Capítulo 6

Segmentación avanzada

6.1. Paralelismo a nivel de instrucción

El paralelismo a nivel de instrucción (*Instruction Level Paralelism*) es la capacidad de procesar instrucciones en paralelo, y viene determinado por el número de instrucciones que puedan solaparse en las etapas de un procesador.

Para explotar el paralelismo a nivel de instrucción es necesario determinar que instrucciones pueden ser ejecutadas en paralelo. Si dos instrucciones son paralelizables, pueden ser ejecutadas simultáneamente en un cauce sin causar ninguna detención, suponiendo que el procesador segmentado tenga recursos suficientes. Si dos instrucciones son dependientes, no son paralelizables y, por lo tanto, se deben ejecutar en orden, aunque a menudo se pueden solapar parcialmente. La clave en ambos casos es determinar si una instrucción es independiente de la otra.

Las dependencias son una propiedad de los programas. Que una dependencia detectada dé lugar a un riesgo, y que éste provoque una detención, son propiedades de la organización del procesador segmentado. La diferencia entre dependencia y riesgo es crítica para entender como puede ser explotado el paralelismo a nivel de instrucción.

Puesto que una dependencia puede limitar la cantidad de paralelismo que se puede explotar a nivel de instrucción, se contemplan dos formas de superarlas: manteniendo la dependencia pero evitando el riesgo, y eliminando la dependencia mediante una transformación del código. El reordenamiento de las instrucciones es el método más utilizado para evitar un riesgo sin alterar el código. Así, utilizando técnicas software o hardware se pueden eliminar algunos riesgos.

La presencia de una dependencia indica un riesgo potencial (situaciones que

impiden que la instrucción siguiente del flujo de instrucciones se ejecutan durante su ciclo de reloj designado), pero es característico del procesador segmentado sus riesgos reales y la duración de cualquier detención.

Por tanto, la importancia de las dependencias de datos consiste en que:

1. indican la posibilidad de un riesgo,
2. determinan el orden en el cual se deben calcular los resultados, y
3. establecen la cantidad máxima de paralelismo que se puede obtener.

6.1.1. Dependencias entre instrucciones

Hay tres tipos diferentes de dependencias:

- las dependencias de datos (también llamadas dependencias verdaderas),
- dependencias de nombre, y
- dependencias de control.

Dependencias de datos

Surgen cuando una instrucción depende de los resultados de una instrucción anterior, de forma que ambas no podrían ejecutarse de forma solapada.

Una instrucción j tiene una dependencia de datos con la instrucción i si ocurre cualquiera de las situaciones siguientes:

- La instrucción i produce un resultado que puede ser usado por la instrucción j. Por ejemplo:

  i: lwc1 $f0, 0($r1)
  j: add.d $f2, $f0, $f4

- La instrucción j tiene una dependencia de datos con la instrucción k, y la instrucción k tiene una dependencia de datos con la instrucción i. Por ejemplo:

  i: lwc1 $f0, 0($r1)
  k: mul.d $f4, $f0, $f10
  j: add.d $f2, $f6, $f4

La segunda condición sólo afirma que una instrucción es dependiente de otra si existe una cadena de dependencias del primer tipo entre dos instrucciones. Esta cadena de dependencias pueden ser tan larga como el programa.

Si dos instrucciones tienen una dependencia de datos, no pueden ejecutarse simultáneamente ni solaparse por completo. La dependencia podría causar una cadena de uno o más riesgos de datos entre las dos instrucciones. En un procesador segmentado con bloqueo, la ejecución simultánea implicará detectar el riesgo y parar, reduciendo y eliminando de este modo el solapamiento. En un procesador sin bloqueo que confíe en la planificación del compilador, éste no puede planificar instrucciones dependientes de tal manera que queden completamente solapadas, ya que en ese caso el programa no se ejecutaría correctamente. La presencia de una dependencia de datos en una secuencia de instrucciones refleja una dependencia de datos en el código fuente a partir del cual se ha generado la secuencia de instrucciones. Por tanto, debe conservarse el efecto de la dependencia de datos original.

El valor de un dato puede pasarse entre instrucciones a través de registros o a través de las posiciones de memoria. Es más sencillo detectar las dependencias de datos entre registros, ya que el nombre de los registros se fija en las instrucciones, aunque se complica cuando intervienen los saltos y la preocupación por la corrección obliga al hardware o al compilador a ser conservadores.

Las dependencias que se mueven a través de posiciones de memoria son más difíciles de detectar ya que dos direcciones pueden remitir a la misma posición de memoria pero parecer diferentes. Además, las direcciones efectivas de una carga o un almacenamiento pueden cambiar de una ejecución a otra de la instrucción, complicando aún más la detección de dependencias.

Dependencias de nombre

El segundo tipo de dependencias son las dependencias de nombre. Una dependencia de nombre se produce, cuando dos instrucciones usan el mismo registro o la misma posición de memoria, lo que se llama un nombre, pero no hay flujo de datos entre las instrucciones asociadas a ese nombre. Hay dos tipos de dependencias de nombre entre una instrucción i que preceda a una instrucción j en el orden del programa:

1. Se produce una **antidependencia** entre la instrucción i y la j cuando la instrucción j escribe un registro o una posición de memoria que lee la instrucción i. Se debe mantener el orden original para asegurar que i lee el valor correcto, por ejemplo:

```
i: mul.d   $f4,  $f0,  $f10
           ↙
j: add.d   $f0,  $f6,  $f8
```

2. Una **dependencia de salida** se produce cuando la instrucción i y la instrucción j escriben en el mismo registro o en la misma posición de memoria. Se debe mantener el orden entre las instrucción para asegurar que el valor finalmente escrito corresponde a la instrucción j. Por ejemplo:

```
i: mul.d   $f0,  $f4,  $f10
           ↓
j: add.d   $f0,  $f6,  $f8
```

Ambas, las antidependecias y las dependencias de salida, son dependencias de nombre, ya que no se trasmiten valores entre instrucciones. Puesto que una dependencia de nombre no es una dependencia verdadera, las instrucciones involucradas en una dependencia de nombre pueden ejecutarse simultáneamente o ser reordenadas, ya que si se cambia el nombre (número de registro o posición de memoria) usado en la instrucción, desaparece el conflicto. El renombrado es más fácil para los registros, y se conoce como renombrado de registros. Puede hacerse estáticamente por el compilador o dinámicamente por el hardware.

Dependencias de control

Una dependencia de control determina el orden de una instrucción, i, con respecto a una instrucción de salto de modo que la instrucción sea ejecutada en el orden correcto del programa y sólo cuando deba ser ejecutada. Todas las instrucciones (excepto aquellas del primer bloque básico del programa) tienen una dependencia de control de algún conjunto de salto, y, en general, es necesario mantener esas dependencias de control para preservar el orden del programa.

En general, hay dos restricciones impuestas por las dependencias de control:

1. Una instrucción que tiene una dependencia de control con un salto, no se puede mover antes del salto de forma que su ejecución ya no esté controlada por el salto. Por ejemplo, no se puede mover una instrucción de la parte **then** de un **if** y colocarla antes del **if**.

2. Una instrucción que no tiene una dependencia de control con un salto, no se puede poner después del salto de forma que su ejecución sea controlada por el salto. Por ejemplo, no se puede mover un instrucción anterior a un **if** y colocarla dentro de él.

En un procesador segmentado básico, hay dos formas de tratar las dependencias de control. En primer lugar, las instrucciones se ejecutan siguiendo el orden del programa. Este ordenamiento asegura que una instrucción que ocurre antes de un salto se ejecute antes de él. Además, se detiene la segmentación garantizando que una instrucción que tiene una dependencia de control con un salto no se ejecute hasta que se conozca la dirección de salto. En segundo lugar, se pueden ejecutar instrucciones que no deberían ser ejecutadas aunque se viole el control de dependencias, siempre que se haga sin influir en la correcta ejecución del programa. Aunque mantener las dependencias de control es una manera simple y útil de preservar el orden del programa, las dependencias de control no son por sí mismas un límite fundamental en la ejecución, ya que no son una propiedad crítica que deba ser preservada.

En cambio, las dos propiedades críticas para la correcta ejecución del programa son el comportamiento de las excepciones y el flujo de datos, y se garantizan por el mantenimiento de las dependencias de control y de datos.

Mantener el comportamiento de las excepciones significa que cualquier cambio en el orden de ejecución de una instrucción no debe modificar la ejecución de las excepciones. Con frecuencia esto se hace más flexible, simplificándose a que el reordenamiento de una instrucción no debe causar ninguna nueva excepción en el programa.

El flujo de datos es la segunda propiedad que garantiza el mantenimiento de las dependencias de datos y de control. Consiste en el movimiento de valores entre instrucciones que producen y utilizan resultados. Los saltos hacen que sea dinámico, puesto que permiten que el origen de los datos de una instrucción dada provenga de diferentes puntos. Dicho de otra forma, no es suficiente con controlar las dependencias de datos, porque una instrucción puede tener dependencias de datos con más de una predecesora. El orden del programa es el que determina cual de las predecesoras proporcionará el valor a una instrucción. Manteniendo las dependencias de control se garantiza el orden de un programa.

Las dependencias de control se mantienen con la implementación de detección de riesgos de control que provocan detenciones de control. Las detenciones de control se pueden eliminar o reducir mediante técnicas software y hardware.

Las dependencias de un programa actúan como el límite del paralelismo a nivel de instrucción que se puede explotar. El reto es acercarse a ese límite intentando minimizar los riesgos reales y las detenciones que surgen. Las técnicas de los apartados 6.4 y 6.5 son una forma sofisticada de intentar explotar todo el paralelismo disponible pero manteniendo las dependencias verdaderas del código.

