

4

*En una materia
fundamental, ningún
detalle es insignificante.*

Proverbio francés

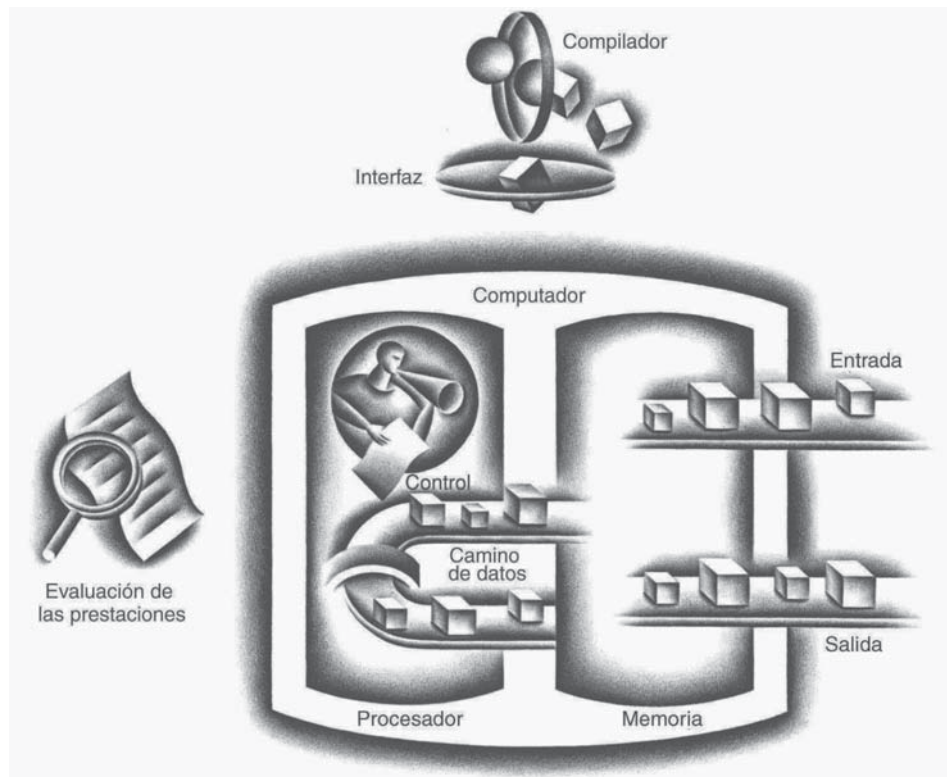
El procesador

- 4.1 Introducción 300**
- 4.2 Convenios de diseño lógico 303**
- 4.3 Construcción de un camino de datos 307**
- 4.4 Esquema de una implementación
simple 316**
- 4.5 Descripción general de la
segmentación 330**
- 4.6 Camino de datos segmentados y control de
la segmentación 344**
- 4.7 Riesgos de datos: anticipación frente a
bloqueos 363**
- 4.8 Riesgos de control 375**
- 4.9 Excepciones 384**

- 4.10 Paralelismo y paralelismo a nivel de instrucciones avanzado 391
- 4.11 Casos reales: El pipeline del AMD Opteron X4 (Barcelona) 404
- 4.12 Tema avanzado: una introducción al diseño digital utilizando un lenguaje de descripción hardware para describir y modelar un pipeline y más figuras sobre segmentación 406
- 4.13 Falacias y errores habituales 407
- 4.14 Conclusiones finales 408
- 4.15 Perspectiva histórica y lecturas recomendadas 409
- 4.16 Ejercicios 409

Nota importante: En la presente edición en castellano, los contenidos del CD incluido en la edición original (en inglés) son accesibles a través de la página web www.reverte.com/microsites/pattersonhennessy. Aunque en la presente edición no se proporciona un CD-ROM físico, a lo largo de todo el texto se menciona el CD y se utiliza el icono que lo representa para hacer referencia a su contenido.

Los cinco componentes clásicos del computador




4.1 Introducción

En el capítulo 1 vimos que las prestaciones de una máquina están determinadas por tres factores clave: el número de instrucciones, el tiempo del ciclo de reloj y los ciclos de reloj por instrucción (*cycles per instruction*, CPI). El compilador y la arquitectura del repertorio de instrucciones, que hemos examinado en el capítulo 2, determinan el número de instrucciones requerido por un cierto programa. Sin embargo, tanto el tiempo de ciclo del reloj como el número de ciclos por instrucción vienen dados por la implementación del procesador. En este capítulo se construye el camino de datos y la unidad de control para dos realizaciones diferentes del repertorio de instrucciones MIPS.

Este capítulo incluye una explicación de los principios y técnicas utilizadas en la implementación de un procesador, comenzando con un resumen altamente abstracto y simplificado en esta sección, seguido por una sección que construye un camino de datos y una versión simple de un procesador suficiente para realizar repertorios de instrucciones como MIPS. El núcleo del capítulo describe una implementación segmentada más realista del MIPS, seguido de una sección que desarrolla los conceptos necesarios para la implementación de un repertorio de instrucciones más complejo, como el x86.

Para el lector interesado en comprender la interpretación a alto nivel de las instrucciones y su impacto en las prestaciones del programa, esta sección inicial y la sección 4.5 proporcionan los conceptos básicos de la segmentación. Las tendencias actuales se indican en la sección 4.10, y la sección 4.11 describe el microprocesador AMD Opteron X4 (Barcelona). Estas secciones proporcionan suficiente bagaje para comprender los conceptos de segmentación a alto nivel.

Las secciones 4.3, 4.4 y 4.6 son útiles para los lectores interesados en entender el procesador y sus prestaciones con mayor profundidad y las secciones 4.2, 4.7, 4.8 y 4.9 para los interesados en como construir un procesador. Para los lectores interesados en diseño hardware moderno, la  [sección 4.12](#) en el CD describe como se utilizan los lenguajes de descripción hardware y las herramientas de CAD para la implementación de hardware y como utilizar un lenguaje de descripción hardware para describir una implementación segmentada. También se incluyen más figuras sobre la ejecución en un hardware segmentado.

Una implementación básica MIPS

Examinaremos una implementación que incluirá un subconjunto básico del repertorio de instrucciones del MIPS formado por:

- Las instrucciones de referencia a memoria cargar palabra (*load word lw*) y almacenar palabra (*store word sw*).
- Las instrucciones aritmético-lógicas *add*, *sub*, *and*, *or* y *sllt*.
- Las instrucciones de saltar si igual (*branch on equal beq*) y salto incondicional (*jump j*), que se añadirán en último lugar.

Este subconjunto no incluye todas las instrucciones con enteros (por ejemplo, se han omitido las de desplazamiento, multiplicación y división), ni ninguna de las instrucciones de punto flotante. Sin embargo, servirá para ilustrar los principios básicos que se utilizan en la construcción del camino de datos y el diseño de la unidad de control. La implementación de las instrucciones restantes es similar.

Al examinar la implementación, tendremos la oportunidad de ver cómo la arquitectura del repertorio de instrucciones determina muchos aspectos de la implementación y cómo la elección de varias estrategias de implementación afecta a la frecuencia del reloj y al CPI de la máquina. Muchos de los principios básicos de diseño introducidos en el capítulo 1, como las directrices «hacer rápido el caso común» y «la simplicidad favorece la regularidad», pueden observarse en la implementación. Además, la mayoría de los conceptos utilizados para realizar el subconjunto MIPS en este capítulo y en el siguiente son las ideas básicas que se utilizan para construir un amplio espectro de computadores, desde servidores de altas prestaciones hasta microprocesadores de propósito general o procesadores empotrados.

Una visión general de la implementación

En el capítulo 2 analizamos el núcleo básico de instrucciones MIPS, incluidas las instrucciones aritmético-lógicas sobre enteros, las instrucciones de referencia a memoria y las instrucciones de salto. La mayor parte de lo necesario para implementar estas instrucciones es común para todas ellas, independientemente de su tipo concreto. Para cada instrucción, los dos primeros pasos son idénticos:

1. Enviar el contador de programa (PC) a la memoria que contiene el código y cargar la instrucción desde esa memoria.
2. Leer uno o dos registros, utilizando para ello los campos específicos de la instrucción para seleccionar los registros a leer. Para la instrucción de cargar palabra es necesario un solo registro, pero la mayoría del resto de las instrucciones requiere la lectura de dos registros.

Después de estos dos pasos, las acciones necesarias para completar la instrucción dependen del tipo de la misma. Afortunadamente, para los tres tipos de instrucciones (referencia a memoria, aritmético-lógicas y saltos) las acciones son generalmente las mismas, independientemente del código de operación exacto. La sencillez y regularidad del repertorio de instrucciones MIPS simplifica la implementación porque la ejecución de muchas clases de instrucciones es similar.

Por ejemplo, todos los tipos de instrucciones, excepto las de salto incondicional, utilizan la unidad aritmético-lógica (ALU) después de la lectura de los registros. Las instrucciones de referencia a memoria utilizan la ALU para el cálculo de la dirección, las instrucciones aritmético-lógicas para ejecutar la operación, y las de salto condicional para comparar. Después de utilizar la ALU, las acciones requeridas para completar la ejecución de los diferentes tipos de instrucciones son distintas. Una instrucción de referencia a memoria necesitará acceder a memoria, ya sea para escribir el dato en una operación de almacenamiento o para leer, en caso de una carga. Una instrucción aritmético-lógica o una de carga debe escribir el resultado calculado por la ALU o el leído de la memoria en un registro. Finalmente, en una instrucción de salto condicional es necesario modificar la dirección de la siguiente

instrucción según el resultado de la comparación; en caso contrario el PC debe incrementarse en 4 para obtener la dirección de la siguiente instrucción.

La figura 4.1 muestra un esquema de alto nivel de una implementación MIPS, en el que se muestran las diferentes unidades funcionales y su interconexión. A pesar de que esta figura muestra la mayor parte del flujo de datos en el procesador, omite dos aspectos importantes de la ejecución de las instrucciones.

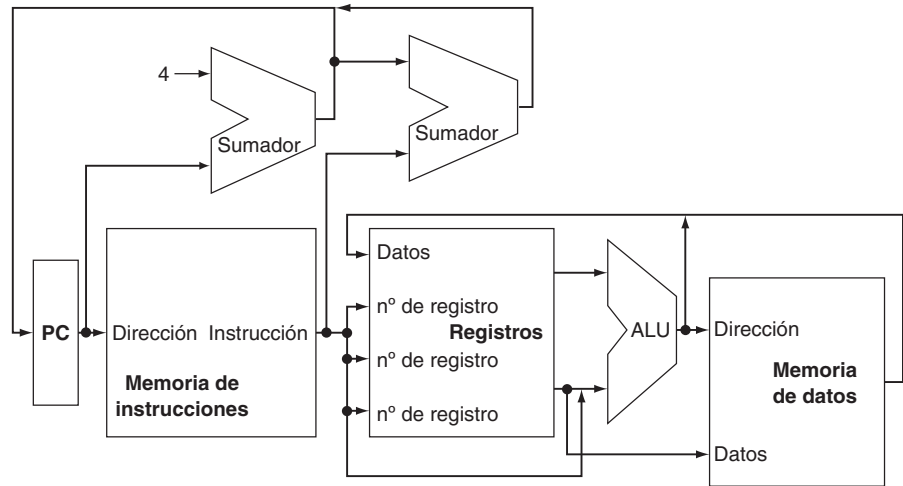




FIGURA 4.1 Una visión abstracta de la implementación del subconjunto MIPS en la que se muestra la mayor parte de las unidades funcionales y las conexiones entre ellas.

Todas las instrucciones comienzan utilizando el contador de programa (PC) para proporcionar la dirección de la instrucción en la memoria de instrucciones. Tras la captación de la instrucción, ciertos campos de ésta especifican los registros que se utilizan como operandos fuente. Una vez que éstos han sido leídos, puede operarse con ellos, ya sea para calcular una dirección de memoria (en una carga o un almacenamiento), para calcular un resultado aritmético (para una instrucción aritmético-lógica entera), o bien para realizar una comparación (en un salto condicional). Si la instrucción es una instrucción aritmético-lógica, el resultado de la ALU debe almacenarse en un registro. Si la operación es una carga o un almacenamiento, el resultado de la ALU se utiliza como la dirección donde almacenar un valor en memoria o cargar un valor en los registros. El resultado de la ALU o memoria se escribe en el banco de registros. Los saltos condicionales requieren el uso de la salida de la ALU para determinar la dirección de la siguiente instrucción, que proviene de la ALU (donde se suma el PC y el desplazamiento) o desde un sumador que incrementa el valor actual del PC en 4. Las líneas gruesas que interconectan las unidades funcionales representan los buses, que consisten en múltiples señales. Las flechas se utilizan para indicar al lector cómo fluye la información. Ya que las líneas de señal pueden cruzarse, se muestra explícitamente la conexión con la presencia de un punto donde se cruzan las líneas.

En primer lugar, en varios puntos, la figura muestra datos dirigidos a una unidad particular provenientes de dos orígenes diferentes. Por ejemplo, el valor escrito en el PC puede provenir de cualquiera de los dos sumadores, el dato escrito en el banco de registros puede provenir de la ALU o de la memoria de datos, y la segunda entrada de la ALU proviene de un registro o del campo inmediato de la instrucción. En la práctica, estas líneas de datos no pueden conectarse directamente; debe añadirse un elemento que seleccione entre los múltiples orígenes y dirija una de estas fuentes al destino. Esta selección se realiza comúnmente mediante un dispositivo denominado *multiplexor*, aunque sería más adecuado

denominarlo *selector de datos*. El multiplexor, que se describe en detalle en el  **apéndice C**, selecciona una de entre varias entradas según la configuración de sus líneas de control. Las líneas de control se configuran principalmente a partir de información tomada de la instrucción en ejecución.

En segundo lugar, varias de las unidades deben controlarse dependiendo del tipo de instrucción. Por ejemplo, la memoria de datos debe leer en una carga y escribir en un almacenamiento. Debe escribirse en el banco de registros en una instrucción de carga y en una instrucción aritmético-lógica. Y, por supuesto, la ALU debe realizar una entre varias operaciones, tal como se mostró en el capítulo 2. (El  **apéndice C** describe el diseño lógico detallado de la ALU.) Al igual que los multiplexores, estas operaciones son dirigidas por las líneas de control que se establecen según los diversos campos de la instrucción.

La figura 4.2 muestra el camino de datos de la figura 4.1 con los tres multiplexores necesarios añadidos, así como las líneas de control para las principales unidades funcionales. Para determinar la activación de las líneas de control de las unidades funcionales y de los multiplexores se utiliza una unidad de control que toma la instrucción como entrada. El tercer multiplexor, que determina si $PC + 4$ o la dirección destino del salto se escribe en el PC, se activa según el valor de la salida Cero de la ALU, que se utiliza para realizar la comparación de la instrucción *beq*. La regularidad y simplicidad del repertorio de instrucciones MIPS implica que un simple proceso de descodificación puede ser utilizado para determinar cómo activar las líneas de control.

En el resto de este capítulo, se refina este esquema para añadir los detalles, lo que va a requerir que se incluyan unidades funcionales adicionales, incrementar el número de conexiones entre las unidades y, por supuesto, añadir una unidad de control que determine las acciones que deben realizarse para cada uno de los distintos tipos de instrucciones. Las secciones 4.3 y 4.4 describen una realización simple que utiliza un único ciclo de reloj largo para cada instrucción y sigue la forma general de las figuras 4.1 y 4.2. En este primer diseño, cada instrucción inicia su ejecución en un flanco de reloj y completa la ejecución en el siguiente flanco de reloj.

Aunque es más fácil de comprender, este enfoque no es práctico, puesto que hay que alargar el ciclo de la señal de reloj para permitir la ejecución de la instrucción más lenta. Una vez diseñado el control para este computador sencillo, se analizará la implementación segmentada en toda su complejidad, incluyendo el tratamiento de las excepciones.

¿Cuántos de los cinco componentes clásicos de un computador, mostrados en la página 299, se incluyen en la figuras 4.1 y 4.2?

Autoevaluación

4.2

Convenios de diseño lógico

Para tratar el diseño de la máquina, se debe decidir cómo operará su implementación lógica y cómo será su sincronización. Esta sección repasa unas cuantas ideas clave de diseño lógico que se utilizarán ampliamente en este capítulo. Si se tiene

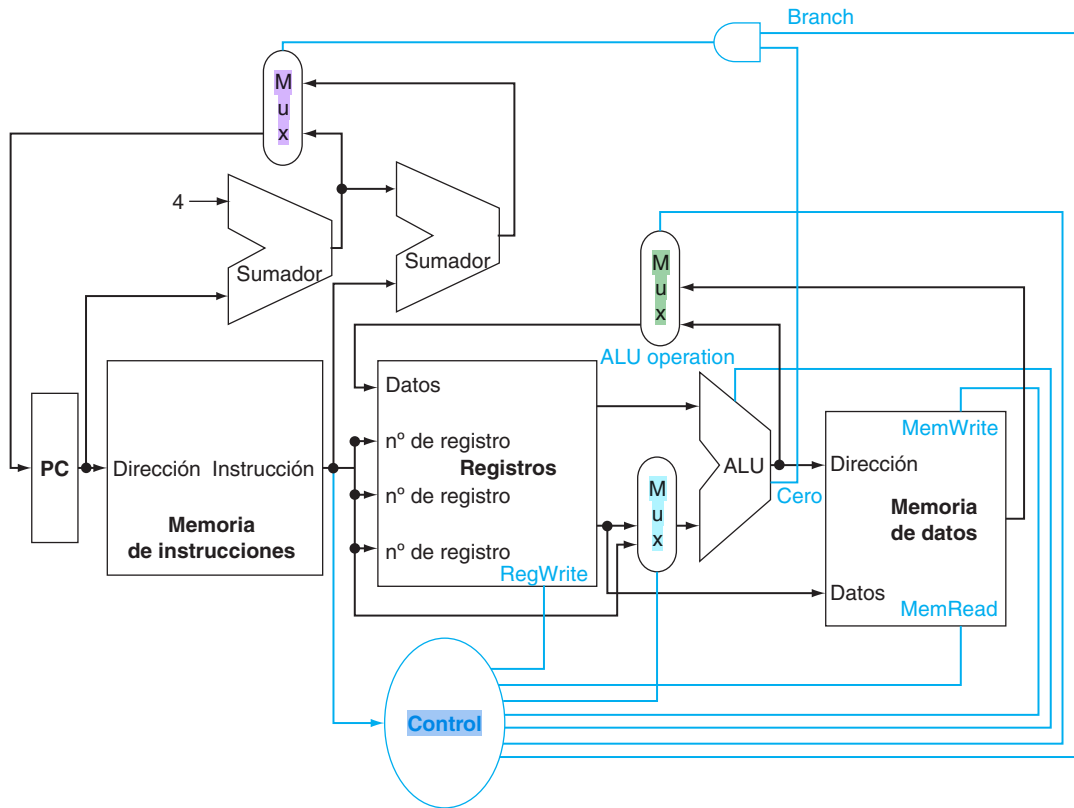



FIGURA 4.2 Implementación básica del subconjunto del MIPS incluyendo los multiplexores y líneas de control necesarias. El multiplexor de la parte superior controla el valor que se va a cargar en el PC ($PC + 4$ o la dirección destino del salto condicional); el multiplexor es controlado por una puerta que realiza una función «and» entre la salida Cero de la ALU y una señal de control que indica que la instrucción es de salto condicional. El multiplexor cuya salida está conectada al banco de registros se utiliza para seleccionar la salida de la ALU (en el caso de una instrucción aritmético-lógica) o la salida de la memoria de datos (en el caso de una carga desde memoria) para escribir en el banco de registros. Por último, el multiplexor de la parte inferior determina si la segunda entrada de la ALU procede de los registros (en una instrucción aritmético-lógica no inmediata) o del campo de desplazamiento de la instrucción (en una operación inmediata, una carga o almacenamiento, o un salto condicional). Las líneas de control añadidas son directas y determinan la operación realizada por la ALU, si se debe leer o escribir en la memoria de datos, y si los registros deben realizar una operación de escritura. Las líneas de control se muestran en color para facilitar su identificación.


escaso o ningún conocimiento sobre diseño lógico, el **apéndice C** resultará muy útil antes de abordar esta sección.

Las unidades funcionales de la implementación MIPS constan de dos tipos de elementos lógicos diferentes: elementos que operan con datos y elementos que contienen el estado. Los elementos que operan con datos son todos **combinacionales**, lo que significa que sus salidas dependen únicamente de los valores actuales de las entradas. Para una misma entrada, un elemento combinacional siempre produce la misma salida. La ALU mostrada en la figura 4.1 y analizada en detalle en el **apéndice C** es un elemento combinacional. Para un conjunto de entradas, siempre produce la misma salida porque no tiene almacenamiento interno.

Elemento combinacional: un elemento operacional tal como una puerta AND o una ALU.

Otros elementos en el diseño no son combinacionales, sino que contienen *estado*. Un elemento contiene estado si tiene almacenamiento interno. Estos elementos se denominan **elementos de estado** porque, si se apaga la máquina, se puede reiniciar cargando dichos elementos con los valores que contenían antes de apagarla. Además, si se guardan y se restauran, es como si la máquina no se hubiera apagado nunca. Así, los elementos de estado caracterizan completamente la máquina. En la figura 4.1, las memorias de instrucciones y datos, así como los registros, son ejemplos de elementos de estado.

Un elemento de estado tiene al menos dos entradas y una salida. Las entradas necesarias son el valor del dato a almacenar en el elemento de estado y el reloj, que determina cuándo se almacena el dato. La salida del elemento de estado proporciona el valor que se almacenó en un ciclo de reloj anterior. Por ejemplo, uno de los elementos de estado más simple es un biestable de tipo D (véase el  **apéndice C**), el cual tiene exactamente estas dos entradas (un dato y un reloj) y una salida. Además de los biestables, la implementación MIPS también utiliza otros dos tipos de elementos de estado: memorias y registros, que aparecen en la figura 4.1. El reloj se utiliza para determinar cuándo debería escribirse el elemento de estado. Un elemento de estado se puede leer en cualquier momento.

Los componentes lógicos que contienen estado también se denominan *secuenciales* porque sus salidas dependen tanto de sus entradas como del contenido de su estado interno. Por ejemplo, la salida de la unidad funcional que representa a los registros depende del identificador del registro y de lo que se haya almacenado en los registros anteriormente. La operación tanto de los elementos combinacionales como secuenciales, así como su construcción, se analiza con más detalle en el  **apéndice C**.

Se utiliza la palabra **activado** para indicar que una señal se encuentra lógicamente a alta y *activar* para especificar que una señal debe ser puesta a alta, mientras que *desactivar* o **desactivado** representan un valor lógico a baja.

Metodología de sincronización

Una **metodología de sincronización** define cuándo pueden leerse y escribirse las diferentes señales. Es importante especificar la temporización de las lecturas y las escrituras porque, si una señal se escribe en el mismo instante en que es leída, el valor leído puede corresponder al valor antiguo, al valor nuevo o ¡incluso a una combinación de ambos! No es necesario decir que los diseños de computadores no pueden tolerar tal imprevisibilidad. La metodología de sincronización se diseña para prevenir esta circunstancia.

Por simplicidad, supondremos una metodología de **sincronización por flanco**. Esta metodología de sincronización implica que cualquier valor almacenado en un elemento lógico secuencial se actualiza únicamente en un flanco de reloj. Debido a que sólo los elementos de estado pueden almacenar datos, las entradas de cualquier lógica combinatorial deben proceder de elementos de este tipo y las salidas también deben dirigirse hacia elementos de este tipo. Las entradas serán valores escritos en un ciclo anterior de reloj, mientras que las salidas son valores que pueden utilizarse en el ciclo siguiente.

Elemento de estado:
elemento de memoria.

Activado: la señal está al nivel lógico alto o verdadero.

Desactivado: la señal está al nivel lógico bajo o falso.

Metodología de sincronización: aproximación que determina cuándo los datos son válidos y estables utilizando como referencia el reloj.

Sincronización por flanco: esquema de sincronización en el cual todos los cambios de estado se producen en los flancos de reloj.

La figura 4.3 muestra dos elementos de estado que rodean a un bloque de lógica combinacional, que opera con un único ciclo de reloj. Todas las señales deben propagarse desde el elemento de estado 1, a través de la lógica combinacional hasta el elemento de estado 2 en un único ciclo de reloj. El tiempo necesario para que las señales alcancen el elemento de estado 2 define la duración del ciclo de reloj.

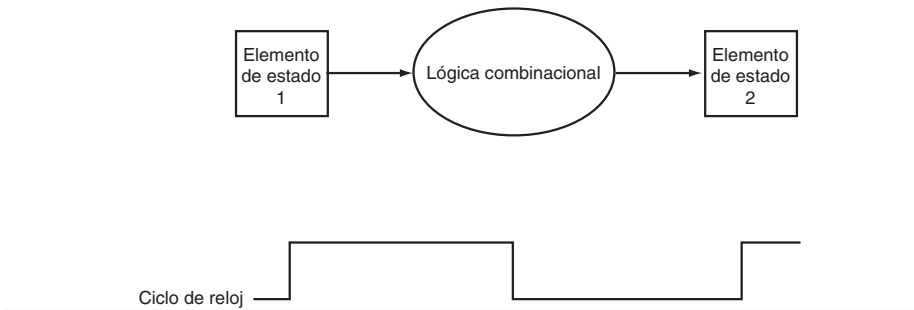


FIGURA 4.3 La lógica combinacional, los elementos de estado y el reloj están estrechamente relacionados. En un sistema digital síncrono, el reloj determina cuándo los elementos con estado van a escribir valores en el almacenamiento interno. Cualquiera de las entradas de un elemento de estado debe alcanzar un valor estable (es decir, alcanzar un valor que no va a cambiar hasta el siguiente flanco del reloj) antes de que el flanco de reloj activo cause la actualización del estado. Se supone que todos estos elementos, incluida la memoria, están sincronizados por flanco.

Señal de control: señal utilizada como selección en un multiplexor o para dirigir la operación de una unidad funcional; contrasta con una **señal de datos**, que contiene información sobre la que opera una unidad funcional.

Por simplicidad, no se muestra una **señal de control** de escritura cuando se escribe en un elemento de estado en cada flanco activo de reloj. En cambio, si el elemento de estado no se actualiza en cada ciclo de reloj, es necesario incluir una señal de control de escritura (*write*). La señal de reloj y la señal de control de escritura son entradas, y el elemento de estado se actualiza únicamente cuando la señal de control de escritura se activa y ocurre un flanco de reloj.

La metodología por flanco permite leer el contenido de un registro, enviar el valor a través de alguna lógica combinacional, y escribir en ese mismo registro en un mismo ciclo de reloj, tal como se muestra en la figura 4.4. Es indiferente donde se asuma la implementación de las escrituras, ya sea en el flanco ascendente o en el descendente, ya que las entradas del bloque combinacional sólo pueden modifi-

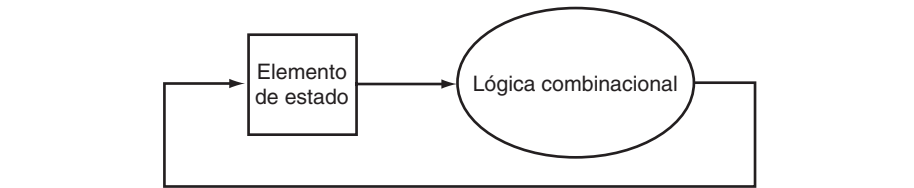



FIGURA 4.4 La metodología de sincronización por flanco permite leer y escribir en un elemento de estado en un mismo ciclo de reloj sin crear una condición de carrera que pueda conducir a valores indeterminados de los datos. Por supuesto, el ciclo de reloj debe ser suficientemente largo para que los valores de entrada sean estables cuando ocurra el flanco de reloj activo. El baipás no puede darse dentro de un mismo ciclo debido a la actualización por flanco del elemento de estado. Si fuera posible el baipás, puede que este diseño no funcione correctamente. Los diseños de este capítulo como los del próximo se basan en este tipo de metodología de sincronización y en estructuras como la mostrada en la figura.

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.

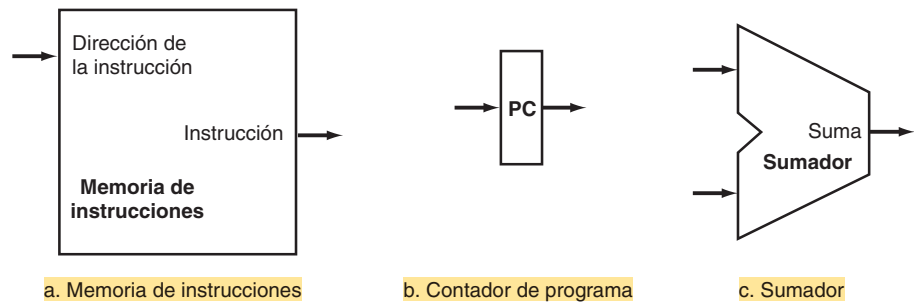


FIGURA 4.5 Se necesitan dos elementos de estado para almacenar y acceder a las instrucciones, y un sumador para calcular la dirección de la instrucción siguiente. Los elementos de estado son la memoria de instrucciones y el contador de programa. La memoria de instrucciones es solo de lectura, ya que el camino de datos nunca escribe instrucciones, y se trata como un elemento de lógica combinacional. La salida en cualquier instante refleja el contenido de la localización especificada por la dirección de entrada, y no se necesita ninguna señal de control de lectura (sólo se necesitará escribir en la memoria de las instrucciones cuando se cargue el programa; esto no es difícil de añadir, por lo que se ignora para simplificar). El contador de programa es un registro de 32 bits que se modifica al final de cada ciclo de reloj y, de esta manera, no necesita ninguna señal de control de escritura. El sumador es una ALU cableada para que sume siempre dos entradas de 32 bits y dé el resultado en su salida.

queta sumador como en la figura 4.5 para indicar que es un sumador permanente y que no puede realizar ninguna otra función propia de una ALU.

Para ejecutar cualquier instrucción se debe empezar por cargar la instrucción desde memoria. Para poder ejecutar la siguiente instrucción se debe incrementar el contador de programa para que apunte hacia ella 4 bytes más allá. La figura 4.6 muestra cómo se combinan los tres elementos de la figura 4.5 para formar un camino de datos que busca instrucciones e incrementa el PC para obtener la dirección de la siguiente instrucción secuencial.

Ahora considérense las instrucciones tipo R (véase la figura 2.20 de la página 136). Todas ellas leen dos registros, operan con la ALU los contenidos de dichos registros como operandos, y escriben el resultado. Estas instrucciones se denominan de tipo R o aritmético-lógicas (ya que realizan operaciones aritméticas o lógicas). En este tipo de instrucciones se incluyen las instrucciones `add`, `sub`, `and`, `or` y `slt`, que fueron introducidas en el capítulo 2. Recuérdese también que el ejemplo típico de este tipo de instrucciones es `add $t1, $t2, $t3`, que lee `$t2` y `$t3` y escribe en `$t1`.

Los registros de 32 bits del procesador se agrupan en una estructura denominada **banco de registros**. Un banco de registros es una colección de registros donde cualquier registro puede leerse o escribirse especificando su número. El banco de registros contiene el estado de los registros de la máquina. Además, se necesita una ALU para poder operar con los valores leídos de los registros.

Debido a que las instrucciones tipo R tienen tres operandos registro, por cada instrucción se necesita leer dos datos del banco y escribir uno en él. Por cada registro que se lee, se necesita una entrada en el banco donde se especifique el número de registro que se quiere leer, así como una salida del banco donde se

Banco de registros:

elemento de estado que consiste en un conjunto de registros que pueden ser leídos y escritos proporcionando el identificador del registro al que se desea acceder.

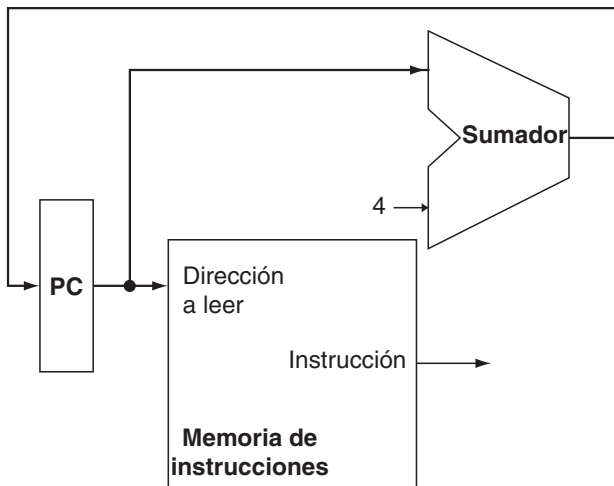


FIGURA 4.6 Parte del camino de datos utilizado para la búsqueda de las instrucciones y el incremento del contador de programa. La instrucción cargada se utiliza en otras partes del camino de datos.

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.

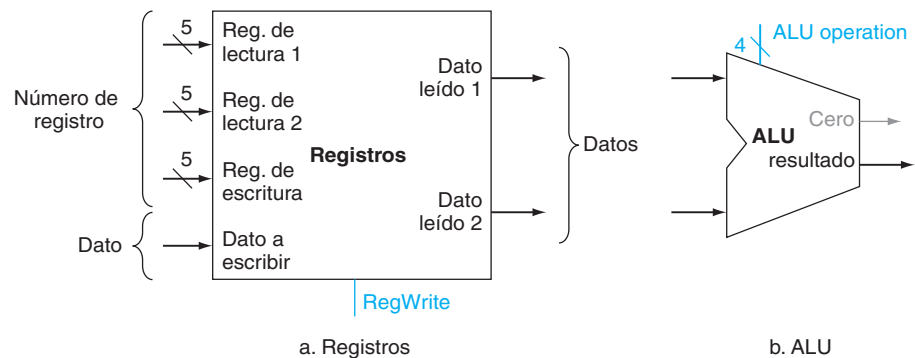


FIGURA 4.7 El banco de registros y la ALU son los dos elementos necesarios para la implementación de instrucciones de tipo R. El banco de registros contiene todos los registros y tiene dos puertos de lectura y uno de escritura. El diseño de los bancos de registros multipuerto se analiza en la sección 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.

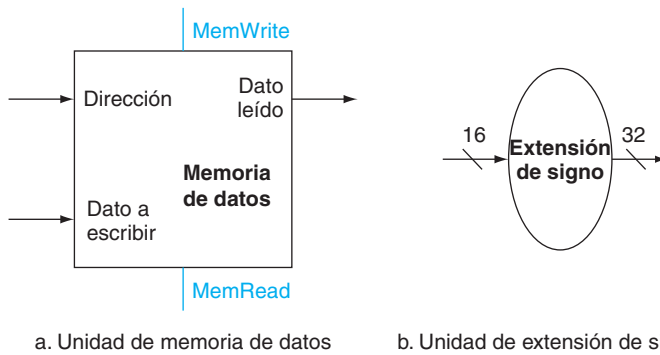


FIGURA 4.8 Las dos unidades necesarias para la implementación de cargas y almacenamientos, además del banco de registros y la ALU de la figura 4.7, son la unidad de memoria de datos y la unidad de extensión de signo. La unidad de memoria es un elemento de estado que tiene como entradas la dirección y 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.

