y todas las siguientes instrucciones leen de \$2. Este registro se escribe en el ciclo 5, por lo que el valor correcto no está disponible antes del ciclo 5. (La lectura de un registro durante un ciclo de reloj retornará el valor escrito al final de la primera mitad del ciclo, si es que esa escritura se produce). Las líneas coloreadas desde la parte superior del camino de datos a la parte inferior muestran las dependencias. Aquellas que deben ir hacia atrás en el tiempo constituyen los *riesgos de datos en el pipeline*.

La figura 4.52 muestra que los valores que se leen del registro \$2 *no* representarían el resultado de la instrucción *sub* a menos que la lectura del registro se hiciera durante el ciclo 5 o después. Las instrucciones que obtendrían el valor correcto de -20 son *add* y *sw*; las instrucciones *and* y *or* obtendrían el valor 10, que es incorrecto. Al utilizar este estilo de dibujo estos problemas se ven claramente porque una línea de dependencia va hacia atrás en el tiempo.

Como se ha mencionado en la sección 4.5, el resultado está disponible al final de la etapa EX, o lo que es lo mismo al final del tercer ciclo. ¿Cuándo se necesita realmente ese dato para las instrucciones *and* y *or*? Al principio de la etapa EX, o lo que es lo mismo en los ciclos 4 y 5, respectivamente. Por tanto, podemos ejecutar este segmento sin bloqueos si simplemente anticipamos los datos tan pronto como estén disponibles a cualquiera de las unidades que necesiten el dato antes de que esté disponible en el banco de registros para ser leído.

¿Cómo funciona la anticipación de resultados? Por simplicidad, en el resto de esta sección se considerará la posibilidad de anticipar datos sólo a las operaciones que están en la etapa EX, que pueden ser operaciones de tipo ALU o el cálculo de

una dirección efectiva. Esto significa que cuando una instrucción trata de usar en su etapa EX el registro que una instrucción anterior intenta escribir en su etapa WB, lo que realmente se necesita es disponer de su valor como entrada a la ALU.

Una notación que ponga nombre a los campos de los registros de segmentación permite una descripción más precisa de las dependencias. Por ejemplo, “ID/EX.RegisterRs” se refiere al identificador de un registro cuyo valor se encuentra en el registro de segmentación ID/EX; esto es, el que viene del primer puerto de lectura del banco de registros. La primera parte del nombre, a la izquierda, es el identificador del registro de segmentación; la segunda parte es el nombre del campo de ese registro. Usando esta notación, existen dos parejas de condiciones para detectar riesgos:

- 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
- 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
- 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
- 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

El primer riesgo en la secuencia de la página 363 se encuentra en el registro \$2, entre el resultado de sub \$2, \$1, \$3 y el primer operando de lectura de la instrucción and \$12, \$2, \$5. Este riesgo se puede detectar cuando la instrucción and está en la etapa EX y la otra instrucción está en la etapa MEM, por lo que corresponde con la condición 1a:

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs} = \$2$$

Detección de dependencias

Clasificar las dependencias en esta secuencia de la página 363:

```
sub    $2,    $1, $3  # Registro $2 escrito por sub
and    $12,   $2, $5  # 1er operando($2) escrito por sub
or     $13,   $6, $2  # 2º operando($2) escrito por sub
add    $14,   $2, $2  # 1er($2) y 2º($2) escrito por sub
sw     $15,   100($2) # Índice($2) escrito por sub
```

Tal como se ha mencionado anteriormente, el riesgo sub-add es de tipo 1a. Los riesgos restantes son:

- El riesgo sub-or es de tipo 2b:
MEM/WB.RegisterRd = ID/EX.RegisterRt = \$2
- Las dos dependencias en sub-add no son riesgos ya que el banco de registros proporciona el valor correcto durante la etapa ID de add.
- No hay riesgo de datos entre sub y sw ya que sw lee \$2 un ciclo de reloj después que sub escriba \$2.

EJEMPLO

RESPUESTA

Ya que algunas instrucciones no escriben en registros, esta política es poco precisa; algunas veces se anticiparía un valor innecesario. Una solución consiste sencillamente en comprobar si la señal RegWrite estará activa: para ello se examina el campo de control WB del registro de segmentación durante las etapas de EX y MEM. Además, MIPS requiere que cada vez que se use como operando el registro \$0, se debe producir un valor para el operando igual a cero. En el caso que una instrucción tenga como destino \$0 (por ejemplo, `sl $0, $1, $2`), se debe evitar la anticipación de un resultado que posiblemente sea diferente de cero. El no anticipar resultados destinados a \$0 libera de restricciones al programador de ensamblador y al compilador para usar \$0 como registro destino. Por lo tanto, las condiciones mencionadas antes funcionarán correctamente siempre que se añada `EX/MEM.RegisterRd ≠ 0` a la primera condición de riesgo y `MEM/WB.RegisterRd ≠ 0` a la segunda condición.