6.1.2. Tipos de riesgos por dependencias entre instrucciones

Riesgos de datos

Los riesgos de datos, como se ha visto ya en el tema anterior, se clasifican en tres tipos, en función del orden de los accesos de escritura y lectura de las instrucciones. Por convenio, los riesgos se nombran por el orden del programa que debe ser mantenido por el procesador segmentado. Considerando dos instrucciones i y j, donde i ocurre antes que j en el orden del programa, los posibles riesgos son:

- **RAW** (*read after write*): j intenta leer un dato antes de que i lo escriba, así j obtiene un valor incorrecto. Este es el tipo de riesgo más común y se corresponde con una dependencia verdadera. Para asegurar que j recibe el valor de i, se debe mantener el orden del programa.

- **WAW** (*write after write*): j intenta escribir un operando antes de que éste sea escrito por i. Se ha realizado la escritura en el orden incorrecto, dejando en el destino el valor escrito por i en vez del valor escrito por j. Este riesgo corresponde a las dependencias de salida. Los riesgos WAW sólo están presentes en cauces que escriben en más de una etapa, o que permiten avanzar a una instrucción incluso si hay instrucciones previas detenidas. Este riesgo no se da en el procesador segmentado básico entero de cinco etapas, pero sí en las técnicas tratadas en los apartado 6.3 y 6.4. También pueden existir riesgos WAW entre un cauce con instrucciones de punto flotante.

- **WAR** (*write after read*): j trata de escribir en su destino antes de que este sea leído por i, así que i obtiene un valor nuevo incorrecto. Este riesgo se corresponde con la antidependencia. No se puede producir en la mayor parte de los procesadores segmentados, incluso en los cauces en punto flotante, debido a que las instrucciones se emiten en orden. Un riesgo WAR sólo ocurre, o bien, cuando hay algunas instrucciones que escriben pronto su resultado y otras retrasan la lectura en el procesador segmentado, o bien, cuando las instrucciones son reordenadas como ocurre en las técnicas descritas en los apartado 6.3 y 6.4.

Riesgos de control

Los riesgos de control se originan a partir de las instrucciones de salto condicional que, según su resultado, determinan la secuencia de instrucciones que hay que procesar tras ellas.

Pueden provocar una mayor pérdida de rendimiento para un cauce MIPS que un riesgo de datos. Cuando se ejecuta un salto, éste puede cambiar o no el contador de programa (PC) a una dirección distinta a la secuencial. Si un salto pone su dirección destino en el PC, es un salto efectivo; en caso contrario es no efectivo. Si una instrucción i es un salto efectivo, normalmente no cambia el PC hasta el fin de la etapa ID, después de completar el cálculo de la dirección y la comparación.

El método más simple para tratar con saltos consiste en repetir la búsqueda de la instrucción siguiente al salto una vez que se detecta el salto durante la ID (decodificación de la instrucción). La primera IF es esencialmente una detención, porque nunca realiza trabajo útil. Hay que destacar que si el salto no es efectivo, es innecesaria la repetición de la etapa IF, ya que, la vez anterior la búsqueda había recogido la instrucción correcta. Existen diferentes técnicas de procesamiento de salto, que veremos en la siguiente sección, para minimizar las detenciones debido a las instrucciones de salto.

6.2. Procesamiento de instrucciones de salto

Como ya se ha comentado anteriormente, los riesgos de control pueden causar una mayor perdida de rendimiento en el procesador MIPS de cinco etapas que estudiamos en el tema pasado, que los riesgos de datos, que en su mayoría se podían solucionar mediante anticipación.

El método más simple para tratar los saltos es detener el flujo de instrucciones en el cauce una vez que hemos detectado la instrucción de salto (en su etapa ID). De esta forma el primer ciclo IF después de la instrucción de salto es en realidad una burbuja, y la etapa IF se vuelve a repetir con la dirección correcta en el PC. Uno se puede dar cuenta ya de que si el salto no se toma (en un salto condicional donde la condición no se cumple), entonces la repetición de la etapa IF es innecesaria, puesto que ya se había realizado la etapa IF de la instrucción correcta. En las siguientes secciones desarrollaremos varios esquemas para sacar partido a esta observación.

Un ciclo de parada en cada salto no parece mucho, pero en la práctica puede suponer una pérdida de rendimiento de entre el 10 % y el 30 %, dependiendo de la frecuencia de los saltos. Así que resulta especialmente interesante encontrar técnicas adecuadas para minimizar esta p'erdida.

6.2.1. Aproximaciones básicas para el tratamiento de saltos

Hay varios métodos para prevenir las paradas del cauce debidas al retardo de los saltos. Vamos a mostrar cuatro alternativas básicas, todas ellas son estáticas, es decir, se mantienen fijas para cada salto durante toda la ejecución. El software puede intentar minimizar la penalización de los saltos si conoce el hardware subyacente y su comportamiento ante saltos.

La técnica más simple es la que comentabamos en la introducción de esta sección, consiste en *purgar* el cauce, borrando todas las instrucciones que vienen después del salto, hasta que el destino del salto se conoce y podemos cambiar el PC. El atractivo de esta solución es su simplicidad tanto para el hardware como para el software. En este caso la penalización por salto es fija y no se puede reducir por software.

Una solución mejor, y sólo ligeramente más compleja, consiste en tratar cada salto como "no tomado", permitiendo que el hardware continue con las instrucciones que vienen por detrás del salto. Aquí hay que tener cuidado de no cambiar el estado del procesador hasta que el resultado del salto se conozca. Si el salto finalmente es tomado, las instrucciones que se estaban ejecutando por detrás de la instrucción de salto se eliminan. En el procesador de 5 etapas que estamos estudiando hasta ahora, este **salto fijo no efectivo** supone una pérdida de un ciclo en caso de que el salto se tome, y no penaliza en caso de que el salto no se tome.

Una alternativa sería la técnica de **salto fijo efectivo**, que como su nombre indica trata cada salto como "tomado". En cuanto la instrucción de salto es decodificada y se conoce la dirección destino, asumimos que el salto se tomará y comenzamos a buscar y ejecutar el destino del salto. Como en nuestro procesador segmentado de cinco etapas conocemos al mismo tiempo (etapa ID) el destino del salto y el resultado de la condición, esta técnica no consigue ninguna mejora. En algunos procesadores, especialmente aquellos con condiciones de salto más potentes y complejas (y por lo tanto más lentas) el destino del salto se puede conocer antes que la evaluación de la condición, y esta técnica tendría sentido.

En ambas técnicas, salto fijo no efectivo y salto fijo efectivo, el compilador puede mejorar el rendimiento reorganizando el código de forma que el camino más frecuente coincida con la predicción que hace el hardware.

Salto Retardado

La cuarta técnica se denomina salto retardado. Esta técnica ha sido muy popular en los primeros procesadores RISC y funciona muy bien en un procesador de 5 etapas como el nuestro. El salto retardado consiste en dejar que las instrucciones

que se captan después de la instrucción de salto, y antes de la modificación del PC, continuen por el cauce, siendo ejecutadas en su totalidad. En el caso del procesador de cinco etapas, donde el salto se resuelve en la etapa ID, sólo hay una instrucción siguiente al salto en esta situación. Esta instrucción se dice que ocupa el hueco de retardo.

Aunque es posible tener un hueco de retardo superior a uno, en la práctica casi todos los procesadores que implementan salto retardado tienen un solo hueco de retardo. Si el cauce tiene una penalización por salto mayor (es decir, el salto se resuelve en etapas posteriores a la ID) se usan otras técnicas para minimizar la penalización.

El trabajo del compilador consiste en conseguir instrucciones válidas y útiles para introducir en el hueco de retardo. Si no es capaz de encontrar una instrucción para el hueco de retardo que no modifique la semántica del programa, entonces introducirá una NOP. Existen varias técnicas que se pueden usar para planificar el hueco de retardo y sacar rendimiento (un ejemplo de ellas se muestra en la Figura 6.1):

- Mover una instrucción **desde antes**: el hueco de retardo se rellena con una instrucción independiente del salto que se tenía que ejecutar antes que este. Es la mejor opción, puesto que esta instrucción siempre será útil (deja de ejecutarse antes del salto para ejecutarse en el hueco de retardo).

- Mover una instrucción **desde destino**: el hueco de retardo se rellena con una instrucción que se tiene que ejecutar si este es tomado. Una precaución que hay que tener es que esta instrucción ha de ser tal que no pueda modificar la semántica del código en caso de que el salto no sea tomado.

- Mover una instrucción **desde instrucciones siguientes**: el hueco de retardo se rellena con una instrucción que se tiene que ejecutar si este no es tomado. Una precaución que hay que tener es que esta instrucción ha de ser tal que no pueda modificar la semántica del código en caso de que el salto sea tomado.