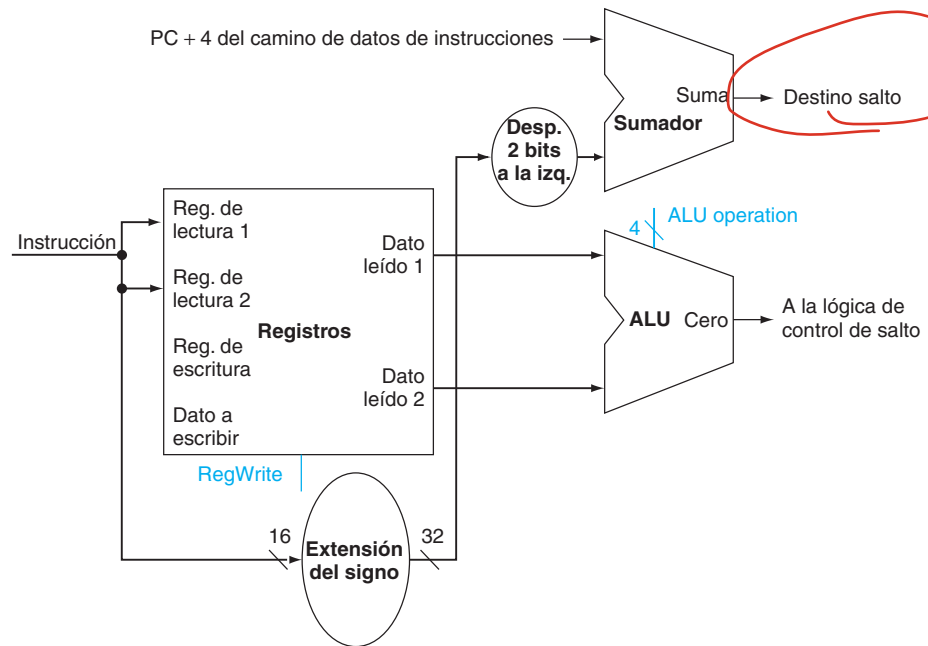


FIGURA 4.9 El camino de datos para un salto condicional utiliza la ALU para evaluar la condición de salto y un sumador aparte para calcular la dirección destino del salto como la suma del PC incrementado y los 16 bits de menor peso de la instrucción con el signo extendido (el desplazamiento de salto) 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.

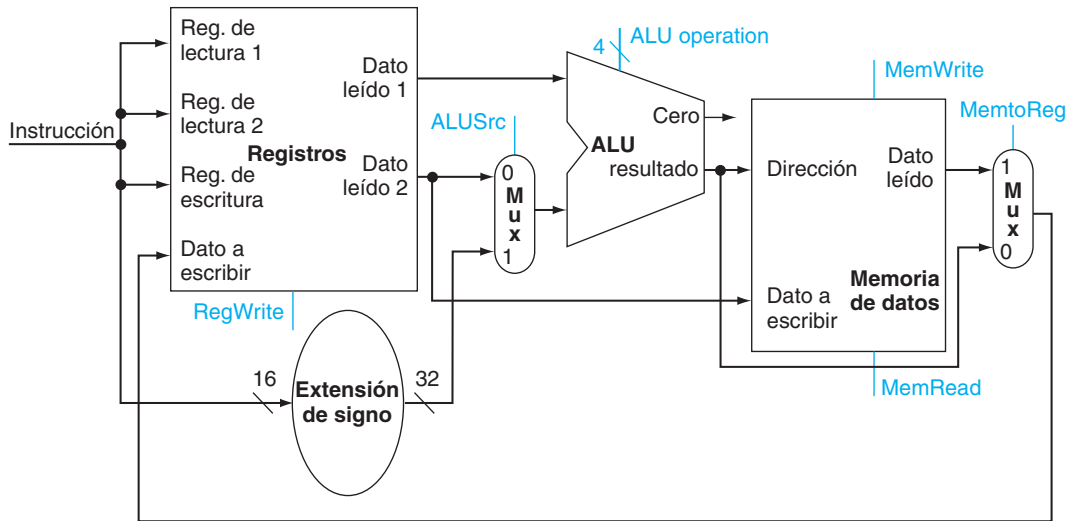


FIGURA 4.10 El camino de datos para las instrucciones de memoria y tipo R. Este ejemplo muestra cómo puede obtenerse un camino uniendo diferentes piezas (figuras 4.7 y 4.8) y añadiendo multiplexores. Se requieren dos multiplexores tal como se describe en el ejemplo.

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.

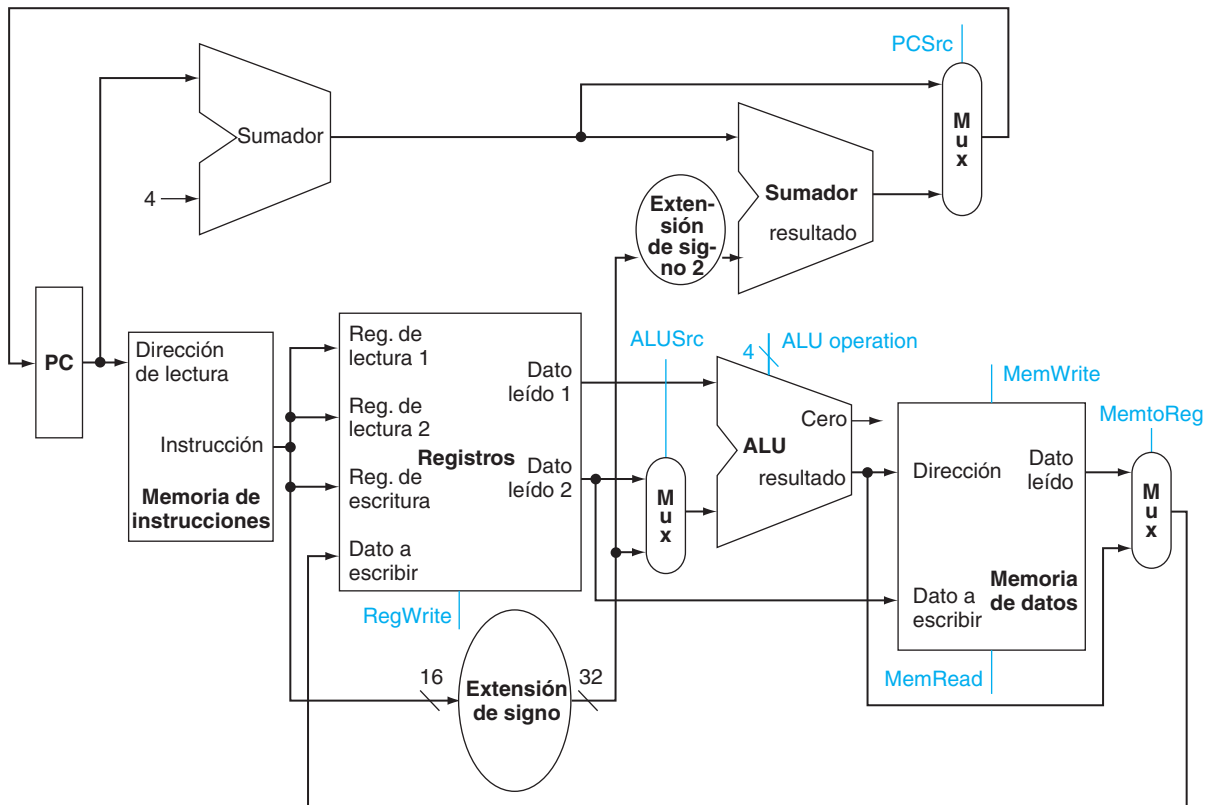


FIGURA 4.11 Un camino de datos sencillo para la arquitectura MIPS que combina los elementos necesarios para los diferentes tipos de instrucciones. Este camino de datos puede ejecutar las instrucciones básicas (carga/almacenamiento de palabras, operaciones de la ALU y saltos) en un solo ciclo de reloj. 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

- a. 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 acceso 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 `sllt`) 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 `sllt`) 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.

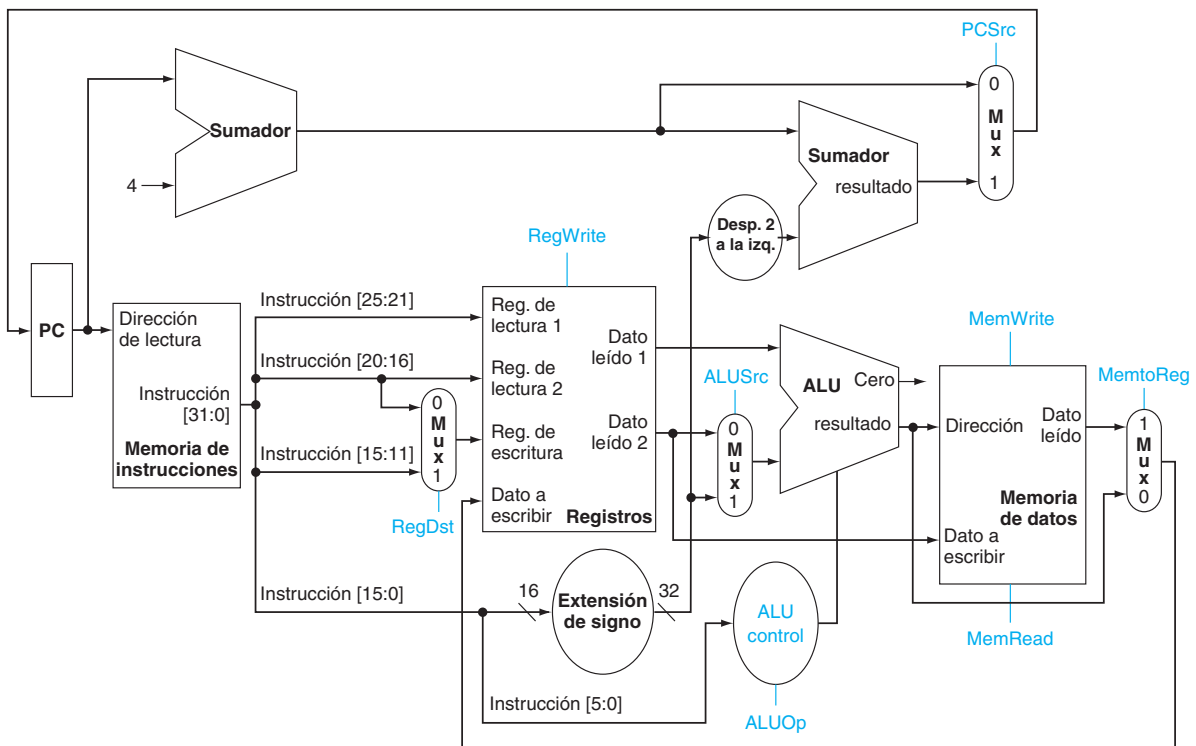


FIGURA 4.15 El camino de datos de la figura 4.12 con todos los multiplexores necesarios y todas las líneas de control identificadas. Las líneas de control se muestran en color. También se ha añadido el bloque encargado de controlar la ALU. El registro PC no necesita un control de escritura ya que sólo 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

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.

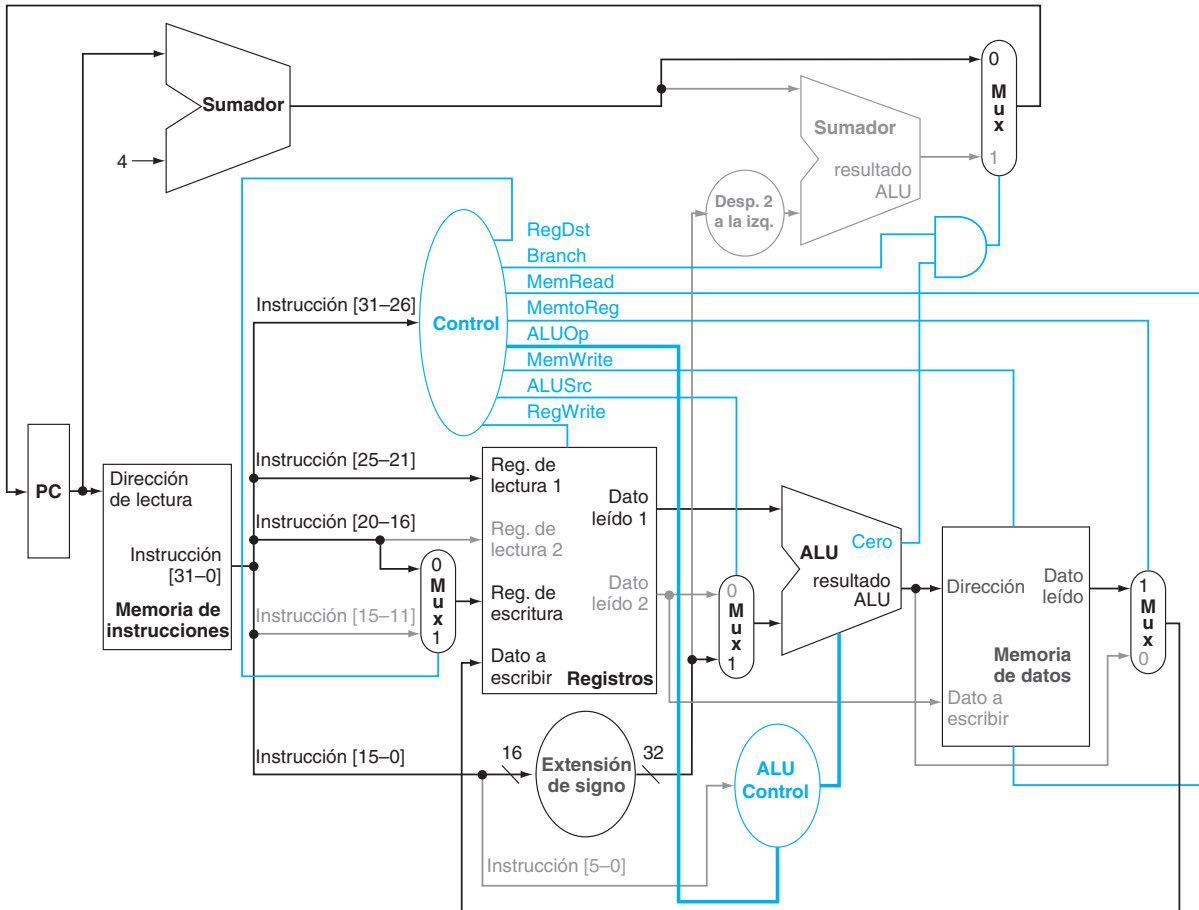


FIGURA 4.20 Funcionamiento del camino de datos para una instrucción de carga. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas. Una instrucción de almacenamiento opera de forma similar. La diferencia principal es que el control de la memoria indicará 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.

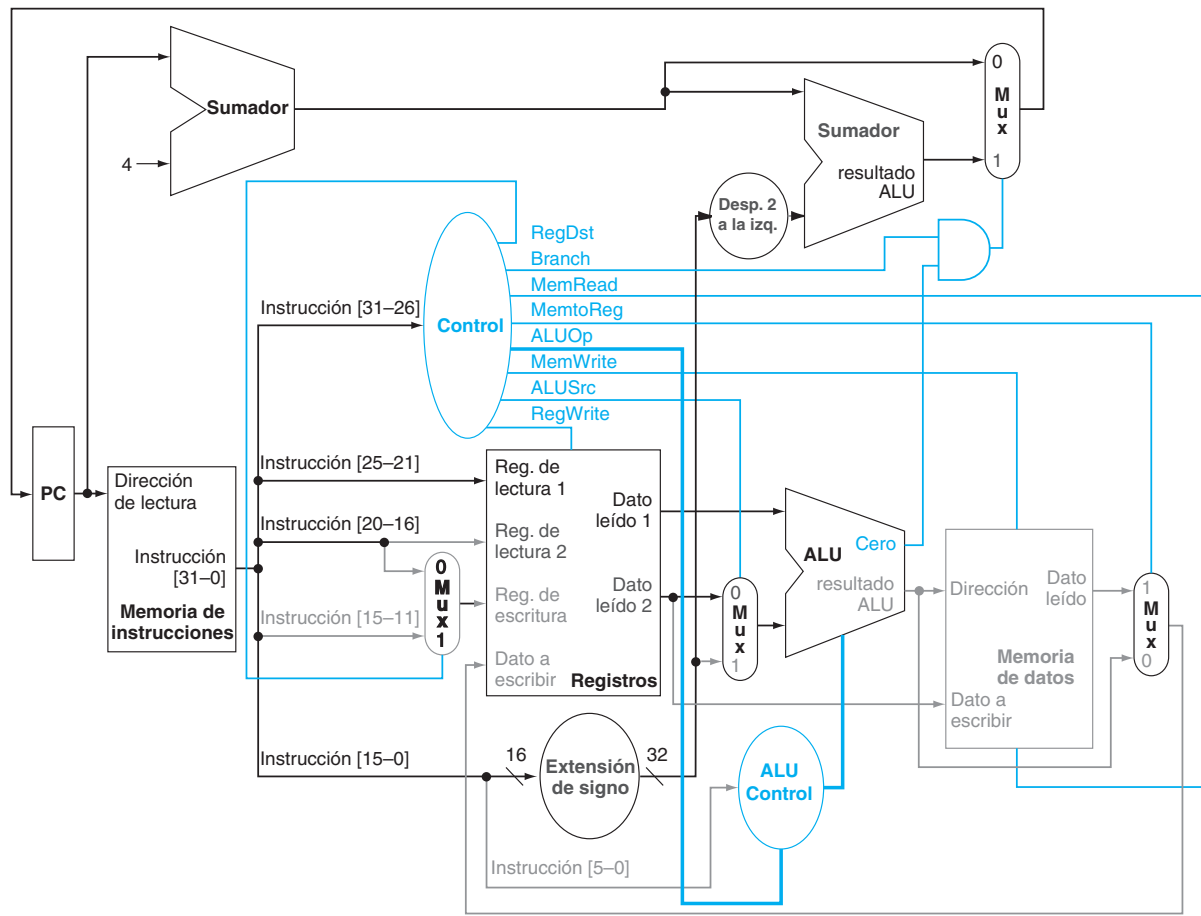


FIGURA 4.21 Funcionamiento del camino de datos para una instrucción de salto si igual. Las líneas de control, las unidades del camino de datos y las conexiones activas se muestran resaltadas. Después de usar el banco de registros y la ALU para realizar la comparación, la salida Cero se utiliza para seleccionar 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 $\text{Op5} \cdot \text{Op2}$, 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

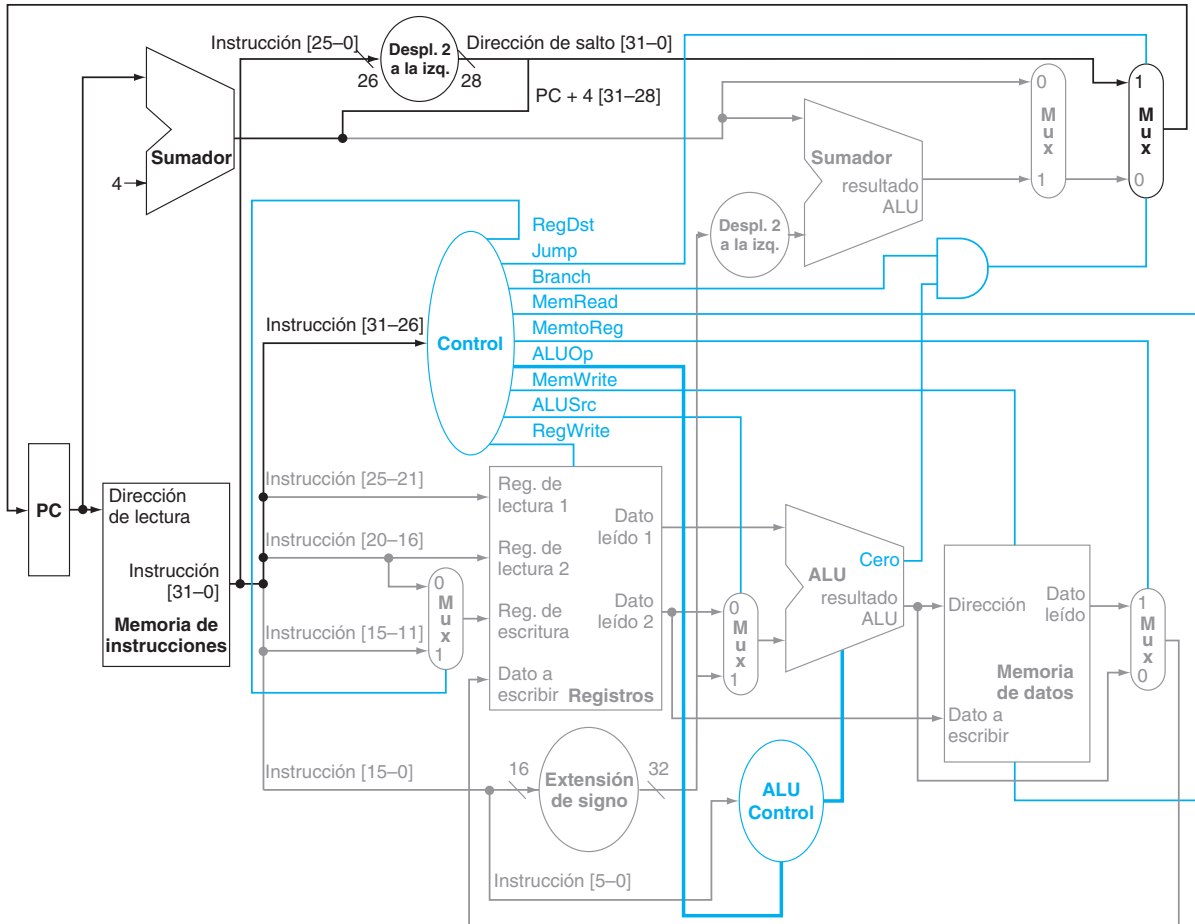


FIGURA 4.24 Un camino de datos sencillo y su control extendido para ejecutar instrucciones de salto incondicional. Se utiliza un multiplexor adicional (en la esquina superior derecha) para elegir entre el destino de la instrucción de salto incondicional y la dirección salto condicional o la siguiente en orden 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

segunda colada de ropa sucia. Cuando la primera colada esté seca, se pone encima de la mesa para empezar a doblarla, se pasa la colada mojada de la lavadora a la secadora, y se mete en la lavadora la tercera colada de ropa sucia. Después, mientras el compañero de piso se lleva la ropa doblada, se empieza a doblar la segunda colada al tiempo que la tercera colada se pasa de la lavadora a la secadora y se introduce la cuarta colada en la lavadora. Llegados a este punto todos los pasos – denominados *etapas* de segmentación (o segmentos)– se llevan a cabo de forma concurrente. Se podrán segmentar las tareas siempre y cuando se disponga de recursos separados para cada etapa.

La paradoja de la segmentación es que el tiempo desde que se pone un calcetín sucio en la lavadora hasta que se seca, se dobla y se guarda no es más corto al utilizar la segmentación; la razón por la cual la segmentación es más rápida para varias coladas es que todas las etapas se llevan a cabo en paralelo, y por tanto se completan más coladas por hora. La segmentación mejora la productividad de la lavandería sin mejorar el tiempo necesario para completar una única colada. Consecuentemente, la segmentación no reducirá el tiempo para completar una colada, pero si tenemos que hacer muchas coladas, la mejora de la productividad reducirá el tiempo total para completar todo el trabajo.

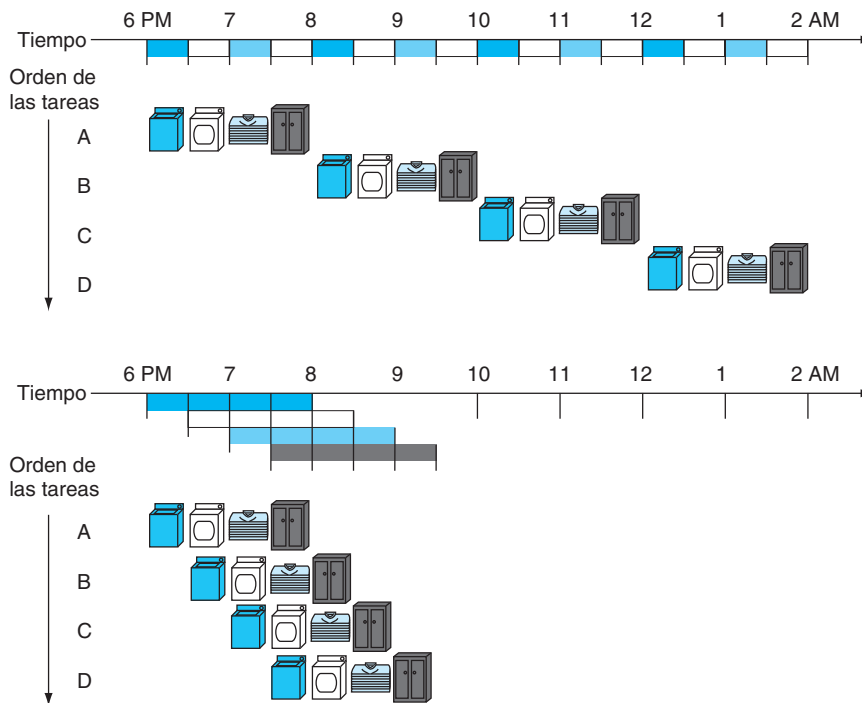


FIGURA 4.25 La analogía de la lavandería. Ann, Brian, Cathy y Don tienen ropa sucia que lavar, secar, doblar y guardar. Cada una de las cuatro tareas (lavar, secar, doblar y guardar) dura 30 minutos. Hacer la colada secuencialmente lleva 8 horas para realizar 4 coladas, mientras que la colada organizada de forma segmentada necesita solamente 3.5 horas. Aunque en realidad se dispone de un único recurso de cada tipo, para mostrar la etapa de segmentación usada por las diferentes coladas a lo largo del tiempo se usan copias de los 4 recursos.

Si todos los segmentos o etapas requieren la misma cantidad de tiempo y si hay el suficiente trabajo por hacer, entonces la ganancia que proporciona la segmentación es igual al número de segmentos en que se divide la tarea, que en este caso son cuatro: lavar, secar, doblar y colocar. Por tanto, el lavado segmentado es potencialmente cuatro veces más rápido que el no segmentado: 20 coladas necesitarán alrededor de 5 veces más tiempo que una única colada, mientras que la estrategia secuencial de lavado requiere 20 veces el tiempo de una colada. En la figura 4.25 la segmentación sólo es 2.3 veces más rápida porque se consideran únicamente 4 coladas. Observe que tanto al principio como al final de la tarea segmentada, el cauce de segmentación (que denominaremos *pipeline*) no está completamente lleno. Estos efectos transitorios iniciales y finales afectan a las prestaciones cuando el número de tareas es pequeño en comparación con el número de etapas. Cuando el número de coladas es mucho mayor que 4, las etapas estarán ocupadas la mayor parte del tiempo y el incremento en la productividad será muy cercano a 4.

Los mismos principios son aplicables a los procesadores cuando se segmenta la ejecución de instrucciones. Clásicamente se consideran cinco pasos para ejecutar las instrucciones MIPS:

1. Buscar una instrucción en memoria.
2. Leer los registros mientras se descodifica la instrucción. El formato de las instrucciones MIPS permite que la lectura y descodificación ocurran de forma simultánea.
3. Ejecutar una operación o calcular una dirección de memoria.
4. Acceder a un operando en la memoria de datos.
5. Escribir el resultado en un registro.

Por tanto, el pipeline MIPS que se explorará en este capítulo tiene cinco etapas. El siguiente ejemplo muestra que la segmentación acelera la ejecución de instrucciones del mismo modo que lo hace para la colada.

EJEMPLO

Las prestaciones en un diseño de ciclo único (monociclo) frente a un diseño segmentado

Para concretar la discusión se creará un pipeline. En este ejemplo y en el resto del capítulo limitaremos nuestra atención a sólo 8 instrucciones: cargar palabra (*lw*), almacenar palabra (*sw*), sumar (*add*), restar (*sub*), y-lógica (*and*), o-lógica (*or*), activar si menor (*set-less-than*, *slt*), y saltar si es igual (*beq*).

Compare el tiempo promedio entre la finalización de instrucciones consecutivas en una implementación monociclo, en la que todas las instrucciones se ejecutan en un único ciclo de reloj, con una implementación segmentada. En este ejemplo, la duración de las operaciones para las unidades funcionales principales es de 200 ps para los accesos a memoria, de 200 ps para las operaciones en la ALU, y de 100 ps para las lecturas o las escrituras en el banco de registros. En el modelo monociclo cada instrucción dura exactamente un ciclo de reloj, y por tanto el ciclo de reloj debe estirarse lo suficiente para poder acomodar a la instrucción más lenta.

RESPUESTA

La figura 4.26 indica el tiempo requerido para cada una de las ocho instrucciones. El diseño monociclo debe permitir acomodar a la instrucción más lenta —que en la figura 4.26 es lw — y por tanto la duración de todas las instrucciones será de 800 ps. Del mismo modo que la figura 4.25, la figura 4.27 compara las ejecuciones segmentadas y no segmentadas de tres instrucciones lw . En el caso del diseño no segmentado, el tiempo transcurrido entre el inicio de la primera instrucción y el inicio de la cuarta instrucción es de 3×800 ps ó 2400 ps.

Todas las etapas de segmentación duran un único ciclo de reloj, de modo que el periodo del reloj debe ser suficientemente largo como para dar cabida a la operación más lenta. Así, del mismo modo que el diseño monociclo debe tener un ciclo de reloj de 800 ps, necesario para el caso más desfavorable, aun cuando algunas instrucciones podrían ser ejecutadas en 500 ps, la ejecución segmentada debe tener un ciclo de reloj de 200 ps para dar cabida a la etapa más desfavorable, aunque algunas etapas sólo necesiten 100 ps. Aún así, la segmentación cuadriplica las prestaciones: el tiempo entre el inicio de la primera instrucción y el inicio de la cuarta instrucción es de 3×200 ps o 600 ps.

Clase de instrucción	Búsqueda de la instrucción	Lectura de registros	Operación ALU	Acceso al dato	Escritura en registro	Tiempo total
Almacenar palabra (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Cargar palabra (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
Formato R (add , sub , and , or , $sllt$)	200 ps	100 ps	200 ps		100 ps	600 ps
Salto (beq)	200 ps	100 ps	200 ps			500 ps

FIGURA 4.26 Tiempo total para cada instrucción calculado a partir del tiempo de cada componente. El cálculo supone que no existe ningún retardo debido a los multiplexores, a la unidad de control, a los accesos al registro PC o a la unidad de extensión de signo.

Es posible convertir en una fórmula la discusión anterior sobre la ganancia de la segmentación. Si las etapas están perfectamente equilibradas, entonces el tiempo entre instrucciones en el procesador segmentado —suponiendo condiciones ideales— es igual a

$$\text{Tiempo entre instrucciones}_{\text{segmentado}} = \frac{\text{Tiempo entre instrucciones}_{\text{no segmentado}}}{\text{Número de etapas de segmentación}}$$

En condiciones ideales y con un gran número de instrucciones, la ganancia debida a la segmentación es aproximadamente igual al número de etapas; un pipeline de cinco etapas es casi cinco veces más rápido.

La fórmula sugiere que con cinco etapas se debe mejorar en cinco veces el tiempo de 800 ps que proporciona el esquema no segmentado, es decir, lograr un ciclo de reloj de 160 ps. Sin embargo, el ejemplo muestra que las etapas pueden estar equilibradas de manera imperfecta. Además, la segmentación implica algún gasto o sobrecarga adicional, cuya fuente será presentada con más claridad en breve. Así, el tiempo por instrucción en el procesador segmentado excederá el mínimo valor posible, y la ganancia será menor que el número de etapas de segmentación.

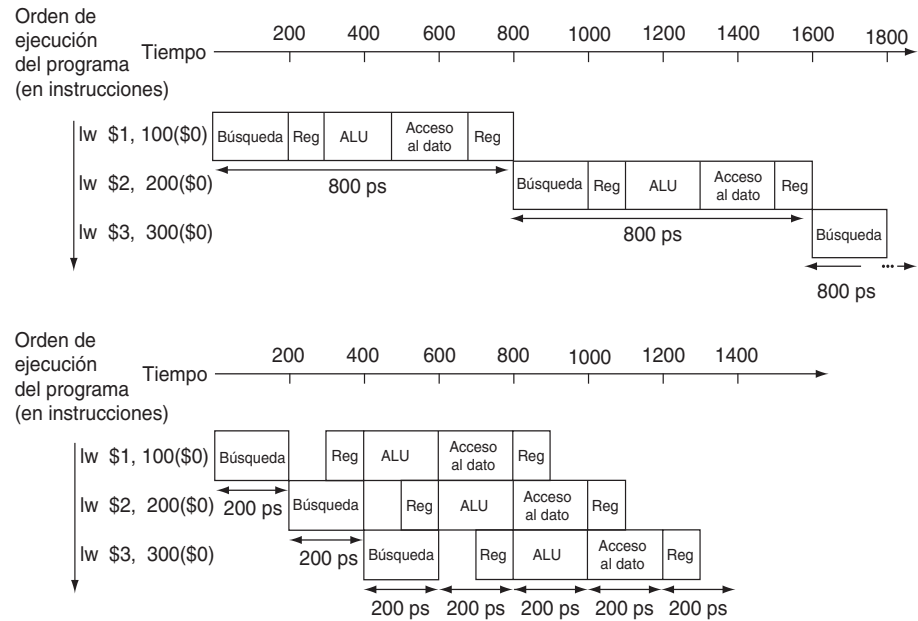


FIGURA 4.27 Arriba: ejecución monociclo sin segmentar; abajo: ejecución segmentada.

Ambos esquemas usan los mismos componentes hardware, cuyo tiempo se lista en la figura 4.26. En este caso se aprecia una ganancia de 4 en el tiempo promedio entre instrucciones, desde 800 ps hasta 200 ps. Comparar esta figura con la figura 4.25. En el caso de la lavandería, se suponía que todas las etapas eran iguales. Si la secadora hubiera sido el elemento más lento, entonces hubiera determinado el tiempo de cada etapa. Los segmentos del computador están limitados a operar a la velocidad del más lento, bien sea la operación de la ALU o el acceso a memoria. Se supondrá que la escritura en el banco de registros ocurre durante la primera mitad del ciclo de reloj mientras que las lecturas del banco de registros ocurren durante la segunda mitad. Se usará esta suposición a lo largo de todo el capítulo.

Incluso nuestra afirmación de que en el ejemplo se mejoran cuatro veces los resultados no queda reflejada en el tiempo total de ejecución para las tres instrucciones: 1400 ps frente a 2400 ps. Por supuesto, esto es debido a que el número de instrucciones analizado es muy pequeño. ¿Qué ocurriría si se incrementara el número de instrucciones? Podemos extender los números del ejemplo previo a 1.000.003 instrucciones. Añadiendo 1.000.000 instrucciones en el ejemplo segmentado se suman 200 ps por instrucción al tiempo total de ejecución. El tiempo total de ejecución sería de $1.000.000 \times 200 \text{ ps} + 1400 \text{ ps}$, ó 200.001.400 ps. En el ejemplo no segmentado, añadiríamos 1.000.000 instrucciones, cada una con una duración de 800 ps, de modo que el tiempo total de ejecución sería $1.000.000 \times 800 \text{ ps} + 2400 \text{ ps}$, ó 800.002.400 ps. En estas condiciones ideales, la razón entre los tiempos totales de ejecución para programas reales en procesadores no segmentados comparados con los tiempos en procesadores segmentados es cercana a la razón de los tiempos entre instrucciones:

$$\frac{800.002.400 \text{ ps}}{200.001.400 \text{ ps}} \approx \frac{800 \text{ ps}}{200 \text{ ps}} \approx 4,00$$

La segmentación mejora las prestaciones *incrementando la productividad (throughput) de las instrucciones, en lugar de disminuir el tiempo de ejecución de cada instrucción individual*, pero la productividad de las instrucciones es la métrica importante, ya que los programas reales ejecutan miles de millones de instrucciones.

Diseño de repertorios de instrucciones para la segmentación

Incluso con esta explicación sencilla de la segmentación es posible comprender el diseño del repertorio de instrucciones del MIPS, diseñado expresamente para la ejecución segmentada.