Una vez detectados los riesgos, la mitad del problema está resuelto, pero todavía falta anticipar el dato correcto.

La figura 4.53 muestra las dependencias entre los registros de segmentación y las entradas de la ALU para la misma secuencia de código de la figura 4.52. El cambio radica en que la dependencia empieza en un registro de segmentación en lugar de esperar a que en la etapa WB se escriba en el banco de registros. Por lo tanto, el dato que se tiene que adelantar ya está disponible en los registros de segmentación con tiempo suficiente para las instrucciones posteriores.

Si se pudieran obtener las entradas de la ALU de *cualquier* registro de segmentación en vez de sólo del registro de segmentación ID/EX, entonces se podría adelantar el dato correcto. Bastaría con añadir multiplexores adicionales en la entrada de la ALU, y el control apropiado para poder ejecutar el pipeline a máxima velocidad en presencia de estas dependencias.

Por ahora, supondremos que las únicas instrucciones que necesitan avanzar su resultado son las cuatro de tipo R: `add`, `sub`, `and` y `or`. La figura 4.54 muestra un primer plano de la ALU y de los registros de segmentación antes y después de añadir la anticipación de datos. La figura 4.55 muestra los valores de las líneas de control de los multiplexores de la ALU que seleccionan bien los valores del banco de registros, o bien uno de los valores anticipados.

El control de la anticipación estará en la etapa EX, ya que los multiplexores de anticipación previos a la ALU se encuentran en esta etapa. Por lo tanto se deben pasar los identificadores de los registros fuente desde la etapa ID a través del registro de segmentación ID/EX para determinar si se deben adelantar los valores. El campo `rt` ya se tiene (bits 20-16). Antes de la anticipación, el registro ID/EX no necesitaba incluir espacio para guardar el campo `rs`. Por lo tanto se debe añadir `rs` (bits 25-21) al registro ID/EX.

Ahora escribiremos tanto las condiciones para detectar riesgos como las señales de control para resolverlos:

1. Riesgo EX:

```

si (EX/MEM.RegWrite
y (EX/MEM.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
si (EX/MEM.RegWrite
y (EX/MEM.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

```

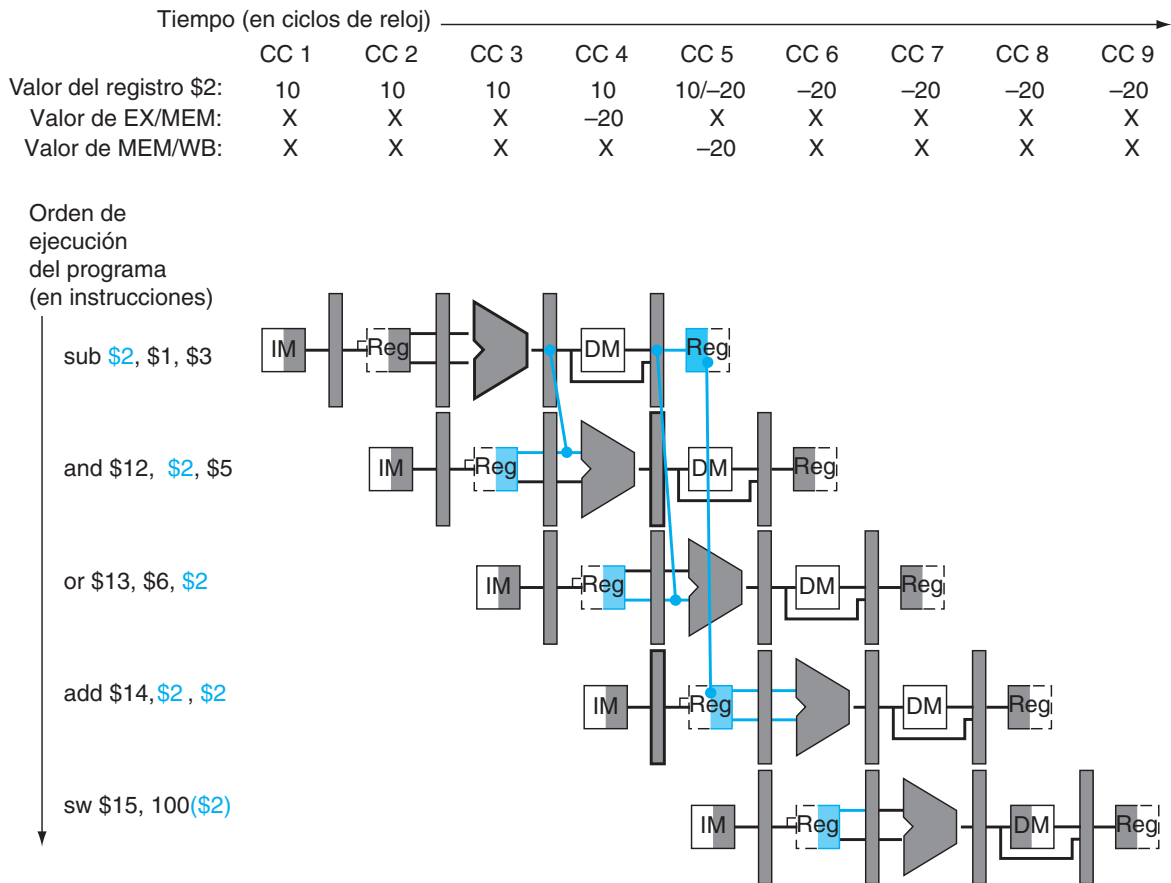
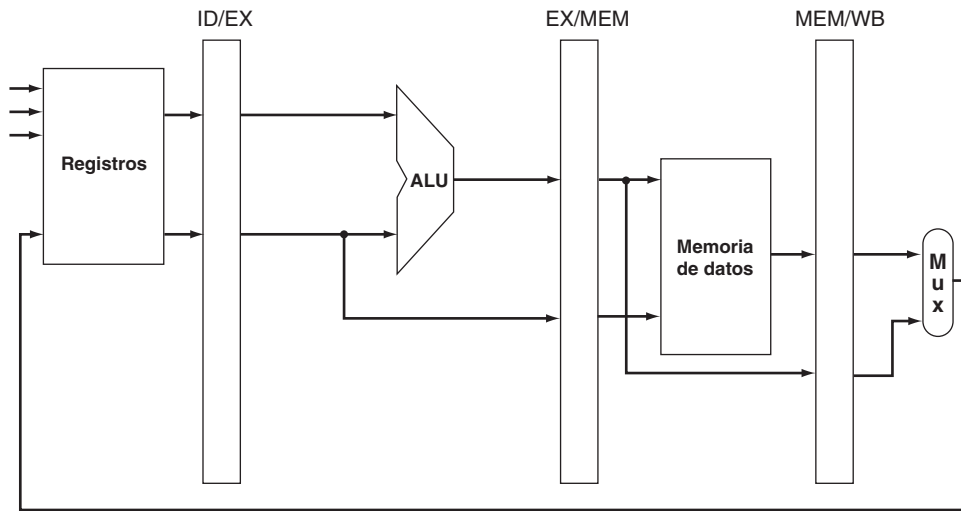