La limitación del salto retardado surge de las restricciones en las instrucciones que se pueden planificar en el hueco de retardo y de nuestra habilidad para predecir el comportamiento del salto. Para mejorar la capacidad del compilador de rellenar los huecos de retardo, muchos procesadores proporcionan saltos condicionales que introducen la propiedad de *concelación* o *anulación* de la instrucción en el hueco de retardo en caso de que el salto no se realice en la dirección adecuada. En un salto retardado con cancelación, la instrucción de salto condicional incluye la dirección en la que se predice que se va a producir el salto. Si el salto se comporta como se predice, entonces la instrucción en el hueco de retardo continúa hasta el final, si el

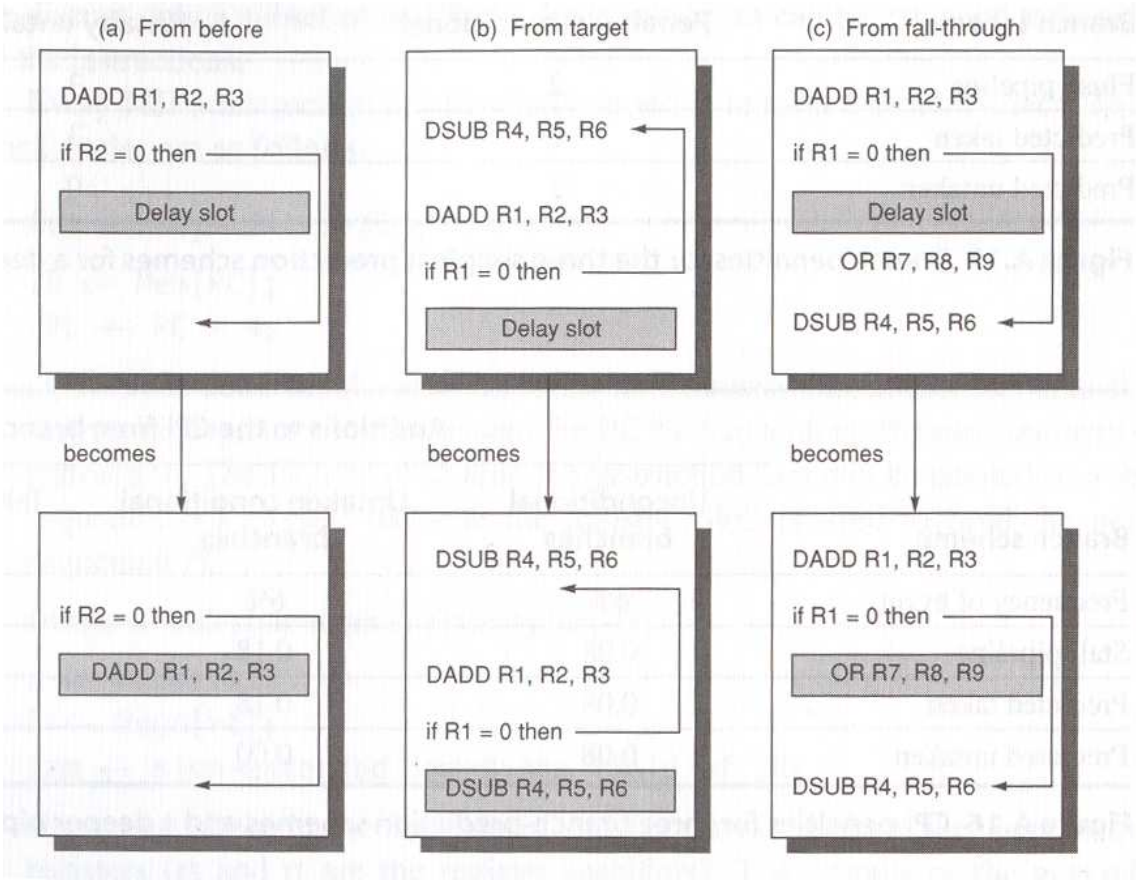


Figura 6.1: Formas de planificar el hueco de retardo

salto no se predice correctamente, entonces la instrucción en el hueco de retardo se anula.

6.2.2. Predicción dinámica de saltos

Las técnicas de procesamiento de saltos vistas en el apartado anterior son esquemas estáticos, la acción tomada no depende del comportamiento dinámico del salto. En este apartado haremos una pequeña introducción al uso del hardware para predecir de forma dinámica el resultado de un salto. La predicción va a depender del comportamiento del salto en tiempo de ejecución y cambiará si el salto cambia su comportamiento durante la ejecución.

Veremos aquí sólo un esquema de predicción muy simple y como se podrían mejorar los mecanimos de predicción. La eficiencia de un esquema de predicción de

saltos depende no sólo de la precisión con la que conseguimos predecir el salto, sino también del coste del salto cuando la predicción es incorrecta, y también cuando la predicción es correcta. La penalización depende de la estructura del cauce, del tipo de predicción, y de las estrategias para recuperar el estado después de una predicción fallida.

El esquema más simple de predicción de saltos es un buffer de predicción de saltos o una tabla de históricos. Un buffer de predicción de saltos es una pequeña memoria indexada por la porción inferior de la dirección de una instrucción de salto. Esta memoria contiene un bit que dice si el salto se ha tomado recientemente o no. Este esquema es útil sólo en los casos en que el retardo del salto es mayor que el tiempo para calcular el posible destino de salto. De hecho, ni siquiera se conoce si la predicción es correcta, podría estar puesta ahí por otro salto que tenga los mismos bits en las posiciones inferiores de su dirección. Si al final la predicción es incorrecta, el bit de predicción del buffer se invierte.

El esquema simple de predicción con 1 bit presenta desventajas desde el punto de vista del rendimiento. Supongamos que un salto se toma casi siempre, cuando el salto no se tome haremos una predicción incorrecta dos veces, en lugar de una. Para remediarlo existen esquemas de predicción que usan 2 bits. En un esquema que use 2 bits, una predicción debe fallar dos veces consecutivas antes de cambiar. La figura 6.2 muestra la máquina de estados para este esquema. El esquema de 2 bits es en realidad un caso particular de un esquema más general de predicción usando n-bits.

6.3. Planificación dinámica

Un procesador segmentado con planificación estática busca una instrucción y la emite, a menos que haya una dependencia entre esa instrucción y otra que ya esté en el cauce y que no pueda ser evitada mediante anticipación. Si hay dependencias de datos que no pueden ser resueltas entonces se para (empezando con la instrucción que usa el resultado). No se busca o emite ninguna instrucción hasta que se solucione la dependencia.

La principal limitación de muchas técnicas de segmentación es que emiten las instrucciones en orden. Si el procesador se detiene con una instrucción, las posteriores no pueden proceder. De esta manera, una dependencia entre dos instrucciones espacialmente cercanas en el procesador segmentado, puede dar lugar a un riesgo y a una detención. Si hay múltiples unidades funcionales, estas permanecerán inactivas. Así, si una instrucción j depende de una instrucción i de larga ejecución, que esté actualmente en ejecución, todas las instrucciones posteriores a j deben detenerse hasta que termine i y se pueda ejecutar j.

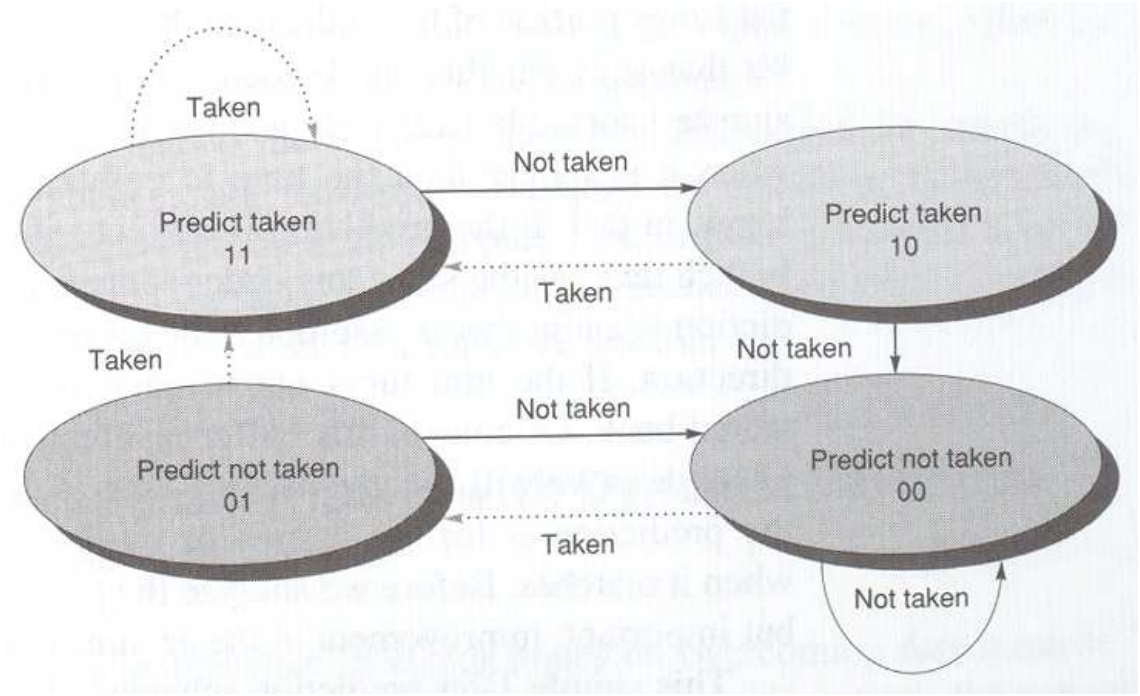


Figura 6.2: Máquina de estados para la predicción usando 2 bits

Otra aproximación diferente es la planificación dinámica, en la cual el hardware reorganiza la ejecución de la instrucción para reducir las detenciones mientras mantiene el flujo de datos y el comportamiento de las excepciones. La planificación dinámica ofrece varias ventajas: habilita el tratamiento de algunos casos cuando las dependencias son desconocidas en tiempo de compilación, y simplifica el compilador. Quizás la ventaja más importante es que permite que un código que fue compilado pensando en una determinada segmentación, se ejecute eficientemente en un procesador con diferente segmentación. Como se refleja en las secciones siguientes, las ventajas de la planificación dinámica se consiguen a costa de un importante aumento en la complejidad hardware.