En primer lugar, todas las instrucciones MIPS tienen la misma longitud. Esta restricción hace que sea mucho más fácil la búsqueda de instrucciones en la primera etapa de la segmentación y la decodificación en la segunda etapa. En un repertorio de instrucciones como el x86, donde las instrucciones varían de 1 byte a 17 bytes, la segmentación es un reto considerable. Las implementaciones recientes de la arquitectura x86 en realidad convierten las instrucciones x86 en microoperaciones simples muy parecidas a instrucciones MIPS. Y se segmenta la ejecución de las microoperaciones en lugar de las instrucciones x86 nativas (véase sección 4.10).

Segundo, MIPS tiene sólo unos pocos formatos de instrucciones, y además en todos ellos los campos de los registros fuentes están situados siempre en la misma posición de la instrucción. Esta simetría implica que la segunda etapa pueda empezar a leer del banco de registros al mismo tiempo que el hardware está determinando el tipo de la instrucción que se ha leído. Si los formatos de instrucciones del MIPS no fueran simétricos, sería necesario partir la segunda etapa, dando como resultado un pipeline de seis etapas. En breve se indicarán la desventaja de los pipelines más largos.

En tercer lugar, en el MIPS los operandos a memoria sólo aparecen en instrucciones de carga y almacenamiento. Esta restricción hace que se pueda usar la etapa de ejecución para calcular la dirección de memoria y en la siguiente etapa se pueda acceder a memoria. Si se permitiera usar operandos en memoria en todas las operaciones, como hace la arquitectura x86, las etapas 3 y 4 se extenderían a una etapa de cálculo de dirección, una etapa de acceso a memoria, y luego una etapa de ejecución.

Y cuarto, como se discutió en el capítulo 2, los operandos deben estar alineados en memoria. Por lo tanto, no hay que preocuparse de que una instrucción de transferencia de datos necesite dos accesos a memoria para acceder a un solo dato; el dato pedido siempre podrá ser transferido entre procesador y memoria en una única etapa.

Riesgos del pipeline

Hay situaciones de segmentación en las que la instrucción siguiente no se puede ejecutar en el ciclo siguiente. Estos sucesos se llaman riesgos (*hazards*) y los hay de tres tipos diferentes

Riesgos estructurales

El primer riesgo se denomina **riesgo estructural** (*structural hazard*). Significa que el hardware no admite una cierta combinación de instrucciones durante el mismo ciclo. Un riesgo estructural en la lavandería ocurriría si se usara una combinación lavadora-secadora en lugar de tener la lavadora y la secadora separadas,

Riesgo estructural: caso en el que una instrucción en curso no se puede ejecutar en el ciclo de reloj adecuado porque el hardware no admite la combinación de instrucciones que se pretenden ejecutar en ese ciclo de reloj.

o si nuestro compañero de piso estuviera ocupado y no pudiera guardar la ropa. De esta forma, se frustraría la tan cuidadosamente planeada segmentación.

Como se ha mencionado con anterioridad, el repertorio de instrucciones MIPS fue diseñado para ser segmentado, por lo que facilita a los diseñadores evitar riesgos estructurales. Supongamos, sin embargo, que se tuviera un solo banco de memoria en lugar de dos. Si el pipeline de la figura 4.27 tuviera una cuarta instrucción, se vería que en un mismo ciclo la primera instrucción está accediendo a datos de memoria mientras que la cuarta está buscando una instrucción de la misma memoria. Sin disponer de dos bancos de memoria, nuestra segmentación podría tener riesgos estructurales.

Riesgos de datos

Riesgo de datos: (riesgo de datos en el pipeline): caso en el que la instrucción en curso no se puede ejecutar en el ciclo de reloj adecuado porque el dato necesario para ejecutar la instrucción no está todavía disponible

Los **riesgos de datos** (*data hazards*), ocurren cuando el pipeline se debe bloquear debido a que un paso de ejecución debe esperar a que otro paso sea completado. Volviendo a la lavandería, supongamos que se está doblando una colada y no se encuentra la pareja de un cierto calcetín. Una estrategia posible es ir al piso a buscar en todos los armarios hasta encontrar la pareja. Obviamente, mientras se busca deberán esperar todas las coladas secas y preparadas para ser dobladas, y todas las coladas limpias y preparadas para ser secadas.

En un pipeline del computador, los riesgos de datos surgen de las dependencias entre una instrucción y otra anterior que se encuentra todavía en el pipeline (una relación que en realidad no se da en la lavandería). Por ejemplo, suponer que se tiene una instrucción `add` seguida inmediatamente por una instrucción `sub` que usa el resultado de la suma (`$s0`):

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

Si no se interviene, un riesgo de datos puede bloquear severamente al procesador. La instrucción `add` no escribe su resultado hasta la quinta etapa, por lo que se tendrían que añadir tres burbujas en el procesador.

Aunque se podría confiar que los compiladores eliminaran todos esos riesgos, los resultados no serían satisfactorios. Estas dependencias ocurren demasiado a menudo y el retardo es demasiado largo para esperar que el compilador nos salve de este problema.

La principal solución se basa en la observación de que no es necesario esperar a que la instrucción se complete antes de intentar resolver el riesgo de datos. Para la secuencia de código anterior, tan pronto como la ALU calcula la suma para la instrucción `add`, se puede suministrar el resultado como entrada de la resta. Se denomina **anticipación de resultados** (*forwarding*) o **realimentación** (*bypassing*) al uso de hardware extra para anticipar antes el dato buscado usando los recursos internos del procesador.

Anticipación de resultados (realimentación): método de resolver un riesgo de datos que consiste en obtener el dato buscado de los búfer internos en lugar de esperar a que llegue a los registros visibles a nivel de programación o a la memoria.

Anticipar datos entre dos instrucciones

Para las dos instrucciones anteriores, mostrar qué etapas del pipeline estarían conectadas por el mecanismo de anticipación de resultados. Usar el dibujo de la figura 4.28 para representar el camino de datos durante las cinco etapas del pipeline. Alinear una copia del camino de datos para cada instrucción, de forma similar a como se hizo en la figura 4.25 para el pipeline de la lavandería.

EJEMPLO

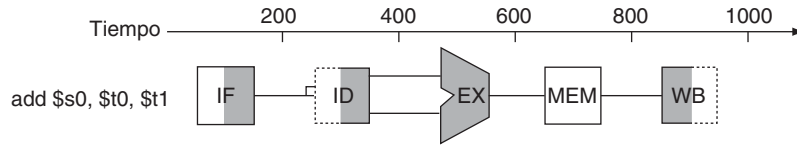


FIGURA 4.28 Representación gráfica del pipeline de instrucciones, similar en espíritu a la segmentación de la lavandería de la figura 4.25. Para representar los recursos físicos se usan símbolos con las abreviaciones de las etapas del pipeline usadas a lo largo del capítulo. Los símbolos para las cinco etapas son: *IF* para la etapa de búsqueda de instrucciones (*instruction fetch*), con la caja que representa la memoria de instrucciones; *ID* para la etapa de descodificación de instrucciones y de lectura del banco de registros (*instruction decode*), con el dibujo que muestra la lectura al banco de registros; *EX* para la etapa de ejecución de instrucciones (*instruction execution*), con el dibujo que representa la ALU; *MEM* para la etapa de acceso a memoria (*memory access*), con la caja que representa la memoria de datos; y *WB* para la etapa de escritura de resultado (*write back*), con el dibujo que muestra la escritura en el banco de registros. El sombreado del dibujo indica el elemento que se usa por parte de la instrucción. De este modo, MEM tiene un fondo blanco porque la instrucción *add* no accede a la memoria de datos. El sombreado en la mitad derecha del banco de registros o de la memoria significa que el elemento se lee en esa etapa, y el sombreado en la mitad izquierda del banco de registros significa que se escribe en esa etapa. Por tanto, la mitad derecha de ID está sombreada en la segunda etapa porque se lee el banco de registros, y la mitad izquierda de WB está sombreada en la quinta etapa porque se escribe en el banco de registros.

La figura 4.29 muestra la conexión para anticipar el valor de *\$s0* después de la etapa de ejecución de la instrucción *add* a la entrada de la etapa de ejecución de la instrucción *sub*.

RESPUESTA

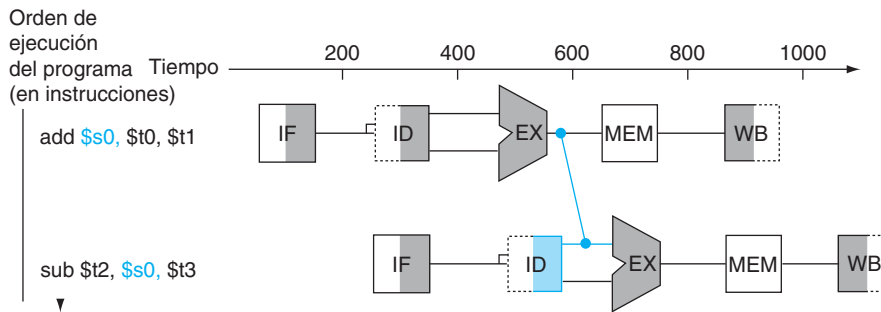


FIGURA 4.29 Representación gráfica de la anticipación de datos. La conexión muestra el camino de anticipación de datos desde la salida de la etapa EX de *add* hasta la entrada de la etapa EX de *sub*, reemplazando el valor del registro *\$s0* leído en la segunda etapa de la instrucción *sub*.

En esta representación gráfica de los sucesos, las líneas de anticipación de datos sólo son válidas si la etapa destino está más adelante en el tiempo que la etapa origen. Por ejemplo, no puede establecerse una línea válida de anticipación desde la salida de la etapa de acceso al dato de la primera instrucción hasta la entrada de la etapa de ejecución de la siguiente, puesto que eso significaría ir hacia atrás en el tiempo.

La anticipación funciona muy bien y será descrita en detalle en la sección 4.7. De todos modos, no es capaz de evitar todos los bloqueos. Por ejemplo, suponer que la primera instrucción fuera una carga del registro *\$s0*, en lugar de una suma. Como se puede deducir a partir de una mirada a la figura 4.29, el dato deseado sólo estaría dis-

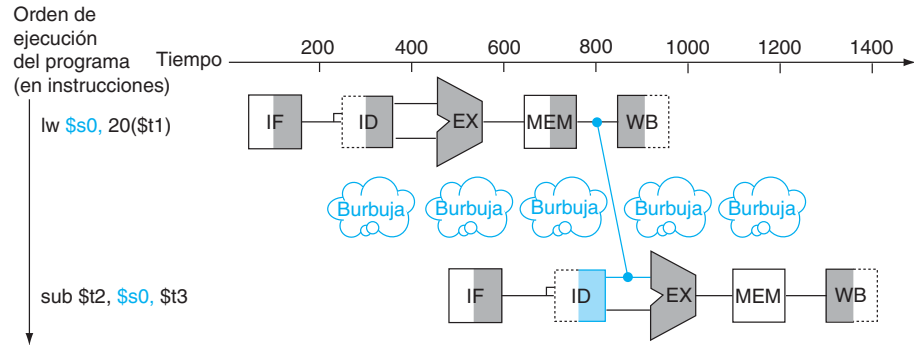


FIGURA 4.30 El bloqueo es necesario incluso con la anticipación de resultados cuando una instrucción de Formato R intenta usar el dato de una instrucción de carga precedente.

Sin el bloqueo, la ruta desde la salida de la etapa de acceso a memoria hasta la entrada de la etapa de ejecución sería hacia atrás en el tiempo, lo cual es imposible. Esta figura es en realidad una simplificación, pues hasta después de que la instrucción de resta es buscada y decodificada no se puede conocer si será o no será necesario un bloqueo. La sección 4.7 muestra los detalles de lo que sucede en realidad en el caso de los riesgos.

Riesgo del dato de una carga: forma específica de riesgo en la que el dato de una instrucción de carga no está aún disponible cuando es pedido por otra instrucción.

Bloqueo del pipeline (burbuja): bloqueo iniciado para resolver un riesgo.

ponible *después* de la cuarta etapa de la primera instrucción, lo cual es demasiado tarde para la *entrada* de la etapa EX de la instrucción *sub*. Por lo tanto, incluso con la anticipación de resultados, habría que bloquear durante una etapa en el caso del **riesgo del dato de una carga** (*load-use data hazard*), tal como se puede ver en la figura 4.30. Esta figura muestra un importante concepto sobre la segmentación, oficialmente denominado un **bloqueo del pipeline** (*pipeline stall*), pero que frecuentemente recibe el apodo de **burbuja** (*bubble*). Se verán bloqueos en otros lugares del pipeline. La sección 4.7 muestra cómo se pueden tratar casos difíciles como éstos, bien con un hardware de detección y con bloqueos, bien por software, que reordena el código para evitar los bloqueos debido a riesgos de datos de una carga (*load-use*), como se ilustra en el siguiente ejemplo.

Reordenación de código para evitar bloqueos del pipeline

Considere el siguiente segmento de código en lenguaje C:

```
a = b + e;
c = b + f;
```

Aquí se muestra el código MIPS generado para este segmento, suponiendo que todas las variables están en memoria y son direccionables como desplazamientos a partir de `$t0`:

EJEMPLO

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

Encuentre los riesgos que existen en el segmento de código y reordene las instrucciones para evitar todos los bloqueos del pipeline.

Las dos instrucciones `add` tienen un riesgo debido a sus respectivas dependencias con la instrucción `lw` que les precede inmediatamente. Observe que la anticipación de datos elimina muchos otros riesgos potenciales, incluidos la dependencia del primer `add` con el primer `lw` y los riesgos con las instrucciones `store`. Mover hacia arriba la tercera instrucción `lw` elimina ambos riesgos.

```
lw      $t1, 0($t0)
lw      $t2, 4($t1)
lw      $t4, 8($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

En un procesador segmentado con anticipación de resultados, la secuencia reordenada se completará en dos ciclos menos que la versión original.

La anticipación de resultados conlleva otra característica que hace comprender la arquitectura MIPS, además de las cuatro mencionadas en la página 335. Cada instrucción MIPS escribe como mucho un resultado y lo hace cerca del final del pipeline. La anticipación de resultados es más complicada si hay múltiples resultados que avanzar por cada instrucción, o si se necesita escribir el resultado antes del final de la ejecución de la instrucción.

Extensión: El nombre “anticipación de resultado” (*forwarding*) viene de la idea de que el resultado se pasa hacia adelante, de una instrucción anterior a una instrucción posterior. El término “realimentación” (*bypassing*) viene del hecho de pasar el resultado a la unidad funcional deseada sin tener que copiarlo previamente en el banco de registros.

Riesgos de control

El tercer tipo de riesgo se llama **riesgo de control** (*control hazard*) y surge de la necesidad de tomar una decisión basada en los resultados de una instrucción mientras las otras se están ejecutando.

Supongamos que los trabajadores de nuestra lavandería tienen la feliz tarea de lavar los uniformes de un equipo de fútbol. Dependiendo de lo sucia que esté la colada, se necesitará determinar si el detergente y la temperatura del agua

RESPUESTA

Riesgo de control (riesgo de saltos): caso en el que la instrucción en curso no se puede ejecutar en el ciclo de reloj adecuado porque la instrucción que ha sido buscada no es la que se requería; esto es, el flujo de direcciones de instrucciones no es el que el pipeline esperaba.

seleccionadas son suficientemente fuertes para lavar los uniformes, pero no tan fuertes como para que éstos se desgasten y se rompan pronto. En nuestra lavandería segmentada, se tiene que esperar a la segunda etapa de la segmentación para examinar el uniforme seco y comprobar si se tiene que cambiar algunos de los parámetros del lavado. ¿Qué se debe hacer?

A continuación se presenta la primera de dos soluciones posibles a los riesgos de control en la lavandería y sus equivalentes para los computadores.

Bloquear (stall): Operar de manera secuencial hasta que la primera carga esté seca, y entonces repetir hasta que se consiga la fórmula correcta.

Esta opción conservadora realmente funciona, pero es lenta.

En un computador, la tarea de decisión equivalente es la instrucción de salto (*branch*). Observe que se debe comenzar a buscar la instrucción que sigue a un salto justo en el siguiente ciclo de reloj. Pero el pipeline puede no conocer cuál es la siguiente instrucción, ya que *¡justo acaba de obtener* la instrucción de salto de la memoria! Igual que con la lavandería, una posible solución es bloquear inmediatamente después de haber ido a buscar un salto, y esperar a que el pipeline determine el resultado del salto y conozca cuál es la instrucción que se debe ir a buscar.

Supongamos que se emplea suficiente hardware adicional para poder examinar los registros, calcular la dirección del salto y actualizar el registro PC durante la segunda etapa del pipeline (véase la sección 4.8 para más detalles). Incluso con este hardware adicional, el pipeline que trata los saltos condicionales se parecería al de la figura 4.31. La instrucción *lw*, ejecutada si el salto no se toma, se bloquea un ciclo extra de 200 ps antes de empezar.

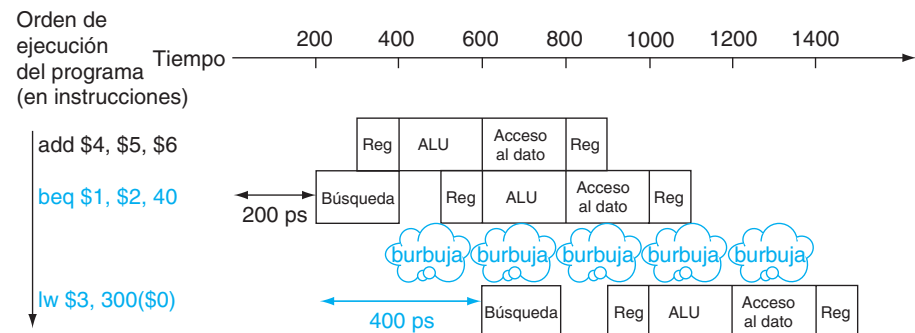


FIGURA 4.31 Pipeline que muestra el bloqueo en cada salto condicional como solución a los riesgos de control. En este ejemplo se supone que el salto condicional es tomado y que la instrucción destino del salto es la instrucción “OR”. Se produce un bloqueo en el pipeline de una etapa, o burbuja, después del salto. En realidad, el proceso de crear el bloqueo es ligeramente más complicado, tal y como se verá en la sección 4.8. El efecto en las prestaciones, en cambio, sí que es el mismo que se daría si realmente se insertara una burbuja.

Rendimiento de “bloquear los saltos”

Estimar el impacto en ciclos de reloj por instrucción (CPI) de bloqueo en los saltos. Suponer que todas las otras instrucciones tienen un CPI de 1.

La figura 3.27 del capítulo 3 muestra que los saltos condicionales representan el 17% de las instrucciones ejecutadas en SPECint2006. Dado que la ejecución del resto de las instrucciones tiene un CPI de 1 y los saltos necesitan un ciclo de reloj adicional debido al bloqueo, entonces se observará un CPI de 1.17 y por lo tanto una desaceleración de 1.17 respecto al caso ideal.

EJEMPLO

RESPUESTA

Si el salto no se puede resolver en la segunda etapa, como ocurre frecuentemente en pipelines más largos, bloquearse ante la presencia de un salto supondrá una mayor disminución del rendimiento. El coste de esta opción es demasiado alto para la mayoría de los computadores y motiva una segunda solución para los riesgos de control:

Predicción: Si se está bastante seguro de que se tiene la fórmula correcta para lavar los uniformes, entonces basta con predecir que funcionará bien y lavar la segunda colada mientras se espera que la primera se seque.

Cuando se acierta, el rendimiento de la segmentación no se reduce. Sin embargo, cuando se falla, se tiene que rehacer la colada que fue lavada mientras se comprobaba si la decisión era correcta.

Los computadores realmente también usan la *predicción* para tratar los saltos. Una estrategia simple es predecir que siempre se dará un salto no tomado. Cuando se acierta, el pipeline funciona a máxima velocidad. El procesador sólo se bloquea cuando los saltos son tomados. La figura 4.32 muestra este ejemplo.

Una versión más sofisticada de la **predicción de saltos (branch prediction)** supondría predecir que algunos saltos saltan (se toman) y que otros no saltan (no se toman). En nuestra analogía, los uniformes oscuros o los de casa podrían necesitar una fórmula y los claros o de calle otra fórmula. En los computadores, los saltos que cierran un lazo saltan hacia atrás hasta el principio del lazo. Ya que probablemente estos saltos serán tomados y saltan hacia atrás, se podría predecir como tomados aquellos saltos cuya dirección destino es una dirección anterior a la de la instrucción de salto.

Este tipo de enfoque tan rígido de la predicción de saltos se basa en un comportamiento estereotipado y no tiene en cuenta las características individuales de cada instrucción de salto específica. Los predictores *dinámicos* realizados en hardware, en claro contraste con los explicados anteriormente, hacen sus predicciones dependiendo del comportamiento de cada salto y pueden cambiar las predicciones para un mismo salto a lo largo de la ejecución de un programa. Siguiendo nuestra analogía, usando predicción dinámica una persona miraría cómo está de sucio el uniforme e intentaría adivinar la fórmula, ajustando la siguiente predicción dependiendo del acierto de las últimas predicciones hechas.

Predicción de saltos: método de resolver los riesgos de saltos que presupone un determinado resultado para el salto y procede a partir de esta suposición, en lugar de esperar a que se establezca el resultado real.

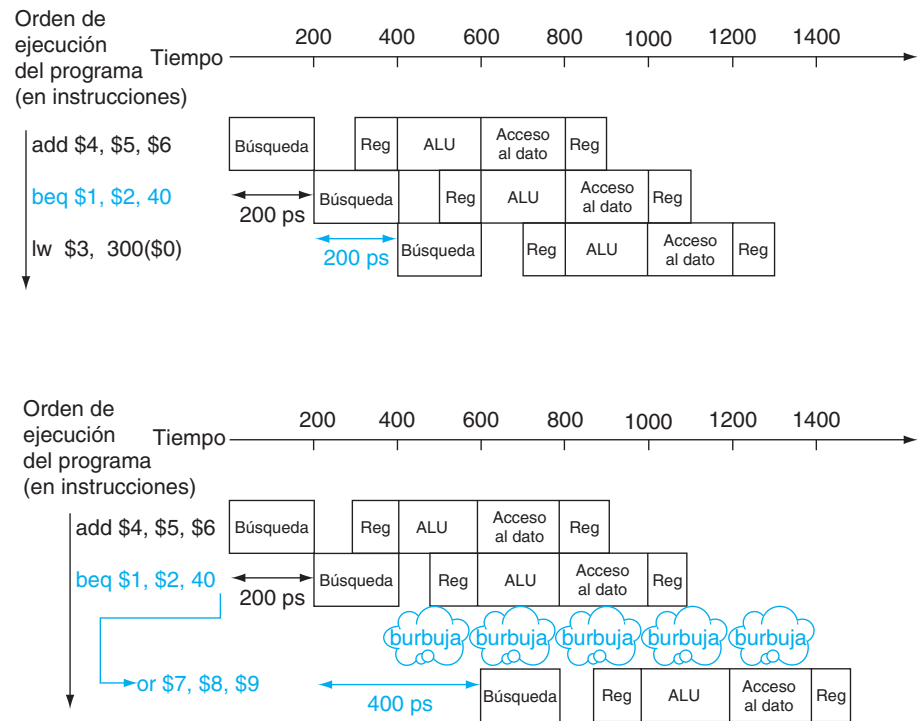


FIGURA 4.32 Predecir que los saltos son no tomados como solución a los riesgos de control. El dibujo de la parte superior muestra el pipeline cuando el salto es no tomado. El dibujo de la parte inferior muestra el pipeline cuando el salto es tomado. Tal como se mostraba en la figura 4.31, la inserción de una burbuja de esta manera simplifica lo que sucede en realidad, al menos durante el primer ciclo de reloj que sigue de forma inmediata al salto. La sección 4.8 revelará los detalles.

Un enfoque muy utilizado en la predicción dinámica de saltos en computadores es mantener una historia para cada salto que indique si ha sido tomado o no tomado, y entonces usar el pasado reciente para predecir el futuro. Tal como se verá más tarde, la cantidad y tipo de historia almacenada ha ido creciendo, con el resultado de que los predictores dinámicos de saltos pueden predecir saltos correctamente con alrededor del 90% de precisión (véase sección 4.8). Cuando la predicción es incorrecta, el control del procesador segmentado debe asegurar que las instrucciones que siguen al salto mal predicho no tienen efecto y debe reinicializar el pipeline desde la dirección correcta de salto. En nuestra analogía de la lavandería, se deben detener las coladas para reiniciar la colada que fue incorrectamente predicha.

Como en el caso de las otras soluciones a los riesgos de control, los pipelines largos agravan el problema, en este caso elevando el coste del fallo de predicción. En la sección 4.8 se describen con más detalle las soluciones a los riesgos de control.

Extensión: Hay un tercer enfoque para los riesgos de control, llamado decisión retardada (*delayed decision*) mencionado anteriormente. En nuestra analogía, cada vez que se vaya a tomar una decisión acerca del lavado se pone en la lavadora ropa que no sea de

fútbol, mientras se espera a que los uniformes de fútbol se sequen. Mientras se tenga suficiente ropa sucia que no se vea afectada por el test, esta solución funciona bien.

En los computadores este enfoque se llama salto retardado (*delayed branch*), y es la solución que se usa en la arquitectura MIPS. El salto retardado siempre ejecuta la siguiente instrucción secuencial, mientras que el salto se ejecuta después del retardo de una instrucción. Esta funcionalidad es oculta al programador de lenguaje ensamblador MIPS, ya que el ensamblador arregla el código automáticamente para que el comportamiento de los saltos sea el que desea el programador. El ensamblador de MIPS pondrá inmediatamente después del salto retardado una instrucción que no dependa del salto, y si el salto es tomado la dirección que se cambiará es la de la instrucción que *sigue* a esta instrucción no dependiente (que es seguro que siempre se va a ejecutar). En el ejemplo, la instrucción `add` que está antes del salto en la figura 4.31 no afecta al salto, y puede moverse después del salto para ocultar completamente su retardo. Como los saltos retardados son útiles cuando los saltos son cortos, ningún procesador usa un salto retardado de más de 1 ciclo. Para retardos mayores de los saltos, generalmente se usa la predicción basada en hardware.

Resumen de la visión general del pipeline

La segmentación es una técnica que explota el paralelismo entre las instrucciones de un flujo secuencial. A diferencia de otras técnicas para incrementar la velocidad del procesador, tiene la ventaja sustancial de ser fundamentalmente invisible al programador.

En las siguientes secciones de este capítulo se analiza el concepto de segmentación utilizando el subconjunto del repertorio de instrucciones MIPS ya utilizado en la implementación de ciclo único de la sección 4.4 y una versión simplificada de su pipeline. Se estudiarán los problemas introducidos por la segmentación y el rendimiento que se puede conseguir en diversas situaciones típicas.

Si el lector desea centrarse más en las implicaciones que suponen la segmentación para el software y para las prestaciones, después de acabar esta sección tendrá suficientes conocimientos básicos como para pasar a la sección 4.10, que introduce conceptos avanzados de la segmentación, como son los procesadores superescalares y la planificación dinámica y la sección 4.11 examina el pipeline de microprocesadores recientes.

Alternativamente, si se está interesado en comprender cómo se implementa el pipeline y los retos de manejar los riesgos, se puede comenzar a examinar el diseño de un camino de datos segmentado y del control básico, que se explican en la sección 4.6. Se debería ser capaz de usar este conocimiento para analizar la implementación de la anticipación de datos y los bloqueos en la sección 4.7. Se puede leer la sección 4.8 para aprender más sobre soluciones para los riesgos de saltos, y después leer en la sección 4.9 cómo se gestionan las excepciones.

Para cada una de las secuencias de código mostradas abajo, afirmar si se deben producir bloqueos, si se pueden evitar los bloqueos usando solamente la anticipación de resultados, o si se pueden ejecutar sin bloquear y sin avanzar resultados:

Secuencia 1	Secuencia 2	Secuencia 3
<pre>lw \$t0,0(\$t0) add \$t1,\$t0,\$t0</pre>	<pre>add \$t1,\$t0,\$t0 addi \$t2,\$t0,#5 addi \$t4,\$t1,#5</pre>	<pre>addi \$t1,\$t0,#1 addi \$t2,\$t0,#2 addi \$t3,\$t0,#2 addi \$t3,\$t0,#4 addi \$t5,\$t0,#5</pre>

Autoevaluación

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

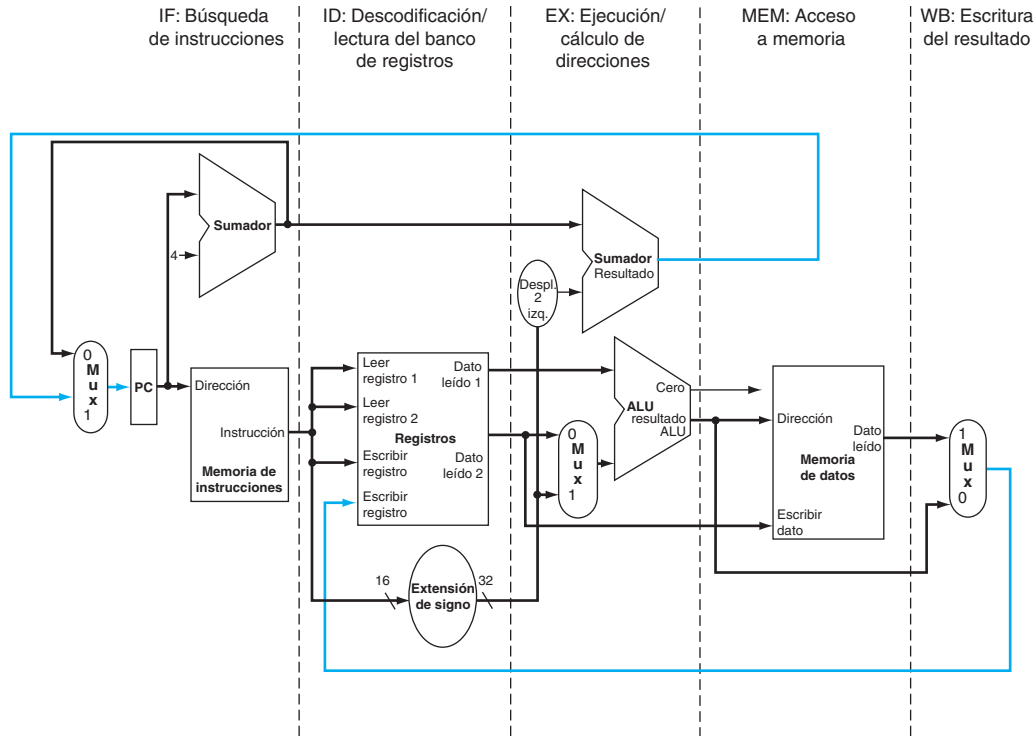


FIGURA 4.33 Camino de datos monociclo extraído de la sección 4.4 (similar al de la figura 4.17). Cada paso de la instrucción se puede situar en el camino de datos de izquierda a derecha. Las únicas excepciones a esta regla son la actualización del registro PC y el paso final de escritura de resultados, 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.

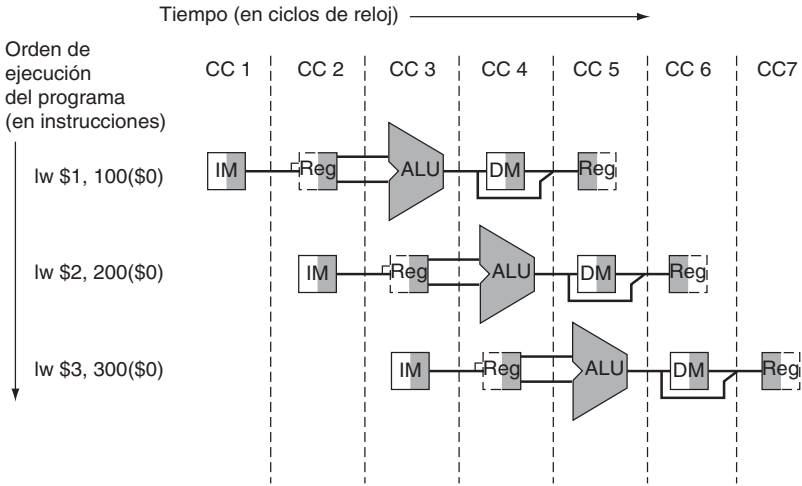


FIGURA 4.34 Ejecución de instrucciones usando el camino de datos monociclo de la figura 4.33, suponiendo que la ejecución está segmentada. De un modo similar a como se hace en las figuras 4.28 a 4.30, esta figura supone que cada instrucción tiene su propio camino de datos, y cada parte del camino de 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 descodificación 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 sombrear 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 sombrear 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.

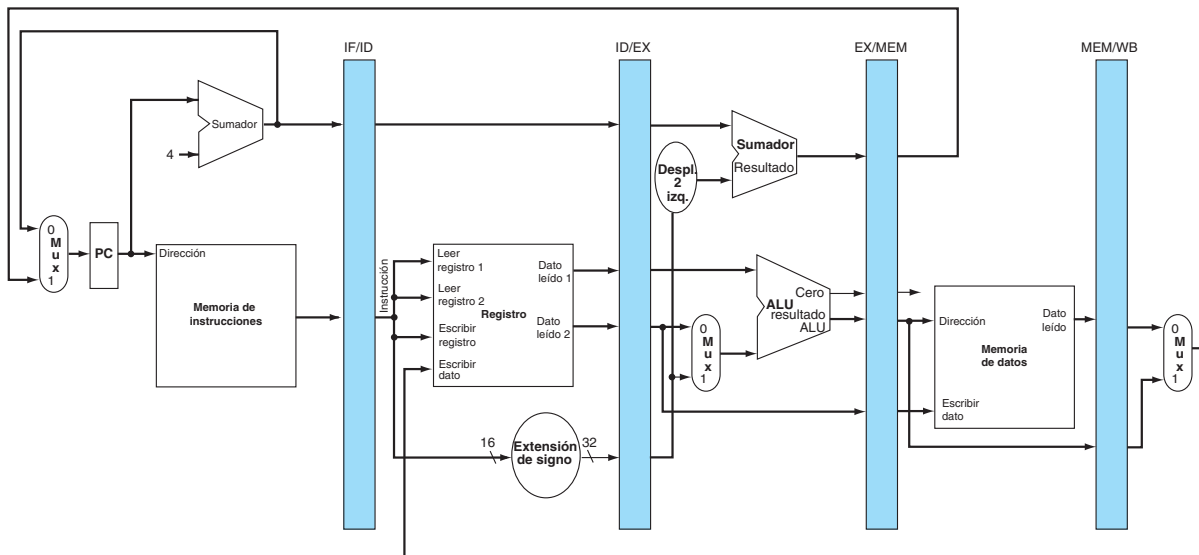


FIGURA 4.35 Versión segmentada del camino de datos de la figura 4.33. Los registros de segmentación, en color, separan cada una de las etapas. Están etiquetados con las etapas que separan; por ejemplo, el primero está etiquetado como IF/ID ya que separa las etapas de búsqueda de instrucciones y de decodificación. 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, $1w$, 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.

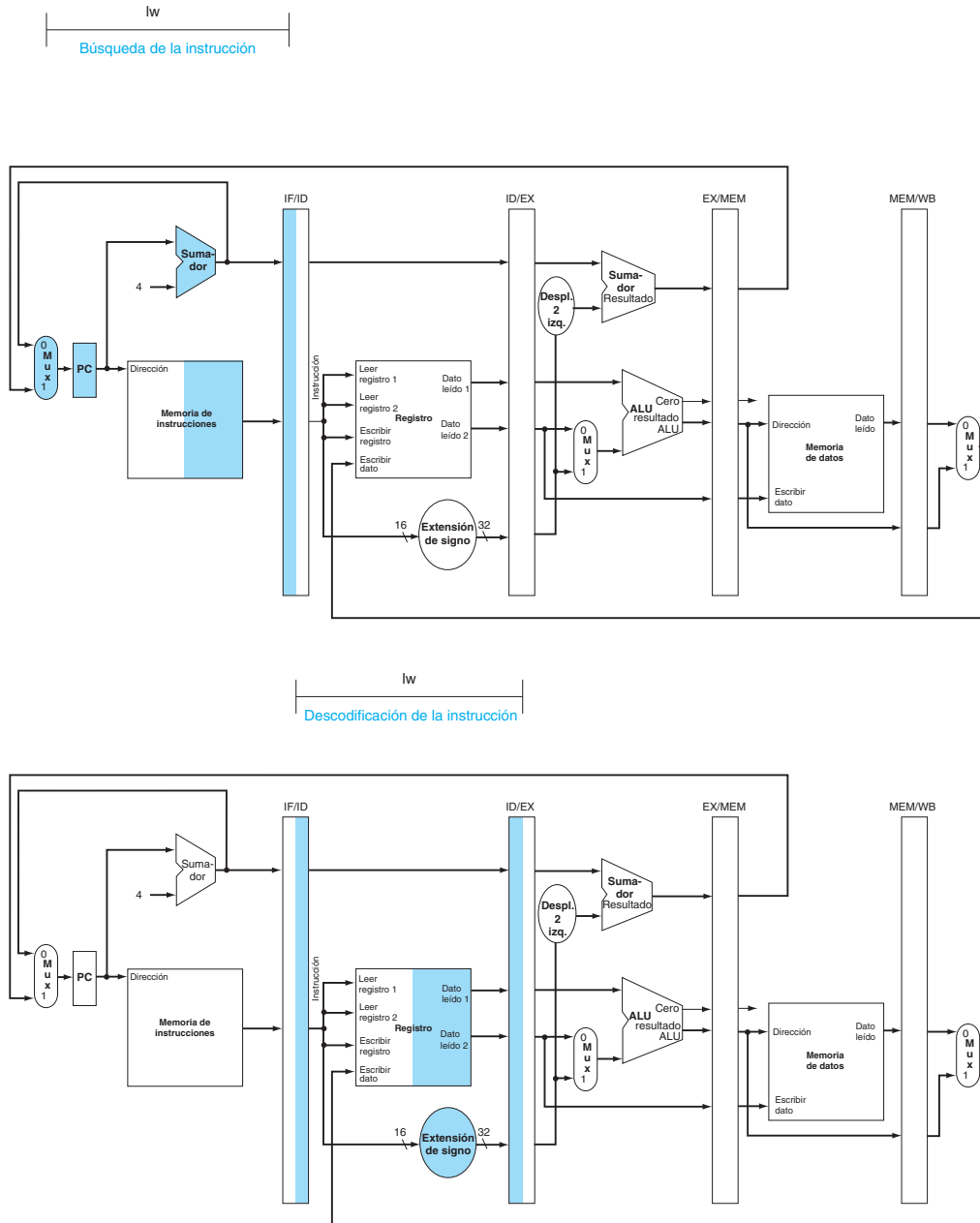


FIGURA 4.36 IF e ID: primera y segunda etapa de segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en estas dos etapas. La convención para resaltar los elementos del camino de datos es la misma que se usó en la figura 4.28. Igual que en la 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 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.

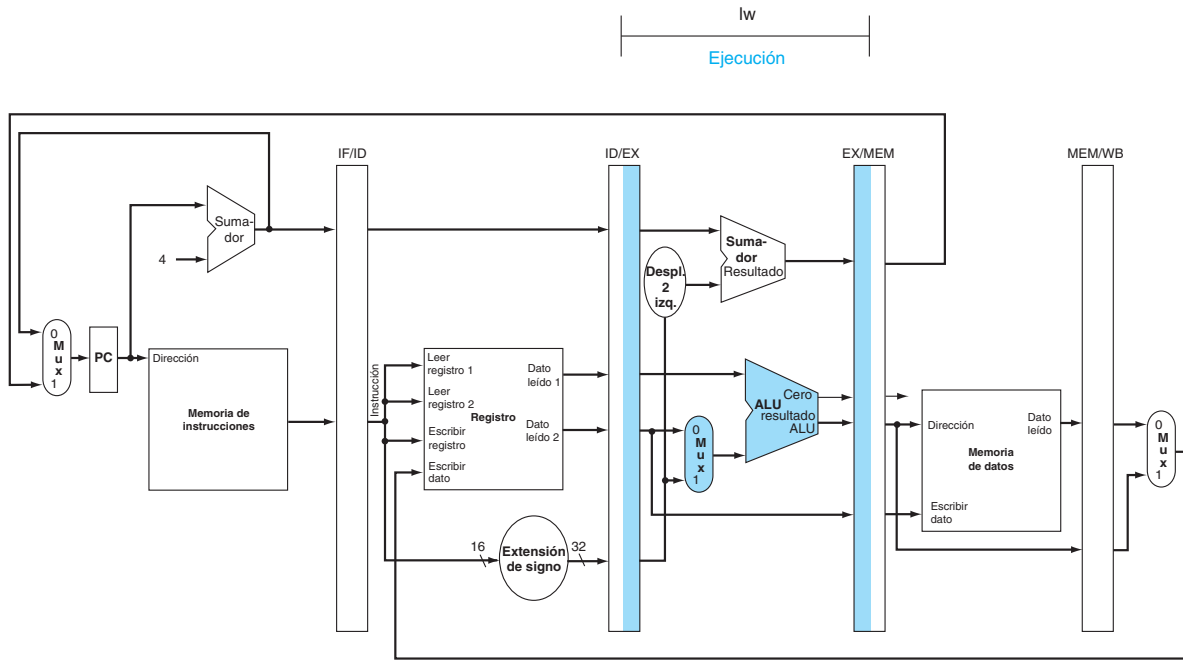


FIGURA 4.37 EX: tercera etapa de la segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en esta etapa. El registro se suma al valor inmediato con el signo extendido, y el resultado de la suma se coloca en el registro de segmentación EX/MEM.

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

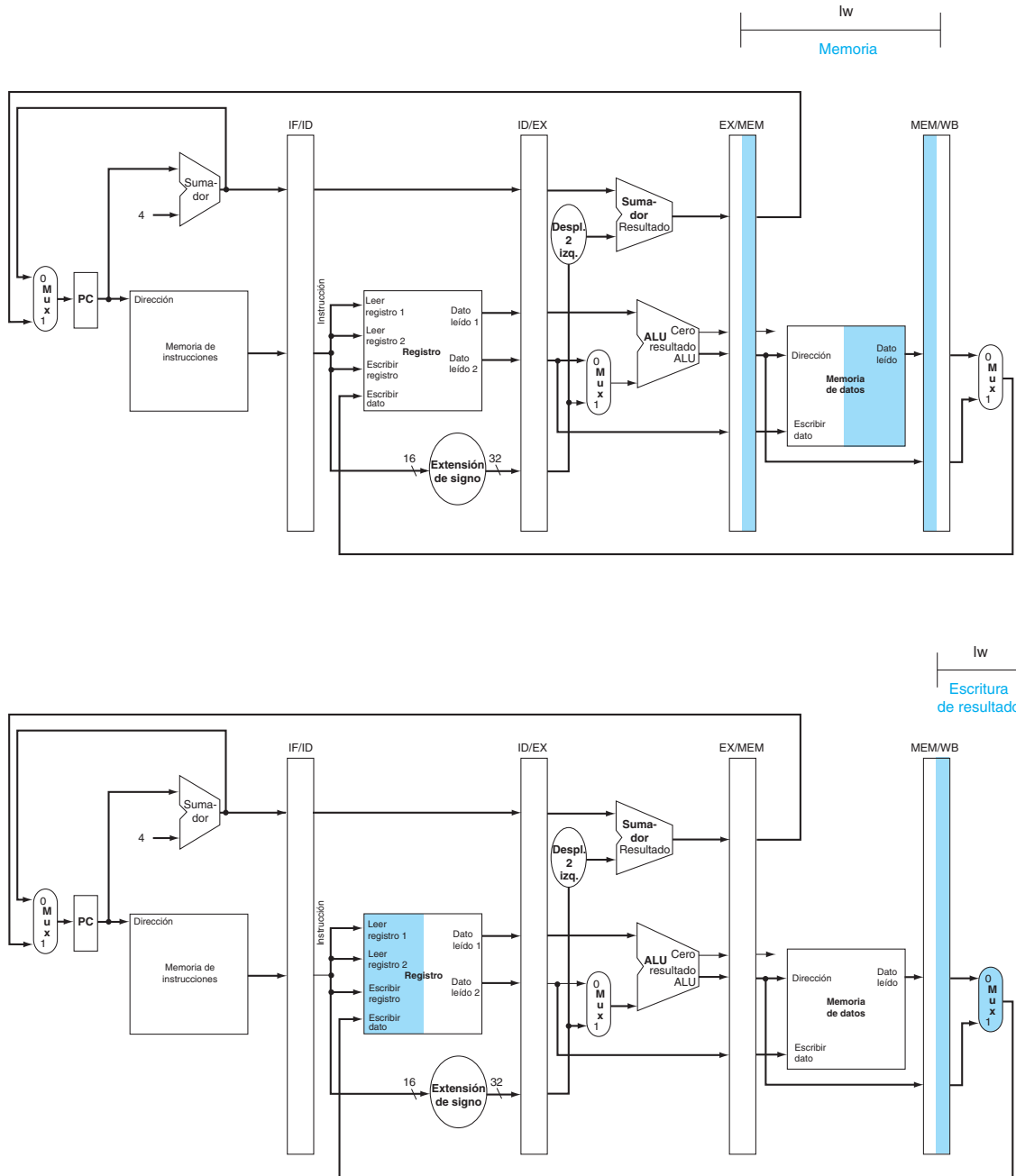


FIGURA 4.38 MEM y WB: cuarta y quinta etapas de la segmentación de una instrucción de carga, resaltando las partes del camino de datos de la figura 4.35 que se usan en esta etapa. Se lee la memoria de datos usando la dirección contenida en el registro EX/MEM, y el dato leído se guarda en el registro 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

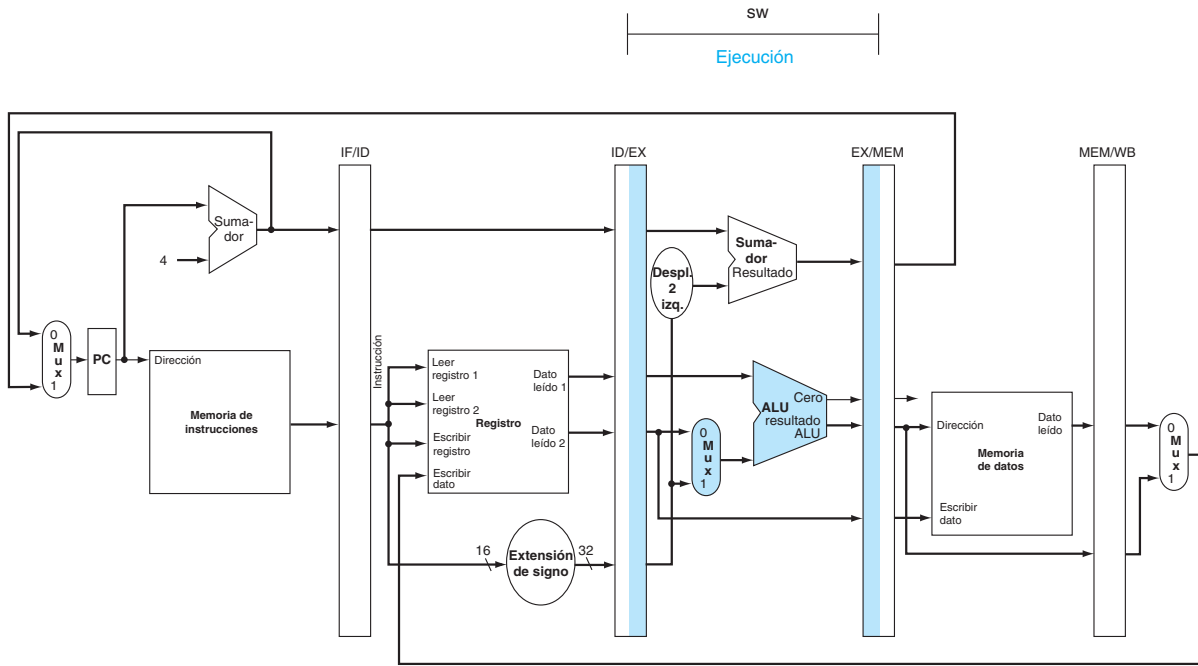


FIGURA 4.39 EX: tercera etapa de la segmentación de una instrucción de almacenamiento. A diferencia de la tercera etapa de la instrucción de carga en la figura 4.37, el valor del segundo registro es cargado en el registro de segmentación EX/MEM para ser usado en la siguiente etapa. Aunque no importaría 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.

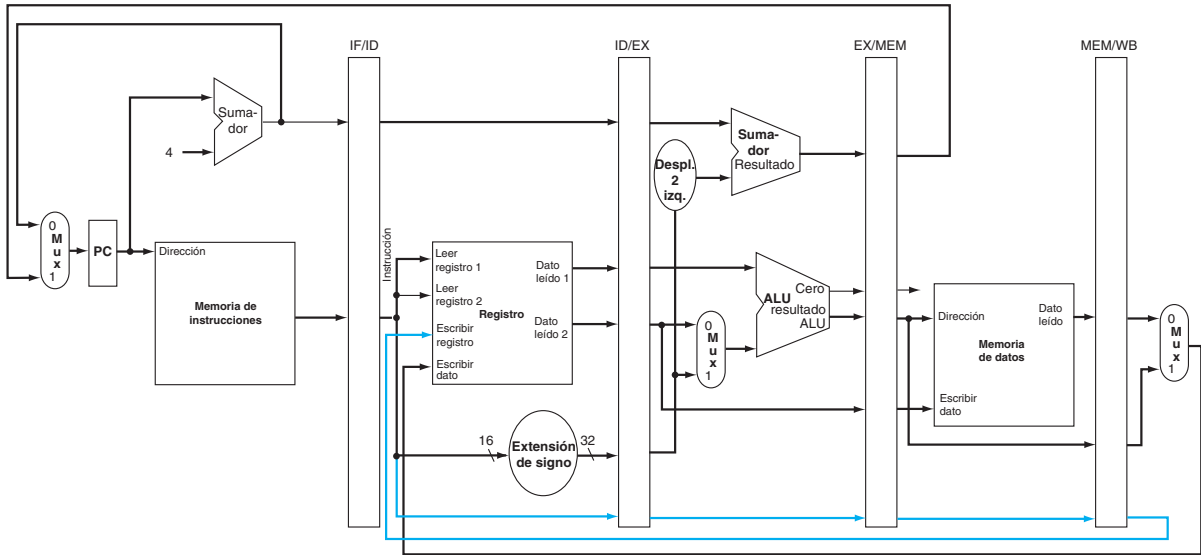


FIGURA 4.41 Camino de datos correctamente modificado para gestionar debidamente la instrucción de carga. Ahora el identificador del registro de escritura viene, junto con el dato a escribir, del registro segmentado MEM/WB. Este identificador se pasa desde la etapa ID hasta que llega al registro de segmentación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

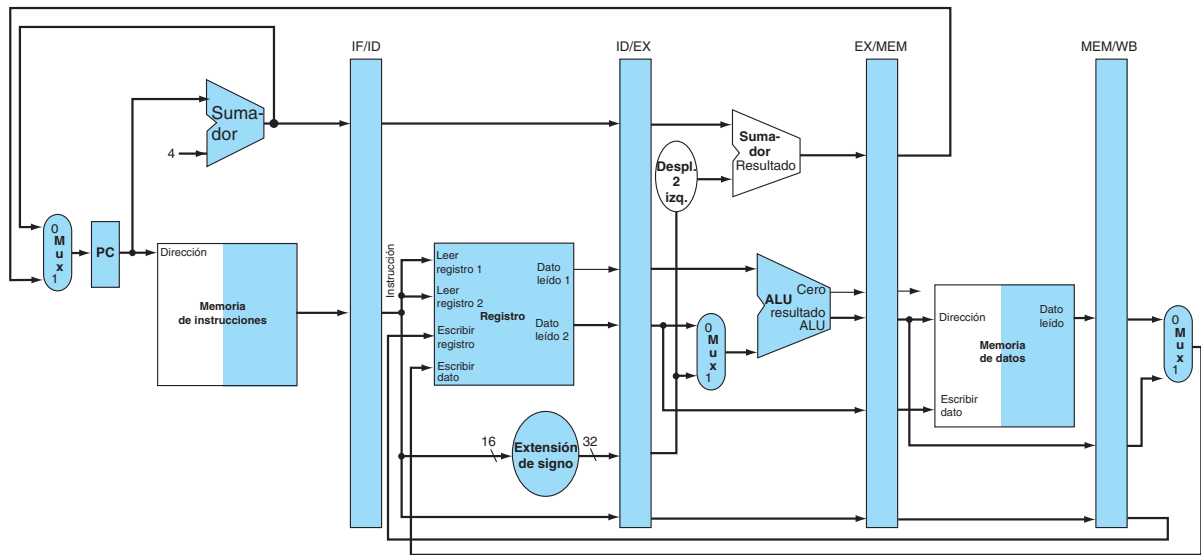


FIGURA 4.42 Porción del camino de datos de la figura 4.41 que es usada por las cinco etapas de la instrucción de carga.

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 la 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 simple, pero no contienen todos los detalles. Por ejemplo, consideremos la siguiente secuencia de cinco instrucciones:

```
lw      $t0, 20($t1)
sub     $t1, $t2, $t3
add     $t2, $t3, $t4
lw      $t3, 24($t1)
add     $t4, $t5, $t6
```

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.

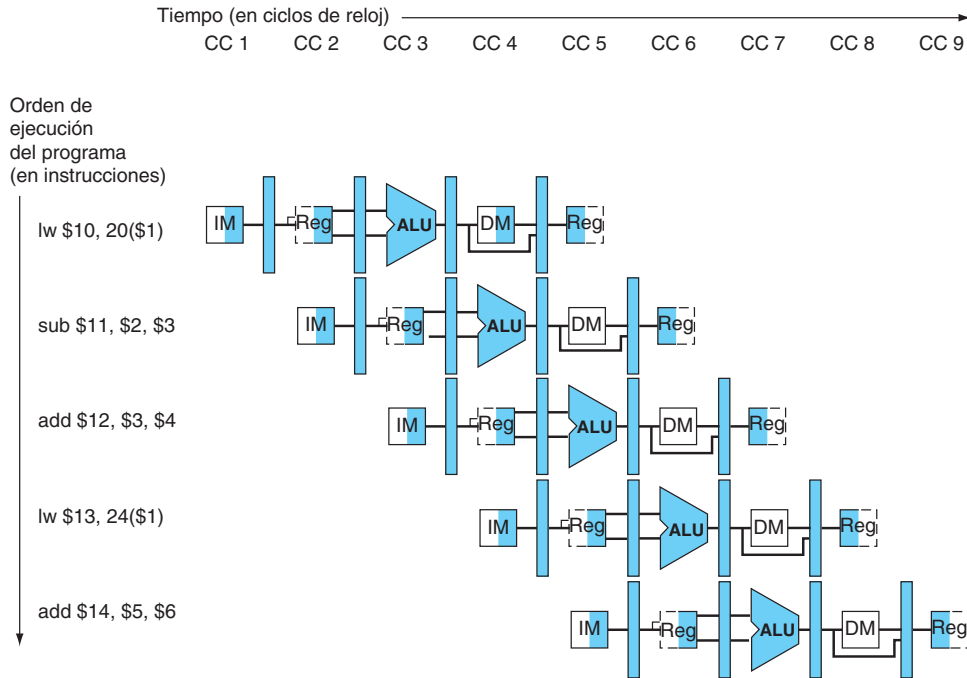


FIGURA 4.43 Diagrama multiciclo de la segmentación de cinco instrucciones. Este estilo de representación del pipeline muestra la ejecución completa de las instrucciones en una sola figura. La relación de instrucciones se hace en orden de ejecución desde la parte superior a la inferior, y los ciclos de 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.

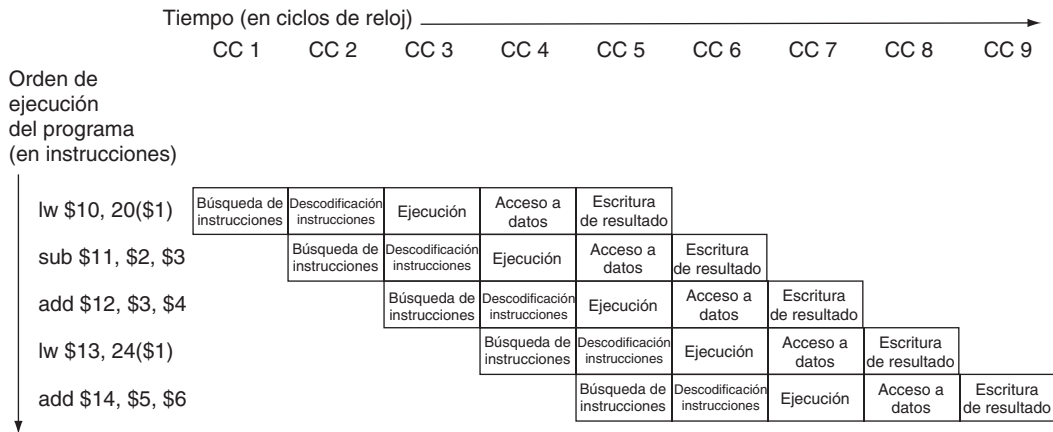


FIGURA 4.44 Versión tradicional del diagrama multiciclo de la segmentación de cinco instrucciones que se muestra en la figura 4.43.

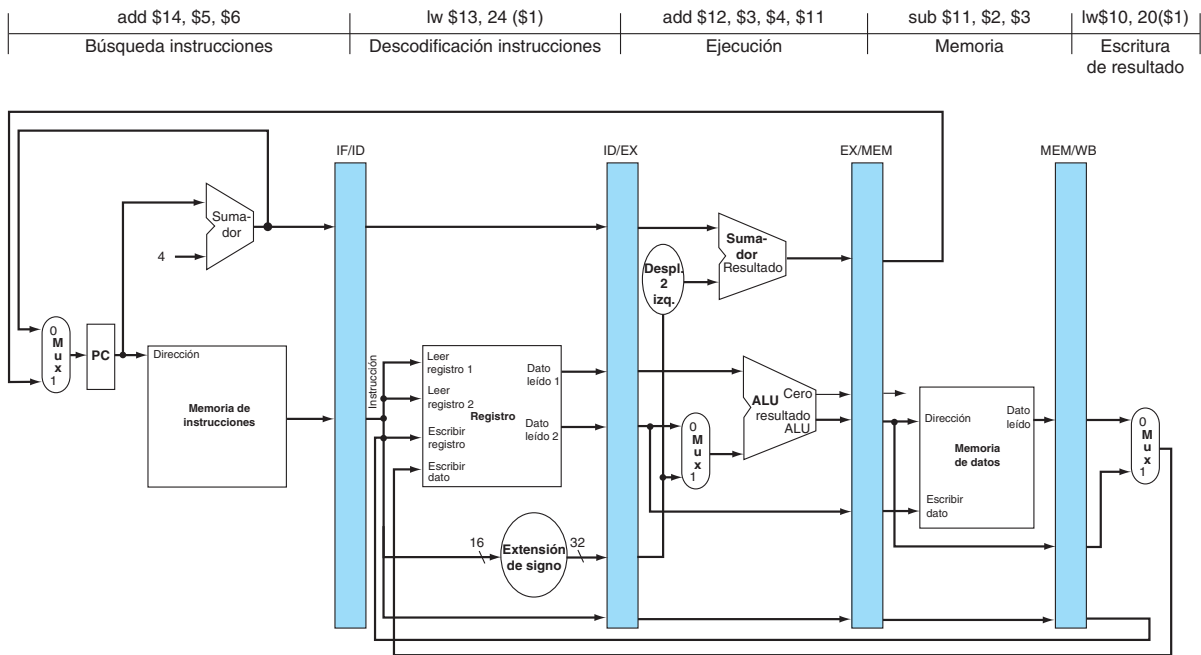


FIGURA 4.45 Diagrama monociclo correspondiente al ciclo 5 del pipeline de las figuras 4.43 y 4.44. Como puede verse, la figura monociclo es una porción vertical del diagrama multiciclo.

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

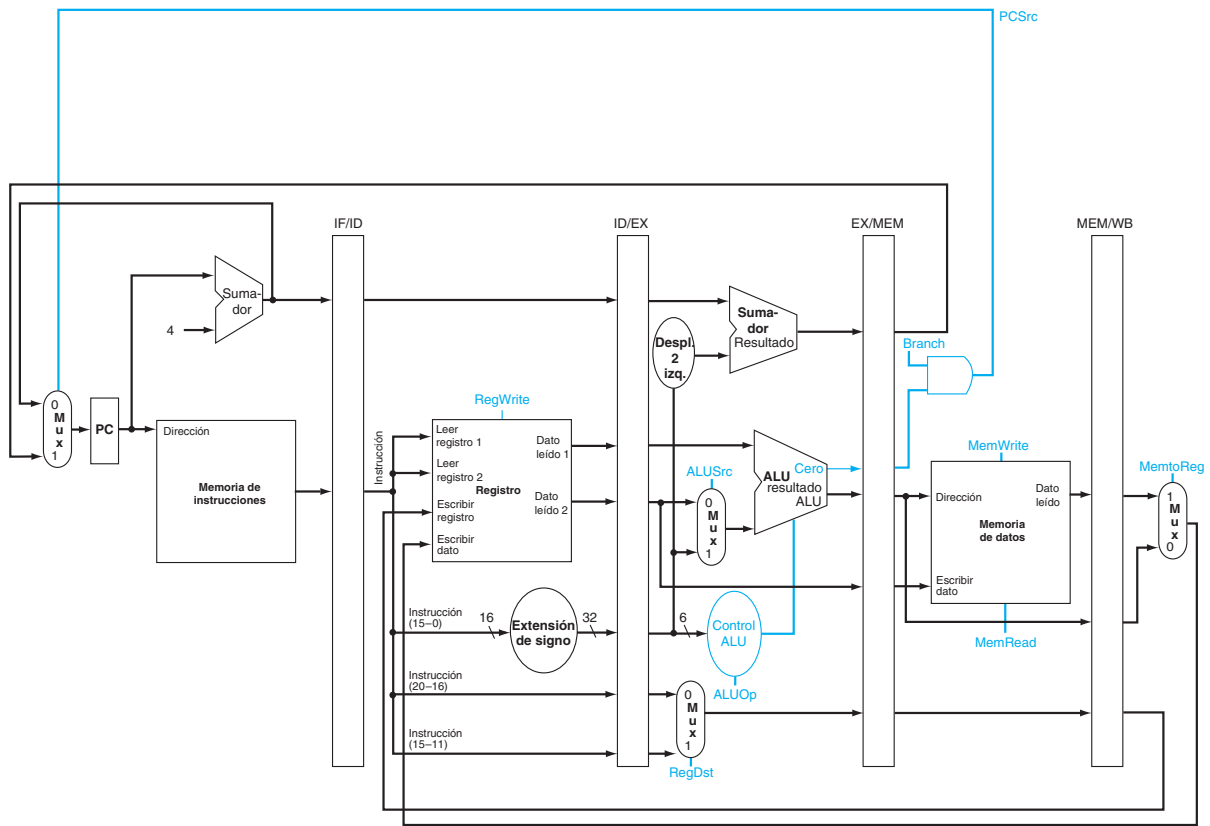


FIGURA 4.46 El camino de datos segmentado de la figura 4.41 en la que se identifican las señales de control. Este camino de datos toma prestada la lógica de control de la sección 4.4 para el PC, el identificador de registro fuente y destino, y el control de la ALU. Debe notarse que en la etapa EX, como 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.
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 4.48 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.

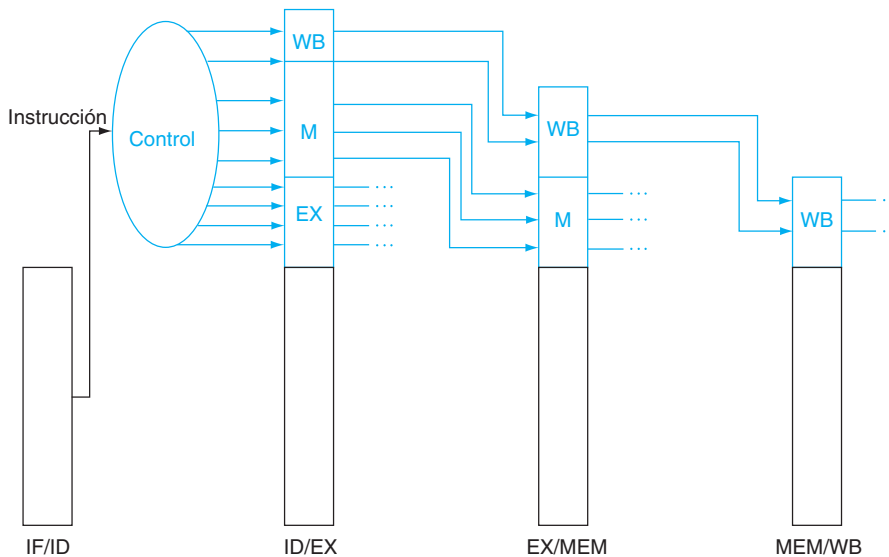


FIGURA 4.50 Líneas de control para las tres etapas finales. Observe que cuatro de las nueve líneas de control se usan en la etapa EX, mientras que las cinco restantes pasan al registro de segmentación EX/MEM, que ha sido extendido para poder almacenar las líneas de control; tres se usan durante la etapa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

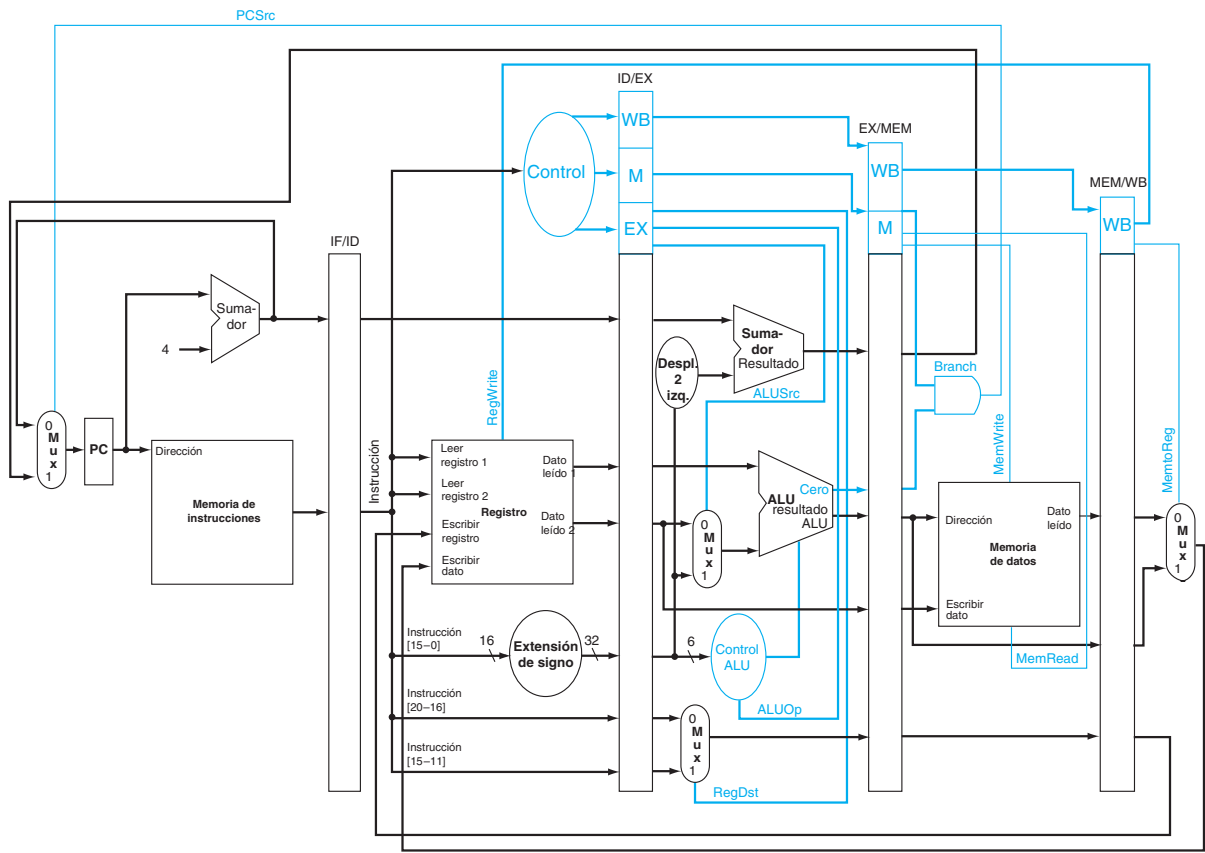


FIGURA 4.51 Camino de datos segmentado de la figura 4.40, con las señales de control conectadas a la parte de control de los registros de segmentación. Los valores de control de las tres últimas etapas se crean durante la decodificación de la instrucción y son escritos en el registro de segmentación 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

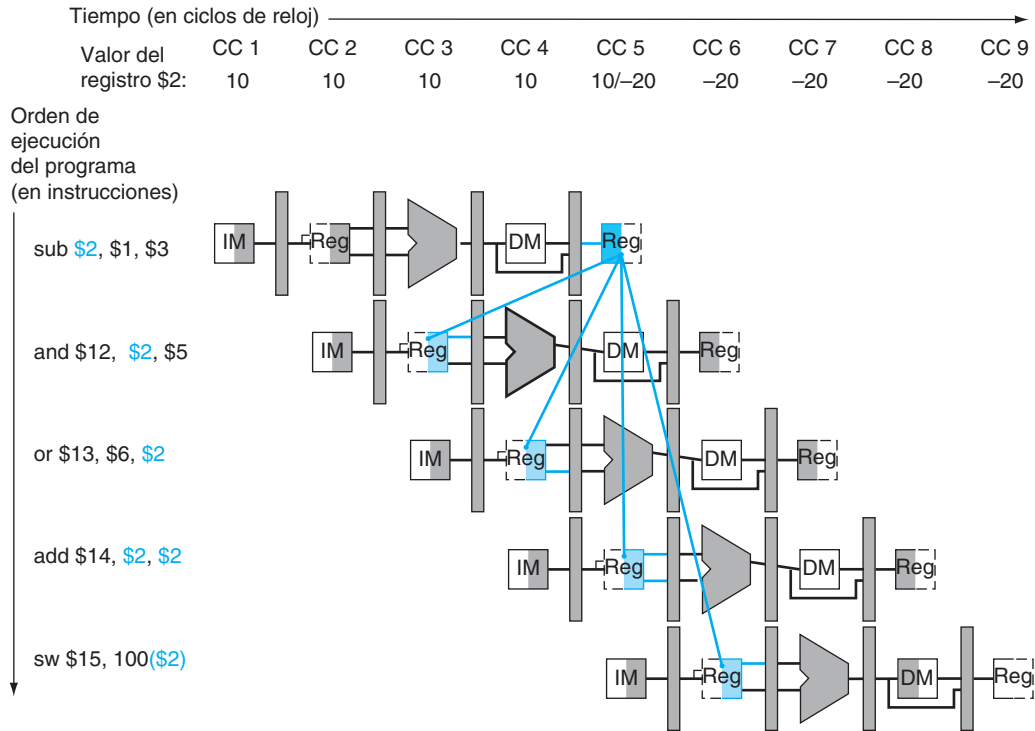


FIGURA 4.52 Dependencias en la segmentación de la ejecución de la secuencia de cinco instrucciones usando caminos de datos simplificados para mostrar las dependencias. Todas las acciones dependientes se muestran en color y “CC *i*” en la parte superior de la figura representa el ciclo de reloj *i*. La 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:

EX/MEM.RegisterRd = 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, `sll $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