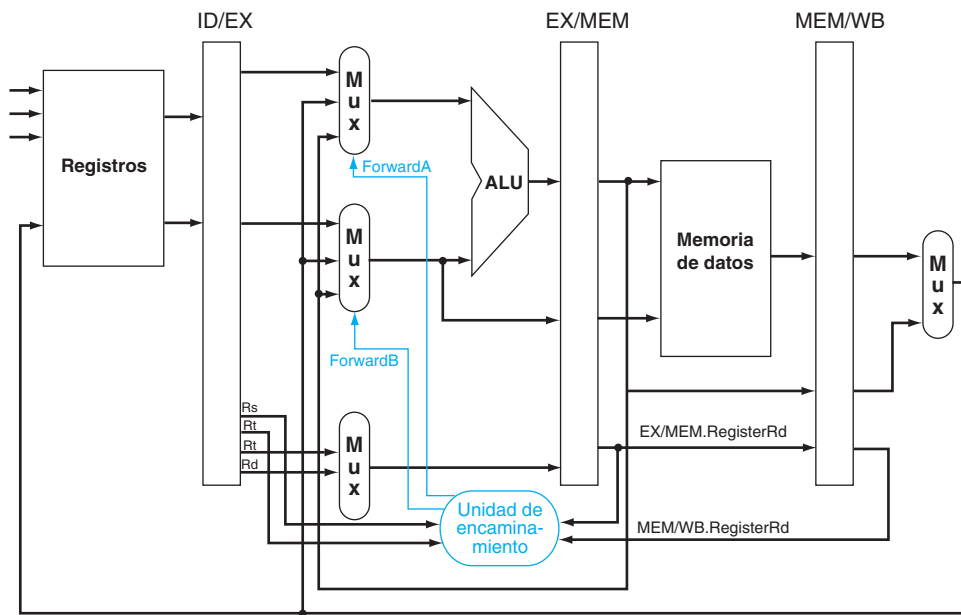
FIGURA 4.53 Las dependencias entre los registros de segmentación se mueven hacia adelante en el tiempo, por lo que es posible proporcionar a la ALU los operandos que necesitan las instrucciones and y or anticipando los resultados que se encuentran en dichos registros. Los valores en los registros de segmentación muestran que el valor deseado está disponible antes de que se escriba en el banco de registros. Se supone que el banco de registros adelanta el valor que se lee y escribe durante el mismo ciclo de reloj, por lo que la instrucción add no se debe bloquear, ya que los valores le vienen desde el banco de registros en vez de desde un registro de segmentación. La anticipación de datos dentro del banco de registros -esto es, la lectura obtiene el valor que se escribe en ese mismo ciclo- explica por qué el ciclo 5 muestra al registro \$2 con el valor 1 al principio y con el valor -20 al final del ciclo de reloj. Como en el resto de la sección, consideraremos todos los casos de anticipación de datos excepto para los valores que deben ser guardados en memoria en el caso de las instrucciones de almacenamiento.

