PROCESADORES SUPER-ESCALARES

ARQUITECTURA DEL PROCESADOR II

1. De los procesadores escalares a los procesadores super-escalares

En la materia ya hemos visto el pipeline de 5 etapas. [1] La idea básica entonces fue que el ciclo de ejecución de la instrucción puede ser descompuesto en etapas no superpuestas, con una instrucción pasando a través de cada etapa en cada uno de los ciclos. Así el llamado procesador *escalar*, tiene un rendimiento (en inglés *troughput*) de 1, es decir, idealmente el número de *Instrucciones Por Ciclo* (IPC) fue 1.
Haciendo memoria recordaremos que la fórmula de tiempo de la CPU es:

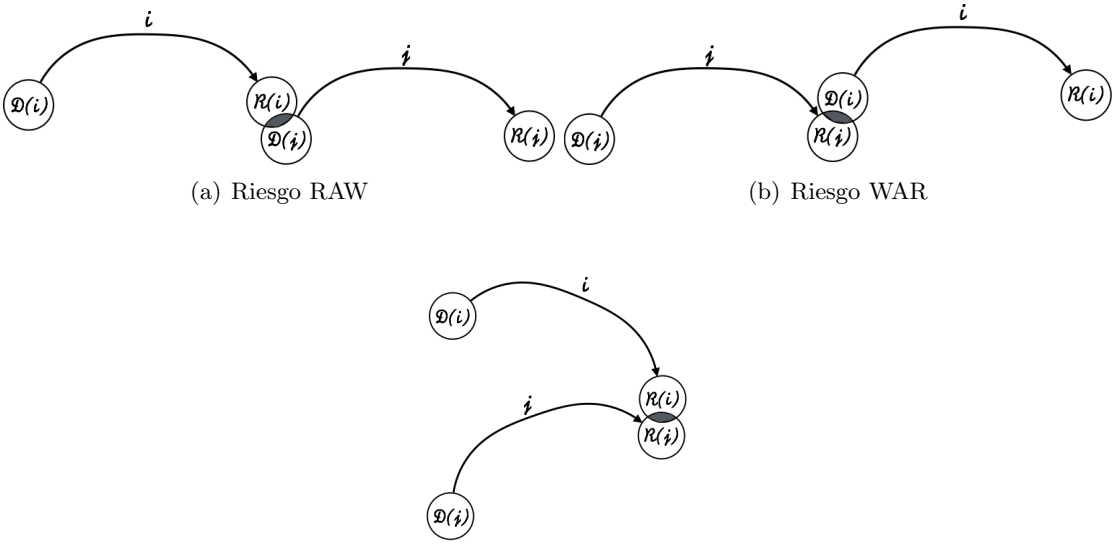
$$CPU_{time} = Cantidad\,de\,Instrucciones \times CPI \times tiempo\,de\,ciclo$$

se observa que para reducir el tiempo de CPU de un procesador manteniendo la Arquitectura del Conjunto de Instrucciones (ISA por sus siglas en inglés)–es decir, sin cambiar la cantidad de instrucciones CI – se debe reducir el CPI (o incrementar el IPC) o reducir el tiempo del ciclo de reloj, o ambas cosas. Veamos las opciones.
La única posibilidad de incrementar el IPC ideal de 1 es modificar radicalmente la estructura del pipeline, permitiendo que en determinado momento, más de una instrucción se encuentre en cada etapa. Al hacerlo, se hace la transición de un procesador escalar a uno super-escalar. Desde el punto de vista de la micro-arquitectura, se requiere un pipeline más *ancho* en el sentido que su representación ya no es una única línea donde ejecuta una sola instrucción. El efecto más evidente es que ahora necesitamos varias unidades funcionales, pero como veremos enseguida, cada etapa del procesador se verá afectada.
La segunda opción es reducir el tiempo de ciclo de reloj, lo que se logra incrementando la frecuencia de reloj. Para eso una etapa debe ser descompuesta en etapas más pequeñas y el pipeline completo se hace más *profundo*.
Los micro-procesadores modernos son más *anchos* y *profundos* que los procesadores de cinco estapas que hemos visto hasta ahora. Para estudiar las decisiones de diseño necesarias para implementar la concurrencia causada por el efecto super-escalar y las consecuencias de un pipeline más profundo, es conveniente distinguir entre el *extremo-inicial* (en inglés *front-end*) y el *extremo-final* (en inglés *back-end*). El extremo-inicial, el cual corresponde a las etapas IF e ID, ahora debe recuperar y decodificar varias instrucciones al mismo tiempo. El número m de instrucciones producidas (idealmente) en cada ciclo, define al procesador como un super-escalar de *m-vías*. El extremo-final, que corresponde con las etapas EX, MEM y WB debe ejecutar varias instrucciones concurrentemente y también escribir los resultados que generen.
Los micro procesadores super-escalares pueden ser divididos en dos categorías. En ambos casos, las instrucciones avanzan a través del extremo-inicial en el orden del programa. En un super-escalar *en-orden* o *estático*, las instrucciones dejan el extremo inicial en el