```

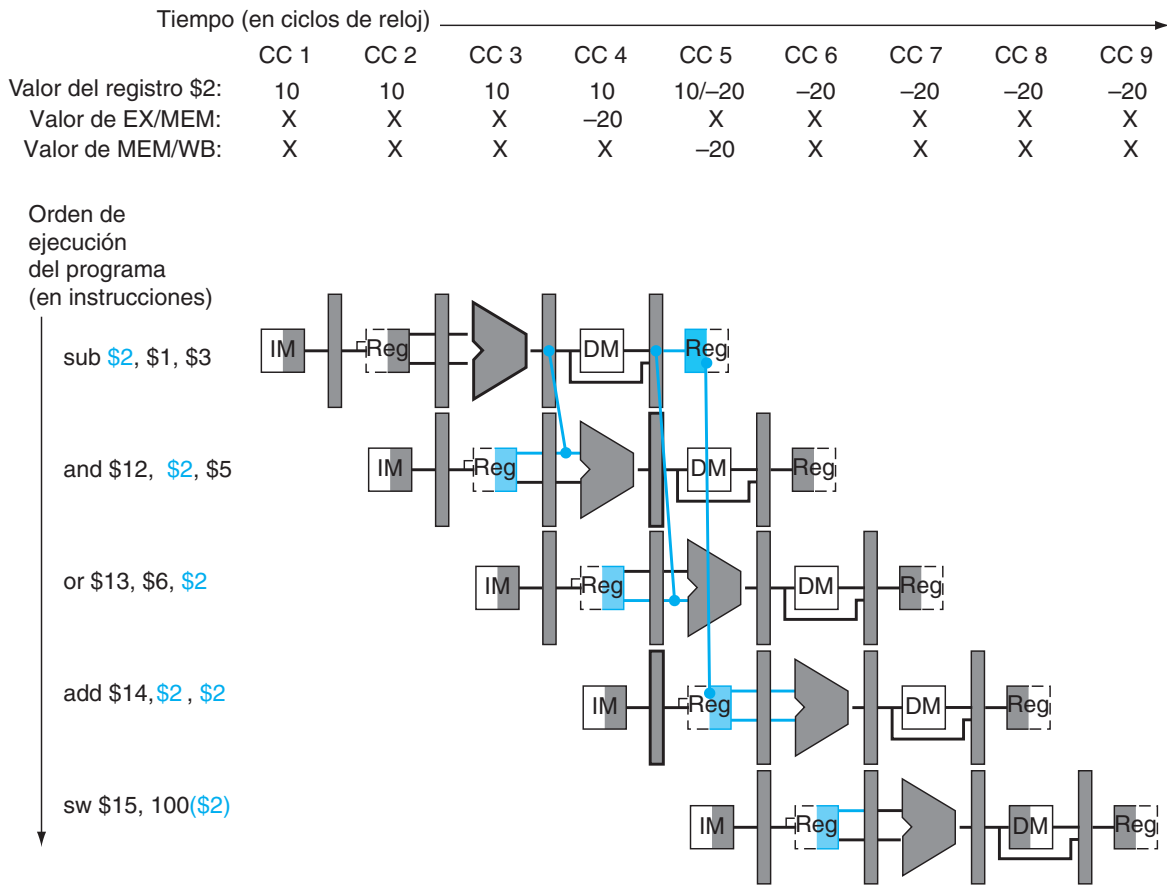
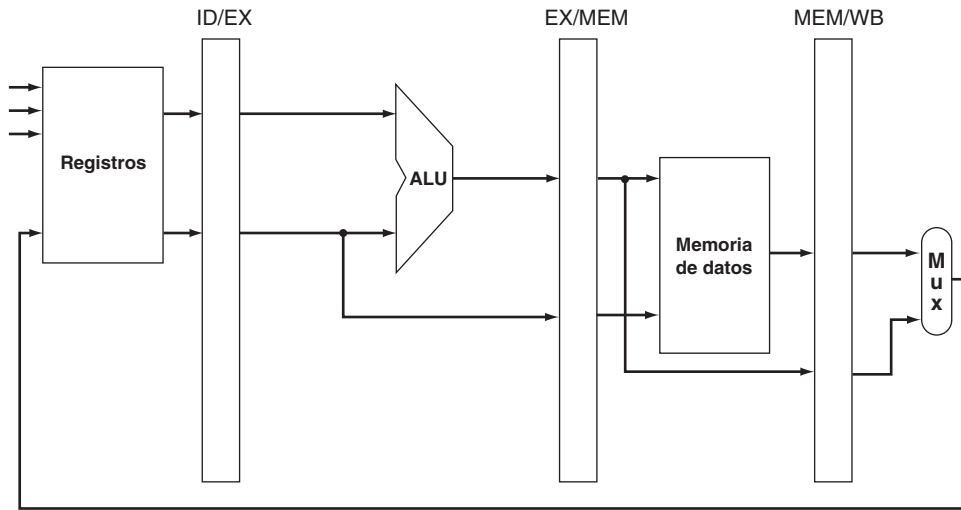


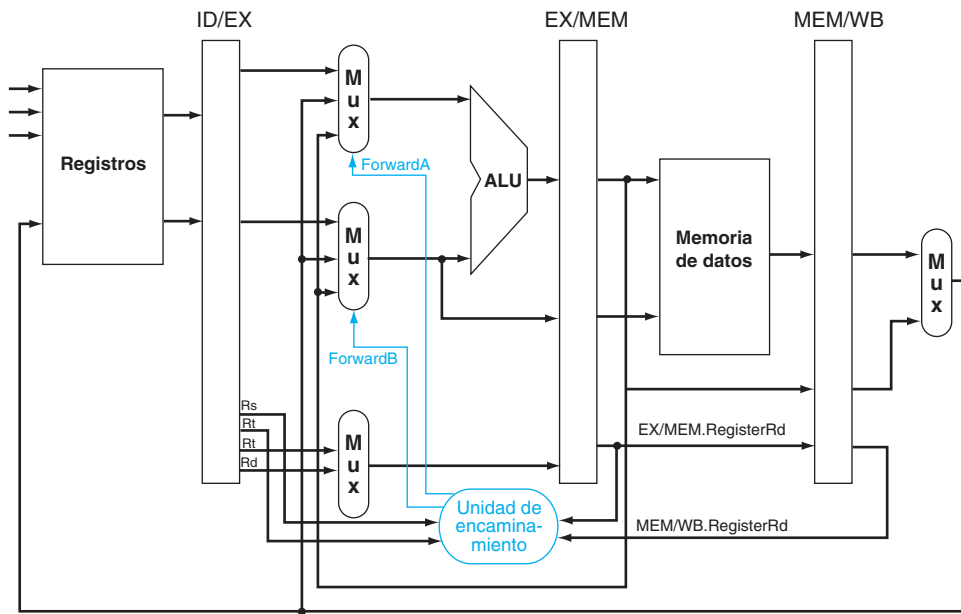
FIGURA 4.53 Las dependencias entre los registros de segmentación se mueven hacia adelante en el tiempo, por lo que es posible proporcionar a la ALU los operandos que necesitan las instrucciones and y or anticipando los resultados que se encuentran en dichos registros. Los valores en los registros de 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.

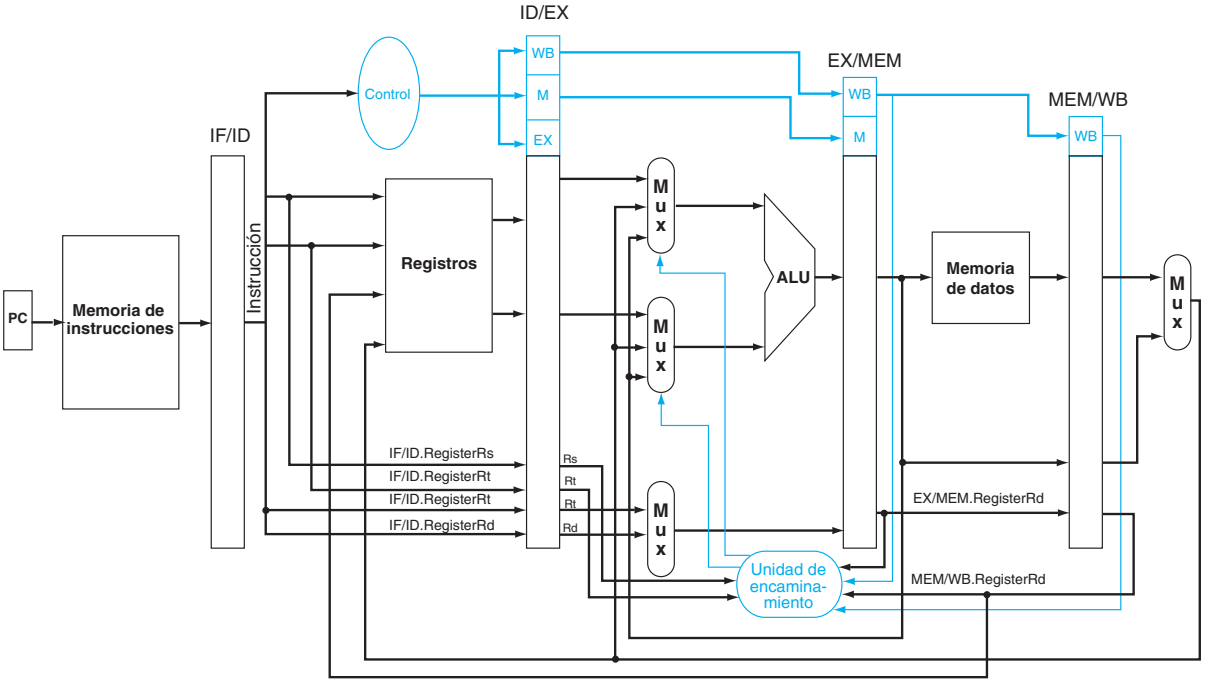


FIGURA 4.56 Camino de datos modificado para resolver los riesgos a través la anticipación de resultados. Comparado con el camino de datos de la figura 4.51, los cambios adicionales son los multiplexores en las entradas de la ALU. Sin embargo, esta figura simplificada deja fuera detalles del camino de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

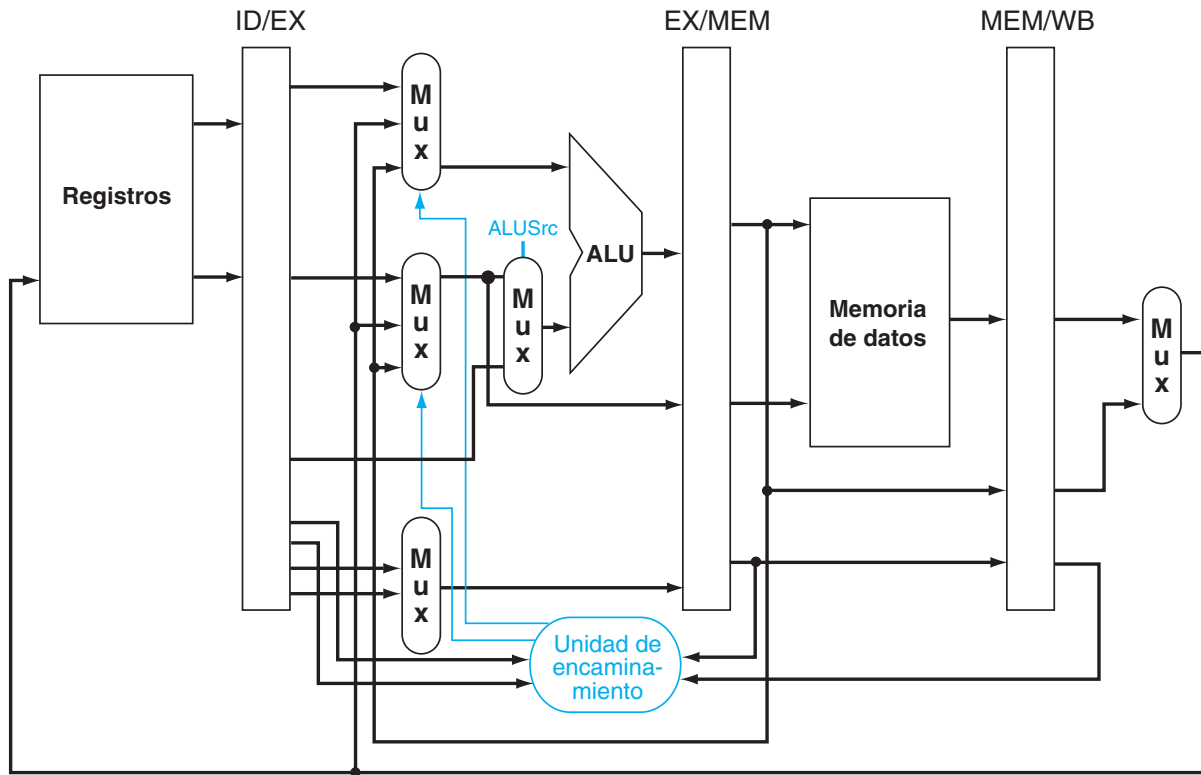


FIGURA 4.57 El zoom del camino de datos de la figura 4.54 muestra un multiplexor 2:1, añadido para seleccionar como entrada de la ALU al valor inmediato con signo.

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
  
```

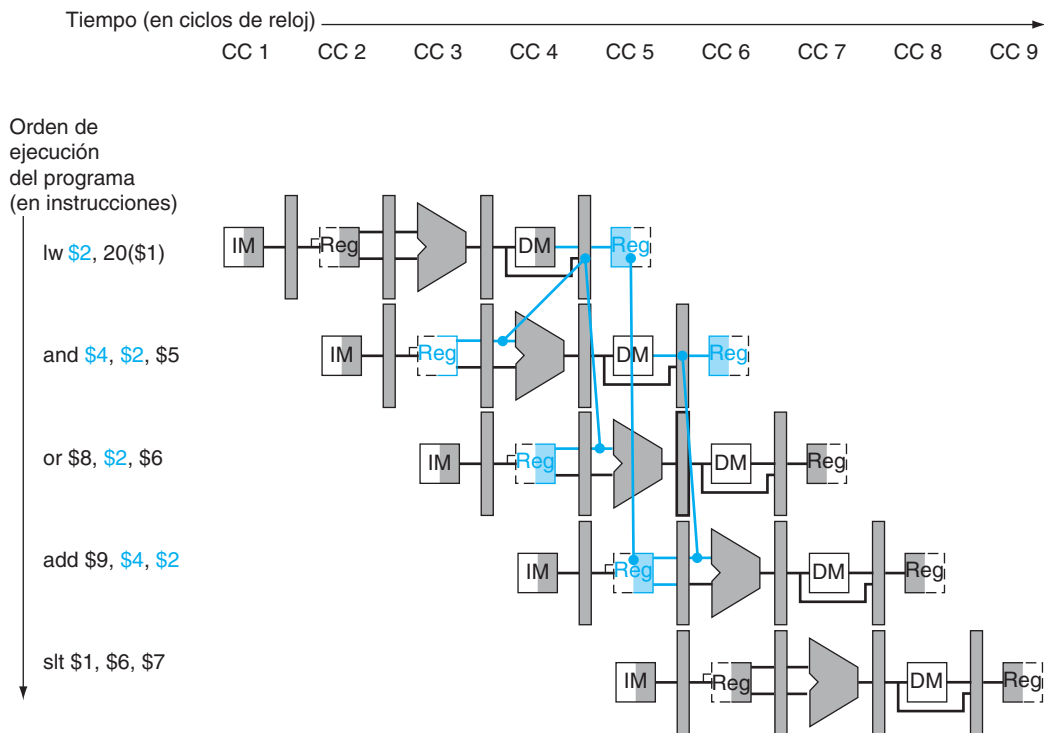


FIGURA 4.58 Secuencia de instrucciones segmentadas. Como la dependencia entre la carga y la instrucción siguiente (*and*) va hacia atrás en el tiempo, este riesgo no puede ser resuelto mediante la anticipación de datos. Por tanto, esta combinación debe resultar en un bloqueo generado por la unidad de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

mentado. 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.

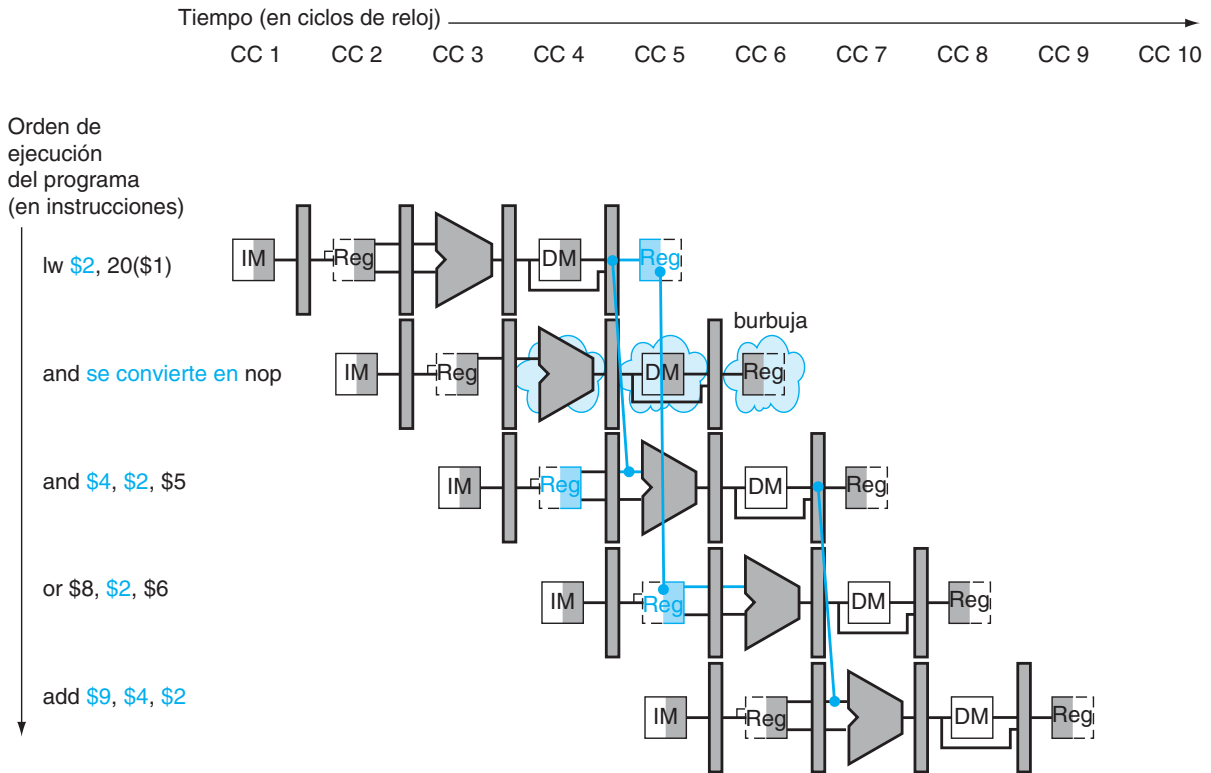


FIGURA 4.59 Como se insertan realmente los bloqueos en el pipeline. Se inserta una burbuja al principio del ciclo 4 cambiando la instrucción **and** por un **nop**. Observe que la instrucción **and** es realmente buscada y descodificada en los ciclos 2 y 3, pero su etapa EX es retrasada hasta el ciclo 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.

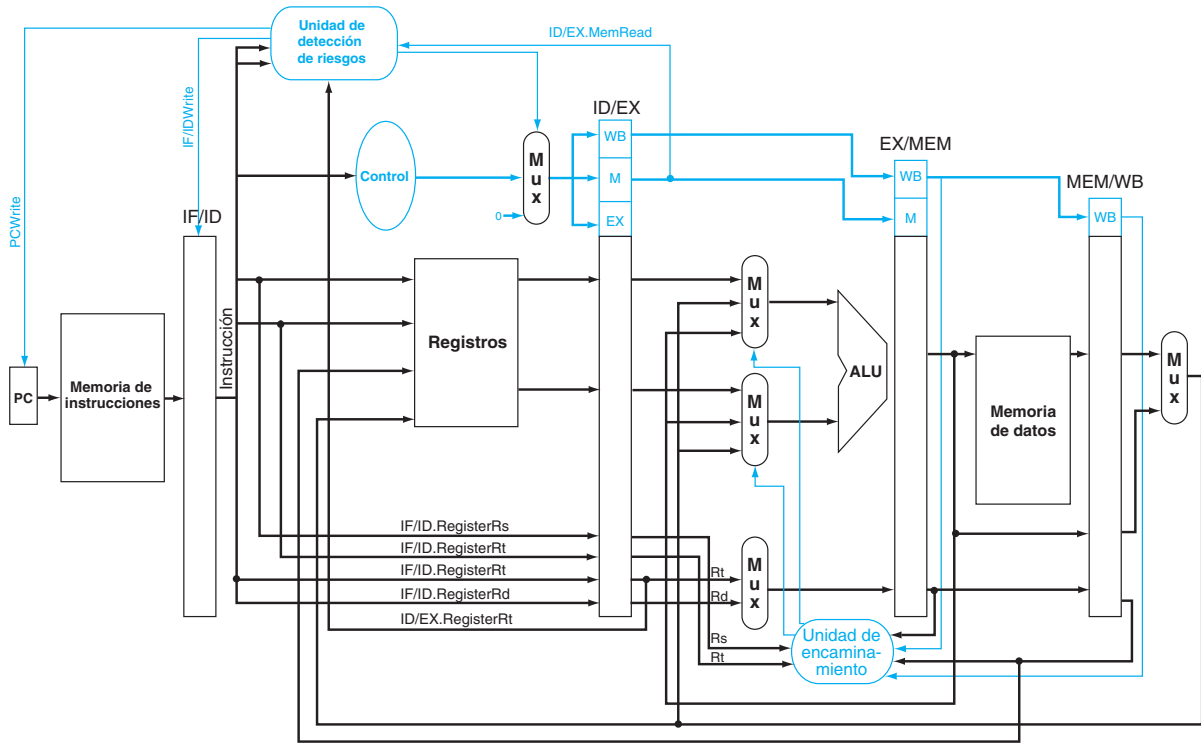


FIGURA 4.60 Perspectiva general del control de la segmentación, mostrando los dos multiplexores para la anticipación de datos, la unidad de detección de riesgos y la unidad de anticipación de datos. A pesar de que las etapas ID y EX se han simplificado (faltan la extensión de signo del valor inmediato y lógica de saltos) este dibujo muestra los requerimientos esenciales para el hardware de anticipación de datos.

Extensión: Con respecto al comentario anterior acerca de poner las líneas de control a 0 para evitar la escritura sobre registros o memoria: sólo se necesita que valgan 0 las señales **RegWrite** y **MemWrite**, mientras que el resto de las señales de control pueden tener un valor “indiferente”.

4.8 Riesgos de control

*Hay miles que atacan las
ramas del mal por cada
uno que golpea en su raíz.*

Henry David Thoreau,
Walden, 1854

Hasta ahora hemos centrado nuestra atención en los riesgos que implican operaciones aritméticas y de transferencia de datos. Pero como ya vimos en la sección 4.5, también existen riesgos en el pipeline en los que están involucrados los saltos. La figura 4.61 muestra una secuencia de instrucciones y señala qué pasa cuando entra un salto en el pipeline. En cada ciclo de reloj se debe buscar una nueva instrucción para poder mantener el pipeline ocupado, pero en nuestro

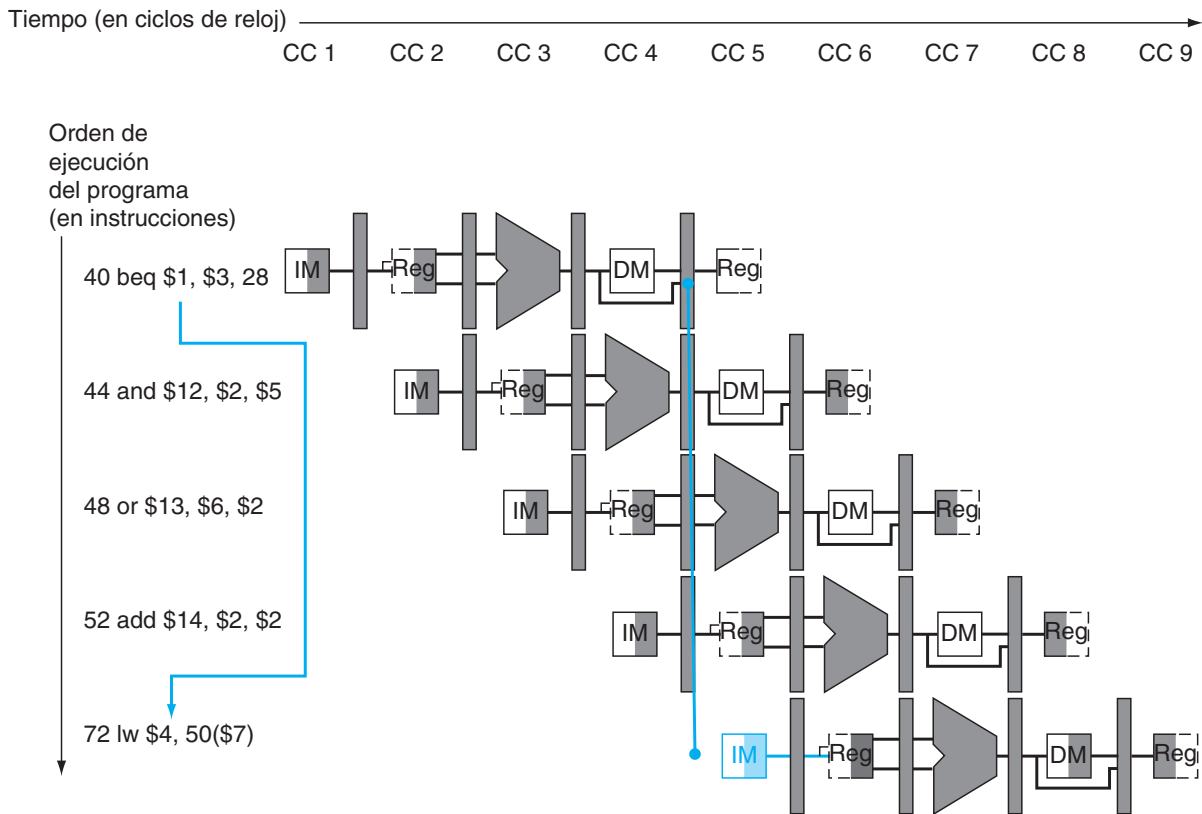


FIGURA 4.61 Impacto de la segmentación en las instrucciones de salto. Los números a la izquierda de las instrucciones (40, 44,...) son sus direcciones. Como una instrucción de salto no toma la decisión de saltar o no saltar hasta la etapa MEM (ciclo 4 para la instrucción beq de arriba), las tres instrucciones que siguen al salto serán leídas de memoria y se empezará su ejecución. Si no se interviene de algún modo, esas tres instrucciones empezarán la ejecución antes que beq salte a la instrucción lw de la posición 72. (En la figura 4.31 se suponía la existencia de circuitería adicional para reducir los riesgos de control a un único ciclo; esta figura utiliza el camino de datos original, sin optimizar).

diseño no se conoce la decisión del salto hasta la etapa MEM. Como se ha mencionado en la sección 4.5, este retardo a la hora de determinar la instrucción correcta que se tiene que buscar se llama riesgo de control o riesgo de saltos, en contraposición con los riesgos de datos que se acaban de estudiar.

Esta sección, que trata sobre los riesgos de control, es más corta que las secciones previas sobre los riesgos de datos. Las razones son que los riesgos de control son relativamente más simples de entender, que aparecen con menos frecuencia que los riesgos de datos y que no existe un mecanismo tan efectivo contra los riesgos de control como lo es la anticipación de resultados para los riesgos de datos. Por tanto, se usarán esquemas más simples. Se estudiarán dos esquemas para resolver los riesgos de control y una optimización para mejorar estos esquemas.

Suponer el salto como no tomado

Como vimos en la sección 4.5, bloquear el procesador hasta que se complete el salto es demasiado lento. Para evitar el bloqueo en los saltos, habitualmente se realiza una mejora que es suponer que el salto no será tomado y por lo tanto la ejecución continúa siguiendo el flujo secuencial de instrucciones. Si el salto es finalmente tomado, se deberá descartar las instrucciones que se están buscando y descodificando en ese momento, y la ejecución continuará a partir del destino del salto. Si la mitad de las veces los saltos son no tomados, y si descartar instrucciones cuesta poco, esta optimización reduce el coste de los riesgos de control a la mitad.

Para descartar instrucciones, basta con cambiar el valor de las señales de control originales y ponerlas a cero, de una forma muy parecida a como se bloquearon las instrucciones para evitar el riesgo de datos de tipo *load-use*. La diferencia estriba en que cuando el salto llega a la etapa de MEM se deben cambiar las señales de control para tres instrucciones, las que hay en las etapas IF, ID y EX; en cambio, en el caso de los bloqueos por uso del dato de una carga, sólo se ponía el control a 0 en la etapa ID y se le dejaba propagar por el pipeline. Por lo tanto, en este caso **descartar instrucciones (*flush instructions*)** significa desechar las instrucciones que haya en las etapas IF, ID y EX.

Reducción del retardo de los saltos

Una manera de mejorar el rendimiento de los saltos es reducir el coste de los saltos tomados. Hasta ahora se ha supuesto que el siguiente valor de PC en el caso de un salto se selecciona en la etapa de MEM, pero si se mueve la ejecución del salto a una etapa anterior del pipeline, entonces se tienen que eliminar menos instrucciones. La arquitectura MIPS fue diseñada para dar soporte a saltos rápidos de un único ciclo de reloj que pudieran ser segmentados con una penalización pequeña. Los diseñadores observaron que muchos de los saltos dependían de comprobaciones muy simples (igualdad o signo, por ejemplo) y que estas comprobaciones no requieren una operación completa en la ALU y pueden hacerse con muy pocas puertas lógicas. Cuando se necesita tomar una decisión de salto más compleja, entonces se debe utilizar una instrucción distinta que realice la comparación requerida mediante la ALU; una situación que es similar al uso de códigos de condición para saltos (véase capítulo 2).

Mover la decisión del salto hacia arriba en el pipeline requiere que dos acciones se realicen antes: el cómputo de la dirección destino del salto, y la evaluación de la decisión de salto. La parte más sencilla de este cambio es adelantar el cálculo de la dirección de salto. En el registro IF/ID ya se dispone del valor del PC y del campo inmediato, por lo que basta sólo con mover el sumador de direcciones desde la etapa MEM a la etapa ID; por supuesto, aunque el cálculo de la dirección destino del salto se haga para todas las instrucciones, sólo se usará cuando realmente sea necesario.

La parte más complicada es la propia decisión de saltar. Para la comprobación de “saltar si igual” se deben comparar los dos registros leídos durante la etapa ID y ver si son iguales. La igualdad se puede verificar haciendo la operación *or-exclusiva* de los bits respectivos de cada uno de los valores leídos y haciendo la operación *OR* con estos resultados. Mover la verificación del salto a la etapa ID implica la necesidad de circuitería nueva para la detección de riesgos y para la anticipación de resultados, ya que una instrucción *branch* que depende de un resultado previo que está todavía en el pipeline debe seguir funcionando correctamente aunque se implemente la optimización. Por ejemplo, para

Descartar instrucciones: descartar instrucciones de un pipeline, generalmente debido a un suceso imprevisto.

implementar la operación “saltar si igual” (y su inversa), se necesitará anticipar los resultados de una instrucción anterior a la lógica de comprobación que opera durante la etapa ID. Existen dos factores que complican las cosas:

1. Durante ID se debe descodificar la instrucción, decidir si se necesita la anticipación a la unidad de comprobación de igualdad, y completar esta comprobación para que si la instrucción es un salto entonces se pueda modificar el registro PC con la dirección destino del salto. La anticipación de los operandos de los saltos era anteriormente manejado por la lógica de anticipación de resultados de la ALU, pero la introducción de la unidad de comprobación de igualdad en la etapa ID requiere su propia lógica de anticipación de resultados. Observe que los operandos fuente que son anticipados a un salto pueden proceder tanto del registro de segmentación EX/MEM como del MEM/WB.
2. Puesto que los valores de una comparación de salto se necesitan en ID pero pueden ser producidos más tarde, es posible que ocurra un riesgo de datos y sea necesario un bloqueo. Por ejemplo, si una instrucción de ALU a la que le sigue inmediatamente una instrucción de salto produce uno de los operandos que necesita la comparación del salto, entonces se necesitará un bloqueo, ya que la etapa EX de la instrucción de tipo ALU se realiza después del ciclo ID que corresponde al salto. Por extensión, cuando a una carga le sigue inmediatamente un salto condicional que chequea el resultado de la carga, se produce un bloqueo durante dos ciclos, porque el resultado de la carga no está disponible hasta el final del ciclo MEM, pero se necesita al principio del ciclo ID del salto.

A pesar de estas dificultades, mover la ejecución de los saltos a la etapa ID es una mejora, ya que cuando el salto es tomado reduce la penalización de los saltos a solamente una instrucción, la que se está buscando en memoria en ese momento. Los ejercicios exploran los detalles de implementación de la ruta de anticipación de resultados y de la detección del riesgo de control.

Para eliminar instrucciones en la etapa de búsqueda (IF), se añade una línea de control llamada IF.Flush, la cual pone a cero el campo de instrucción del registro de segmentación IF/ID. Al poner este registro a cero se transforma la instrucción leída en una instrucción *nop*, una instrucción que no realiza ninguna acción ni cambia ningún estado.

EJEMPLO

Salto segmentado

Mostrar qué ocurre cuando en la siguiente secuencia de instrucciones el salto es tomado, suponiendo que el pipeline está optimizado para los saltos no tomados y que se ha movido la ejecución del salto a la etapa ID:

```

36 sub $10, $4, $8
40 beq $1, $3, 7 # salto relativo a PC a 40+4+7*4=72
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7