Observe que el campo EX/MEM.RegisterRd es el registro destino tanto para una instrucción ALU (que proviene del campo Rd de la instrucción) como para una carga (que proviene del campo Rt).

Este caso adelanta el resultado desde una instrucción anterior a cualquiera de las entradas de la ALU. Si la instrucción previa va a escribir en el banco de registros y el identificador del registro destino es igual al identificador de registro de lectura A o B en la entrada de la ALU, suponiendo que no es el registro 0, entonces se hace que el multiplexor tome el valor del registro de segmentación EX/MEM.



a. Sin encaminamiento



b. Con encaminamiento

FIGURA 4.54 En la parte superior se encuentra la ALU y los registros de segmentación antes de añadir la anticipación de resultados. En la parte inferior, los multiplexores se han expandido para añadir los caminos de anticipación, y también se muestra la unidad de anticipación. El hardware nuevo se muestra en color. Esta figura es, sin embargo, un dibujo simplificado que deja fuera detalles del camino de datos completo, como por ejemplo el hardware de extensión de signo. Observe que el campo ID/EX.RegisterRt se muestra dos veces, una vez conectado al multiplexor y otra vez conectado a la unidad de anticipación, pero es una sola señal. Como en la discusión anterior, se ignora la anticipación de datos para el valor que debe ser guardado en memoria en el caso de las instrucciones de almacenamiento. Obsérvese que esta técnica funciona también para instrucciones `slt`.

2. Riesgo MEM:

```

si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

```

Como ya hemos mencionado anteriormente, no hay riesgo en la etapa WB, ya que se supone que el banco de registros proporciona el valor correcto si la instrucción en la etapa ID lee el mismo registro que escribe la instrucción situada en la etapa WB. Un banco de registros de tal funcionalidad realiza otro tipo de anticipación, pero ocurre dentro del propio banco de registros.

Una posible complicación consiste en tener riesgos de datos potenciales entre el resultado de la instrucción situada en WB, el resultado de la instrucción en la etapa MEM y el operando fuente de la instrucción en la etapa de ALU. Por ejemplo, cuando se suma un vector de números en un único registro, todas las instrucciones de la secuencia leerán y escribirán en el mismo registro:

```

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

```

En este caso, el resultado se anticipa desde la etapa MEM ya que el resultado en esta etapa es el más reciente. Por lo tanto, el control del riesgo en la etapa MEM sería (con los cambios adicionales resaltados):

```

si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
y (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

si (MEM/WB.RegWrite
y (MEM/WB.RegisterRd ≠ 0)
y (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
y (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

```

La figura 4.56 muestra el hardware necesario para implementar la anticipación de datos con las operaciones que usan resultados en la etapa EX.(20) Observe que el campo EX/MEM.RegisterRd es el registro destino de tanto de una instrucción de

Control del multiplexor	Fuente	Explicación
ForwardA = 00	ID/EX	EL primer operando de la ALU viene del banco de registros
ForwardA = 10	EX/MEM	EL primer operando de la ALU se anticipa del resultado anterior de la ALU
ForwardA = 01	MEM/WB	EL primer operando de la ALU se anticipa de la memoria de datos o de un resultado de la ALU anterior
ForwardB = 00	ID/EX	EL segundo operando de la ALU viene del banco de registros
ForwardB = 10	EX/MEM	EL segundo operando de la ALU se anticipa del resultado anterior de la ALU
ForwardB = 01	MEM/WB	EL segundo operando de la ALU se anticipa de la memoria de datos o de un resultado de la ALU anterior

FIGURA 4.55 Los valores de control para los multiplexores de anticipación de datos de la figura 4.54. El valor inmediato con signo que es otra entrada de la ALU se describe en la Extensión que se encuentra al final de esta sección.

la ALU (que se obtiene del campo Rd de la instrucción) como de una carga (que se obtiene del campo Rt).

Para más detalles, la [sección 4.12](#) en el CD muestra dos fragmentos de código MIPS con riesgos y anticipación, ilustrados con diagramas monociclo.