[1] Este material de estudio es la traducción de la sección 3.1 y 3.3 del libro de Jean-Loup Baer [1]

orden estricto del programa y todas las dependencias de datos son resueltas antes que las instrucciones pasen al extremo-final. En un super-escalar *fuera-de-orden* o *dinámico* las instrucciones pueden dejar el extremo-inicial y ejecutar en el extremo-final antes que algunas de las instrucciones que le predeceden. En un super-escalar dinámico la etapa de WB ahora es denominada etapa de *consolidación* (en inglés *c*ommit), esta última etapa debe ser diseñada de forma tal que se respete la semántica (lógica) del programa, es decir que los resultados deben ser almacenados en el orden pretendido por el código fuente. Se puede decir que los procesadores fuera-de-orden son más complejos de diseñar que los super-escalares en-orden. Se estima que para el mismo número de unidades funcionales, ellos requieren un 30 % más de lógica y naturalmente, la complejidad de diseño se traduce en demoras para ganar mercado.

Los riesgos en la planificación dinámica. Como consecuencia de que tanto el comienzo como la finalización de la ejecución de las instrucciones pueden ocurrir en un orden diferente al establecido por el programa, existen tres tipos de riesgos entre las instrucciones que son dependientes. Consideremos dos instrucciones i y j, donde i precede a j en el orden del programa y además existe algún tipo de dependencia entre ellas. Teniendo en cuenta que la ejecución de las instrucciones producen cambios en el estado del procesador de manera semejante a funciones, se puede considerar que los registros empleados como operandos representan el dominio de dichas funciones y el rango serían los registros donde se preservan los resultados obtenidos. Es posible determinar la existencia de riesgo de dato a partir de la intersección del dominio y el rango de las instrucciones i y j mientras permanecen dentro de la tubería. El primer tipo de riesgo de



(a) Riesgo RAW

(b) Riesgo WAR

(c) Riesgo WAW

FIGURA 1. **Riesgos en ejecución fuera de orden.** a)La intesección del rango de i y el dominio de j es distinto de vacío. b)La intesección del dominio de i y el rango de j es distinto de vacío. c)La intesección del rango de i y el rango de j es distinto de vacío.

dato es RAW (read after write) que ocurre cuando la intersección del rango de i ($R(i)$) y el dominio de j ($D(j)$) es distinto de vacío; hay riesgo RAW cuando $R(i) \cap D(j) \neq \emptyset$. Ver Figura 1 a. El segundo tipo de riesgo de dato es WAR (write after read) que ocurre cuando la intersección del dominio de i y el rango de j es distinto de vacío; hay riesgo WAR cuando $D(i) \cap R(j) \neq \emptyset$. Ver Figura 1 b. El tercer tipo de riesgo de dato es WAW (write after write) que ocurre cuando la intersección del rango de i y el rango de j es distinto de vacío; hay riesgo WAW cuando $R(i) \cap R(j) \neq \emptyset$. Ver Figura 1 c.

Volviendo al desempeño de estos procesadores, en teoría si tenemos un super-escalar de m-vías con una frecuencia de reloj k veces mayor que la de un procesador escalar con la misma ISA, se debería conseguir una aceleración (o *speedup* en inglés) de mk. Mientras que esta ganancia teórica podría ser alcanzada para m y k pequeños, existen factores tecnológicos, de diseño y de comportamiento que limitan el incremento de m y de k. La que sigue es una lista no exhaustiva de los factores más importantes:

- Para sostener la ejecución concurrente de varias instrucciones, es decir cuando hay un m grande, es necesario descubrir una gran cantidad de paralelismo a nivel de instrucción (ILP por sus siglas en inglés). Esto es especialmente difícil en los procesadores en-orden, en donde las instrucciones que tienen dependencias reales (riesgos RAW) no pueden abandonar el extremo-inicial hasta que no se resuelvan las dependencias. En un procesador fuera-de-orden hay más posibilidades de encontrar instrucciones independientes, pero esto se logra con un mayor costo en diseño y complejidad. Si bien algunos investigadores anticipan super-escalares de 8-vías y aún de 16-vías, actualmente no hay implementaciones de más de 6-vías y la tendencia es no incrementar m.
- Un segundo factor que limita m es que el extremo-final requiere al menos m unidades funcionales. Si bien la implementación de un gran número, digamos n, de unidades funcionales no está limitada por la cantidad de lógica necesaria, el número de caminos de adelantamientos crece como n^2. Algunos de estos caminos necesitarán ser largos y los cables largos impiden que el adelantamiento se realice en un único ciclo.
- Varios factores imponen límites en la mejora del tiempo de reloj. Una primera limitación, como ya se mencionó, es que la disipación de calor se incrementa con la frecuencia. Una segunda restricción es que los registros del pipe (es decir, el almacenamiento estable entre etapas, se debe leer y escribir en cada ciclo), esto impone un límite físico inferior sobre el tiempo del ciclo.
- Los pipelines más profundos (es decir, aquellos que tienen un k grande), presentan una desventaja importante. La resolución de cualquier actividad especulativa, como la predicción de saltos, se conoce tarde en el pipe. En el caso de una mala predicción la recuperación es demorada y puede contribuir a una pérdida en el desempeño (un IPC bajo) comparado con un pipe más corto. La tendencia de pipes más largos, como el caso del Pentium IV con una penalidad de 20 ciclos por error en la predicción de saltos vs sólo 10 etapas en el Pentium III, fue revertida en la Arquitectura del Intel Core, el cual tiene un pipe de 14 etapas.

Los super-escalares en-orden, son los sucesores lógicos de los procesadores segmentados convencionales y la produción de super-escalares comenzó con ellos. Cuando los micro-procesadores en-orden y fuera-de-orden se hicieron disponibles, la relativa sencillez de implementación de los procesadores en-orden permitió una mayor velocidad de reloj. Hasta mediados de los 90 el desempeño de los procesadores en-orden dominó a los fuera-de-orden, los cuales tenían una velocidad de reloj más baja, pero un IPC más alto. Después de esa etapa, los procesadores fuera-de-orden tomaron el control, en parte gracias a la ley de Moore con la cual se permite mayor funcionalidad y más hardware de asistencia en el mismo chip.
Hoy en día los micro-procesadores de alto desempeño con un solo procesador son exclusivamente fuera-de-orden. La ventaja de velocidad que tenían los procesadores en-orden ha desaparecido debido a que la disipación del calor ha podido superar la limitación que imponía el incremento en la frecuencia del reloj. Como consecuencia de los siguientes tres hechos: el límite impuesto a la velocidad por la disipación de calor, la validez de la ley de Moore, (lo que permite mayor integración y más lógica dentro de un chip) y finalmente debido a que el desempeño alcanzable con un único procesador no se puede incrementar sin agregar una complejidad de diseño extrema, los sistemas con un único procesador están siendo reemplazados por multi-procesadores. Si los multi-procesadores en un chip consistirán de procesadores en-orden simples o procesadores fuera-de-orden complejos, o una mezcla de ambos, es aún una pregunta abierta. Por eso es importante estudiar ambos tipos de arquitecturas. En la próxima sección se comienza con un ejemplo clásico de super-escalar en-orden.

2. Caso de estudio: El Scoreboar del CDC 6600

El objetivo del scoreboard es ejecutar una instrucción tan pronto como sea posible. [2] Si una instrucción no puede comenzar a ejecutar detiene a otras instrucciones que podrían ser despachadas y ejecutadas. Tanto la información considerada, como la decisión tomada, es total responsabilidad del scoreboard, ya que cada instrucción pasa a través suyo, y le permite crear un registro de las dependencias de datos; este paso corresponde a la emisión de instrucciones y reemplaza una parte de la etapa ID en el pipeline del MIPS. En base a ésto el scoreboard determina cuando una instrucción puede leer sus operandos y comenzar su ejecución. Si una instrucción no puede ejecutar inmediatamente, se considera cada cambio en el hardware hasta determinar que la instrucción **puede** ejecutar. También controla cuando una instrucción puede escribir su resultado en el registro destino. Luego, toda la detección y resolución de hazards está centralizada. A continuación se detallan las etapas por las que pasa una instrucción (solamente se consideran instrucciones de punto flotante por lo que no se considera la etapa de acceso a memoria).