72 lw $4, 50($7)

```

RESPUESTA

La figura 4.62 muestra lo que ocurre cuando el salto es tomado. Al contrario de la figura 4.61, al tomar el salto sólo aparece una burbuja en el pipeline.

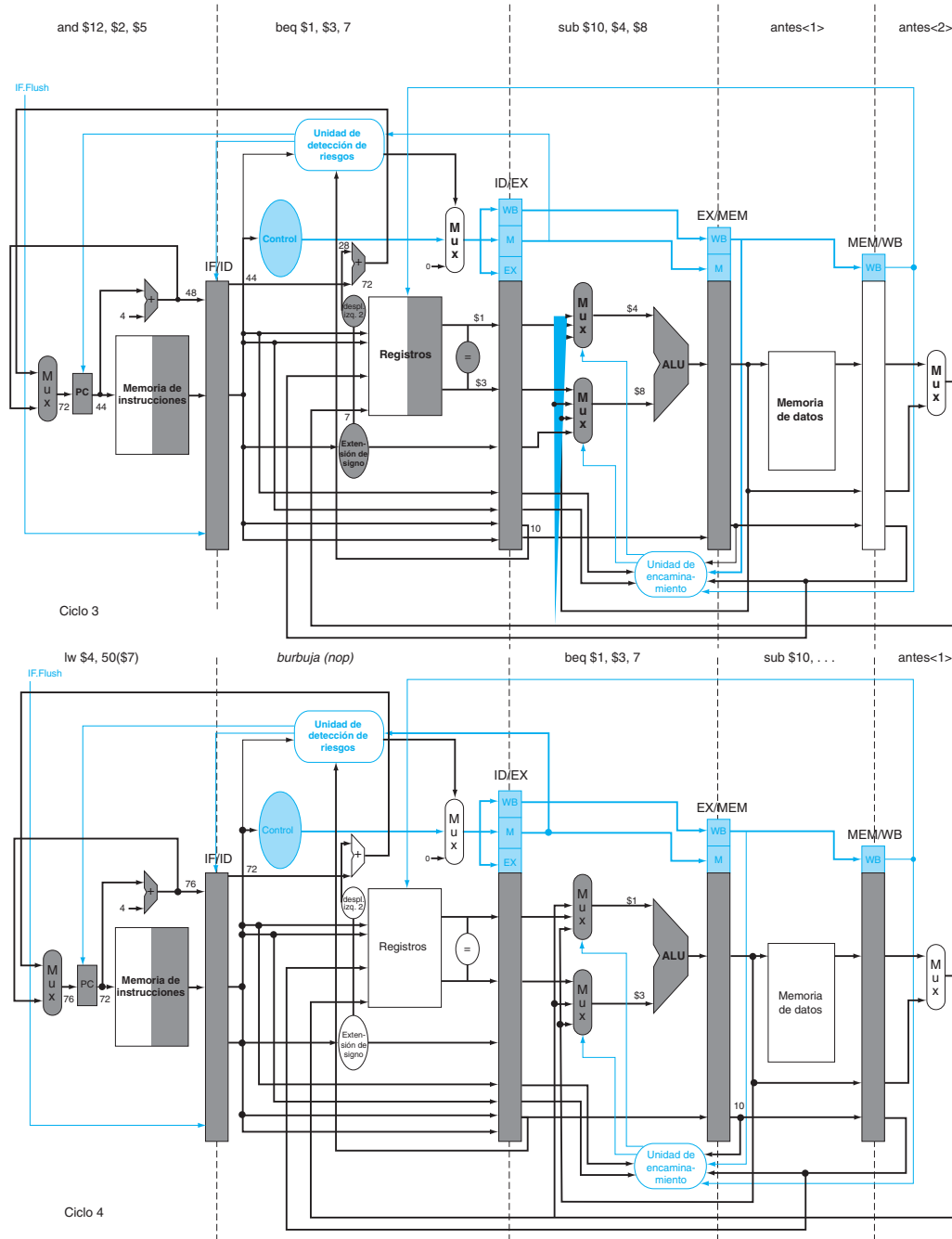


FIGURA 4.62 La etapa ID del ciclo 3 determina si el salto se debe tomar, de modo que selecciona el valor 72 como siguiente PC y pone a cero la instrucción que ha sido buscada para ser descodificada en el siguiente ciclo. El ciclo 4 muestra que se está buscando la instrucción situada en la posición 72 y que ha aparecido una burbuja o `nop` en el pipeline como resultado del salto tomado. (Ya que en realidad la instrucción `nop` corresponde con la instrucción `sl $0, $0, $0` es discutible si en el ciclo 4 se debería o no resaltar la etapa ID)

Predicción dinámica de saltos

Suponer que los saltos no son tomados es una forma simple de *predicción de saltos*. En este caso, la predicción es que los saltos no se toman, y se vacía el pipeline cuando la predicción es incorrecta. En el caso del pipeline simple de cinco etapas, esta estrategia, en muchas ocasiones realizada en colaboración con el compilador, es probablemente adecuada. En cambio, con pipelines más profundos (mayor número de etapas), la penalización de los fallos de la predicción de saltos, si se mide en ciclos de reloj, es mayor. De modo similar, la ejecución múltiple de instrucciones (véase la sección 4.10) incrementa la penalización en términos de oportunidades perdidas de ejecución de instrucciones. Esta combinación significa que en un diseño más agresivo del pipeline, un esquema simple de predicción estática no alcanzará unas buenas prestaciones. Tal como se mencionó en la sección 4.5, dedicando más hardware es posible tratar de predecir el comportamiento de los saltos durante la propia ejecución del programa.

Una estrategia es fijarse en la dirección de las instrucciones para ver si se trata de un salto que fue tomado la última vez que la instrucción se ejecutó, y en ese caso, comenzar a buscar la siguiente instrucción en el mismo sitio que se hizo la vez anterior. Esta técnica se denomina **predicción dinámica de saltos (*dynamic branch prediction*)**.

Una manera de realizar esta estrategia es utilizar un **búfer de predicción de saltos (*branch prediction buffer*)** o una **tabla de historia de saltos (*branch history table*)**. Un búfer de predicción de saltos es una memoria pequeña indexada con la parte menos significativa de la dirección de la instrucción de salto. Esta memoria contiene un bit en cada entrada que dice si el salto ha sido recientemente tomado o no.

Esta es la clase más sencilla de búfer: en realidad no se sabe si la predicción es la correcta, ya que esa entrada podría contener la información de otro salto que coincida en los bits menos significativos de la dirección. Pero este problema no afecta a la corrección del mecanismo. La predicción es simplemente una apuesta que se supone que va a ser acertada, así que la búsqueda se inicia en la dirección predicha. Si la apuesta finalmente resulta ser falsa, entonces se eliminan las instrucciones predichas de forma incorrecta, se invierte el bit de predicción y se vuelve a guardar en la tabla, y a continuación se busca y ejecuta la secuencia correcta de instrucciones.

Este sencillo mecanismo de predicción basado en 1 bit tiene una desventaja en cuanto a las prestaciones: aunque un salto sea tomado prácticamente siempre, es muy probable que cada vez que no sea tomado se hagan dos predicciones incorrectas en lugar de una. El siguiente ejemplo muestra este dilema.

Predicción dinámica de saltos: predicción de saltos en tiempo de ejecución que usa la información que se va generando durante la ejecución.

Búfer de predicción de saltos (tabla de historia de saltos): pequeña memoria que se indexa con la porción menos significativa de los bits de la dirección de la instrucción de salto y que contiene uno o más bits que indican si el salto ha saltado recientemente o no.

Lazos y predicción

Considere un salto de final de lazo que salta nueve veces seguidas para iterar dentro del lazo hasta que la última vez deja de saltar para salir del lazo. ¿Cuál es la precisión de la predicción para este salto, suponiendo que el bit de predicción de este salto se mantiene en el búfer?

EJEMPLO

RESPUESTA

Durante el estado estable de la ejecución del programa, la predicción siempre fallará tanto en la primera como en la última iteración del lazo. Fallar en la última iteración es inevitable, ya que el bit de predicción siempre indicará que se debe tomar el salto, ya que en ese momento el salto ha sido tomado nueve veces seguidas. El fallo de predicción en la primera iteración del lazo se debe a que el bit se cambió justo al ejecutar el salto en la última iteración del lazo, ya que para salir del lazo el salto tuvo que ser no tomado. Por lo tanto, la precisión de la predicción de este salto, que es tomado el 90% de las veces, es sólo del 80% (dos predicciones incorrectas y ocho correctas).

Idealmente, para estos saltos tan regulares, la precisión del predictor tendría que ser igual a la frecuencia de saltos tomados. Para remediar el punto débil de este esquema con frecuencia se usan esquemas de predicción de 2 bits. Usando este esquema, la predicción ha de ser incorrecta 2 veces seguidas antes de cambiarla. La figura 4.63 muestra la máquina de estados finitos para un esquema de predicción de 2 bits.

Un búfer de predicción de saltos se puede implementar como un pequeño búfer especial al que se accede con la dirección de la instrucción durante la etapa IF. Si se predice la instrucción como tomada, la búsqueda de instrucciones empieza en la dirección destino del salto tan pronto como se conozca el PC; y esto, tal como se menciona en la página 377, puede ser como muy pronto en la etapa ID. En caso contrario, la búsqueda y ejecución secuencial de instrucciones continúa. Si la predicción resulta ser incorrecta, se cambian los bits de predicción tal como se muestra en la figura 4.63.

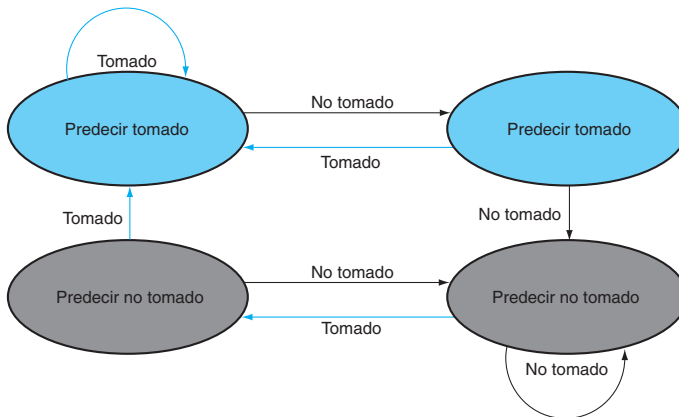


FIGURA 4.63 Estados en el esquema de predicción de 2 bits. Usando 2 bits en vez de 1, un salto que tiene una gran tendencia o bien a ser tomado o bien a ser no tomado (como hacen muchos saltos) será predicho incorrectamente sólo una vez. Los dos bits se utilizan para codificar los cuatro estados del sistema. El esquema de dos bits en un ejemplo general de los predictores basados en un contador, que se incrementa cuando la predicción es adecuada y se decrementa en el caso contrario, y el punto medio del rango de valores que puede tomar el contador se utiliza para tomar la decisión de predicción.

Extensión: Tal como se describió en la sección 4.5, en un pipeline de cinco etapas se puede evitar el problema de los riesgos de control redefiniendo el salto. Se define el salto retardado de forma que siempre se ejecuta la instrucción que hay después del salto, y es la segunda instrucción que sigue al salto la que será afectada por el salto.

Los compiladores y ensambladores intentan colocar después del salto, en el **hueco del salto retardado (branch delay slot)**, una instrucción que siempre se deba ejecutar. La tarea del software es conseguir que las instrucciones que suceden a los saltos retardados sean

Hueco de salto retardado: hueco que se encuentra directamente después de una instrucción de salto retardado, y que en la arquitectura MIPS es rellenado por una instrucción que no afecta al salto.

tanto válidas como útiles. La figura 4.64 muestra tres maneras de planificar el uso del hueco del salto retardado.

Las limitaciones de la planificación del uso de los saltos retardados provienen de (1) las restricciones en las instrucciones que se colocan en el hueco del salto retardado y de (2) la capacidad que se tiene en tiempo de compilación para predecir si un salto será tomado o no.

La utilización de saltos retardados fue una solución simple y efectiva para los procesadores segmentados en cinco etapas que ejecutaban una instrucción por ciclo. Desde que los procesadores se diseñan con pipelines más largos y con la capacidad para ejecutar más instrucciones por ciclo (véase la sección 4.10), el retardo de los saltos se hace mayor, y tener un único hueco de salto retardado resulta insuficiente. Además, el uso del salto retardado pierde popularidad en comparación con las estrategias dinámicas, que son más costosas pero más flexibles. Simultáneamente, el crecimiento en el número de transistores disponibles en el chip ha hecho que la predicción dinámica sea relativamente más barata.

Extensión: Un predictor de saltos nos indica si un salto debe o no ser tomado, pero todavía es necesario calcular la dirección destino del salto. En el pipeline de cinco etapas, este

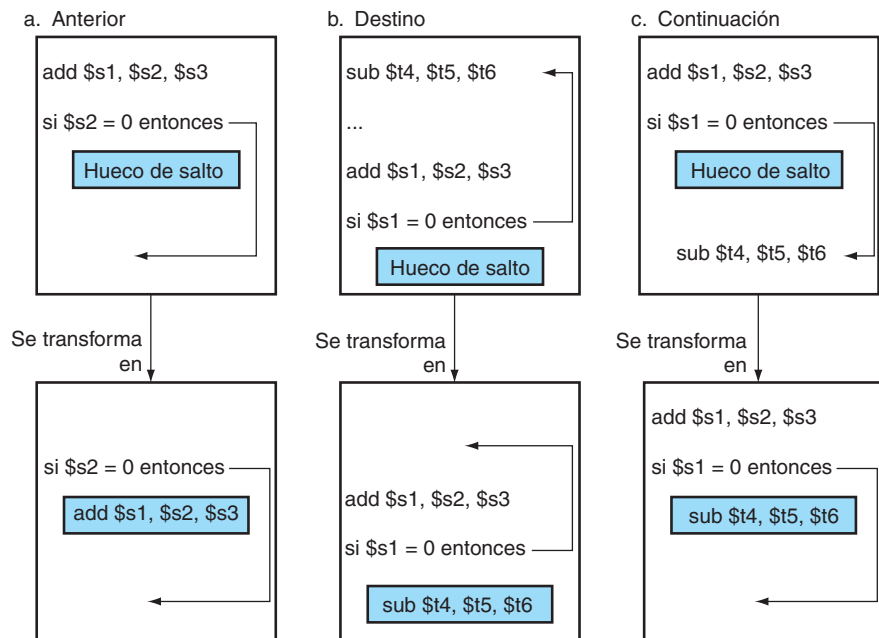


FIGURA 4.64 Planificación del hueco del salto retardado. El recuadro de la parte superior muestra el código antes de la planificación, mientras que el recuadro de la parte inferior muestra el código después de planificar la utilización del hueco. En el caso (a), en el hueco de salto retardado se coloca una instrucción independiente anterior al salto. Si es posible, ésta es la mejor elección, en caso contrario se emplean las estrategias (b) y (c). En las secuencias de código para (b) y (c), el uso del registro `$s1` en la condición del salto evita que la instrucción `add` (cuyo destino es `$s1`) se pueda mover al hueco del salto retardado. En (b) es la instrucción destino del salto la que se coloca en el hueco, y normalmente se necesitará mantener una copia de esta instrucción destino en su lugar de origen, ya que se puede llegar a ella también desde otros caminos. Esta estrategia es la preferida si el salto es tomado con mucha frecuencia, como es el caso del salto de un lazo. Finalmente, se puede colocar en el hueco una instrucción del camino que sigue al salto cuando éste no es tomado, como se muestra en (c). Para que las optimizaciones (b) y (c) funcionen correctamente, debe ser correcta la ejecución de la instrucción `sub` cuando el salto toma la dirección no esperada. Esto significa que se desperdicia trabajo, pero que el programa se ejecuta correctamente. Este sería el caso, por ejemplo, si `$t4` fuera un registro temporal que no tiene ningún uso cuando el salto va a la dirección no esperada.

cálculo necesita 1 ciclo, lo cual significa que los saltos tomados van a tener una penalización de un ciclo. Los saltos retardados representan un estrategia para eliminar esta penalización. Otra estrategia es usar una caché para almacenar la dirección destino del salto, o la instrucción destino del salto, usando un **búfer de destinos de salto (branch target buffer)**.

El esquema de predicción dinámica con 2 bits utiliza la información de cada salto en particular. Los investigadores se dieron cuenta de que el uso de la información de un salto en particular junto con la información global sobre el comportamiento de los saltos ejecutados recientemente logra una precisión mucho mayor utilizando la misma cantidad total de bits de predicción. Este tipo de predictores se denominan **predictores de correlación (correlating predictors)**. Un predictor de correlación típico puede dedicar dos predictores de 2 bits a cada salto permitiendo escoger entre los dos predictores en función de si el último salto ejecutado (no necesariamente el salto que se quiere predecir) ha sido tomado o no. De este modo, el comportamiento global de los saltos se puede interpretar como añadir más bits al índice que se utiliza para el acceso a la predicción dentro de una tabla.

Una innovación aún más reciente en la predicción de saltos es el uso de predictores de torneo. Un **predictor de torneo (tournament predictor)** utiliza múltiples predictores, y además toma nota, para cada salto particular, de cuál de los predictores básicos proporciona mejores resultados. Un predictor de torneo típico podría generar dos predicciones para cada índice de saltos: uno basado en información local y otro basado en el comportamiento global de los saltos. Para cada predicción un selector se encargaría de escoger entre uno de los dos predictores. El selector puede funcionar de forma similar a un predictor de 1 o 2 bits, favoreciendo al predictor que haya sido más preciso. Muchos de los microprocesadores avanzados más recientes hacen uso de estos predictores tan elaborados.

Extensión: Una forma de reducir el número de instrucciones de salto condicional es añadir instrucciones de *copia condicional (conditional move)*. En lugar de cambiar el PC con un salto condicional, la instrucción cambia condicionalmente el registro destino de la copia. Si la condición falla, la copia se comporta como un *nop*. Por ejemplo, una versión de la arquitectura del repertorio de instrucciones MIPS tiene dos nuevas instrucciones llamadas *movn* (copia si no es cero) y *movz* (copia si es cero). Así, *movn \$8, \$11, \$4* copia el contenido del registro 11 en el registro 8 si el valor del registro 4 es distinto de cero; en caso contrario no hace nada.

El repertorio de instrucciones ARM tiene un campo de condición en la mayoría de las instrucciones. De este modo, los programas de ARM podrían tener menos saltos condicionales que los programas MIPS.

Resumen de la segmentación

Este capítulo comenzó en una sala de lavado, donde se mostraban los principios de la segmentación en una tarea cotidiana. Usando esta analogía como guía, explicamos la segmentación de instrucciones paso a paso, comenzando con el camino de datos monociclo y después añadiendo registros de segmentación, caminos de anticipación de resultados, lógica de detección de riesgos, predicción de saltos, y la capacidad de vaciar el pipeline de instrucciones en el caso de excepciones. La figura 4.65 muestra la evolución final del camino de datos y de su control. Ahora estamos listos para otro riesgo de control: la cuestión peliaguda de las excepciones.

Considere los tres esquemas de predicción siguiente: predecir siempre no tomado, predecir siempre tomado, y predicción dinámica. Suponga que los tres tienen una penalización de cero ciclos cuando su predicción es correcta, y de 2 ciclos cuando es errónea. Suponga que la precisión promedio del predictor dinámico es del 90%. ¿Qué predictor supone la mejor elección en los siguientes casos?

Búfer de destinos de salto:

estructura que almacena las direcciones destino o las instrucciones destino de los saltos. Generalmente se organiza como una caché con etiquetas, lo cual la hace más costosa que un simple búfer de predicción.

Predictor de correlación:

predictor de saltos que combina el comportamiento local de un salto en particular con la información global del comportamiento de un cierto número de saltos ejecutados recientemente.

Predictor de torneo:

predictor de saltos que genera múltiples predicciones para un mismo salto y que para cada salto concreto utiliza un mecanismo de selección que escoge cuál de estas predicciones habilitar.

Autoevaluación

1. Los saltos se toman con una frecuencia del 5%
2. Los saltos se toman con una frecuencia del 95%
3. Los saltos se toman con una frecuencia del 70%

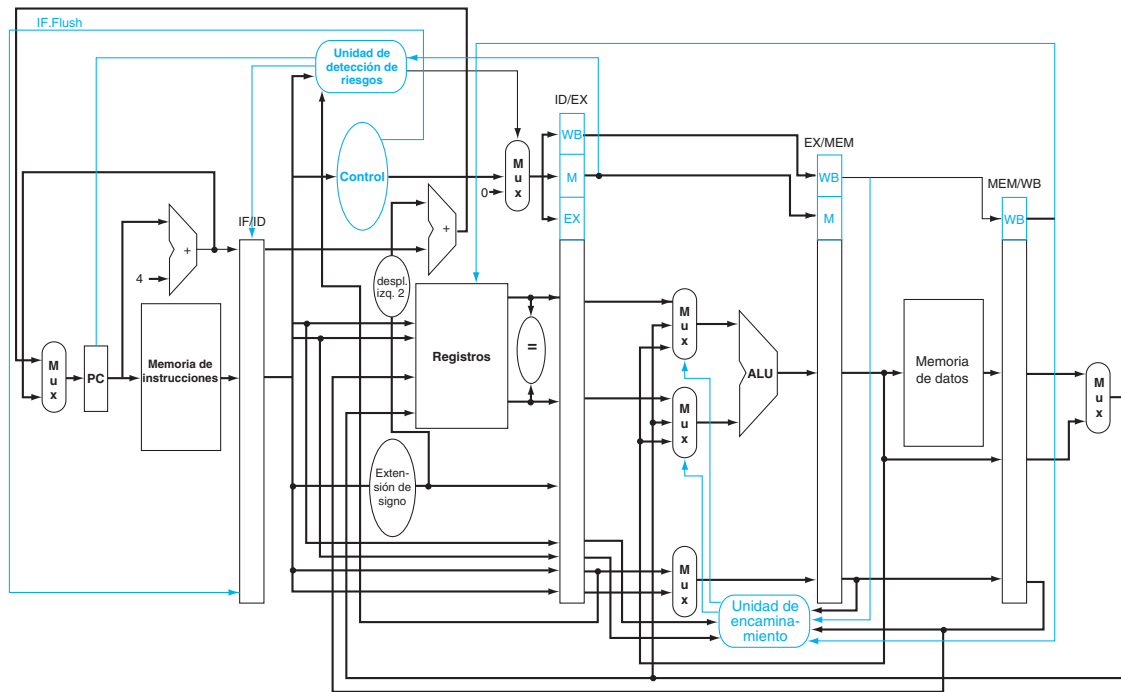


FIGURA 4.65 El resultado final del camino de datos y del control para este capítulo. Obsérvese que ésta es una figura simplificada en lugar de un camino de datos detallado, de modo que no está representado ni el multiplexor ALUsrc de la figura 4.57 ni las señales de control de los multiplexores de la figura 4.51.

Hacer que un computador dotado de un sistema automático de interrupción del programa funcione [de manera secuencial] no fue una tarea fácil, ya que cuando se produce una señal de interrupción el número de instrucciones que se encuentran en diferentes etapas de su procesamiento puede llegar a ser muy grande.

Fred Brooks Jr., *Planning a Computer System: Project Stretch*, 1962

4.9 Excepciones

El control es el aspecto más exigente del diseño del procesador: es la parte más difícil de construir, pues no sólo ha de funcionar bien, sino que también ha de hacerlo rápido. Una de las partes más problemáticas del control es la implementación de las **excepciones** y las **interrupciones** (sucesos, aparte de los saltos, que cambian el flujo normal de ejecución de las instrucciones). Se crearon inicialmente para procesar sucesos inesperados en el procesador, tales como el desbordamiento aritmético. Este mismo mecanismo básico fue extendido para la comunicación de los dispositivos de E/S con el procesador, como se verá en el capítulo 6.

Muchas arquitecturas y autores no distinguen entre interrupciones y excepciones, y a menudo denominan interrupciones a los dos tipos de sucesos. Por ejemplo, el x86 de Intel usa interrupción para todos los sucesos. Siguiendo con la convención del MIPS, se utiliza el término excepción para referirse a cualquier

cambio inesperado en el flujo de control sin distinguir si la causa es interna o externa; y se usa el término interrupción sólo cuando el suceso es externo. Aquí hay algunos ejemplos que muestran si la situación es generada internamente por el procesador o externamente:

Tipo de suceso	¿De dónde?	Terminología MIPS
Demanda de dispositivos de entrada y salida	Externo	Interrupción
Invocar al sistema operativo desde el programa del usuario	Interno	Excepción
Desbordamiento aritmético	Interno	Excepción
Utilizar una instrucción no definida	Interno	Excepción
Errores hardware	Cualquiera	Excepción o interrupción

Una gran parte de la necesidad de soportar excepciones proviene de las situaciones específicas que las causan. Por lo tanto, se volverá a este tema en el capítulo 5, cuando se aborden las jerarquías de memoria, y en el capítulo 6, cuando se vean los dispositivos de entrada y salida, y se entenderá mejor la motivación de las capacidades adicionales en el mecanismo de excepciones. En esta sección se estudiará la implementación del control para detectar dos tipos de excepciones que ya se han tratado.

La detección de situaciones excepcionales y la respuesta mediante la acción apropiada está a menudo en el camino crítico del tiempo del procesador, que determina el tiempo de ciclo de reloj y sus prestaciones. Si no se ha prestado atención a las excepciones durante el diseño de la unidad de control, el intento de añadirlas a posteriori a una implementación complicada puede reducir las prestaciones significativamente, así como complicar la tarea de conseguir un diseño correcto.

Cómo se tratan las excepciones en la arquitectura MIPS

Los dos tipos de excepciones que la implementación actual puede generar son la ejecución de una instrucción no definida y un desbordamiento aritmético. En las siguientes páginas, usaremos el desbordamiento aritmético en la instrucción `add $1, $2, $1`, como ejemplo de excepción. La acción básica que la máquina debe realizar cuando ocurre una excepción es guardar la dirección de la instrucción responsable (*offending*) en el contador de programa de excepción (EPC) y entonces transferir el control al sistema operativo en alguna dirección específica.

Entonces, el sistema operativo puede efectuar la acción apropiada, que puede incluir el darle algún servicio al programa de usuario, tomar alguna acción predefinida en respuesta a un desbordamiento, parar la ejecución del programa o mostrar un mensaje de error. Después de realizar la acción necesaria causada por la excepción, el sistema operativo puede finalizar el programa o continuar su ejecución, utilizando el EPC para saber dónde se ha de retomar la ejecución. En el capítulo 5 se analiza detalladamente el tema del restablecimiento de la ejecución.

Para que el sistema operativo trate la excepción, debe conocer las razones por las que se ha producido, además de la instrucción que la ha causado. Hay dos métodos principales para comunicar la causa de una excepción. El método utilizado en la arquitectura MIPS es incluir un registro de estado denominado registro de causa (*cause register*), que contiene un campo que indica la razón de la excepción.

Excepción (interrupción): suceso no planificado que interrumpe la ejecución del programa; se utiliza para detectar el desbordamiento

Interrupción: excepción que proviene del exterior del procesador. (Algunas arquitecturas utilizan el término *interrupción* para todas las excepciones).

Interrupción vectorizada: interrupción para la cual la dirección a la que se transfiere el control viene determinada por la causa de la excepción.

Un segundo método es utilizar **interrupciones vectorizadas**. En una interrupción vectorizada, la dirección a la que se transfiere el control viene determinada por la causa de la excepción. Por ejemplo, para conformar los dos tipos de excepción antes mencionados se podrían definir las siguientes direcciones en los vectores de excepción:

Tipo de excepción	Dirección del vector de excepciones (en hexadecimal)
Instrucción no definida	8000 0000 _{hex}
Desbordamiento aritmético	8000 0180 _{hex}

El sistema operativo conoce la razón de la excepción por la dirección donde se inicia. Las direcciones están separadas por 32 bytes u 8 instrucciones, y el sistema operativo debe guardar la causa de la excepción y ejecutar algún proceso limitado a esta secuencia. Cuando la excepción no está vectorizada, se puede utilizar un único punto de entrada para todas las excepciones, y el sistema operativo descodificaría el registro de estado para encontrar la causa.

Se puede realizar el proceso requerido por las excepciones añadiendo algunos registros extra y algunas señales de control a nuestra implementación básica y ampliando ligeramente el control. Suponga que se está implementando el sistema de excepciones utilizado en la arquitectura MIPS (la implementación de excepciones vectorizadas no es más difícil). Se tendrán que añadir dos registros adicionales al camino de datos:

- **EPC:** un registro de 32 bits utilizado para guardar la dirección de la instrucción afectada (este registro se necesita aún cuando las excepciones están vectorizadas).
- **Causa:** un registro para almacenar la causa de la excepción. En la arquitectura MIPS, este registro es de 32 bits, aunque algunos de estos bits no se utilizan normalmente. Se supone que existe un campo de cinco bits para codificar las dos posibles fuentes de excepciones mencionadas anteriormente, con 10 representando una instrucción no definida y 12 representando un desbordamiento aritmético.

Excepciones en una implementación segmentada

Una implementación segmentada trata las excepciones como cualquier otro riesgo de control. Por ejemplo, supóngase que hay un desbordamiento aritmético en una instrucción de suma. Tal como se ha hecho en las secciones previas para un salto tomado, se descartan las instrucciones que han entrado en el pipeline a continuación de la suma y se comienza la búsqueda de instrucciones a partir de la nueva dirección. Para las excepciones usaremos el mismo esquema que se ha utilizado para los saltos tomados, pero en esta ocasión la excepción hace que se desactiven las señales de control.

Al gestionar los fallos de predicción de saltos ya vimos cómo eliminar la instrucción en la etapa IF transformándola en una *nop*. Para eliminar instrucciones en la etapa ID se usa el multiplexor que ya existe en la etapa ID y que se usaba para poner a cero las señales de control en caso de bloqueos. Una nueva señal de control, llamada ID.Flush, y la señal de bloqueo producida por la unidad de detección de riesgos se combinan con una O-lógica para eliminar la instrucción

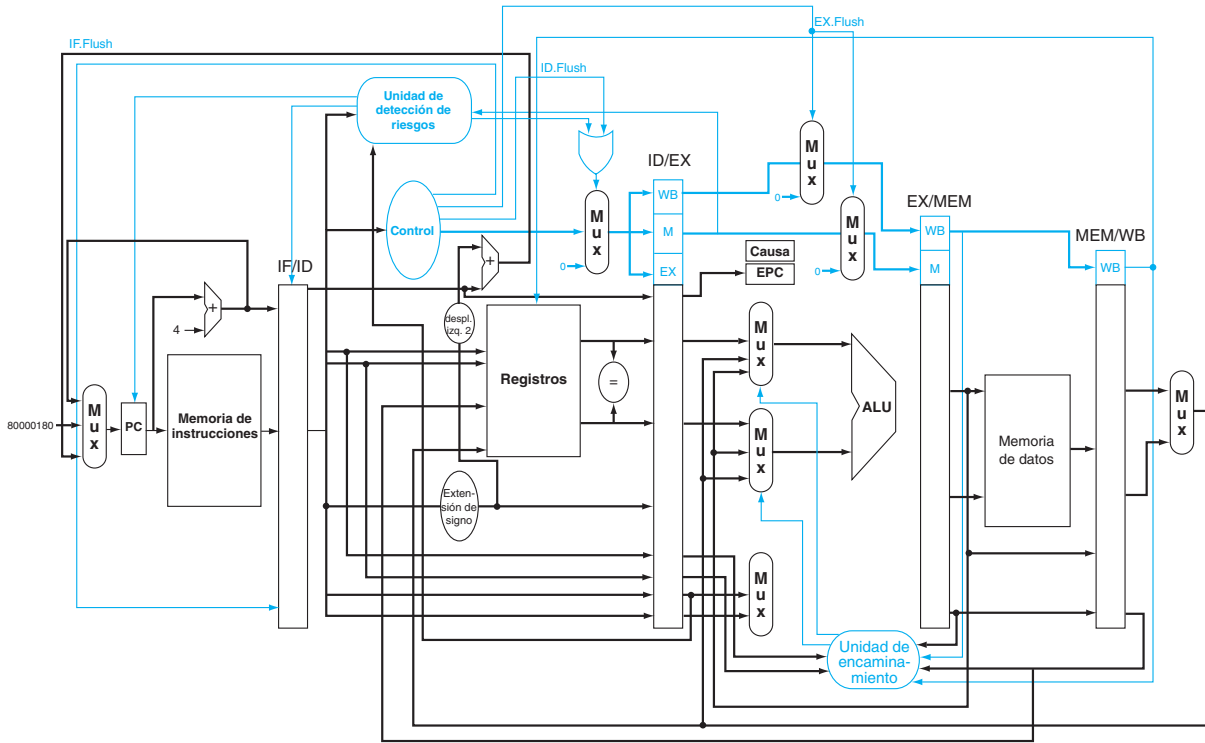


FIGURA 4.66 Camino de datos con el control para gestionar excepciones. Los cambios clave en el diseño incluyen una nueva entrada con el valor `8000 0180hex` en el multiplexor que proporciona el nuevo valor del PC; un registro de Causa para registrar el motivo o causa de la excepción, y un registro PC de excepciones (*Exception PC*) para guardar la dirección de la instrucción que ha provocado la excepción. La entrada `8000 0180hex` al multiplexor es la dirección inicial desde donde se debe empezar la búsqueda de instrucciones cuando ocurre una excepción. A pesar de que no se muestra, la señal de desbordamiento de la ALU es una entrada a la unidad de control.

en la etapa ID. Para eliminar la instrucción situada en la etapa EX se usará una nueva señal, llamada `EX.Flush`, que hará que nuevos multiplexores pongan a cero las líneas de control. Para empezar la búsqueda de instrucciones en la posición `8000 0180hex`, que es la localización de la excepción en el caso de un desbordamiento aritmético, basta con añadir una entrada adicional al multiplexor del PC que envíe el valor `8000 0180hex` al PC. La figura 4.66 muestra estos cambios.

Este ejemplo nos muestra un problema que ocurre con las excepciones: si no se para la ejecución de la instrucción a la mitad, el programador no podrá ver el valor del registro `$1` que ha ayudado a causar la excepción, ya que este valor se perderá al ser `$1` el registro destino de la propia instrucción `add`. Mediante una cuidadosa planificación se consigue detectar el desbordamiento en la etapa EX, de modo que se podrá usar la señal `EX.Flush` para evitar que la instrucción que se encuentra en la etapa EX escriba su resultado en la etapa de WB. Muchas excepciones requieren que se acabe completando la ejecución de la instrucción que causó la excepción como si se hubieran ejecutado de forma normal. La forma más sencilla de llevar esto a cabo es primero anular la ejecución de la instrucción vaciando del pipeline, y después de que la excepción haya sido ya gestionada reiniciar su ejecución desde el principio.

El paso final consiste en guardar el PC de la instrucción que ha causado la excepción en el registro PC de excepciones (EPC). En realidad, se guarda la dirección de la instrucción + 4, por lo que la rutina de gestión de la excepción deberá primero restar el valor 4 al valor guardado. La figura 4.66 muestra una versión simplificada del camino de datos, que incluye la circuitería para saltos y los cambios que son necesarios para gestionar excepciones.

EJEMPLO

Excepciones en un computador segmentado

Dada la siguiente secuencia de instrucciones,

```
40hex    sub    $11, $2, $4
44hex    and    $12, $2, $5
48hex    or     $13, $2, $6
4Chex    add    $1,  $2, $1
50hex    slt    $15, $6, $7
54hex    lw     $16, 50($7)
...
```

suponga que las instrucciones que se han de invocar cuando se produce una excepción empiezan así:

```
80000180hex    sw      $24, 1000($0)
80000184hex    sw      $26, 1004($0)
...
```

Mostrar qué ocurre en el pipeline si se produce una excepción de desbordamiento en la instrucción add.

RESPUESTA

La figura 4.67 muestra los sucesos que se producen, empezando con la instrucción add situada en la etapa EX. Durante esta fase se detecta el desbordamiento, y se fuerza al PC a que tome el valor 8000 0180_{hex}. El ciclo 7 muestra cómo se eliminan tanto la instrucción add como las siguientes, y se busca la primera instrucción del código de excepción. Observe que se guarda la dirección de la instrucción que sigue a add: 4C_{hex} + 4 = 50_{hex}.

En la página 385 se han mencionado cinco ejemplos de excepciones y veremos algunos más en los capítulo 5 y 6. Con cinco instrucciones activas en cada ciclo de reloj, el reto consiste en asociar la excepción con la instrucción apropiada. Además, pueden aparecer múltiples excepciones de forma simultánea en un mismo ciclo. La solución más corriente es dar prioridad a las excepciones de tal manera que sea fácil determinar cuál de ellas se ha de servir primero. En la mayoría de las implementaciones de procesadores MIPS, el hardware ordena las excepciones de modo que sea la instrucción más antigua la que se vea interrumpida.

Las peticiones de un dispositivo de E/S y el mal funcionamiento del hardware no están asociados a una instrucción específica, por lo que hay cierta flexibilidad a la hora de tratarlas, como al decidir el momento en el que se debe interrumpir el pipeline. Así, usar el mismo mecanismo que se usa para las otras excepciones funciona perfectamente.

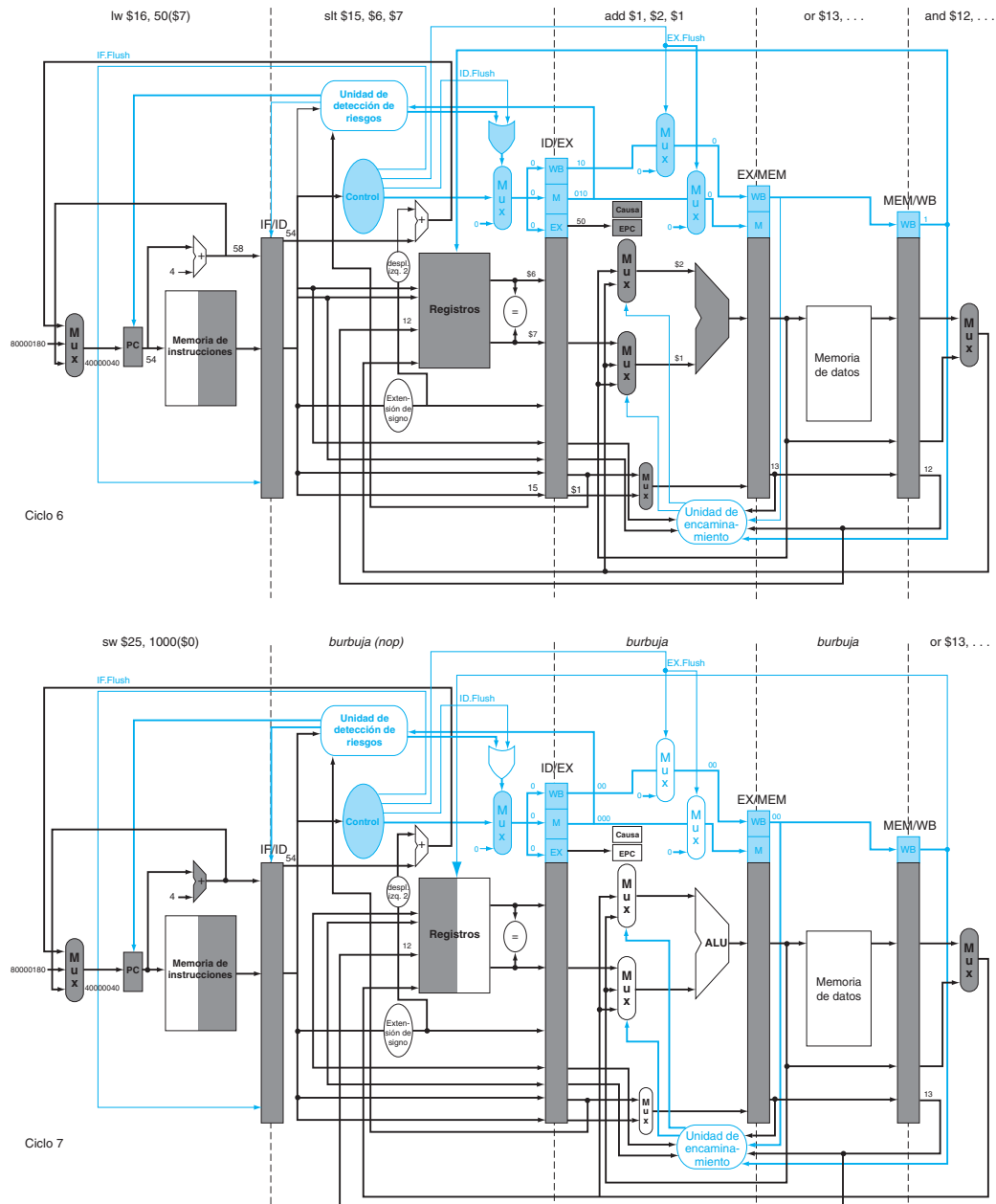


FIGURA 4.67 Resultado de una excepción causada por un desbordamiento aritmético en la instrucción de suma. El desbordamiento se detecta durante la etapa EX, en el ciclo 6, guardando la dirección de la instrucción que sigue a `add` en el registro EPC ($4C + 4 = 50_{\text{hex}}$). El desbordamiento hace que cerca del final de este ciclo de reloj se activen todas las señales de vaciado, desactivando los valores de las líneas control (poniéndolos a 0) para la instrucción `add`. El ciclo 7 muestra las instrucciones convertidas en burbujas dentro del pipeline y la búsqueda en la posición $8000\ 0180_{\text{hex}}$ de la primera instrucción de la rutina de excepción `sw $25 1000 ($0)`. Debe notarse que las instrucciones `and` y `or`, que estaban antes del `add`, se acaban completando. A pesar de que no se muestra, la señal de desbordamiento de la ALU es una entrada en la unidad de control.

El registro EPC captura la dirección de la instrucción interrumpida, y el registro de Causa de MIPS registra todas las posibles excepciones que se pueden dar en un ciclo de reloj, por lo que la rutina de gestión de la excepción debe asociar el tipo de la excepción producida a la instrucción que la provoca. Una pista importante para hacerlo es saber qué tipo de excepción puede ocurrir en cada una de las etapas de segmentación. Por ejemplo, el caso de una excepción provocada por una instrucción no definida se descubre en la etapa ID, mientras que una llamada al sistema operativo ocurre en la etapa de EX. Las excepciones se recogen en el registro de Causa de modo que una vez resuelta la gestión de la excepción de una instrucción anterior, el hardware se pueda interrumpir en una nueva excepción causada por una instrucción posterior.

Interfaz hardware software

La máquina y el sistema operativo deben trabajar conjuntamente para que las excepciones se comporten tal como se espera. Normalmente, el contrato por parte del hardware consiste en detener la instrucción que causa la excepción en medio del flujo de instrucciones, dejar que las instrucciones anteriores se completen, eliminar todas las instrucciones posteriores, modificar un registro para que muestre la causa de la excepción, guardar la dirección de la instrucción que la ha provocado y entonces saltar a una dirección predefinida. El contrato por parte del sistema operativo consiste en considerar la causa de la excepción y actuar en consecuencia. En el caso de una instrucción no definida, del mal funcionamiento del hardware o de una excepción producida por desbordamiento, el sistema operativo normalmente detiene la ejecución del programa y devuelve un indicador de la razón de esta detención. En el caso de una petición por parte de un dispositivo de E/S o de una llamada a una rutina de servicio del sistema operativo, éste guarda el estado del programa, realiza la tarea demandada y en algún punto del futuro restaura el programa para reanudar su ejecución. En el caso de pedir el uso de un dispositivo de E/S, frecuentemente se decide ejecutar otra tarea diferente antes de reanudar la tarea que realizó la petición de E/S, ya que generalmente la tarea no será capaz de continuar hasta que la tarea de E/S haya sido completada. Por esta razón es crítica la habilidad de guardar y restaurar el estado de una tarea. Uno de los usos más importantes y frecuentes de las excepciones es la gestión de fallos de página y de fallos de TLB. El capítulo 5 describe estas excepciones y su gestión de forma más detallada.

Interrupción (excepción) imprecisa: interrupción o excepción en un computador segmentado y que no está asociada a la instrucción exacta que ha causado la interrupción o excepción.

Interrupción (excepción) precisa: interrupción o excepción que en un computador segmentado siempre se asocia con la instrucción correcta.

Extensión: La dificultad en los computadores segmentados para asociar siempre de forma correcta la excepción con la instrucción que la ha causado ha llevado a algunos diseñadores de computadores a relajar los requerimientos en los casos que no son críticos. De estas máquinas se dice que tienen **interrupciones imprecisas o excepciones imprecisas**. En el caso del ejemplo anterior, el PC normalmente valdría 58_{hex} al comienzo del ciclo siguiente a la detección de la excepción, aún cuando la instrucción que causa la excepción se encuentra en la dirección $4C_{\text{hex}}$. Una máquina con excepciones imprecisas podría poner el valor 58_{hex} en EPC y dejar que el sistema operativo detecte cuál fue la instrucción que causó el problema. MIPS y la mayoría de los computadores actuales soportan **interrupciones precisas o excepciones precisas**. (Una razón es que han de poder soportar memoria virtual, lo cual se verá en el capítulo 5.)

Extensión: Aunque en MIPS la dirección a la que transfiere el control en casi todas las excepciones es la $8000\ 0180_{\text{hex}}$, utiliza también la dirección $8000\ 0000_{\text{hex}}$ para mejorar las prestaciones del procesamiento de las excepciones de fallo de TLB (véase el capítulo 5).

¿Qué excepción debería reconocerse en primer lugar en esta secuencia?

1. add \$1, \$2, \$1 # desbordamiento aritmético
2. XXX \$1, \$2, \$1 # instrucción no definida
3. sub \$1, \$2, \$1 # error del hardware

4.10

Paralelismo y paralelismo a nivel de instrucciones avanzado

Se advierte con antelación que esta sección proporciona una visión general breve de temas fascinantes pero avanzados. Si se desea aprender más detalles, se debería consultar el libro más avanzado, *Arquitectura de Computadores: Un enfoque cuantitativo*, ¡donde el material cubierto por las siguientes 13 páginas se amplía a más de 200 páginas (incluyendo apéndices)!

La segmentación aprovecha el paralelismo potencial entre las instrucciones. Este paralelismo se denomina **paralelismo a nivel de instrucciones** (*instruction-level Parallelism*, ILP). Existen dos estrategias básicas para incrementar la cantidad potencial del ILP. La primera consiste en aumentar la profundidad del pipeline para solapar la ejecución de más instrucciones. Empleando la analogía de la lavandería y suponiendo que el ciclo de lavado fuera más largo que los otros, se podría dividir nuestra lavadora en tres máquinas que realizaran los pasos de lavado, aclarado y centrifugado de la máquina tradicional. De este modo, convertiríamos el pipeline de cuatro etapas en un pipeline de seis etapas. Para conseguir la máxima ganancia en velocidad, tanto en el caso del procesador como en el caso de la lavandería, los pasos restantes se deberían volver a equilibrar para que tuvieran la misma longitud. La cantidad de paralelismo que se aprovecharía es mayor, puesto que se solapa la ejecución de más instrucciones. Las prestaciones también serían potencialmente mayores puesto que se puede reducir el ciclo de reloj.

La segunda estrategia consiste en replicar los componentes internos del computador para poder ejecutar múltiples instrucciones dentro de cada etapa de segmentación. El nombre general para esta técnica es **ejecución múltiple** (*multiple issue*). Una lavandería con ejecución múltiple reemplazaría nuestra lavadora y secadora, por ejemplo, por tres lavadoras y tres secadoras. También se tendrían que reclutar más asistentes para doblar y guardar en el mismo tiempo una cantidad de ropa tres veces mayor. El inconveniente de la estrategia es que se requiere trabajo adicional para conseguir mantener todas las máquinas ocupadas y para transferir las cargas de ropa de una etapa de segmentación a la siguiente.

Ejecutar varias instrucciones por etapa permite que la frecuencia de instrucciones ejecutadas sea mayor que la frecuencia del reloj, o, dicho de otra manera, que el CPI sea menor que 1. En ocasiones es beneficioso invertir la métrica del CPI y usar el IPC (*instructions per clock cycle*, instrucciones ejecutadas por ciclo de reloj). Por ejemplo, un microprocesador con ejecución múltiple de hasta cuatro instrucciones que funcione a 4 GHz podrá llegar a ejecutar instrucciones a una velocidad pico de 16 mil millones de instrucciones por segundo, y tener en el mejor de los casos un CPI de 0,25, o un IPC de 4. Suponiendo un pipeline de cinco etapas, este procesador tendría en todo momento hasta 20 instrucciones válidas en ejecución. Los microprocesadores de gama alta de hoy en día tratan de ejecutar entre tres y seis nuevas instrucciones (4 vías) en cada ciclo de

Autoevaluación

Paralelismo a nivel de instrucciones: paralelismo entre instrucciones.

Ejecución múltiple: esquema que permite lanzar varias instrucciones para ejecutarse durante un ciclo de reloj.

Ejecución múltiple con planificación estática:

estrategia de implementación de un procesador con ejecución múltiple en la que el compilador toma muchas decisiones antes de la ejecución del programa.

Ejecución múltiple con planificación dinámica:

estrategia de implementación de un procesador con ejecución múltiple en la que el propio procesador toma muchas decisiones durante la ejecución del programa.

Ranuras de ejecución:

posiciones desde las que las instrucciones pueden ser enviadas a ejecutar en cada ciclo de reloj; usando una analogía, podrían corresponder a las posiciones de salida para una carrera.

Especulación: estrategia utilizada por el compilador o por el procesador para predecir el resultado de una instrucción, y de ese modo poder eliminar la dependencia que esa instrucción genera en la ejecución de otras instrucciones.

reloj. Sin embargo, es habitual encontrar muchas restricciones en el tipo de instrucciones que se pueden ejecutar de forma simultánea y diferencias en la forma de tratar las dependencias que se van encontrando entre las instrucciones.

Hay dos formas fundamentales de implementar un procesador de ejecución múltiple. La diferencia principal entre ellas es la forma de repartir el trabajo entre el compilador y el hardware. Puesto que esta división del trabajo determina si las decisiones se hacen estáticamente (esto es, en tiempo de compilación) o dinámicamente (durante la ejecución), a estas estrategias frecuentemente se las denomina **ejecución múltiple con planificación estática** (*static multiple issue*) y **ejecución múltiple con planificación dinámica** (*dynamic multiple issue*). Veremos que ambas estrategias tienen otros nombres más conocidos, pero que pueden ser menos precisos o más restrictivos.

Un pipeline de ejecución múltiple debe responsabilizarse de dos tareas principales muy distintas:

1. Empaquetar instrucciones en **ranuras de ejecución** (*issue slots*): ¿Cómo determina el procesador cuántas y qué instrucciones pueden ser lanzadas a ejecutar en un determinado ciclo de reloj? En muchos procesadores que planifican el lanzamiento de instrucciones de forma estática, este proceso es gestionado, al menos parcialmente, por el compilador; en procesadores que planifican el lanzamiento de instrucciones de forma dinámica, esta tarea se realiza en tiempo de ejecución por parte del procesador, aunque es frecuente que el compilador haya tratado previamente de colocar las instrucciones en un orden beneficioso, que ayude a mejorar la velocidad de ejecución.
2. Gestionar los riesgos de datos y de control: En procesadores con ejecución de instrucciones planificado estáticamente algunas o todas las consecuencias de los riesgos de datos y de control son gestionadas por el compilador. Por el contrario, muchos procesadores con lanzamiento de instrucciones planificado dinámicamente intentan aliviar al menos algunas clases de riesgos usando técnicas hardware aplicadas en tiempo de ejecución.

Aunque se describan las dos estrategias como si fueran completamente distintas, en realidad algunas técnicas usadas por una de las estrategias son tomadas prestadas por la otra, y ninguna de las dos puede proclamarse como perfectamente pura.

El concepto de la especulación

Uno de los métodos más importantes para encontrar y aprovechar más el ILP es la **especulación** que es una estrategia que permite al compilador o al procesador “adivinar” las propiedades de una instrucción, y de este modo habilitar la ejecución inmediata de otras instrucciones que puedan depender de la instrucción sobre la cual se ha especulado. Por ejemplo, es posible especular con el resultado de un salto, de forma que las instrucciones que siguen al salto puedan ser ejecutadas antes. O se puede especular que una instrucción de almacenamiento que precede a una instrucción de carga no hace referencia a la misma dirección, lo cual permitiría que la carga se ejecute antes que el almacenamiento. La dificultad de la especulación es que puede ser errónea. Así, cualquier mecanismo de especulación debe incluir tanto un método para verificar si la predicción fue correcta, como un método para deshacer los efectos de las instrucciones que fueron ejecutadas especulativamente. La implementación de esta capacidad de vuelta atrás añade complejidad a los procesadores que soportan la especulación.

La especulación la puede realizar tanto el compilador como el hardware. Por ejemplo, el compilador puede usar la especulación para reordenar las instrucciones, moviendo una instrucción antes o después de un salto, o una instrucción de carga antes o después de un almacenamiento. El procesador puede realizar la misma transformación por hardware y durante la ejecución usando técnicas que veremos un poco más adelante en esta sección.

Los mecanismos usados para recuperarse de una especulación incorrecta son bastante diferentes. En el caso de la especulación por software, el compilador generalmente inserta instrucciones adicionales para verificar la precisión de la especulación y proporciona rutinas que arreglan los desperfectos para que sean utilizadas en el caso de que la especulación sea incorrecta. En la especulación por hardware el procesador normalmente guarda los resultados especulativos hasta que se asegura de que ya no son especulativos. Si la especulación ha sido correcta, las instrucciones se completan permitiendo que el contenido de los búferes en los que se guardan temporalmente los resultados se escriban en los registros o en memoria. Si la especulación ha sido incorrecta, el hardware vacía el contenido de los búferes y se vuelve a ejecutar la secuencia de instrucciones correcta.

La especulación introduce otro posible problema: especular con ciertas instrucciones puede introducir excepciones que previamente no se producían. Por ejemplo, suponga que una instrucción de carga se mueve para ser ejecutada de forma especulativa, pero que la dirección que utiliza no es válida cuando la especulación es incorrecta. El resultado sería la generación de una excepción que nunca debería haber ocurrido. El problema se complica por el hecho de que si la instrucción de carga se está ejecutando de forma especulativa y la especulación es correcta, ¡entonces la excepción sí que debe ocurrir! En la especulación basada en el compilador estos problemas se evitan añadiendo un soporte especial a la especulación que permite que las excepciones sean ignoradas hasta que se pueda clarificar si éstas realmente deben ocurrir o no. En la especulación basada en hardware las excepciones son simplemente retenidas temporalmente en un búfer hasta que se verifica que la instrucción que causa la excepción ya no es especulativa y está lista para ser completada; llegados a este punto es cuando se permite que la excepción sea visible y se procede a su manejo siguiendo el mecanismo normal.

Puesto que la especulación puede mejorar las prestaciones cuando se hace de forma adecuada y puede reducirlas si se hace sin cuidado, se debe dedicar un esfuerzo importante a decidir cuándo es apropiado especular y cuándo no. Más adelante, en esta sección, veremos técnicas especulativas tanto estáticas como dinámicas.

Ejecución múltiple con planificación estática

Todos los procesadores con ejecución múltiple y planificación estática utilizan el compilador para que les asista en la tarea de empaquetar las instrucciones y en la gestión de los riesgos. En estos procesadores se puede interpretar que el conjunto de instrucciones que comienzan su ejecución en un determinado ciclo de reloj, lo que se denomina **paquete de ejecución (issue packet)**, es una instrucción larga con múltiples operaciones. Esta visión es mucho más que una analogía. Como este tipo de procesadores generalmente restringen las posibles combinaciones de instrucciones que pueden ser iniciadas en un determinado ciclo, es muy conveniente pensar en el paquete de ejecución como en una instrucción única que contiene ciertos campos predefinidos que le permiten especificar varias operaciones. Esta visión fue la que dio pie al nombre original de esta estrategia: **Very Long Instruction Word (VLIW, literalmente: palabra de instrucción de tamaño muy grande)**.

Paquete de ejecución:

conjunto de instrucciones que se lanzan a ejecutar juntas durante un mismo ciclo de reloj; el paquete de instrucciones puede ser determinado de forma estática por parte del compilador o puede ser determinado de forma dinámica por parte del procesador.

Very Long Instruction Word, VLIW (palabra de instrucción de tamaño muy grande):

estilo de arquitectura de repertorio de instrucciones que junta muchas operaciones independientes en una única instrucción más ancha, típicamente con muchos campos de código de operación separados.

La mayoría de los procesadores de planificación estática de la ejecución también confían al compilador cierta responsabilidad en la gestión de los riesgos de datos y de control. Entre estas responsabilidades se suele incluir la predicción estática de saltos y la reordenación de instrucciones para reducir o incluso evitar todos los riesgos de datos. Antes de describir el uso de las técnicas de planificación estática de ejecución con procesadores más agresivos, echaremos un vistazo a una versión simple del procesador MIPS.

Ejemplo: ejecución múltiple con planificación estática con MIPS

Para dar una idea de la planificación estática consideraremos un procesador MIPS capaz de ejecutar dos instrucciones por ciclo (dos vías, *two issue*), en el cual una de las instrucciones puede ser una operación entera en la ALU o un salto condicional, mientras que la otra puede ser una instrucción de memoria (carga o almacenamiento). Este tipo de diseño es similar al que se usa en algunos procesadores MIPS empotrados. Ejecutar dos instrucciones por ciclo requerirá la búsqueda y decodificación de un total de 64 bits de instrucciones. En muchos procesadores de planificación estática, y esencialmente en todos los procesadores VLIW, las posibilidades para seleccionar las instrucciones que se ejecutan simultáneamente están restringidas, para así simplificar la tarea de decodificar las instrucciones y de lanzarlas a ejecución. Así, suele ser necesario que la pareja de instrucciones esté alineada en una frontera de 64 bits, y que la operación ALU o de salto aparezca siempre en primer lugar. Si una de las instrucciones de la pareja no se puede utilizar, entonces será necesario reemplazarla por una no-operación (*nop*). Por tanto, las instrucciones se lanzarán a ejecutar siempre en parejas, aunque en ocasiones una de las ranuras de ejecución contenga un *nop*. La figura 4.68 muestra cómo avanzan las instrucciones a pares dentro del pipeline.

Los procesadores con planificación estática de la ejecución varían en la forma en que gestionan los riesgos potenciales de datos y de control. En algunos diseños es el compilador quien asume la total responsabilidad de eliminar *todos* los riesgos, planificando el código e insertando instrucciones *nop* para que se ejecute sin necesidad de la lógica de detección de riesgos y sin necesidad de que el hardware deba bloquear la ejecución. En otros diseños, el hardware detecta los riesgos de datos y genera bloqueos entre dos paquetes de ejecución consecutivos, aunque requiere que sea el compilador quien evite

Tipo de instrucción	Etapas del pipeline							
Instrucción ALU o salto condicional	IF	ID	EX	MEM	WB			
Instrucción carga o almacenamiento	IF	ID	EX	MEM	WB			
Instrucción ALU o salto condicional		IF	ID	EX	MEM	WB		
Instrucción carga o almacenamiento		IF	ID	EX	MEM	WB		
Instrucción ALU o salto condicional			IF	ID	EX	MEM	WB	
Instrucción carga o almacenamiento			IF	ID	EX	MEM	WB	
Instrucción ALU o salto condicional				IF	ID	EX	MEM	WB
Instrucción carga o almacenamiento				IF	ID	EX	MEM	WB

FIGURA 4.69 Funcionamiento de un pipeline de ejecución múltiple de dos instrucciones con planificación estática (*static two-issue*). Las instrucciones ALU y de transferencias de datos se lanzan al mismo tiempo. Se ha supuesto que se tiene la misma estructura de cinco etapas que la que se usa en un pipeline de ejecución simple (*single-issue*). Aunque esto no es estrictamente necesario, proporciona algunas ventajas. En particular, la gestión de las excepciones y el mantenimiento de un modelo de excepciones precisas, que se complica en los procesadores de ejecución múltiple, se ven simplificados por el hecho de mantener las escrituras en los registros en la etapa final del pipeline.

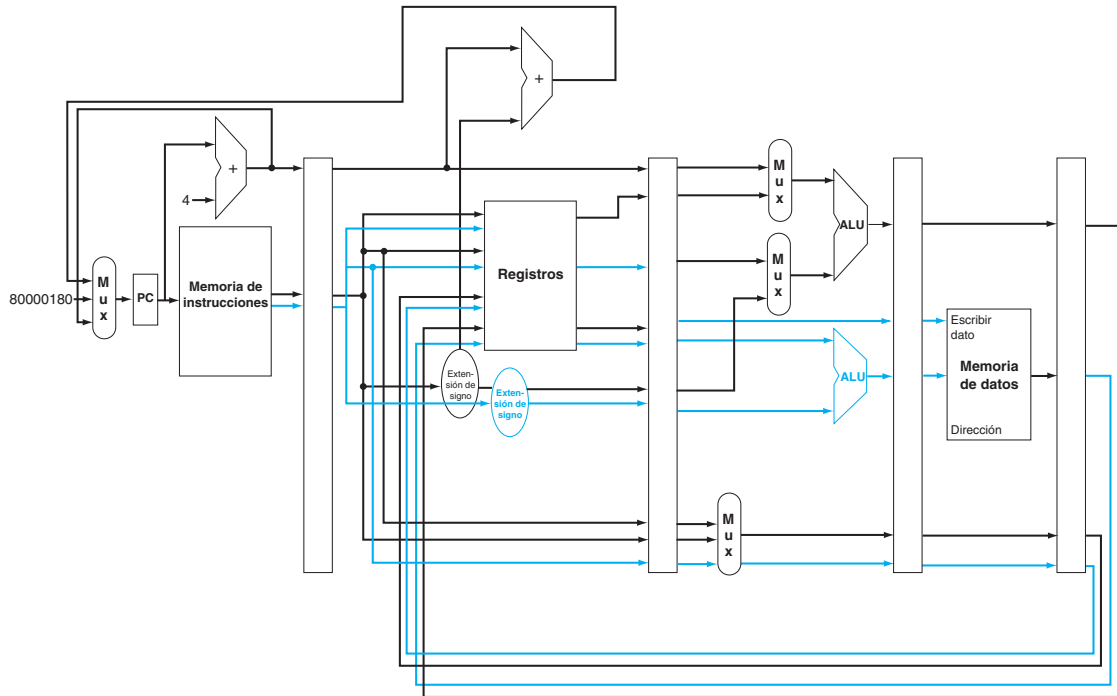


FIGURA 4.68 Camino de datos para la ejecución múltiple de dos instrucciones con planificación estática. Se resaltan los elementos adicionales para hacer que el camino de datos permita dos instrucciones simultáneas: 32 bits más desde la memoria de instrucciones, dos puertos de lectura más y uno más de escritura en el banco de registros, y otra ALU. Suponga que la ALU de la parte inferior calcula las direcciones de las transferencias de datos con memoria y que la ALU de la parte superior gestiona todas las demás operaciones enteras.

las dependencias entre las instrucciones internas de cada paquete de ejecución. Incluso así, un riesgo generalmente forzará a que se bloquee el paquete de ejecución entero que contiene la instrucción dependiente. Tanto si el software debe manejar todos los riesgos como si sólo debe tratar de reducir el porcentaje de riesgos entre diferentes paquetes de ejecución, se refuerza la apariencia de tener una única instrucción compuesta de múltiples operaciones. En el siguiente ejemplo se supondrá la segunda estrategia.

Para ejecutar en paralelo una operación de la ALU y una transferencia de datos a memoria, el primer hardware adicional que sería necesario, además de la habitual lógica de detección de riesgos, es disponer de más puertos de lectura y de escritura en el banco de registros (véase la figura 4.69). En un ciclo podríamos necesitar leer dos registros para la operación ALU y dos más para un almacenamiento, y también podría ser necesario un puerto de escritura para la operación ALU y uno más para una carga. Ya que la ALU está ocupada ejecutando su operación, también se necesitará un sumador adicional para calcular la dirección efectiva de las transferencias de datos a memoria. Sin estos recursos adicionales, el pipeline de dos instrucciones se vería limitado por riesgos estructurales.

Claramente, este procesador de ejecución doble puede llegar a mejorar las prestaciones en un factor 2. Pero para ello es necesario que puedan llegar a solapar su ejecución el doble de instrucciones, y este solapamiento adicional incrementa la pérdida relativa de rendimiento a causa de los riesgos de datos y de control. Por ejemplo, en nuestro pipeline simple de cinco etapas, las cargas tienen una **latencia de uso** de 1 ciclo de reloj, lo

Latencia de uso:

número de ciclos de la señal de reloj entre una instrucción de carga y una instrucción que utiliza el resultado de la carga sin bloquear el pipeline.

cual impide que la instrucción siguiente pueda usar el dato inmediatamente y hace que deba bloquearse. En el pipeline de cinco etapas con doble capacidad de ejecución, el resultado de una instrucción de carga tampoco puede usarse en el siguiente ciclo de reloj, y esto significa ahora que las siguientes *dos* instrucciones no pueden usar el dato de la carga inmediatamente, y tendrían que bloquearse. Además, las instrucciones de la ALU que no tenían latencia de uso en el pipeline sencillo de cinco etapas tienen ahora una latencia de uso de 1 ciclo, ya que el resultado no puede utilizarse en la carga o el almacenamiento correspondiente. Para explotar el paralelismo disponible en un procesador de ejecución múltiple de forma eficiente, se necesitan técnicas más ambiciosas de planificación por parte del compilador o del hardware. En un esquema de planificación estática es el compilador el que debe asumir este rol.

EJEMPLO

Planificación simple de código para ejecución múltiple

¿Cómo se debería planificar este lazo de forma estática en un procesador MIPS con ejecución de dos instrucciones?