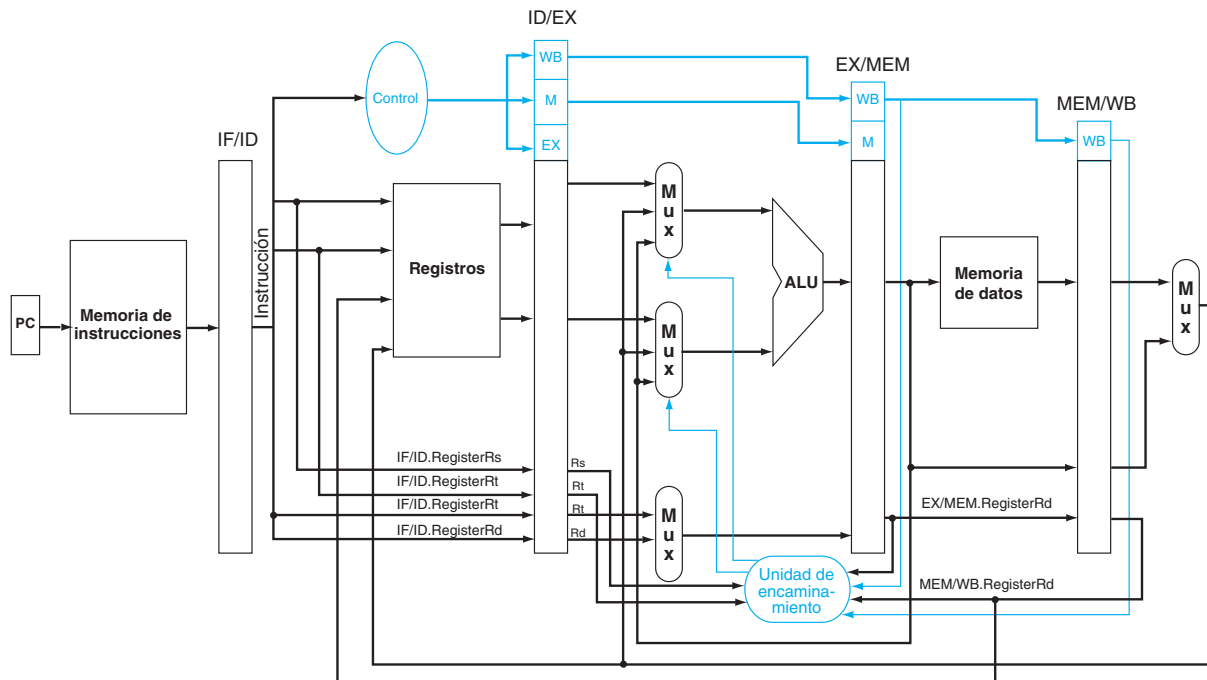


FIGURA 4.56 Camino de datos modificado para resolver los riesgos a través la anticipación de resultados. Comparado con el camino de datos de la figura 4.51, los cambios adicionales son los multiplexores en las entradas de la ALU. Sin embargo, esta figura simplificada deja fuera detalles del camino de datos completo, como por ejemplo el hardware para los saltos y para la extensión de signo.

Extensión: La anticipación de datos también puede ayudar con los riesgos cuando las instrucciones de almacenamiento en memoria dependen de otras instrucciones. En este caso la anticipación es sencilla, ya que las instrucciones sólo usan un valor durante la etapa MEM. Pero considere cargas seguidas inmediatamente de almacenamientos, útiles para realizar copias memoria-a-memoria en la arquitectura MIPS. Se necesita añadir más hardware de anticipación de datos para hacer que se ejecuten más rápido. Si se dibujara otra vez la figura 4.53, reemplazando las instrucciones *sub* y *and* por *lw* y *sw*, se vería que es posible evitar el bloqueo, ya que el dato está en el registro MEM/WB de la instrucción de carga con tiempo para que sea usado en la etapa MEM de la instrucción de almacenamiento. Para esta opción, se necesitaría añadir el hardware de anticipación de datos en la etapa de acceso a memoria. Se deja esta modificación en el camino de datos como ejercicio.

Por otra parte, el valor inmediato con signo extendido que necesitan las cargas y los almacenamientos en la entrada de la ALU no está en el camino de datos de la figura 4.56. Debido a que el control central decide entre el valor del registro o el inmediato, y debido a que la unidad de anticipación de datos elige el registro de segmentación que irá a la entrada de la ALU, la solución más fácil es añadir un multiplexor 2:1 que escoja entre la salida del multiplexor ForwardB y el valor inmediato con signo. La figura 4.57 muestra este cambio adicional.