Arquitectura del MIPS con scoreboard. Esta técnica de planificación dinámica fue usada en la primera supercomputadora de la historia, la CDC 6600, diseñada por Seymour Cray en 1962 (salió al mercado en 1964). La arquitectura CDC 6600 presenta 16 unidades funcionales separadas, incluyendo 4 unidades de punto flotante, 5 unidades para referencias a memoria y 7 unidades para operaciones enteras. La figura 2 muestra

[2] Este caso de estudio es la traducción de parte de la sección A7 del libro de Hennessy & Patterson [2]

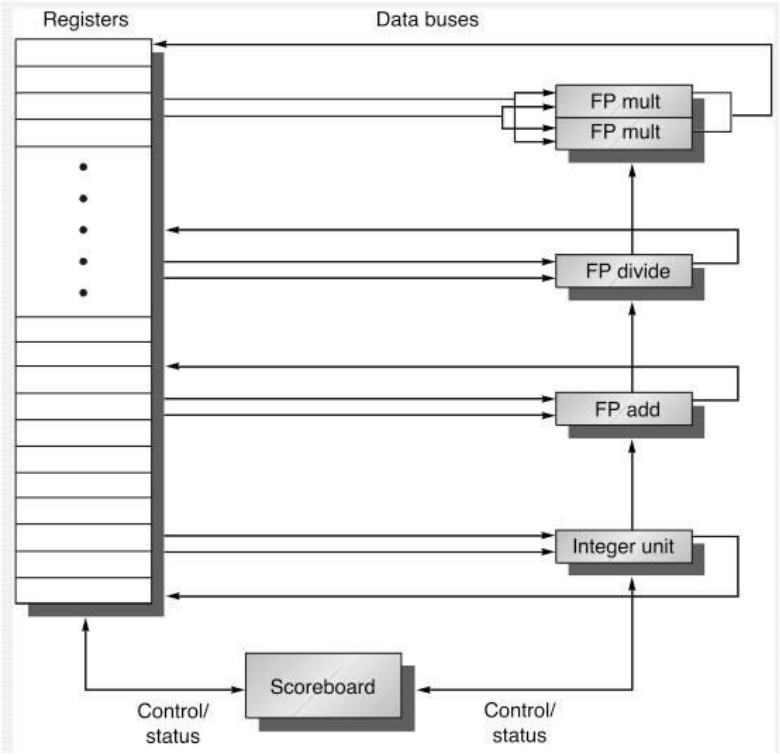


FIGURA 2. Estructura básica de un procesador MIPS con un scoreboard

la arquitectura del MIPS controlada con un scoreboard, consiste en 2 multiplicadores, un sumador, una unidad de división, una única unidad entera.

Etapas. Cada instrucción pasa por cuatro etapas, las cuales reeemplazan a las etapas de ID, EX y WB de pipeline estándar. Las etapas son:

1. **Etapa de Emisión o Issue**: Si está libre la unidad funcional que requiere la instrucción y ninguna otra instrucción activa tiene el mismo registro destino, el scoreboard emite la instrucción a la unidad funcional y actualiza su estructura interna. La detección de un riesgo WAW o estructural detiene la emisión de la instrucción que lo provoca y las que le siguen en el programa, hasta que el hazard sea solucionado. Asegurando que ninguna otra unidad funcional activa quiere escribir un resultado en el mismo registro destino, garantizamos que un hazard WAW no puede ocurrir.
2. **Etapa de Lectura de Operandos**: El scoreboard controla la disponibilidad de los operandos fuentes. Un operando fuente se encuentra disponible si ninguna instrucción activa ya emitida quiere escribir en él. Cuando los operandos fuentes están disponibles, el scoreboard indica a la unidad funcional que puede proceder a leer los operandos desde los registros y comienzar la ejecución. De esta manera, los hazards RAW son resueltos dinámicamente en esta etapa, y las instrucciones pueden ser enviadas a ejecución fuera de orden.

3. **Etapa de Ejecución**: La unidad funcional comienza la ejecución cuando recibe los operandos. Cuando el resultado está listo, notifica al scoreboard que ha completado la ejecución.
4. **Etapa de Escritura de Resultados**: Una vez que el scoreboard conoce que la unidad funcional ha completado su ejecución, chequea por hazard WAR y detiene la instrucción, si es necesario.

Ejemplo de Hazard WAR. A continuación veremos un ejemplo para mostrar como el scoreboard soluciona hazards WAR que pueden surgir en un código, para ésto considere el siguiente fragmento de código:

```
div.d   f0 , f2, f4
add.d   f10, f0, f8
sub.d   f8 , f8, f14
```

Podemos observar que existe un hazard WAR entre la instrucción de suma y la de resta, debido a que la primera posee a F8 como operando fuente, el cual es también destino para la instrucción de resta. Sin embargo, la suma depende de la instrucción de división. El scoreboard detendrá la instrucción de resta en su etapa de escritura de resultados hasta que la suma lea sus operandos. En general, a una instrucción no se le permite escribir sus resultados cuando:

- Existe una instrucción que le precede en orden de emisión pero aún no ha leido sus operandos , y
- Uno de esos operandos es el registro destino de la instrucción a punto de finalizar.