Aunque un procesador con planificación dinámica no puede cambiar el flujo de datos, intenta evitar detenciones cuando hay dependencias que pueden generar riesgos. Por contra, un procesador segmentado con planificación estática en compilación, intenta minimizar las detenciones separando las instrucciones dependientes, de forma que no puedan dar lugar a riesgos. Desde luego, la planificación del compilador también puede usarse con código destinado a ejecutarse en procesadores segmentados con planificación dinámica.

En la ejecución clásica en cinco etapas, los riesgos estructurales y de datos podían

ser comprobados durante la decodificación de la instrucción (*etapa ID*). Cuando una instrucción se podía ejecutar sin riesgos, se emitía sabiendo desde ID que todos los riesgos de datos habían sido resueltos. Con esta técnica se comprueban los riesgos estructurales cuando se emite la instrucción; todavía se utiliza emisión en orden (i.e., las instrucciones se emiten en el orden del programa), pero las instrucciones comienzan a ejecutarse tan pronto como todos sus operandos están disponibles. De esta manera, el procesador segmentado realizará ejecución fuera de orden, lo que implica terminación fuera de orden.

Para mostrar las diferencias en la ejecución entre un código no planificado y el mismo código usando planificación dinámica, suponemos una configuración en la que la latencia de las unidades de multiplicación sea de cuatro ciclos y de las de suma de dos ciclos, y además, donde no haya riesgos estructurales. Partimos del siguiente código:

```
mul.d   $f0,  $f2,  $f6
add.d   $f4,  $f0,  $f2
add.d   $f8,  $f2,  $f6
sub.d   $f10, $f6,  $f2
```

En este código existe una dependencia de datos entre la primera y la segunda instrucción, que será la que resalte las diferencias entre la ejecución sin planificación y con planificación dinámica.

Esta dependencia de datos influirá en ambas ejecuciones, tal y como se ve en la Figura 6.3, ya que en ambos casos detiene la ejecución de la segunda instrucción (*add.d $f4, $f0, $f2*). Lo que varía es el tratamiento de aquellas instrucciones que son posteriores a la que tiene el riesgo, pero que no tienen ninguna dependencia, y por lo tanto podrían ejecutarse correctamente.

En el caso de la ejecución original (ver Figura 6.3(a)), que se caracteriza por una emisión y terminación en orden, el resto de las instrucciones son detenidas, debido a que la instrucción que tiene el riesgo ha sido parada en la *etapa ID*, deteniendo de esta forma todas aquellas instrucciones que estén en una etapa previa (en este caso sólo la *etapa IF*). Es decir, hasta que la instrucción de la *etapa ID* conozca el valor correcto de su operando fuente no avanzará a la *etapa EX*, permitiendo así el avance del resto del cauce. Esto ocasiona, que durante cinco ciclos haya recursos del procesador que estén infrautilizados. Este problema se agudizará, cuanto mayor sea la latencia de la instrucción que tiene que escribir el operando fuente. Para minimizar este inconveniente se utiliza la anticipación, que permite que una instrucción que está esperando por un operando fuente lo reciba tan pronto ha sido calculado y no tenga que esperar a la escritura. En el ejemplo, esto permitiría que la segunda

instrucción pasará a la *etapa EX* en el ciclo siete.

Por su parte, la aproximación de la planificación dinámica permite una ejecución fuera de orden y puede tener varias instrucciones en la *etapa de emisión* simultáneamente, con lo que aunque la segunda instrucción se quede parada, las instrucciones siguientes del ejemplo puedan avanzar su ejecución, ya que no existe ninguna dependencia.

En las Figura 6.3, se puede comparar el rendimiento conseguido por ambas ejecuciones. Se observa que en la ejecución en orden, en el ciclo octavo sólo ha finalizado una instrucción y el resto están comenzando a ejecutarse. Por otro lado, en la planificación dinámica, dos instrucciones han finalizado su ejecución y a otra de ellas sólo le falta la *etapa de escritura*.

La ejecución fuera de orden introduce la posibilidad de tener que gestionar más riesgos, que no existen en un cauce de cinco etapas ni en su extensión a un procesador segmentado con operaciones en punto flotante y ejecución en orden.

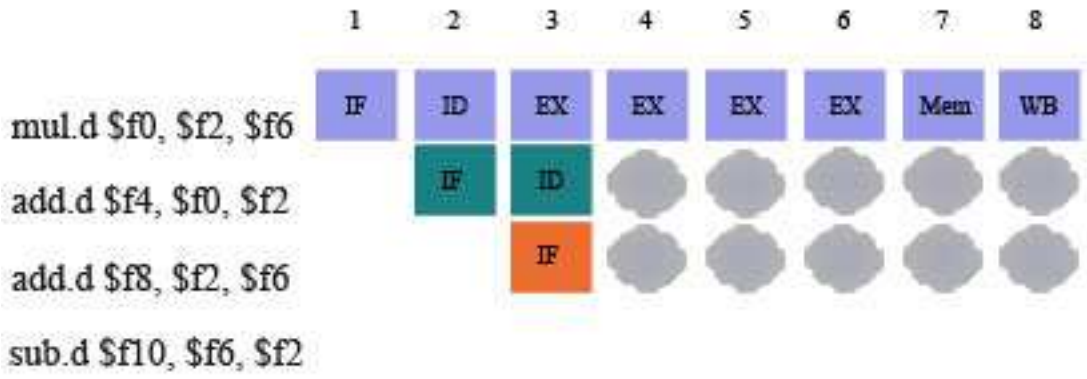
Para permitir la ejecución fuera de orden, es necesario dividir la *etapa ID* del procesador segmentado clásico de cinco etapas, en dos:

- **Emisión:** Decodificación de instrucciones, comprobación por riesgos estructurales.
- **Lectura de operandos:** Se espera hasta que no haya riesgos de datos, luego se leen los operandos.

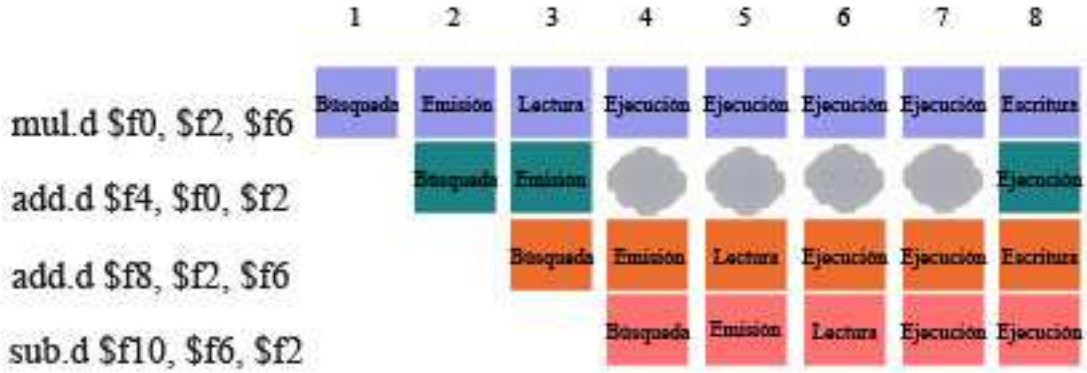
La búsqueda de instrucciones precede a la etapa de emisión. La ejecución (*etapa EX*) es la etapa siguiente a la *lectura de operandos*, al igual que en el cauce de cinco etapas. La *ejecución* puede necesitar múltiples ciclos, dependiendo de la operación.

Se puede distinguir cuando una instrucción comienza su ejecución y cuando la completa; entre estos dos momentos, la instrucción está en ejecución. El cauce permite multiples instrucciones en ejecución al mismo tiempo. Tener múltiples instrucciones en ejecución a la vez requiere múltiples unidades funcionales, unidades funcionales segmentadas, o ambas. Se asume que el procesador tiene múltiples unidades funcionales.

A continuación, vamos a presentar las dos principales técnicas de planificación dinámica: Marcador y Tomasulo. Marcador es una técnica que permite que las instrucciones se ejecuten fuera de orden si hay recursos suficientes y no hay dependencias de datos. Existe una técnica más sofisticada, el algoritmo de Tomasulo, que mejora sensiblemente a Marcador.



(a) Sin planificación



(b) Planificación dinámica

Figura 6.3: Diagrama multiciclo

6.4. Técnica de Marcador

En un procesador segmentado con planificación dinámica, todas las instrucciones pasan a través de la etapa de decodificación en orden; sin embargo pueden ser detenidas o anticipadas en la segunda etapa (*lectura de operandos*) y, así, entrar en la ejecución fuera de orden. Marcador es una técnica que permite que las instrucciones se ejecuten fuera de orden si hay suficientes recursos y no existen dependencias de datos.

Es importante destacar, que los riesgos WAR que no existen en las segmentaciones enteras o en punto flotante del procesador segmentado básico, pueden surgir aquí debido a la ejecución fuera de orden.

El objetivo de Marcador es mantener una velocidad de ejecución de una instrucción por ciclo de reloj (cuando no haya riesgos estructurales) ejecutando cada instrucción lo antes posible. Así, cuando se detiene la próxima instrucción a ser ejecutada, se puede emitir y ejecutar otra instrucción si no tiene dependencias con ninguna otra instrucción activa o detenida. El marcador es el responsable de la emisión y la ejecución de las instrucciones, incluyendo todas las detecciones de riesgos. Para obtener beneficios de la ejecución fuera de orden se requiere que múltiples instrucciones estén simultáneamente en su *etapa EX*. Esto se consigue con múltiples unidades funcionales, con unidades funcionales segmentadas o con ambas. Como estas dos posibilidades son esencialmente equivalentes para los propósitos del control de la segmentación, supondremos que la máquina tiene múltiples unidades funcionales. En la Figura 6.4 se puede ver un diseño de un procesador con la técnica de Marcador.