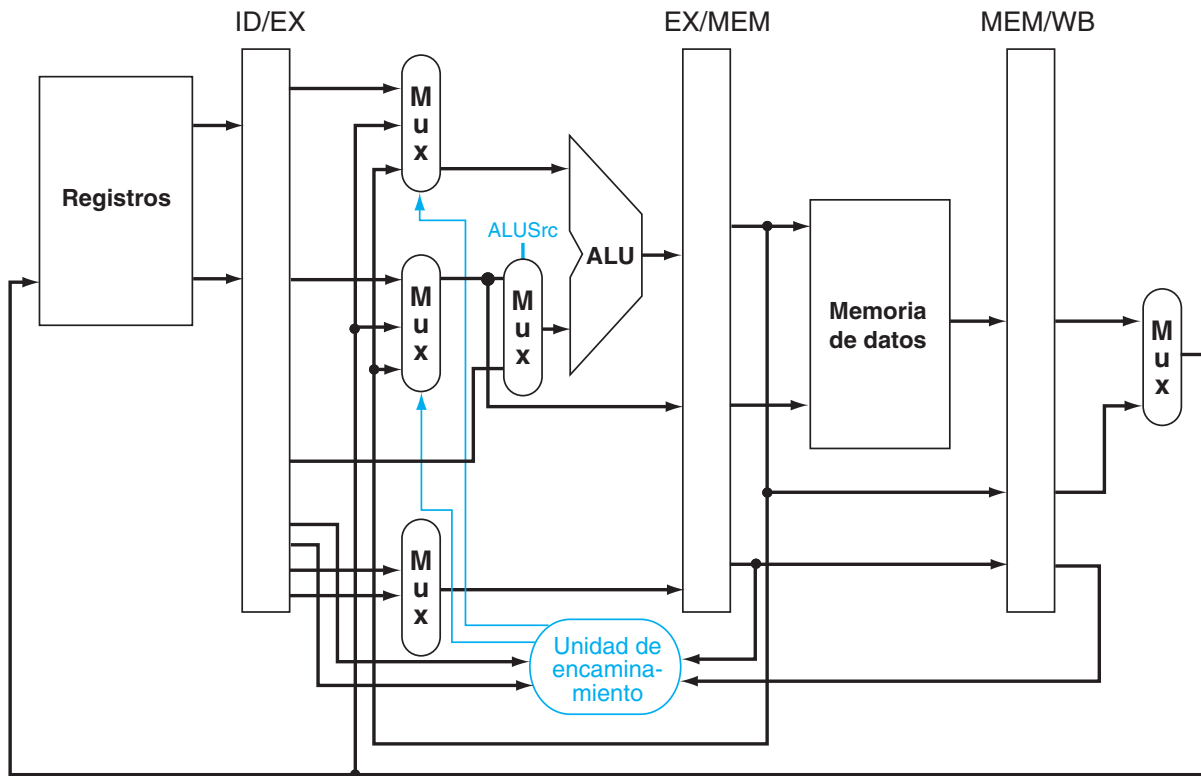


FIGURA 4.57 El zoom del camino de datos de la figura 4.54 muestra un multiplexor 2:1, añadido para seleccionar como entrada de la ALU al valor inmediato con signo.

Si al principio no tienes éxito, redefine el éxito.

Anónimo

Riesgos de datos y bloqueos

Como se ha dicho en la sección 4.5, un caso donde la anticipación de resultados no nos puede resolver la papeleta es cuando una instrucción situada después de una carga intenta leer el registro en el que escribe la carga. La figura 4.58 ilustra el problema. El dato todavía se está leyendo de memoria en el ciclo 4 mientras la ALU está ejecutando la operación de la siguiente instrucción. Es necesario que se bloquee el pipeline para la combinación de una carga seguido de una instrucción que lee su resultado.

Por lo tanto, además de una unidad de anticipación de datos, se necesita una unidad de detección de riesgos. Debe funcionar durante la etapa ID de tal manera que pueda insertar un bloqueo entre el *load* y la instrucción que lo usa. El control para la detección de riesgos cuando se tiene que comprobar una instrucción *load* es la siguiente condición:

```

si (ID/EX.MemRead y
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) o
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    bloquear el pipeline
  