Estructuras de Datos del Scoreboard. En esta sección se detallan las estructuras de datos mantenidas por un MIPS con scoreboard.
El scoreboard está compuesto de tres tablas:

1. *Estado de Instrucción*: Indica que etapa ha superado cada una de las instrucciones.
2. *Estado de las Unidades Funcionales*: Indica cual es el estado de las unidades funcionales, FU. Hay nueve campos en cada unidad funcional:
   - *Busy*: Indica si la unidad está ocupada o no.
   - *Op*: La operación que se realiza en la unidad (por ejemplo suma o resta).
   - F_i: El registro destino.
   - F_j, F_k: Los registros fuentes.
   - Q_j, Q_k: Nombre de la unidad funcional que producirá los resultados para los registros fuentes F_j, F_k.
   - R_j, R_k: Flags indicando cuando F_j y F_k se encuentran listos y aún no han sido leidos. Estos flags son seteados a **No** después de leer los operandos.
3. *Estado de los Registros Resultados*: Indica que unidad funcional escribirá en un registro determinado. Para aquellos registros sobre los cuales ninguna instrucción deba escribir, este campo permanecerá en blanco.

Algoritmo del Scoreboard. A continuación se detalla como trabaja el algoritmo scoreboard en cada etapa.

1. Emisión o Issue (Ver Tabla 1). Durante la etapa de emisión o issue, el scoreboard espera hasta que la unidad funcional requerida para ejecutar la instrucción a emitir se libere (chequeo por hazard estructural) y ninguna otra instrucción activa

Esperar hasta	**Tareas**
Not Busy[FU] y no Result[D]	Busy[FU] ← yes; Op[FU] ← op; F_i ← D F_j[FU] ← S1; F_k[FU] ← S2; Q_j ← Result[S1]; Q_k ← Result[S2]; R_j ← No Q_j; R_k ← No Q_k; Result[D] ← FU;

Tabla 1. Algoritmo de Scoreboard - Etapa Issue

quiera escribir en el mismo registro (chequeo por hazard WAW). Una vez que estas condiciones se han cumplido la instrucción se emite y procede a realizar las siguientes actualizaciones sobre sus estructuras de datos:

- Busy[FU] ← yes, indica que la unidad funcional FU se encuentra ocupada;
- Op[FU] ← op, indica la operación que realizará la unidad funcional FU;
- F_i ← D, indica en que registro quedará el resultado;
- F_j[FU] ← S1, indentifica el registro empleado como primer operando fuente;
- F_k[FU] ← S2, indentifica el registro empleado como segundo operando fuente;
- Q_j ← Result[S1], indica que unidad funcional producirá el resultado para el primer operando, obtenido desde la estructura de estado de registros de resultados;
- Q_k ← Result[S2], indica que unidad funcional producirá el resultado para el segundo operando, obtenido desde la estructura de estado de registros de resultados;
- R_j ←, coloca **No** si alguna unidad fucional producirá un resultado para el primer operando, **Yes** en otro caso;
- R_k ←, coloca **No** si alguna unidad fucional producirá un resultado para el segundo operando, **Yes** en otro caso;
- Result[D] ← FU, coloca en la estructura que mantiene el estado de registros de resultados, que la unidad funcional FU producirá un resultado para el registro destino.

Una vez que todas estas tareas han sido completadas por una instrucción se considera que ha finalizado la etapa de emisión.

2. Lectura de Operandos (Ver Tabla 2). Luego de que la instrucción ha sido emitida avanza a la etapa de lectura de operandos. En esta etapa el scoreboard debe esperar hasta que los flags R_j y R_k se activen indicando que los operandos ya se encuentran disponibles en el banco de registros. Recién cuando ambos operandos se encuentren disponibles, el scoreboard permitirá la lectura de los mismos. De esta manera son evitados los hazard RAW.

Esperar hasta	**Tareas**
R_j y R_k	R_j ← No; R_k ← No; Q_j ← 0; Q_k ← 0

Tabla 2. Algoritmo de Scoreboard - Etapa de Lectura de Operandos

El scoreboard realiza las siguientes actualizaciones en sus estructuras de datos:

- R_j ← No; R_k ← No, indica que los operandos fuentes ya han sido leido;

- $Q_j \leftarrow 0$; $Q_k \leftarrow 0$, indica que ninguna unidad funcional produce resultados para los operandos fuentes.