Cada instrucción va a través del Marcador, donde se construye un cuadro de dependencias; este paso se corresponde con la emisión de una instrucción y sustituye parte del paso *ID* en un procesador segmentado MIPS. El Marcador decide cuando una instrucción puede leer sus operandos y comenzar su ejecución. Si el Marcador decide que la instrucción no se puede ejecutar inmediatamente, vigila cualquier cambio en el hardware y decide cuando puede ejecutarse. El Marcador también controla cuando una instrucción puede escribir su resultado en el registro destino. Así, todas las detecciones y resoluciones de riesgos están centralizadas en el Marcador.

6.4.1. Pasos en la ejecución a través de Marcador

Cada instrucción pasa por cuatro pasos en ejecución. Como nos centramos en ejecuciones en punto flotante no consideraremos un paso para acceso a memoria. Los cuatro pasos que reemplazan el *ID*, *EX* y *WB* del procesador segmentado MIPS

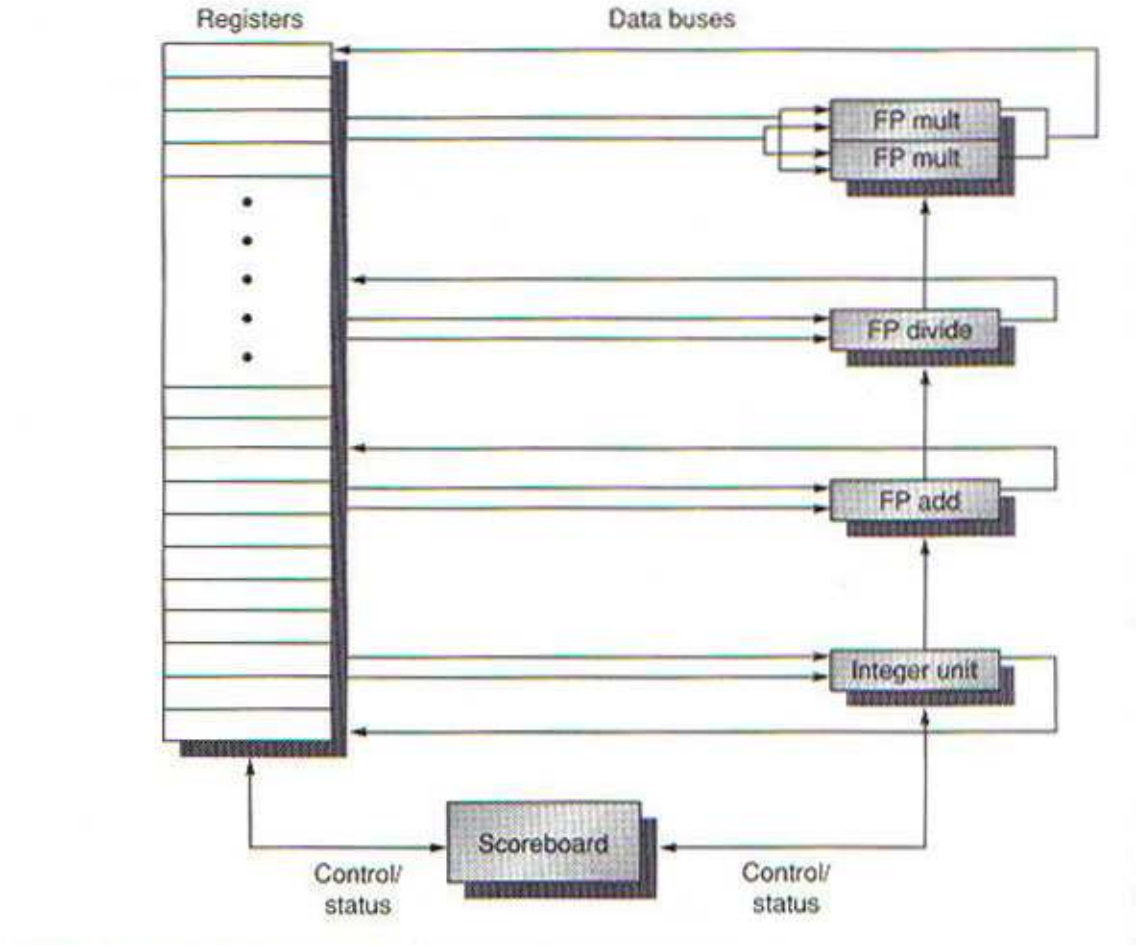


Figura 6.4: Estructura básica de un procesador MIPS con Marcador

estándar, son los siguientes:

- **Emisión** (*issue*): Si una unidad funcional está libre para la instrucción, y ninguna otra instrucción activa tiene el mismo registro destino, el marcador facilita la instrucción a la unidad funcional y actualiza su estructura interna de datos. Este paso sustituye una parte del paso *ID* del cauce MIPS. Al asegurar que ninguna otra unidad funcional activa quiere escribir su resultado en el registro destino, se garantiza la correcta gestión de los riesgos WAW. Si existe un riesgo estructural o WAW, entonces se detiene la emisión de la instrucción, y no se emiten más instrucciones hasta que desaparezcan estos riesgos. Cuando la etapa de emisión se detiene, se llena el buffer que comunica la búsqueda de emisiones y la decodificación; si el buffer es de una única entrada, se detiene

la búsqueda de instrucciones inmediatamente. Si el buffer es una cola con múltiples instrucciones, se detiene cuando se llena la cola.

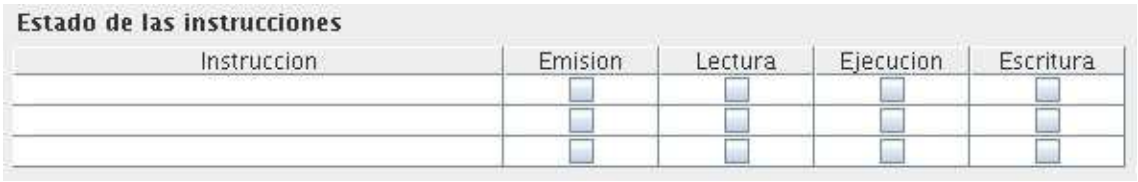
- **Lectura de operandos**: El marcador vigila la disponibilidad de los operandos fuente. Un operando fuente está disponible si ninguna instrucción activa emitida previamente, va a escribirlo. Cuando los operandos fuente están disponibles, el marcador indica a la unidad funcional que proceda a leer los operandos de los registros y que comience la ejecución. El marcador gestiona los riesgos RAW en este paso, y las instrucciones pueden ser enviadas a ejecución fuera de orden. Este paso junto con el de emisión, completa la función del paso *ID* en el procesador segmentado MIPS básico.

- **Ejecución**: La unidad funcional comienza la ejecución sobre los operandos recibidos. Cuando el resultado está listo, se notifica al marcador que se ha completado la ejecución. Este paso sustituye al paso *EX* en la procesador segmentado MIPS y emplea múltiples ciclos en un procesador segmentado MIPS en punto flotante.

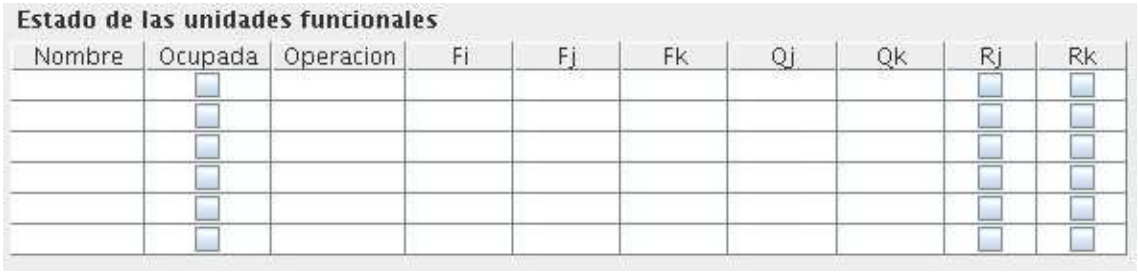
- **Escritura del resultado** Una vez que el marcador es consciente de que la unidad funcional ha completado la ejecución, el marcador comprueba los registros WAR y detiene la finalización de la instrucción, si fuera necesario. En general, a una instrucción que está finalizando no se le permitirá escribir sus resultados cuando hay una instrucción que no ha leído sus operandos y que precede (en una emisión en orden) a la instrucción que está finalizando, y si uno de los operandos está en el mismo registro que el resultado de la instrucción que se está completando.

  Cuando no existe este riesgo WAR, o cuando desaparece, el marcador indica a la unidad funcional que almacene su resultado en el registro destino. Este paso sustituye al paso WB de cauce MIPS básico.

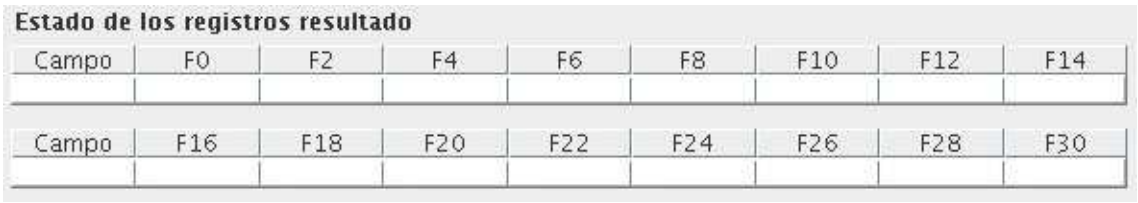
Debido a que los operandos para una instrucción sólo se leen cuando están disponible en el archivo de registros, el Marcador no se aprovecha de la técnica de anticipación. En cambio, los registros sólo se leen cuando están ambos disponibles. Esto no es una penalización tan grande como podría parecer inicialmente. A diferencia del procesador segmentado básico de cinco etapas, las instrucción escribirán su resultado en los registros tan pronto como completen su ejecución (asumiendo que no hay riesgos WAR), más que esperar una asignación estática del *slot* de escritura que puede necesitar varios ciclos. Tiene como efecto la reducción de latencia y los beneficios de la anticipación. Sigue habiendo un ciclo adicional de latencia, puesto que la escritura de resultados y la lectura de operandos no se puede solapar. Para eliminar este ciclo se necesitaría un buffer adicional.