```

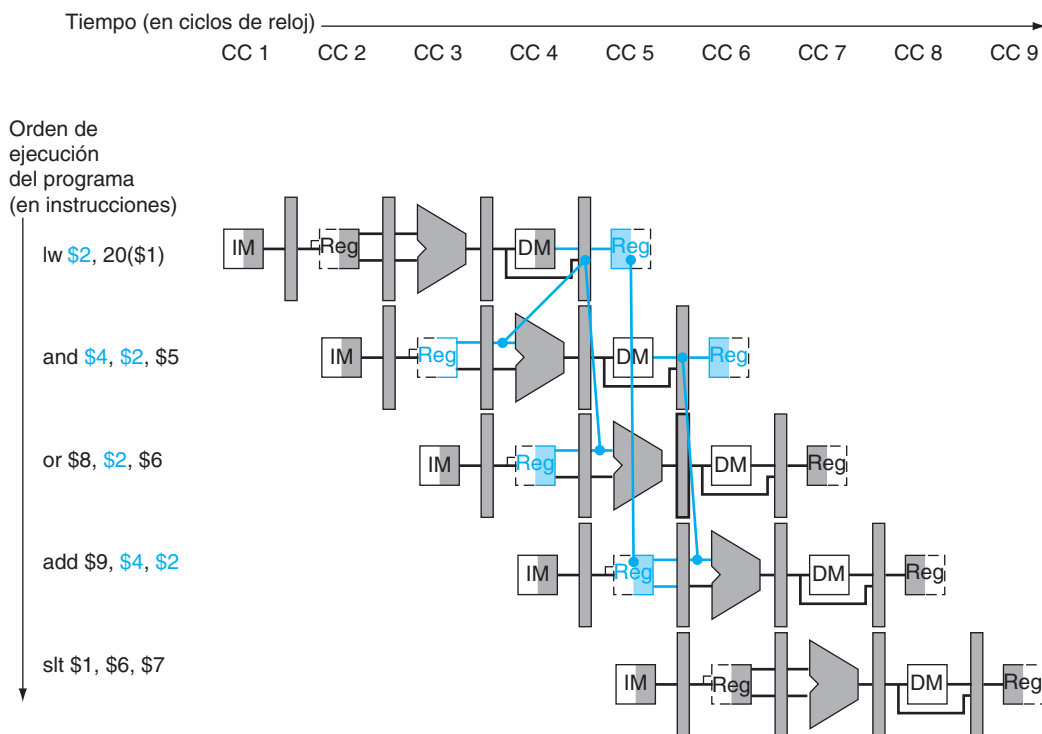


FIGURA 4.58 Secuencia de instrucciones segmentadas. Como la dependencia entre la carga y la instrucción siguiente (*and*) va hacia atrás en el tiempo, este riesgo no puede ser resuelto mediante la anticipación de datos. Por tanto, esta combinación debe resultar en un bloqueo generado por la unidad de detección de riesgos.

La primera línea comprueba si la instrucción es una carga: ésta es la única instrucción que lee de memoria. Las dos líneas siguientes comprueban si el campo de registro destino de la carga en la etapa EX es igual a cualquiera de los dos registros fuente de la instrucción en ID. Si la condición se cumple, la instrucción se bloquea durante 1 ciclo. Después de este bloqueo de un ciclo, la lógica de anticipación de datos puede resolver la dependencia y la ejecución continúa. (Si no hubiera anticipación de datos, entonces las instrucciones en la figura 4.58 tendrían que bloquearse otro ciclo).

Si la instrucción situada en la etapa ID se bloquea, entonces también debe bloquearse la que esté en IF; de no hacerse así se perdería la instrucción buscada de memoria. Evitar que estas dos instrucciones progresen en el procesador supone simplemente evitar que cambien el registro PC y el registro de segmentación IF/ID. Si estos valores no cambian, la instrucción en la etapa IF continuará leyendo de memoria usando el mismo PC y se volverán a leer los registros en la etapa ID usando los mismos campos de instrucción en el registro de segmentación IF/ID. Volviendo a nuestra analogía favorita, es como si la lavadora volviera a empezar con la misma ropa y se dejara a la secadora que diera vueltas vacía. Por supuesto, igual que con la secadora, la mitad posterior de pipeline que comienza en la etapa EX debe hacer alguna cosa. Lo que hace es ejecutar instrucciones que no tienen ningún efecto: **nops**.

¿Cómo se pueden insertar estos nops, que actúan como burbujas en el pipeline? En la figura 4.49 se observa que negando las nueve señales de control (poniéndolas a 0) en las etapas EX/MEM y WB se creará una instrucción “no hacer nada” o nop. Al identificar el riesgo en la etapa ID, se puede insertar una burbuja en el pipeline poniendo a 0 los campos de control de EX, MEM y WB en el registro de segmentación ID/EX. Estos nuevos valores de control se propagan hacia adelante en cada ciclo con el efecto deseado: si los valores de control valen todos 0 no se escribe sobre memoria o sobre registros.

La figura 4.59 muestra lo que realmente sucede en el hardware: el segmento de ejecución asociado a la instrucción `and` se convierte en un nop, y todas las instrucciones a partir de `and` son retrasadas un ciclo. Como una burbuja de aire en una tubería de agua, una burbuja de bloqueo retrasa todo lo que viene detrás, y avanza por el pipeline hasta salir por su extremo final. El riesgo fuerza a que las instrucciones `and` y `or` repitan en el ciclo 4 lo que hicieron en el 3: `and` lee los registros y se decodifica, `or` se busca de nuevo en la memoria de instrucciones. Es realmente esta repetición de trabajo lo que realiza un bloqueo, pero su efecto es alargar el tiempo de las instrucciones `and` y `or`, y retrasar la búsqueda de la instrucción `add`.