Cuando estas tareas han sido cumplidas se ha completado la etapa de lectura de operandos.

3. Ejecución (Ver Tabla 3). En esta etapa se realiza la ejecución, el scoreboard no realiza ninguna tarea.

Esperar hasta	**Tareas**
FU finalice	

Tabla 3. Algoritmo de Scoreboard - Etapa de Ejecución

4. Escritura de Resultados (Ver Tabla 4). Luego de que se ha completado la etapa de ejecución, la instrucción avanza a la etapa de escritura de resultados. La instrucción es mantenida en esta etapa hasta que ninguna unidad funcional f tenga como operando al mismo registro en el cual escribirá y aún no haya leido sus operandos ($\forall f(F_j[f] \neq F_i[FU])$ y $\forall f(F_k[f] \neq F_i[FU])$), de esta manera evita que ocurran hazard WAR. Una vez que se cumple esta condición, el scoreboard le permite a la instrucción escribir su resultado en el banco de registros y realiza las siguientes actualizaciones en sus estructuras de datos:

Esperar hasta	**Tareas**
$\forall f((F_j[f] \neq F_i[FU] \vee R_j[f] = No) \wedge$ $(F_k[f] \neq F_i[FU] \vee R_k[f] = No))$	$\forall f$ (if $Q_j[f]$ = FU then $R_j[f]$ = yes); $\forall f$ (if $Q_k[f]$ = FU then $R_k[f]$ = yes); Result[F_i[FU]] $\leftarrow$ 0; Busy[FU] $\leftarrow$ 0;

Tabla 4. Algoritmo de Scoreboard - Etapa de Escritura de Resultados

- $\forall f$ (if $Q_j[f]$ = FU then $R_j[f]$ = yes), toda unidad funcional f que espera el resultado producido por la instrucción que se encuentra en esta etapa coloca el flag de R_j en yes, indicando que el operando ya se encuentra disponible en el banco de registros.
- $\forall f$ (if $Q_k[f]$ = FU then $R_k[f]$ = yes), idem al caso anterior, pero considerando el segundo operando.
- Result[F_i[FU]] $\leftarrow$ 0, borra el campo correspondiente en la estructura de datos que mantiene el estado de registros de resultados para indicar que el registro ya ha sido actualizado.
- Busy[FU] $\leftarrow$ 0, indica que la unidad funcional FU se encuentra ahora libre.

2.1. Ejemplo SocreBoard. Asuma las siguientes latencias para el ciclo EX de las unidades de punto flotante: La suma toma 2 ciclos, la multiplicación toma 10 ciclos y la división 40 ciclos. En las siguientes figuras mostramos el estado de las tablas mantenidas por el scoreboard para algunos ciclos importantes de la ejecución de un fragmento de código.

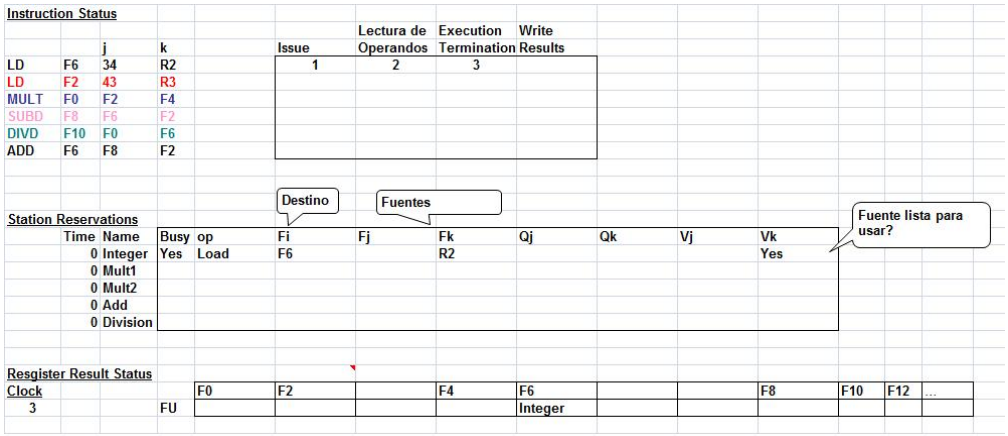


FIGURA 3. Tablas de Scoreboard antes de que LD escriba su resultado

Ciclo3. El primer load ha completado su ejecución pero aún no ha escrito su resultado. El segundo load no es despachado debido a un hazard estructural por la unidad entera, produciendo a su vez que ninguna otra instrucción posterior en el programa se despache hasta que este hazard sea resuelto.