(a) Tabla de instrucciones



(b) Tabla de las unidades funcionales



(c) Tabla de registros

Figura 6.5: Tablas de Marcador

Basado en su propia estructura de datos, el Marcador controla mediante comunicación con las unidades funcionales, la progresión de las instrucciones de un paso al siguiente. Tiene, sin embargo, una pequeña complicación. Sólo hay un número limitado de buses desde los operandos fuente y resultado al archivo de registros, lo cual representa un riesgo estructural. El marcador debe garantizar que el número de unidades funcionales permitidas para proceder en los pasos 2 y 4 no superen el número de buses disponible.

6.4.2. Estructura del Marcador

El Marcador está constituido por tres tablas:

- **Estado de las instrucciones:** indica en cual de los cuatro pasos del Marcador está la instrucción (ver Figura 6.5(a)).
- **Estado de las unidades funcionales**(ver Figura 6.5(b)): indica el estado de

la unidad funcional (FU). Hay nueve campos para cada unidad funcional:

- *Ocupado*: indica si la unidad está ocupada o no.
- *Op*: operación a realizar en la unidad (por ejemplo: suma o resta).
- F_i: Registro destino.
- F_j, F_k: número de los registros fuente.
- Q_j, Q_k: unidades funcionales que producen los registros fuentes F_j, F_k.
- R_j, R_k: etiquetas que indican cuando F_j y F_k están listos.

- **Estado de los registros resultado**(ver Figura 6.5(c)): indica que unidad funcional va a escribir cada registro, si una instrucción activa tiene el registro como destino. Este campo está en blanco si no hay ninguna instrucción pendiente de escribir en ese registro.

6.4.3. Funcionamiento del marcador

La tabla 6.1 muestra las condiciones que el marcador comprueba para que avance cada instrucción y las acciones necesarias cuando avanza la instrucción.

Estado de la instrucción	**Espera hasta**	**Acción**
Emisión	$\overline{ocupado[FU]}$ & $\overline{resultado[D]}$	$Ocupado[FU] \leftarrow$ Sí; $Op[FU] \leftarrow$op; $F_i[FU] \leftarrow D$; $F_j[FU] \leftarrow S_1$; $F_k[FU] \leftarrow S_2$; $Q_j \leftarrow Resultado[S_1]$; $Q_k \leftarrow Resultado[S_2]$; $R_j \leftarrow notQ_j$; $R_k \leftarrow notQ_k$; $Resultado[D] \leftarrow FU$;
Lectura de operandos	R_j and R_k	$R_j \leftarrow$No; $R_k \leftarrow$No; $Q_j \leftarrow 0$; $Q_k \leftarrow 0$
Ejecución	Finaliza unidad funcional	
Escritura de resultados	$\forall f((F_j[f] \neq F_i[FU]$ or $R_j[f] =$ No)& $(F_k[f] \neq F_i[FU]$ or $R_k[f]=$No))	$\forall f(if Q_k[f] =$FU$then R_k[f] \leftarrow$Sí); $\forall f($if $Q_j[f] = FU$ then $R_j[f] \leftarrow$Sí); Resultado$[F_i[FU]] \leftarrow 0$; $Ocupado[FU] \leftarrow$No)

Cuadro 6.1: Comprobaciones requeridas y acciones en cada paso de la ejecución de una instrucción

En esta tablas se utiliza la siguiente notación, FU indica la unidad funcional utilizada por la instrucción que estamos procesando, D es el registro destino, S_1 y S_2 son los registros fuente, y *op* es la operación básica que se va a realizar. Para acceder a campo de una unidad funcional utilizamos la notación $Campo[FU]$. Por ejemplo, para denotar el campo F_j de la unidad funcional FU, utilizamos el valor $F_j[FU]$. $Resultado[D]$ es el valor del campo de resultado del registro para el registro D.

La comprobación en el caso de escritura de resultado impide la escritura cuando hay un riesgo WAR, que existe si otra instrucción tiene este registro destino de la instrucción ($F_i[FU]$) como operando fuente ($F_j[f]$ o $F_k[f]$), y si esta otra instrucción no ha leido el registro (R_j=No o R_k=No).

6.4.4. Ventajas e inconvenientes

El Marcador utiliza el ILP disponible para minimizar el número de detenciones procedentes de las dependencias de datos verdaderas del programa. En la eliminación de detenciones, el Marcador está limitado por varios factores:

- **La cantidad de paralelismo existente entre instrucciones.** Esto determina si es posible encontrar instrucciones independientes para ejecutar. Si cada instrucción depende de su predecesora, el esquema de planificación dinámica no disminuyen las detenciones.

- **El número de entradas de marcador.** Esto determina cuanto puede avanzar el procesador segmentado buscando instrucciones independientes. El conjunto de instrucciones examinadas como candidatas para una posterior ejecución se llama ventana. El tamaño del marcador determina el tamaño de la ventana.

- **El número y los tipos de las unidades funcionales.** Determina la importancia de los riesgos estructurales, los cuales pueden aumentar si se utiliza planificación dinámica.

- **La presencia de antidependencias y dependencias de salida.** Esto da lugar a las detenciones WAR y WAW.

El segundo y el tercer factor se pueden reducir, al incrementar el tamaño del Marcador y el número de unidades funcionales; sin embargo estos cambios tienen consecuencias en el coste y pueden afectar también al tiempo del ciclo. Los riesgos WAW y WAR se vuelven más importantes en procesadores con planificación dinámica porque el cauce revela más dependencias de nombre. Los riesgos WAW también se hacen más importantes con la planificación dinámica con un esquema de predicción de salto que permite solapar múltiples iteraciones de un bucle.

6.5. Técnica de Tomasulo

La unidad de punto flotante del IBM 360/91, contaba con una aproximación clave que permitía avanzar a la ejecución en presencia de dependencias. Inventado por Robert Tomasulo, este esquema sigue la pista de cuando los operandos de las instrucción están disponibles, para minimizar los riesgos RAW, e introduce el renombrado de registros para eliminar los riesgos WAW y WAR. Hay muchas variaciones de este esquema en procesadores modernos aunque siguen siendo características comunes el concepto clave de seguir la pista de las dependencias de instrucciones para permitir ejecutarlas tan pronto como los operandos estén disponibles, y el renombrado de registros para evitar riesgos WAR y WAW.

Nos centraremos en la unidad de punto flotante y la unidad de carga/almacenamiento, en el contexto de un conjunto de instrucciones MIPS. La propuesta inicial de Tomasulo consideraba unidades funcionales segmentadas, más que múltiples unidades funcionales, pero aquí se tratara el algoritmo como si tuviera múltiples unidades funcionales, ya que añadir esas unidades funcionales segmentadas al procesador segmentado es una simple extensión conceptual.

Como hemos visto anteriormente, los riesgos RAW se gestionan ejecutando las instrucciones sólo cuando sus operandos están disponibles. Los riesgos WAR y WAW,

que surgen de dependencias de nombres, se eliminan mediante el renombrado de registros. El renombrado de registros elimina esos riesgos renombrado los registros destino, incluyendo aquellos que tienen una lectura o escritura pendiente de una instrucción anterior, de esta forma, la escritura fuera de orden no afecta a ninguna instrucción que dependa del valor anterior de un operando.

En el esquema de Tomasulo, el renombrado de registros se proporciona gracias a las estaciones de reserva, que almacenan los operandos de las instrucciones que están esperando para emitirse. La idea básica consiste en buscar las estación de reserva que necesiten el dato y almacenar un operando tan pronto como esté disponible, eliminando la necesidad de traer el operando de un registro. Además, las instrucciones pendientes señalan la estación de reserva que les va a proporcionar su entrada. Finalmente, cuando se solapan en ejecución sucesivas escrituras a un registro, sólo se actualiza la última. Puesto que puede haber más estaciones de reserva que registros, esta técnica puede eliminar riesgos que surgen de dependencias de datos que incluso no pueden ser eliminados por un compilador.

El uso de estaciones de reserva, más que un archivo de registros centralizado, da lugar a dos propiedades importantes. En primer lugar, la detección de riesgos y el control de la ejecución están distribuidos: la información mantenida en las estaciones de reserva de cada unidad funcional determina si una instrucción puede empezar su ejecución en esa unidad. En segundo lugar, los resultados pasan directamente a la unidad funcional desde las estaciones de reserva donde estaban almacenados, en vez de acceder a ellos a través de los registros. Se hace esta anticipación con un bus común de resultados, que permite cargar simultáneamente un resultado en todas las unidades funcionales que lo estaban esperando (en el 360/91 se llama *common data bus*, o CDB). En procesadores segmentados con múltiples unidades de ejecución y emisión múltiple de instrucciones en cada ciclo se puede necesitar más de un bus.

La Figura 6.6 muestra la estructura básica de un procesador MIPS que utiliza el algoritmo de Tomasulo, incluyendo la unidad de punto flotante y la unidad de carga/almacenamiento. Cada estación de reserva almacena una instrucción que ya ha sido emitida y que está a la espera de ejecutarse en la unidad funcional, y también

almacena los valores de los operandos de la instrucción si ya han sido calculados, o si no, los nombres de las estaciones de reserva que los proporcionarán.