La figura 4.60 resalta las conexiones en el pipeline tanto para la unidad de detección de riesgos como para la unidad de anticipación de datos. Como antes, la unidad de anticipación controla los multiplexores de la ALU para reemplazar el valor del registro de propósito general por el valor del registro de segmentación adecuado. La unidad de detección de riesgos controla la escritura sobre los registros PC e IF/ID, además del multiplexor que elige entre los valores de control reales o todo ceros. La unidad de riesgos bloquea y desactiva los campos de control si es cierta la comprobación del riesgo de uso de una carga (*load-use hazard*) antes

nops: instrucción que no realiza ninguna operación para cambiar el estado. Viene de “no operación”.

mencionado. Para más detalles, la [sección 4.12](#) en el CD muestra un fragmento de código MIPS con riesgos que causan bloqueos del pipeline, ilustrado con diagramas monociclo.

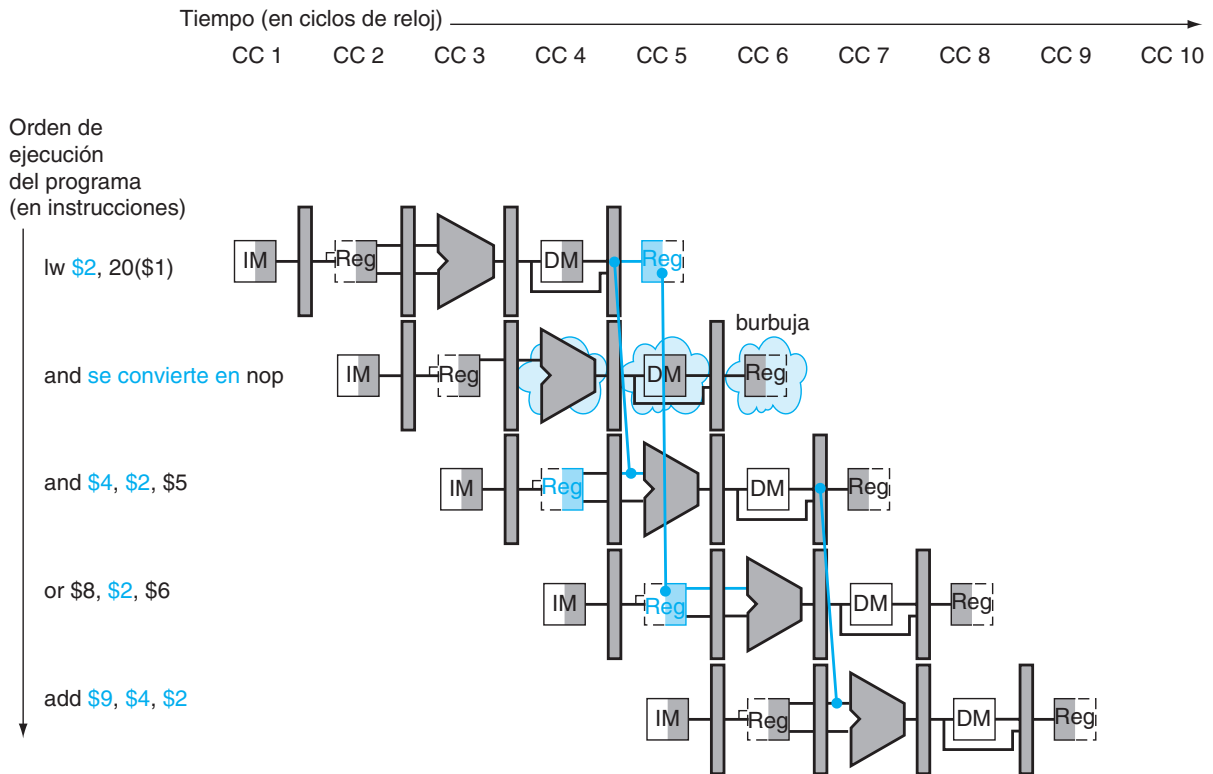


FIGURA 4.59 Como se insertan realmente los bloqueos en el pipeline. Se inserta una burbuja al principio del ciclo 4 cambiando la instrucción **and** por un **nop**. Observe que la instrucción **and** es realmente buscada y descodificada en los ciclos 2 y 3, pero su etapa EX es retrasada hasta el ciclo 5 (en lugar del ciclo 4 cuando no se produce bloqueo). De un modo similar, la instrucción **or** se busca en el ciclo 4, pero su etapa IF es retrasada hasta el ciclo 5 (en lugar del ciclo 4 cuando no se produce bloqueo). Después de insertar la burbuja, todas las dependencias avanzan en el tiempo y no ocurren más riesgos de datos.

IDEA clave

El hardware puede depender del compilador o no para resolver los riesgos de dependencias y asegurar una ejecución correcta. Sea como sea, para conseguir las mejores prestaciones es necesario que el compilador comprenda el pipeline. De lo contrario, los bloqueos inesperados reducirán las prestaciones del código compilado.