Ciclo9. El primer y segundo load han completado su escritura de resultados. Tanto la multiplicación como la resta proceden a leer sus operandos durante este ciclo. La división aún no ha leido sus operandos, ya que existe una dependencia con el resultado de la multiplicación. La suma no es despachada debido a un hazard estructural con la instrucción de resta.

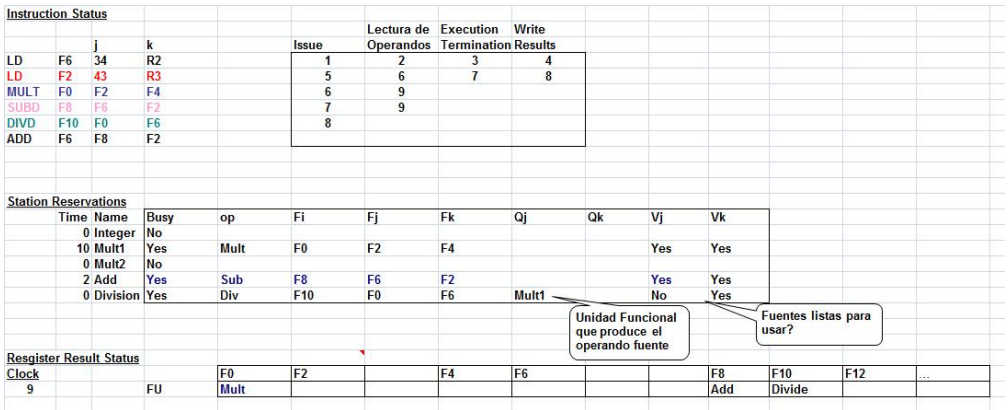


FIGURA 4. Tablas de Scoreboard durante el ciclo 9

Ciclo17. La división aún no ha leido sus operandos debido a la dependencia sobre el resultado de la multplicación. La suma ha leido sus operandos y ejecutado, sin embargo no puede proceder a escribir su resultado debido al hazard WAR sobre F6, el cual es usado por la división.

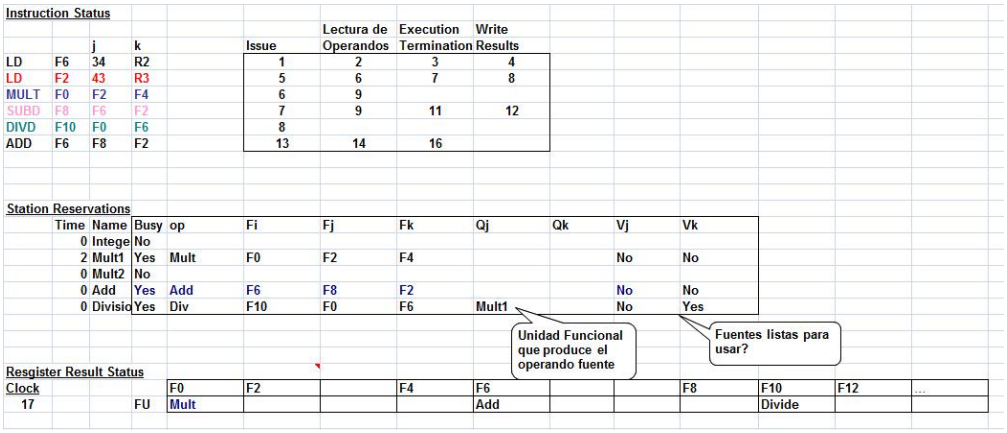


FIGURA 5. Tablas de Scoreboard durante el ciclo 17

Ciclo62. La multplicación finalizó su escritura de resultados durante el ciclo 20, por lo que en el ciclo 21 la división procedió a leer sus operandos y en el ciclo 22 se le permitió a la suma escribir su resultado, la cual se encontraba detenida desde el ciclo 17. Se observa