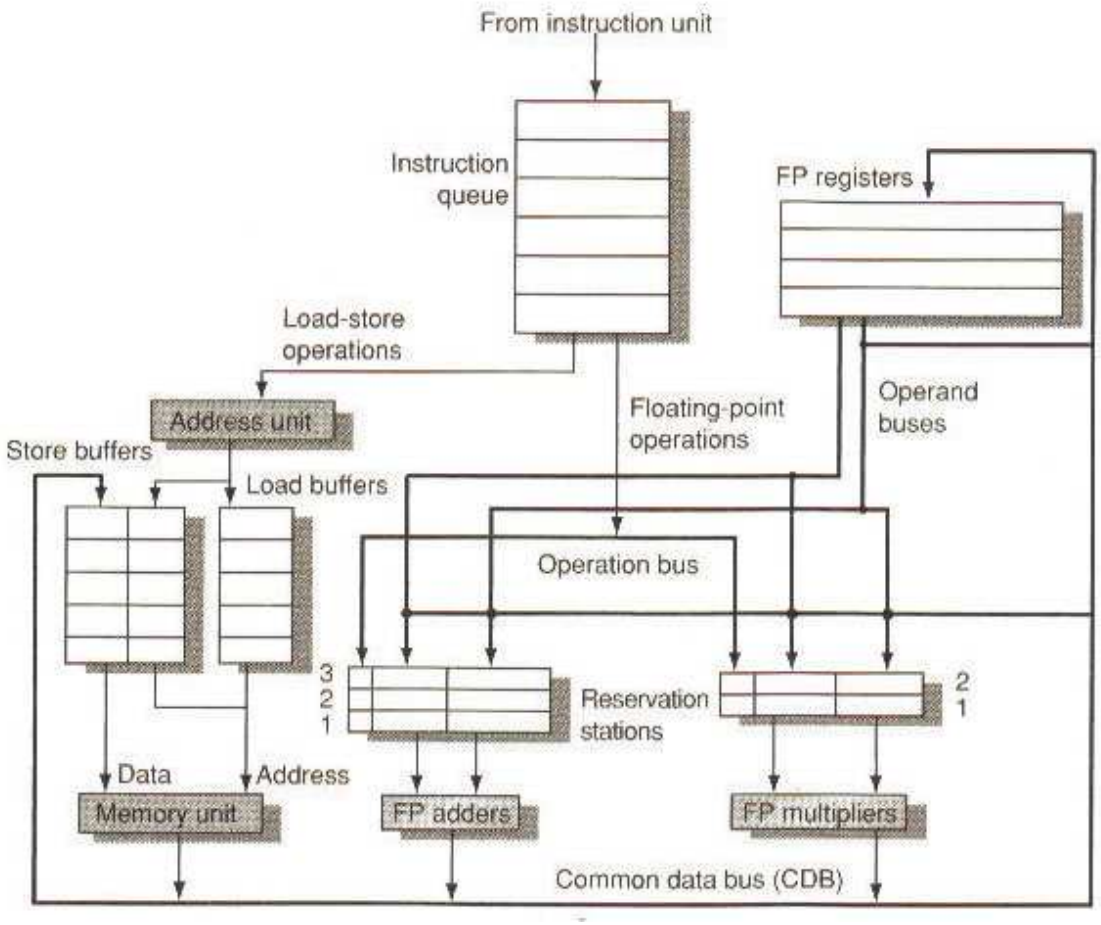


Figura 6.6: Estructura básica de un procesador MIPS de punto flotante usando el algoritmo de Tomasulo.

Los buffers de carga y almacenamiento contienen datos o direcciones que vienen y van a memoria, y se comportan casi como estaciones de reserva, por lo que de aquí en adelante sólo se distinguirán cuando sea necesario. Los registros de punto flotante están conectados por un par de buses a las unidades funcionales y por un único bus a los buffers de almacenamiento. Todos los resultados de las unidades funcionales y de memoria se envían al bus común de datos, que va a todos los sitios excepto al

buffer de carga. Todas las estaciones de reserva tienen etiquetas identificativas que se utilizan en el control del procesador segmentado.

6.5.1. Pasos en la ejecución del algoritmo de Tomasulo

Antes de describir en detalle las estaciones de reserva y el algoritmo, examinemos los pasos de una instrucción en el algoritmo de Tomasulo. Consta sólo de tres etapas (aunque cada una puede necesitar un número diferente de ciclos de reloj) ya que la estructura es drásticamente diferente a la de Marcador y a la del cauce clásico de cinco etapas.

- **Emisión:** Obtiene la instrucción de la cola de instrucciones, que se mantiene en orden FIFO para asegurar la conservación del flujo correcto de datos. Si la estación de reserva que se necesita está vacía, se emite la instrucción a la estación con los valores de los operandos que hay actualmente en el registro. Si no hay ninguna estación de reserva vacía, se produce un riesgo estructural y se detiene la instrucción hasta que la estación o el buffer sea liberado. Si los operandos no están en los registros, se mantiene el rastro de la unidad funcional que los escribirá. Este paso renombra los registros, eliminando los riesgos WAR y WAW.

- **Ejecución:** Si todavía no está disponible alguno de los operandos, se observa el *CDB* a la espera de ese valor. Cuando un operando está disponible, se sitúa en la estación de reserva correspondiente. Cuando todos los operandos están disponibles, la operación se puede ejecutar en la unidad funcional correspondiente. Se detectan los riesgos RAW y se retrasa la ejecución de instrucciones hasta que los operandos están disponibles. Observar que varias instrucciones puede estar preparadas en el mismo ciclo de reloj para la misma unidad funcional. Aunque unidades funcionales independientes pueden comenzar la ejecución de diferentes instrucciones en el mismo ciclo de reloj, si más de una instrucción está preparada para una unidad funcional, la unidad tiene que escoger una. Para las estaciones de reserva en punto flotantes, se puede escoger

arbitrariamente; sin embargo, las cargas y los almacenamientos, plantean una complicación adicional.

Las cargas y los almacenamientos necesitan un proceso de ejecución en dos pasos. En el primer paso se calcula la dirección efectiva si el registro base está disponible. Las cargas en el buffer de carga se ejecutan tan pronto como la unidad de memoria está disponible. En el buffer de almacenamiento se espera por el valor que va a ser almacenado antes de acceder a la unidad de memoria. Las cargas y almacenamientos se mantienen en el orden del programa, lo cual sirve para prevenir riesgos debidos a la memoria, como veremos después brevemente.

Para preservar el comportamiento de las instrucciones, no se permite que ninguna instrucción inicie su ejecución hasta que todos los saltos que la preceden en el orden del programa se hayan completado. Esta restricción garantiza que una instrucción que causa una excepción durante la ejecución realmente debería ser ejecutada.

- **Escritura del resultado:** Cuando el resultado está disponible, se escribe en el CDB y desde allí en el archivo de registros y en cualquier estación de reserva (incluidos los buffers de almacenamiento) que estén esperando por él. Las instrucciones de almacenamiento también escriben el dato en memoria durante este paso. Cuando la dirección y el dato están disponible, se envían a la unidad de memoria y se completa el almacenamiento

6.5.2. Estructura

Las estructuras de datos usadas para detectar y eliminar riesgos se adjuntan a las estaciones de reserva, al archivo de registros y a los buffer de carga y almacenamiento con información ligeramente diferente en cada uno de los objetos.

Al describir la operación de este esquema, se usa la terminología usada en la técnica de Marcador.

La primera tabla indica el estado de las instrucciones (ver Figura 6.7(a))

Estado de las instrucciones

Instruccion	Emision	Ejecucion	Escritura
	☐	☐	☐
	☐	☐	☐
	☐	☐	☐

(a) Tabla de las instrucciones

Estado de las estaciones de reserva

Nombre	Ocupada	Operacion	Vj	Vk	Qj	Qk	A
	☐						
	☐						
	☐						
	☐						
	☐						
	☐						
	☐						

(b) Tabla de las estaciones de reserva

Estado de los registros resultado

Campo	F0	F2	F4	F6	F8	F10	F12	F14

Campo	F16	F18	F20	F22	F24	F26	F28	F30

(c) Tabla de registros

Figura 6.7: Tablas de Tomasulo

La segunda tabla indica el estado de las estaciones de reserva. Cada estación de reserva tiene siete campos (ver Figura 6.7(b)):

- *Op*: La operación a realizar sobre los operandos fuente S_1 y S_2.
- Q_j, Q_k: Las estaciones de reserva que producirán el correspondiente operando fuente; un valor cero indica que el operando fuente ya esta disponible en V_j o V_k, o es innecesario.
- V_j, V_k: El valor de los operandos fuente. Observar que sólo uno de los campos V o Q es válido para cada operando.
- A: Se utiliza para almacenar información para el cálculo de la dirección de memoria en una carga o almacenamiento. Inicialmente, el desplazamiento de la instrucción se almacena aquí; después de calcular la dirección, se almacena la dirección efectiva.
- *Ocupado*: Indica que esta estación de reserva y su unidad funcional correspondiente están ocupadas.

El archivo de registros tiene como campo, Q_i (ver Figura 6.7(c)):

- Q_i El número de la estación de reserva que contiene la operación cuyo resultado debería ser almacenado en este registro. Si el valor de Q_i está en blanco ó 0, ninguna instrucción activa está calculando un resultado destinado a este registro, quiere decir que el valor es sencillamente el que contiene el registro.

Los buffers de carga y almacenamiento tienen un campo, A, que almacena el resultado de la dirección efectiva una vez que el primer paso de la ejecución se ha completado.