```
Loop: lw      $t0, 0($s1)      # $t0=elemento de un vector
      addu    $t0,$t0,$s2      # sumar valor escalar en $s2
      sw      $t0, 0($s1)      # almacenar resultado
      addi    $s1,$s1,-4       # decrementar puntero
      bne     $s1,$zero,Loop   # saltar si $s1!=0
```

Reordenar las instrucciones para evitar el máximo número de bloqueos que sea posible. Suponer que los saltos son predichos, de forma que los riesgos de control son gestionados por el hardware.

RESPUESTA

Las primeras tres instrucciones tienen dependencias de datos, así como las dos últimas. La figura 4.70 muestra la mejor planificación posible para estas instrucciones. Observe que sólo un par de instrucciones completan ambas ranuras del paquete de ejecución. Se tardan 4 ciclos por cada iteración del lazo; al ejecutar 5 instrucciones cada 4 ciclos se obtiene un decepcionante CPI de 0.8 frente un máximo valor alcanzable de 0.5, o un IPC de 1.25 frente a 2. Observe también que al calcular CPI o IPC los *nops* ejecutados no se cuentan como instrucciones útiles. Hacer eso podría mejorar el CPI, ¡pero no las prestaciones!

	Instrucción ALU o branch	Instrucción de transferencia de dato	Ciclo de reloj
Loop:		lw \$t0, 0(\$s1)	1
	addi \$s1,\$s1,-4		2
	addu \$t0,\$t0,\$s2		3
	bne \$s1,\$zero,Loop	sw \$t0, 4(\$s1)	4

FIGURA 4.70 El código planificado para ser ejecutado en un pipeline MIPS de dos instrucciones por ciclo. Los huecos vacíos representan nops.

Una técnica importante aplicada por el compilador para conseguir un mayor rendimiento en los lazos es el **desenrollado de lazos (loop unrolling)**, que consiste en realizar múltiples copias del cuerpo del lazo. Después del desenrollado, se dispone de más ILP para poder solapar la ejecución de instrucciones que pertenecen a diferentes iteraciones.

Desenrollado de lazos para pipelines con ejecución múltiple

Investigue cómo se mejora las prestaciones del ejemplo anterior desenrollando el lazo y planificando de nuevo la ejecución. Por simplicidad, suponga que el índice del lazo es múltiplo de cuatro.

Se necesita hacer cuatro copias del cuerpo del lazo para poder planificar las instrucciones sin que haya retardos. Después de desenrollar el lazo y de eliminar las instrucciones innecesarias, el lazo contendrá cuatro copias de cada instrucción `lw`, `add` y `sw`, además de una instrucción `addi` y una `bne`. La figura 4.71 muestra el código desenrollado y planificado.

Durante el proceso de desenrollado el compilador introduce el uso de registros adicionales (`$t1`, `$t2`, `$t3`). El objetivo de este proceso, llamado **renombrado de registros (register renaming)**, es eliminar aquellas dependencias que no son dependencias de datos verdaderas, pero que pueden conducir a riesgos potenciales o limitar la flexibilidad del compilador para planificar el código. Consideremos cuál habría sido el resultado de desenrollar el código si sólo se hubiera usado `$t0`. Habría copias repetidas en las instrucciones `lw $t0, 0($s1)`, `addu $t0, $t0, $s2` seguidas por la instrucción de `sw $t0, 4($s1)`. Aunque únicamente se use el registro `$t0`, estas secuencias de instrucciones siguen siendo en realidad completamente independientes (entre un par de instrucciones y el siguiente par de instrucciones no se produce ninguna transferencia de datos). El orden entre las instrucciones está forzado única y exclusivamente por la reutilización de un nombre (el registro `$t0`), en lugar de estar forzado por una dependencia de datos verdadera, y a esto se le conoce como **antidependencia (antidependence) o dependencia de nombre (name dependence)**.

Renombrar los registros durante el proceso de desenrollado permite que posteriormente el compilador pueda mover estas instrucciones independientes y así planificar mejor el código. El proceso de renombrado elimina las dependencias de nombre, mientras que preserva las dependencias de datos reales.

Observe que ahora se logra que 12 de las 14 instrucciones del lazo se puedan ejecutar combinadas en una pareja. Cuatro iteraciones del lazo tardan 8 ciclos, que son 2 ciclos por iteración, lo cual supone un CPI de $8/14 = 0.57$. El desenrollado de lazos y la subsiguiente planificación de instrucciones dan un factor de mejora de dos, en parte debido a la reducción en el número de instrucciones de control del lazo y en parte debido a la ejecución dual de instrucciones. El coste de esta mejora de las prestaciones es el uso de cuatro registros temporales en vez de uno, además de un incremento significativo del tamaño del código.

Procesadores con ejecución múltiple y planificación dinámica

Los procesadores con planificación dinámica de la ejecución múltiple de instrucciones se conocen también como procesadores **superescalares (superscalar)**, o simplemente superescalares. En los procesadores superescalares más simples, las instrucciones se

Desenrollado de lazos:

técnica para obtener mayor rendimiento en los lazos de acceso a vectores, en la que se realizan múltiples copias del cuerpo del lazo para poder planificar conjuntamente la ejecución de instrucciones de diferentes iteraciones.

EJEMPLO

RESPUESTA

Renombrado de registros:

volver a nombrar registros, mediante el compilador o el hardware, para eliminar las antidependencias.

Antidependencia (dependencia de nombre):

ordenación forzada por la reutilización de un nombre, generalmente un registro, en lugar de una dependencia de dato verdadera que acarrea un valor entre dos instrucciones.

Superescalar:

técnica avanzada de segmentación que permite al procesador ejecutar más de una instrucción por ciclo de reloj.

	Instrucción ALU o branch	Instrucción de transferencia de dato	Ciclo de reloj
Loop:	addi \$s1,\$s1,-16	lw \$t0, 0(\$s1)	1
		lw \$t1,12(\$s1)	2
	addu \$t0,\$t0,\$s2	lw \$t2, 8(\$s1)	3
	addu \$t1,\$t1,\$s2	lw \$t3, 4(\$s1)	4
	addu \$t2,\$t2,\$s2	sw \$t0, 16(\$s1)	5
	addu \$t3,\$t3,\$s2	sw \$t1,12(\$s1)	6
		sw \$t2, 8(\$s1)	7
	bne \$s1,\$zero,Loop	sw \$t3, 4(\$s1)	8

FIGURA 4.71 Código de la figura 4.70 desenrollado y planificado para un procesador MIPS de ejecución dual. Los huecos vacíos representan *nops*. Como la primera instrucción de lazo decrementa \$s1 en 16 unidades, las direcciones de donde se cargan los datos son el valor original de \$s1, y luego esta dirección menos 4, menos 8 y menos 12.

lanzan a ejecutar en orden, y en un determinado ciclo de reloj el procesador decide si se pueden ejecutar una o más instrucciones nuevas o ninguna. Obviamente, para alcanzar unas buenas prestaciones en estos procesadores, se requiere que el compilador planifique las instrucciones de forma que se eliminen dependencias y se mejore la frecuencia de ejecución. Incluso con esa planificación realizada por el compilador, existe una diferencia importante entre este superescalar sencillo y un procesador VLIW: el hardware garantiza que el código, tanto si está planificado como si no lo está, se ejecutará correctamente. Más aún, independientemente de la estructura del pipeline o de la capacidad del procesador para ejecutar instrucciones simultáneamente, el código compilado siempre funcionará de forma correcta. Esto no es así en algunos diseños VLIW, que requieren que se recompile el código cuando éste se quiere ejecutar en diferentes modelos del procesador. En otros procesadores de planificación estática el código sí que se ejecuta correctamente en las diferentes implementaciones de la arquitectura, pero frecuentemente el rendimiento acaba siendo tan pobre que la recompilación acaba siendo necesaria.

Muchos superescalares extienden el ámbito de las decisiones dinámicas para incluir la **planificación dinámica del pipeline** (*dynamic pipeline scheduling*). La planificación dinámica del pipeline escoge las instrucciones a ejecutar en cada ciclo de reloj intentando en lo posible evitar los riesgos y los bloqueos. Comenzaremos con un ejemplo simple que evita un riesgo de datos. Considere la siguiente secuencia de código

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

Aunque la instrucción `sub` está preparada para ser ejecutada, debe esperar a que primero se completen las instrucciones `lw` y `addu`, lo cual puede suponer muchos ciclos si los accesos a memoria son lentos. (El capítulo 5 explica el funcionamiento de las cachés, la razón de que los accesos a memoria sean en ocasiones muy lentos). La planificación dinámica permite evitar estos riesgos parcial o completamente.

Planificación dinámica del pipeline: soporte hardware para reordenar la ejecución de las instrucciones de modo que se eviten bloqueos.

Planificación dinámica del pipeline

La planificación dinámica del pipeline escoge las siguientes instrucciones a ejecutar, posiblemente reordenándolas para así evitar bloqueos. En estos procesadores, el pipeline se divide en tres unidades fundamentales: una unidad de búsqueda de instrucciones y de decisión de qué instrucciones ejecutar (*issue unit*), múltiples unidades funcionales (12 o incluso más en los diseños de gama alta en el 2008), y una **unidad de confirmación** (*commit unit*). La figura 4.72 muestra el modelo. La primera unidad busca instrucciones, las descodifica y envía cada una de ellas a la unidad funcional que corresponda para ser ejecutada. Cada unidad funcional dispone de búferes, llamados **estaciones de reserva** (*reservation stations*), que almacenan la operación y los operandos. (En la siguiente sección veremos una alternativa a las estaciones de reserva que usan muchos procesadores recientes). Tan pronto como el búfer contiene todos los operandos de la operación y la unidad funcional está preparada, se ejecuta la instrucción y se calcula el resultado. Una vez obtenido el resultado, se envía a todas las estaciones de reserva que puedan estar esperando por él, además de enviarlo a la unidad de confirmación, que lo guardará temporalmente hasta que sea seguro escribirlo en el banco de registros o, si es un almacenamiento, escribirlo en memoria. Este búfer que se encuentra en la unidad de confirmación, llamado con frecuencia **búfer de reordenación** (*reorder buffer*), se usa también para proporcionar operandos de una forma muy similar a como hace la lógica de anticipación de resultados en un procesador planificado de forma estática. Una vez que el resultado es confirmado en el banco de registros, puede ser accedido directamente desde allí, igual que se hace en un pipeline normal.

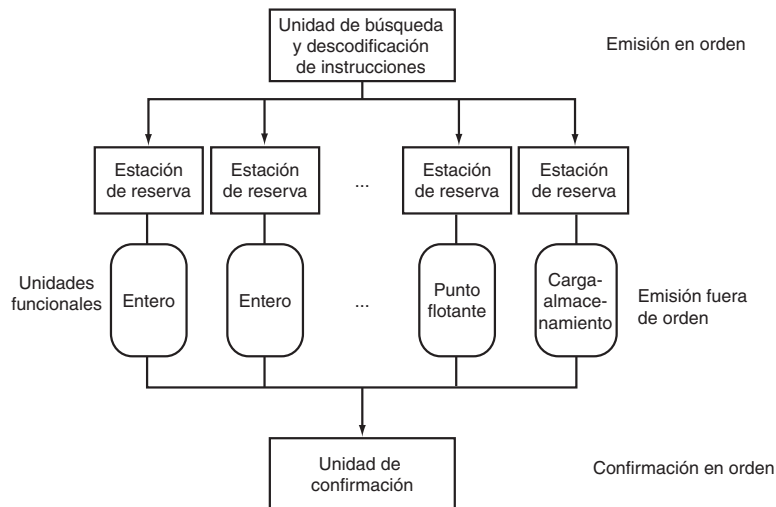


FIGURA 4.72 Las tres unidades principales de un pipeline planificado dinámicamente. El paso final de actualización del estado también se llama jubilación o graduación.

La combinación de almacenar los operandos temporalmente en las estaciones de reserva y de almacenar los resultados temporalmente en el búfer de reordenación supone una forma de renombrado, justamente como la usada por el compilador en el ejemplo anterior del desenrollado de lazos mostrado en la página 397. Para ver cómo funciona todo esto de forma conceptual es necesario considerar los siguientes pasos:

Unidad de confirmación:

en un procesador con ejecución fuera de orden, es la unidad que decide cuando es seguro confirmar el resultado de una operación y hacerlo visible, bien a los registros, bien a la memoria.

Estación de reserva:

búfer acoplado a las unidades funcionales que almacena temporalmente los operandos y las operaciones pendientes de ser ejecutadas.

Búfer de reordenación:

búfer que almacena temporalmente los resultados producidos en un procesador planificado de forma dinámica hasta que es seguro almacenarlos en memoria o en un registro y hacerlos así visibles.

1. Cuando se lanza una instrucción a la unidad de ejecución, si alguno de sus operandos se encuentra en el banco de registros o en el búfer de reordenación, entonces se copia inmediatamente a la estación de reserva correspondiente, donde se mantiene temporalmente hasta que todos los operandos y la unidad funcional están disponibles. Para la instrucción que se ha lanzado, la copia del operando que hay en el registro ya no es necesaria, y se puede permitir realizar una escritura sobre ese registro y sobrescribir su valor actual.
2. Si un operando no está en el banco de registros ni en el búfer de reordenación, entonces debe esperarse a que sea producido por una unidad funcional. Se toma nota de la unidad funcional que producirá el resultado y, cuando finalmente esto ocurra, se copiará directamente desde la unidad funcional, mediante la red de anticipación, a la estación de reserva que está esperando.

Estos pasos usan de forma efectiva el búfer de reordenación y las estaciones de reserva para implementar el renombrado de registros.

Conceptualmente se puede pensar que un pipeline con planificación dinámica analiza la estructura del flujo de datos del programa. El procesador entonces ejecuta las instrucciones en un cierto orden que preserva el orden del flujo de datos del programa. Este estilo de ejecución se llama **ejecución fuera de orden**, ya que las instrucciones pueden ser ejecutadas en un orden diferente del orden en que fueron capturadas.

Para que los programas se comporten como si fueran ejecutados en un pipeline simple con ejecución ordenada, las etapas de búsqueda de instrucciones y de decodificación se deben realizar también en orden, lo cual permite tomar nota de las dependencias, y la unidad de confirmación de la ejecución también debe escribir los resultados en registros y en memoria en el orden original del programa. Este modo conservador de operación se denomina **confirmación en orden** (*in-order commit*). De este modo, si ocurre una excepción, el computador podrá apuntar a la última instrucción ejecutada, y los únicos registros que habrán sido actualizados serán aquellos escritos por las instrucciones anteriores a la que ha causado la excepción. Aunque las etapas denominadas *front-end* (búsqueda y lanzamiento de instrucciones) y la etapa denominada *back-end* (confirmación) del pipeline funcionen en orden, las unidades funcionales son libres de iniciar la ejecución de las instrucciones cuando los datos que se necesitan estén disponibles. Hoy en día, todos los pipelines planificados dinámicamente usan finalización en orden.

La planificación dinámica frecuentemente se extiende para incluir especulación basada en hardware, especialmente en lo que se refiere a los resultados de los saltos. Prediciendo el sentido y la dirección destino de un salto condicional, un procesador con planificación dinámica puede seguir buscando y ejecutando instrucciones a lo largo del camino del flujo de control predicho. Puesto que las instrucciones se confirman en orden, antes de que alguna instrucción en el camino predicho sea confirmada se sabrá si el salto ha sido correctamente predicho o no. Es también fácil para un procesador segmentado con ejecución especulativa y planificación dinámica que dé soporte a la especulación de direcciones de las instrucciones de carga, permitiendo la reordenación entre cargas y almacenamientos, usando la unidad de confirmación para evitar los problemas de una especulación incorrecta. En la siguiente sección revisaremos el uso de la ejecución con planificación dinámica y especulativa en el diseño del AMD Opteron X4 (Barcelona).

Ejecución fuera de orden: propiedad de la ejecución segmentada que permite que el bloque de la ejecución de una instrucción no provoque que las instrucciones que le sigan tengan que esperarse.

Confirmación en orden: confirmación de los resultados de la ejecución segmentada que se escriben en el estado visible para el programador en el mismo orden en que las instrucciones se traen de memoria.

**Comprender
las prestaciones
de los programas**

Puesto que los compiladores son capaces de planificar el código para sortear las dependencias de datos, uno puede preguntarse para qué los procesadores superescalares usan la planificación dinámica. Existen tres razones principales. En primer lugar, no todos los bloqueos en el pipeline son predecibles. En particular, los fallos de caché (véase capítulo 5) son la causa de muchos bloqueos impredecibles. La planificación dinámica permite que el procesador oculte el efecto de algunos de estos bloqueos al seguir ejecutando instrucciones mientras el bloqueo no finaliza.

En segundo lugar, si un procesador utiliza la predicción dinámica de saltos para especular con los resultados de los saltos, entonces no será posible conocer el orden exacto de las instrucciones en tiempo de compilación, puesto que el orden dependerá del propio comportamiento de los saltos. La incorporación de la especulación dinámica para aprovechar mejor el paralelismo de instrucciones ILP sin disponer a su vez de una planificación dinámica de la ejecución, restringiría de forma significativa el beneficio de la especulación.

En tercer lugar, como la latencia en el pipeline y la anchura de ejecución varían de una implementación a otra, la mejor forma de compilar una secuencia de código también varía. Por ejemplo, la mejor manera de planificar la ejecución de una secuencia de instrucciones dependientes se ve afectada tanto por la anchura de ejecución como por la latencia de las operaciones. La estructura del pipeline afecta al número de veces que un lazo debe ser desenrollado para evitar los bloqueos, así como al proceso de renombrado de registros realizado por el compilador. La planificación dinámica permite al hardware ocultar muchos de estos detalles. De este modo, los usuarios y los distribuidores de software no necesitan preocuparse de tener diferentes versiones de un programa para diferentes implementaciones del mismo repertorio de instrucciones. De forma similar, el código generado para versiones previas de un procesador podrá aprovechar mucha parte de las ventajas de una nueva implementación del procesador sin necesidad de tener que recompilarlo.

Tanto la segmentación como la ejecución múltiple de instrucciones intentan aprovechar el paralelismo de instrucciones ILP para aumentar el ritmo máximo de ejecución de instrucciones. Pero las dependencias de datos y de control determinan un límite superior en las prestaciones que se puede conseguir de forma sostenida, ya que a veces el procesador debe esperar a que una dependencia se resuelva. Las estrategias para aprovechar el paralelismo de instrucciones ILP que están centradas en el software dependen de la habilidad del compilador para encontrar estas dependencias y reducir sus efectos, mientras que las estrategias centradas en el Hardware dependen de extensiones a los mecanismos de segmentación y de lanzamiento múltiple de instrucciones. La especulación, tanto si es realizada por el compilador como por el hardware, incrementa la cantidad de paralelismo de instrucciones ILP que puede llegar a ser aprovechado, aunque se debe tener cuidado, ya que la especulación incorrecta puede llegar a reducir las prestaciones.

**IDEA
clave**

Interfaz hardware software

Los modernos microprocesadores de altas prestaciones son capaces de lanzar a ejecutar bastantes instrucciones por ciclo, pero desafortunadamente es muy difícil alcanzar esa capacidad de forma sostenida. Por ejemplo, a pesar de que existen procesadores que ejecutan 4 y 6 instrucciones por ciclo, muy pocas aplicaciones son capaces de ejecutar de forma sostenida más de dos instrucciones por ciclo. Esto se debe básicamente a dos razones.

Primero, dentro del pipeline, los cuellos de botella más importantes se deben a dependencias que no pueden evitarse o reducirse, y de este modo el paralelismo entre instrucciones y la velocidad sostenida de ejecución se reducen. Aunque poca cosa se puede hacer con las dependencias reales, frecuentemente ni el compilador ni el hardware son capaces de asegurar si existe o no una dependencia y, de una forma conservadora, deben suponer que la dependencia existe. Por ejemplo, un código que haga uso de punteros, particularmente si lo hace de forma tal que crea muchas formas de referenciar la misma variable (*aliasing*), dará lugar a muchas dependencias potenciales implícitas. En cambio, la mayor regularidad de los accesos a un vector frecuentemente permite que el compilador deduzca que no existen dependencias. De forma similar, las instrucciones de salto que no pueden ser predichas de forma precisa tanto en tiempo de ejecución como en tiempo de compilación limitarán las posibilidades de aprovechar el paralelismo de instrucciones ILP. En muchas ocasiones existe paralelismo ILP adicional disponible, pero al encontrarse muy alejado (a veces a una distancia de miles de instrucciones) la habilidad del compilador o del hardware para encontrar el paralelismo ILP es muy limitada.

En segundo lugar, las deficiencias del sistema de memoria (el tema del capítulo 7) también limitan la habilidad de mantener el pipeline lleno. Algunos bloqueos producidos por el sistema de memoria pueden ser ocultados, pero si se dispone de una cantidad limitada de paralelismo ILP, esto también limita la cantidad de estos bloqueos que se puede ocultar.

Eficiencia energética y segmentación avanzada

El inconveniente de aumentar la explotación del paralelismo a nivel de instrucciones vía la planificación dinámica con búsqueda de múltiples instrucciones y la especulación es la eficiencia energética. Cada innovación fue capaz de transformar los transistores adicionales en más prestaciones, pero a menudo se hizo de forma muy ineficiente. Ahora que tenemos que luchar contra el muro de la potencia, estamos buscando diseños con varios procesadores por chip, donde los procesadores no tienen pipelines tan profundos ni técnicas de especulación tan agresivas como sus predecesores.

La opinión general es que aunque los procesadores más sencillos no son tan rápidos como sus hermanos más sofisticados, tienen mejores prestaciones por vatio, de modo que pueden proporcionar mejores prestaciones por chip cuando los diseños están limitados más por la potencia disipada que por el número de transistores.

La figura 4.73 muestra el número de etapas, el ancho de emisión, el nivel de especulación, la frecuencia del reloj, el número de núcleos por chip y la potencia de varios microprocesadores pasados y recientes. Obsérvese el descenso del número de etapas y la potencia a medida que los fabricantes introducen diseños multinúcleo.

Microprocesador	Año	Frecuencia de reloj	Etapas	Ancho de ejecución	Fuera de orden/ especulación	Núcleos/ chip	Potencia
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Si	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Si	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Si	1	103 W
Intel Core	2006	2930 MHz	14	4	Si	2	75 W
Sun UltraSPARC III	2003	1950 MHz	14	4	No	1	90 W
Sun UltraSPARC T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W

FIGURA 4.73 Relación de microprocesadores Intel y Sun en términos de complejidad del pipeline, número de núcleos y potencia. Entre las etapas del pipeline del Pentium4 no se incluyen las etapas de confirmación (commit). Si estas etapas se incluyen, los pipelines del Pentium 4 serían todavía más profundos.

Extensión: Los controles de una unidad de confirmación actualizan el banco de registro y la memoria. Algunos procesadores con planificación dinámica actualizan el banco de registros de forma inmediata durante la ejecución, utilizando registros adicionales para implementar el renombrado y manteniendo una copia de los contenidos anteriores hasta que la instrucción que actualiza los registros ya no es especulativa. Otros procesadores guardan los resultados, típicamente en una estructura llamada búfer de reordenación, y la actualización real de los registros se realiza más tarde como parte de la finalización. Los almacenamientos en memoria deben mantenerse en un búfer hasta la finalización, bien en un búfer de almacenamientos (véase el capítulo 5) o bien en un búfer de reordenación. La unidad confirmación permite que la instrucción de almacenamiento escriba el contenido del búfer en memoria cuando el búfer tiene una dirección y un dato válidos y el almacenamiento ya no depende de saltos condicionales con predicción.

Extensión: Los accesos a memoria se benefician de las caches no bloqueantes (*nonblocking caches*), que continúan procesando accesos a la cache durante un fallo de cache (véase el capítulo 5). Los procesadores con ejecución fuera-de-orden necesitan que la cache esté diseñada de forma que permita la ejecución de instrucciones durante un fallo de cache.

Diga si las siguientes técnicas o componentes se asocian fundamentalmente con estrategias bien software o hardware de aprovechamiento del paralelismo de instrucciones ILP. En algunos casos, la respuesta puede ser ambas estrategias.

Autoevaluación

1. Predicción de saltos
2. Ejecución múltiple
3. VLIW
4. Superescalar
5. Planificación dinámica
6. Ejecución fuera-de-orden
7. Especulación
8. Búfer de reordenación
9. Renombrado de registros

4.11

Casos reales: El pipeline del AMD Opteron X4 (Barcelona)

Como la mayoría de los computadores modernos, los microprocesadores x86 incorporan técnicas de segmentación sofisticadas. Sin embargo, estos procesadores están afrontando todavía el reto de implementar el complejo repertorio de instrucciones x86, descrito en el capítulo 2. Tanto AMD como Intel capturan instrucciones x86 y las traducen internamente a instrucciones tipo MIPS, que AMD llama operaciones RICS (Rops) e Intel llama microoperaciones. Las operaciones RISC se ejecutan en un pipeline sofisticado con planificación dinámica y especulativo, capaz de mantener una velocidad de tres operaciones RISC por ciclo de reloj en el AMD Opteron X4 (Barcelona). En esta sección nos centramos en el pipeline de las operaciones RISC.

Cuando se considera el diseño de sofisticados procesadores con planificación dinámica, el diseño de las unidades funcionales, de la caché, del banco de registros, de la lógica para lanzar a ejecutar las instrucciones, y el control total del pipeline está muy interrelacionado, haciendo muy difícil separar lo que es sólo el camino de datos de la totalidad del pipeline. Por esta razón, muchos ingenieros e investigadores han adoptado el término **microarquitectura** para referirse a la arquitectura interna detallada de un procesador. La figura 4.74 muestra la microarquitectura del X4, centrándose en las estructuras que se utilizan para ejecutar las operaciones RISC.

Otra forma de considerar el X4 es mirando las etapas de segmentación que atraviesa una instrucción típica. La figura 4.75 muestra la estructura del pipeline y el número típico de ciclos de reloj que se gastan en él. Por supuesto, el número de ciclos variará debido a la naturaleza de la planificación dinámica, así como a los requerimientos individuales propios de cada operación RISC.

Extensión: El Pentium 4 usa un esquema para resolver las antidependencias y para arreglar las especulaciones incorrectas. Este esquema utiliza un búfer de reordenación junto al renombrado de registros. El renombrado de registros renombra de forma explícita los **registros de la arquitectura** (*architectural registers*) del procesador (16 en el caso de la versión de 64 bits de la arquitectura X86) a un conjunto mayor de registros físicos (72 en el X4). El X4 usa el renombrado de registros para eliminar las antidependencias. El renombrado de registros necesita que el procesador mantenga un asignación entre los registros de la arquitectura y los registros físicos, indicando qué registro físico es la copia más actual de cada registro de la arquitectura. Si se mantiene la información de los renombrados que se han producido, el renombrado de registros ofrece una estrategia alternativa para recuperarse cuando se produce una especulación incorrecta: simplemente se deshacen los mapeos que han ocurrido desde la primera instrucción producto de la especulación incorrecta. Esto provocará que se retorne al estado del procesador que se tenía después de la última instrucción correctamente ejecutada, logrando recuperar la asignación correcto entre registros de la arquitectura y físicos.

Microarquitectura: organización del procesador, que incluye sus principales unidades funcionales, sus elementos de interconexión y el control.

Registros de la arquitectura: registros visibles en el repertorio de instrucciones del procesador; por ejemplo, en MIPS, estos son 32 registros enteros y 16 registros de punto flotante.

Autoevaluación

Las siguientes afirmaciones, ¿son ciertas o falsas?

1. El pipeline de ejecución múltiple del Opteron X4 ejecuta directamente instrucciones x86.
2. El X4 usa planificación dinámica pero sin especulación.

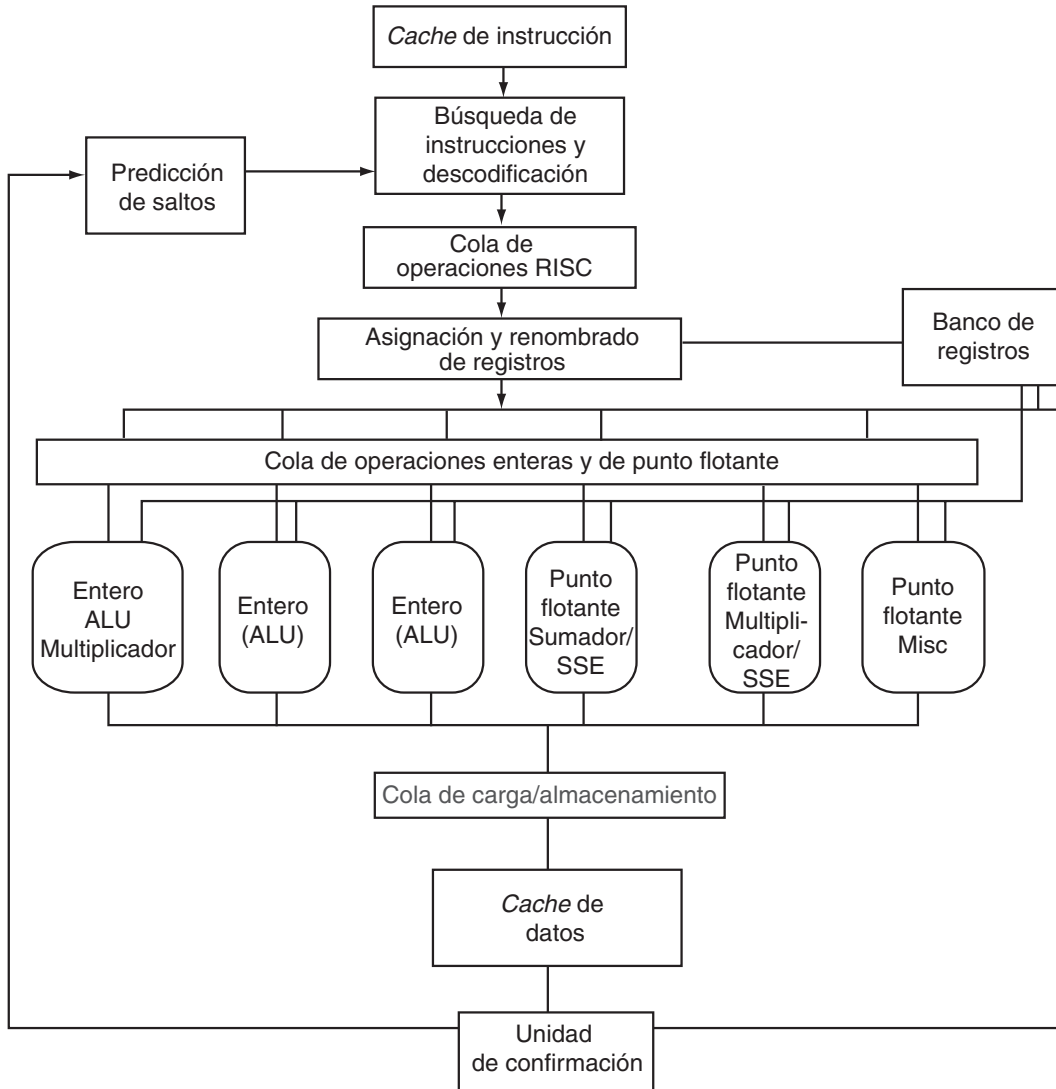


FIGURA 4.74 La microarquitectura del AMD Opteron X4. Las amplias colas disponibles permiten que hasta 106 operaciones RISC estén pendientes de entrar en ejecución, incluyendo 24 operaciones de enteros, 36 operaciones de punto flotante/SSE y 44 cargas y almacenamientos. Las unidades de carga y almacenamiento están realmente separadas en dos partes, la primera parte se encarga del cálculo de la dirección en unidad ALU de enteros y la segunda parte es responsable de la referencia a memoria. Hay un amplio circuito de anticipación entre unidades funcionales; puesto que el pipeline es dinámico en lugar de estático, la anticipación se hace etiquetando los resultados y rastreando los operandos fuente, para detectar cuando un resultado ha sido obtenido por una instrucción en una de las colas que están esperando por este resultado.

3. La microarquitectura del X4 tiene muchos más registros de los que requiere la arquitectura x86.
4. El pipeline del X4 tiene menos de la mitad de etapas que el Pentium 4 Prescott (véase la figura 4.73).

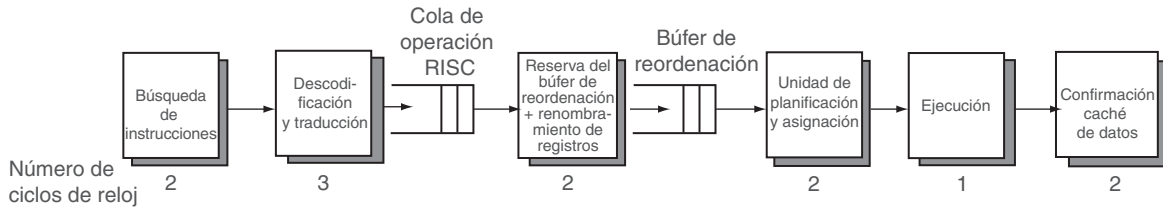


FIGURA 4.75 El pipeline del Opteron X4 mostrando el flujo de una instrucción típica y el número de ciclos de reloj de los principales pasos del pipeline de 12 etapas de las operaciones RISC de enteros. La cola de ejecución de punto flotante tiene una longitud de 17 etapas. Se muestran también los principales búferes donde esperan las operaciones RISC.

Comprender las prestaciones de los programas

El Opteron X4 combina un pipeline de 12 etapas con una ejecución múltiple agresiva para poder alcanzar altas prestaciones. El impacto de las dependencias de datos se reduce haciendo que las latencias entre operaciones consecutivas (*back-to-back*) sean reducidas. Para los programas que se ejecutan en este procesador, ¿cuáles son los cuellos de botella potenciales más serios para las prestaciones? La siguiente lista incluye algunos de estos problemas potenciales para las prestaciones; los tres últimos también se pueden aplicar de alguna manera a cualquier procesador segmentado de altas prestaciones.

- El uso de instrucciones x86 que no se traducen en pocas operaciones RISC simples.
- Las instrucciones de salto difíciles de predecir, que causan bloqueos debido a fallos de predicción y hacen que se deba reiniciar la ejecución por especulación errónea.
- Largas dependencias, generalmente causadas por instrucciones con latencias elevadas o por fallos en la caché de datos, que dan lugar a bloqueos durante la ejecución.
- Retrasos en los accesos a la memoria que provocan que el procesador se bloquee (véase capítulo 5).



Tema avanzado: una introducción al diseño digital utilizando un lenguaje de descripción hardware para describir y modelar un pipeline y más figuras sobre segmentación

El diseño digital moderno se hace utilizando lenguajes de descripción hardware y modernas herramientas de síntesis asistida por computador que son capaces de crear diseños hardware detallados a partir de una descripción con bibliotecas y síntesis lógica. Se han escrito libros enteros sobre estos lenguajes y su utilización en diseño digital. Esta sección, incluida en el CD, da una breve introducción y muestra como un lenguaje de descripción hardware, Verilog en este caso, puede utilizarse para des-

cribir el control del MIPS tanto a nivel de comportamiento como a un nivel adecuado para la síntesis del hardware. Así, proporciona una serie de modelos de comportamiento para el pipeline de cinco etapas de MIPS en Verilog. El modelo inicial ignora los riesgos y las variaciones posteriores resaltan los cambios introducidos para incorporar la anticipación, los riesgos de datos y los riesgos de los saltos.

Proporciona, también, alrededor de una docena de figuras con la representación de ciclo único del pipeline para aquellos lectores interesados en conocer con más detalle qué ocurre en el pipeline para unas cuantas secuencias de instrucciones MIPS.

4.13 Falacias y errores habituales

Falacia: La segmentación es sencilla.

Nuestros libros certifican lo delicado que es conseguir una ejecución segmentada correcta. La primera edición de nuestro libro avanzado tenía un error en el *pipeline*, a pesar de que fue revisado con anterioridad por más de 100 personas y fue probado en las aulas de 18 universidades. El error sólo fue descubierto cuando alguien intentó construir el computador explicado en el libro. El hecho de que el código en lenguaje Verilog que describe un pipeline como el del Opteron X4 será de miles de líneas es una indicación de su complejidad. ¡Tenga cuidado!

Falacia: Las ideas de la segmentación se pueden llevar a cabo independientemente de la tecnología.

Cuando el número de transistores por chip y la velocidad de los transistores hicieron que la segmentación de cinco etapas fuera la mejor solución, el salto retardado (véase la primera sección Extensión de la página 381) fue considerado una solución sencilla para los conflictos de control. Con pipelines más largos, ejecución superescalar y predicción dinámica de saltos, esa técnica es ahora redundante. A comienzos de los 90, la planificación dinámica del pipeline necesitaba demasiados transistores y no fue utilizada para aumentar las prestaciones, pero a medida que la cantidad de transistores disponibles se continuó doblando y la lógica se fue haciendo cada vez más rápida que la memoria, tuvo más sentido el uso de múltiples unidades funcionales y de la planificación dinámica. Hoy en día, la preocupación por la potencia está llevando a diseños menos agresivos.

Error habitual: Olvidar que el diseño del repertorio de instrucciones puede afectar negativamente a la segmentación.

Muchas de las dificultades de la segmentación se deben a complicaciones implícitas en el repertorio de instrucciones. Estos son algunos ejemplos:

- Si la longitud y los tiempos de ejecución de las instrucciones son muy variables, en un diseño segmentado a nivel del repertorio de instrucciones se puede crear un desequilibrio entre las etapas de segmentación que complique seriamente la detección de conflictos. Este problema fue resuelto por primera vez a finales de la década de 1980 en el DEC VAX 8500, usando para ello el mismo esquema de microsegmentación que emplea actualmente el Opteron 4. Por supuesto, la sobrecarga de tener que transformar y mantener la correspondencia entre instrucciones reales y microoperaciones se sigue manteniendo.

- Los modos de direccionamiento muy sofisticados pueden conducir a diferentes tipos de problemas. Los modos de direccionamiento que actualizan registros complican la detección de conflictos. Otros modos de direccionamiento que requieren múltiples accesos a memoria complican sustancialmente el control de la segmentación y hacen difícil mantener las instrucciones avanzando por el pipeline de forma uniforme.

Quizás el mejor ejemplo es el DEC Alpha y el DEC NVAX. Con una tecnología comparable, la nueva arquitectura del repertorio de instrucciones del Alpha permitió una implementación cuyo rendimiento es más del doble de rápido que la del NVAX. En otro ejemplo, Bhandarkar y Clark [1991] compararon el MIPS M/2000 y el DEC VAX 8700 contando el número total de ciclos de los programas de evaluación SPEC; concluyeron que, a pesar de que el MIPS M/2000 ejecuta más instrucciones, el VAX ejecuta un promedio de 2.7 veces más ciclos, por lo que el MIPS es más rápido.

Un noventa por ciento de la sabiduría consiste en ser sabios a tiempo.

Proverbio americano

4.14

Conclusiones finales

Como hemos visto en este capítulo, tanto el camino de datos como el control de un procesador pueden diseñarse tomando como punto de partida la arquitectura del repertorio de instrucciones y una comprensión de las características básicas de la tecnología. En la sección 4.3 vimos cómo el camino de datos de un procesador MIPS podía construirse basándonos en la arquitectura y en la decisión de hacer una implementación de ciclo único. Desde luego, la tecnología subyacente afecta también a muchas decisiones de diseño, imponiendo los componentes que pueden utilizarse en el camino de datos, y determinando si una implementación de ciclo único tiene sentido.

La segmentación mejora la productividad, pero no el tiempo de ejecución inherente, o **latencia de instrucciones**; para algunas instrucciones la latencia es similar en longitud a la de la estrategia de ciclo único. La ejecución múltiple de instrucciones añade un hardware adicional al camino de datos para permitir que varias instrucciones comiencen en cada ciclo de reloj, pero a costa de un aumento en la latencia efectiva. La segmentación se presentó como una técnica para reducir la duración del ciclo de reloj del camino de datos de ciclo único. En comparación, la ejecución múltiple de instrucciones se centra claramente en la reducción de los ciclos por instrucción (CPI).

Tanto la segmentación como la ejecución múltiple de instrucciones intentan aprovechar el paralelismo entre las instrucciones. La presencia de dependencias de datos y de control, que pueden convertirse en conflictos, representan las limitaciones más importantes para la utilización del paralelismo. La planificación de instrucciones y la especulación, tanto en hardware como en software, son las técnicas principales que se usan para reducir el impacto de las dependencias en las prestaciones.

El cambio realizado a mediados de la década de 1990 hacia pipelines más largos, ejecución múltiple de instrucciones y planificación dinámica ha ayudado a mantener el 60% de incremento anual en el rendimiento de los procesadores con el que nos hemos visto beneficiados desde principios de los años 1980. Como se ha mencionado en el capítulo 1, estos microprocesadores mantenían el modelo de programación secuencial,

Latencia de instrucciones: tiempo de ejecución inherente a una instrucción.

pero chocaron con el muro de la potencia. Así, los fabricantes se vieron forzados a introducir los multiprocesadores, que explotan el paralelismo en niveles mucho más gruesos (el objetivo del capítulo 7). Esta tendencia ha causado también que los diseñadores reconsideren las implicaciones del binomio potencia-prestaciones de algunas de las innovaciones introducidas desde mediados de la década de 1990, resultando en una simplificación de los pipelines en las microarquitecturas más recientes.

Para mantener los avances en las prestaciones mediante la utilización de procesadores paralelos, la ley de Amdahl sugiere que será otra parte del sistema la que se convertirá en cuello de botella. Ese cuello de botella es el tema del siguiente capítulo: el sistema de memoria.



Perspectiva histórica y lecturas recomendadas

Esta sección, que aparece en el CD, discute la historia de los primeros procesadores segmentados, los primeros superescalares, el desarrollo de las técnicas de ejecución fuera de orden y especulativa, así como el importante desarrollo de la tecnología de compiladores que les acompaña.



Ejercicios

Contribución de Milos Prvulovic, Georgia Tech.

Ejercicio 4.1

En la implementación básica de ciclo único, instrucciones diferentes utilizan bloques hardware diferentes. Los tres problemas siguientes en este ejercicio se refieren a las siguientes instrucciones:

	Instrucción	Interpretación
a.	add Rd, Rs, Rt	$\text{Reg}[\text{Rd}] = \text{Reg}[\text{Rs}] + \text{Reg}[\text{Rt}]$
b.	lw Rt, Offs(Rs)	$\text{Reg}[\text{Rt}] = \text{Mem}[\text{Reg}[\text{Rs}] + \text{Offs}]$

4.1.1 [5]<4.1> ¿Cuáles son los valores de las señales de control generadas por el control de la figura 4.2 para esta instrucción?

4.1.2 [5]<4.1> ¿Qué recursos (bloques) hacen algo útil para esta instrucción?

4.1.3 [10]<4.1> ¿Qué recursos (bloques) producen salidas que no son útiles para esta instrucción? ¿Qué recursos no generan ninguna salida para esta instrucción?