El esquema de Tomasulo ofrece importantes ventajas sobre esquemas anteriores o más simples:

1. la distribución de la lógica de detección de riesgos,

2. la eliminación de detecciones para los riesgos WAW y WAR.

La primera ventaja surge de las estaciones de reserva distribuidas y el uso del CDB. Si varias instrucciones están esperando por un resultado, y cada una de ellas tiene ya el otro operando, entonces son informadas simultáneamente mediante una difusión por el CDB. Si se usara un archivo de registros centralizado, las unidades tendrían que leer sus resultados desde los registros cuando estuviera disponible el bus de los registros.

La segunda ventaja, la eliminación de riesgos WAW y WAR, se debe al renombrado de registros usando estaciones de reserva, y al proceso de almacenar operandos en las estaciones de reserva tan pronto como están disponibles.

6.5.3. Funcionamiento de la técnica de Tomasulo

En la tabla 6.2 se describen todos los pasos y comprobaciones que debe realizar una instrucción.

Utilizamos la misma notación que en la subsección 6.4.3, pero necesitamos los siguientes valores adicionales: imm es el desplazamiento con extensión de signo, y r es la estación de reserva o buffer al que ha sido asignado la instrucción. RS es la estructura de datos de la estación de reserva. $RegisterStat$ es la estructura de datos de estado de registros (no el archivo de registro, que es $Regs[\,]$).

Si se emite una instrucción se actualiza el campo Q_i del registro destino, al número del buffer o la estación de destino a la que ha sido emitida. Si los operandos están disponibles en los registros, se almacenan en los campos V. De lo contrario, los campos Q indican la estación de reserva que producirán el valor que necesitan los operandos fuente. La instrucción espera en la estación de reserva hasta que están disponibles ambos operandos, indicado por el cero en el campo Q. El campo Q se pone a cero si la instrucción se ha emitido, o si la instrucción de la que depende esa se ha completado y ha escrito el resultado. Si una instrucción ha terminado su ejecución y el CDB está disponible, entonces escribe su resultado. Todos los buffers, registros y estaciones de reserva cuyos valores de Q_j y Q_k son iguales que

Estado de la instrucción	**Espera hasta**	**Acción**
EMISIÓN		
Operación FP	Estación r vacía	if($RegisterStat[S_1].Q_i \neq 0$) $\{RS[r].Q_j \leftarrow RegisterStat[S_1].Q_i\}$ else$\{RS[r].V_j \leftarrow Regs[S_1]$; $RS[r].Q_j \leftarrow 0\}$; if($RegisterStat[S_2].Q_i \neq 0$) $\{RS[r].Q_k \leftarrow RegisterStat[S_2].Q_i\}$ else$\{RS[r].V_k \leftarrow Regs[S_2]$; $RS[r].Q_k \leftarrow 0\}$; $RS[r].Ocupado \leftarrow$Sí; $RegisterStat[D].Q_i = r$;
Carga o almacenamiento	Buffer r vacío	if($RegisterStat[S_1].Q_i \neq 0$) $\{RS[r].Q_j \leftarrow RegisterStat[S_1].Q_i\}$ else$\{RS[r].V_j \leftarrow Regs[S_1]$; $RS[r].Q_j \leftarrow 0\}$; $RS[r].A \leftarrow imm$; $RS[r].Ocupado \leftarrow$Sí;
Solo carga		$RegisterStat[S_2].Q_i = r$;
Solo almacenamiento		if($RegisterStat[S_2].Q_i \neq 0$) $\{RS[r].Q_k \leftarrow RegisterStat[S_1].Q_i\}$ else$\{RS[r].V_k \leftarrow Regs[S_2]$;$RS[r].Q_k \leftarrow 0\}$;
EJECUCIÓN		
Operación FP	$RS[r].Q_j$=0 & $RS[r].Q_k$=0	Calcular el resultado: los operandos están en V_j y V_k
Paso 1 carga o almacenamiento	$RS[r].Q_j$=0 & r está en la cabeza de la cola de carga / almacenamiento	$RS[r].A \leftarrow RS[r].V_j + RS[r].A$;
Paso 2 de carga	Paso 1 completo	Leer desde $Mem[S_1[r].A]$;
ESCRITURA DEL RESULTADO		
Operación FP o carga	Ejecución completa en r & CDB disponible	$\forall$x(if($RegisterStat[x].Q_i = r$) $\{Regs[x] \leftarrow$resultado; $RegisterStat[x].Q_i \leftarrow 0\}$) $\forall$x(if($RS[x].Q_j = r$) $\{RS[x].V_j \leftarrow$resultado;$RS[x].Q_j \leftarrow 0\}$); $\forall$x(if($RS[x].Q_k = r$)$\{RS[x].V_k \leftarrow$resultado; $RS[x].Q_k \leftarrow 0\}$);$RS[r].Ocupado \leftarrow$No;
Almacenamiento	Ejecución completa en r & $RS[r].Q_k = 0$;	$Mem[RS[r].A] \leftarrow RS[r].V_k$; $RS[r].Ocupado \leftarrow$No;

Cuadro 6.2: Pasos del algoritmo y qué se requiere en cada paso

el número de la estación de reserva que está escribiendo, actualizan sus valores con el dato del CDB y marca el campo Q indicando que se ha recibido ese valor. De esta manera, el CDB puede difundir su resultado a muchos destinatarios en un ciclo de reloj. Las cargas necesitan dos pasos en ejecución, y los almacenamientos funcionan ligeramente diferente durante la escritura de resultados, donde pueden tener que esperar por el valor a almacenar. Para asegurar el comportamiento de las excepciones, ninguna instrucción se puede ejecutar si hay un salto previo en el orden del programa que no se ha completado. Debido a que muchos conceptos del orden del programa no se mantiene después de la etapa de emisión, está restricción se implementa normalmente para prevenir que ninguna instrucción abandone el paso de emisión, si hay un salto pendiente en el procesador segmentado.

Al utilizar estaciones de reserva, se pueden permitir múltiples ejecuciones simultáneas de un bucle si se utiliza predicción de saltos. Esta ventaja se consigue sin cambiar el código, de hecho, el bucle es desenrollado dinámicamente por el hardware, utilizando las estaciones de reserva como si fueran registros adicionales gracias al renombrado de registros.

Una carga y un almacenamiento se pueden hacer de una forma segura en un orden diferente, proporcionándoles direcciones de acceso diferentes. Si una carga y un almacenamiento acceden a la misma dirección, entonces:

- La carga es anterior al almacenamiento en el orden del programa, existe una antidependencia.
- El almacenamiento es anterior a la carga en el orden del programa, hay una dependencia verdadera.

Igualmente, al intercambiar dos almacenamientos a la misma dirección da lugar a un riesgo WAW.

Por lo tanto, para determinar si una carga puede ser almacenada en un momento dado, el procesador debe comprobar si hay algún almacenamiento incompleto que preceda a la carga en el orden del programa que tenga la misma dirección de

memoria que la carga. Asimismo, un almacenamiento debe esperar si hay cargas o almacenamientos con la misma dirección de memoria, que lo precedan en el orden del programa y que aún no se hayan ejecutado.

Para detectar tales riesgos, el procesador debe calcular la dirección de memoria asociada con cualquier operación anterior. Una forma simple, pero no necesariamente óptima, de garantizar que el procesador tiene las direcciones correctas, es efectuar en el orden del programa los cálculos de las direcciones efectivas. En realidad, sólo es necesario mantener el orden relativo entre almacenamientos y las otras referencias a memoria; es decir, las cargas se pueden reordenar libremente.

Consideremos primero la situación de las cargas. Si se calcula una dirección efectiva de acceso a memoria en el orden del programa, entonces si una carga ha completado el cálculo de su dirección efectiva de acceso a memoria, se puede comprobar si hay un conflicto examinando el campo A de todas las entradas activas en los buffers de almacenamiento, la instrucción de carga no se enviará al buffer de carga hasta que se solucione el conflicto del almacenamiento. Algunas implementaciones anticipan el valor directamente a la carga desde el almacenamiento, reduciendo el retraso para este riesgo RAW.

De modo parecido operan los almacenamientos, excepto que el procesador debe comprobar los conflictos tanto en los buffers de carga como en los de almacenamiento, puesto que los almacenamientos conflictivos no pueden ser reordenados con respecto a una carga ni a otro almacenamiento.

Un procesador segmentado con planificación dinámica puede dar un alto rendimiento, siempre que se predigan los saltos con bastante precisión. El principal inconveniente de esta aproximación es la complejidad del esquema de Tomasulo, ya que requiere una gran cantidad de hardware. En particular, cada estación de reserva necesita un buffer de alta velocidad, así como lógica de control compleja. Finalmente, el rendimiento se puede ver limitado si hay un único CDB. Aunque se pueden añadir CDBs adicionales, cada uno de ellos debe interactuar con cada una de las estaciones de reserva, y sería necesario duplicar el hardware de asociación de coincidencia de etiquetas para cada estación por cada CDB.

El esquema de Tomasulo es especialmente atractivo si el diseñador necesita segmentar una arquitectura para la cual es difícil planificar el código, tiene escasez de registros, o se desea obtener un gran rendimiento sin una compilación de segmentación específica. Por otro lado, las ventajas de la aproximación de Tomasulo son menores que los costes de implementación, frente a una planificación del compilador para un procesador segmentado eficiente de una única emisión. Pero, como los procesadores son cada vez más agresivos en su capacidad de emisión y los diseñadores se preocupan por el rendimiento del código difícil de planificar (como la mayor parte del código no numérico), técnicas como el renombrado de registros y la planificación dinámica son cada vez más importantes. Además, esta aproximación ha sido muy popular en los últimos años debido al papel de la planificación dinámica como base para la especulación hardware.