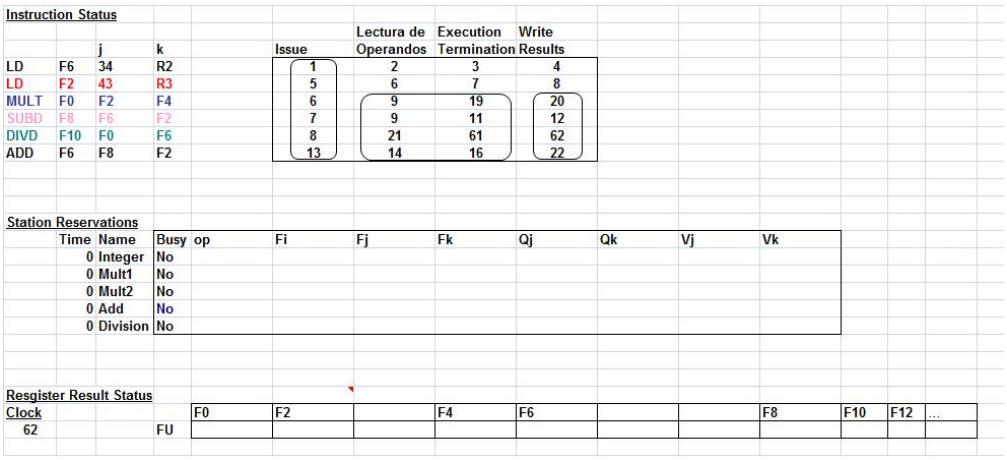


FIGURA 6. Tablas de Scoreboard durante el ciclo 62

que se realizó despacho en orden, ejecución fuera de orden con finalización fuera de orden pero manteniendo la lógica del programa fuente.

3. INTRODUCIÓN AL RENOMBRAMIENTO DE REGISTROS, BUFFER DE REORDENAMIENTO Y ESTACIONES DE RESERVACIÓN

Las etapas de recuperación y decodificación de instrucciones no necesitan ser diferentes en los procesadores que preservan el orden y en aquellos que no lo preservan. Las tareas del *extremo incial* (en inglés front-end) del procesador tales como la recuperación de múltiples instrucciones, la predicción de saltos y la decodificación, son independientes

de como se ejecutará la corriente de instrucciones en el *extremo final* (en inglés back-end). La diferencia entre los dos tipos de procesadores super-escalares comienzan con la decisión de *cuándo* las instrucciones dejan el extremo inicial y de *cómo* ellas son procesadas en el extremo final.
En un procesador que preserva el orden, las instrucciones deben dejar el extremo inicial en el orden del programa. En este caso aquellas instrucciones que tengan riesgos RAW o WAW no podrán ser despachadas. En un procesador fuera de orden, los riesgos RAW y WAW serán evitados mediante *renombramiento de registros*. Como las instrucciones pueden finalizar en un orden diferente al establecido por el programa, se hace necesario volver a imponer el orden del programa, esta tarea corresponde al *buffer de reordenamiento* o ROB por sus siglas en inglés. Además, puede ser que las instrucciones dejen el extremo inicial y aún no estén listas para ejecutar, ya sea por dependencias RAW o riesgos estructurales tales como unidades funcionales ocupadas. Estas instrucciones tendrán que esperar en las *estaciones de reservación*, o lo que es lo mismo, permanecer en la *ventana de instrucciones*.

3.1. Renombramiento de registros. Mientras que los riesgos RAW tiene su origen en el algoritmo codificado, los riesgos WAW y WAR se producen por una limitación de recursos; como ser cuando hay pocos registros disponibles. En lugar de ser dependencias verdaderas, las dos últimas son llamadas dependencias de *nombre*. Los riesgos WAW y WAR pueden ser evitados con renombramiento de registros

Ejemplo 1. *Considere que todos los registros están disponibles al comienzo de la siguiente secuencia de instrucciones y que tres instrucciones pueden ser despachadas por ciclo:*

i1: R1 ← R2/R3 # la división toma varios ciclos en finalizar
i2: R4 ← R1+R5 # riesgo RAW con *i1*
i3: R5 ← R6+R7 # riesgo WAR con *i2*
i4: R1 ← R8+R9 # riesgo WAW con *i1*

Debido al riesgo RAW entre *i1* e *i2*, en un procesador que preserva el orden solamente la primer instrucción podría ser despachada. La instrucción *i2* podría ser despachada cuando la división estuviese en su último ciclo de ejecución y las instrucciones *i3* e *i4* podrían ser despachadas al mismo tiempo que *i2*, o podría ser en el siguiente ciclo dependiendo de las reglas de despacho del procesador particular. Los riesgos WAW y WAR podrían desaparecer tan pronto como el riesgo RAW sea superado. En un procesador que no preserva el orden, el riesgo RAW también será respetado. Pero su presencia no debería evitar que las instrucciones que le siguen a la que es dependiente comiencen su ejecución si no necesitan los resultados que aún no han sido generados. Los riesgos WAW y WAR se evitan recurriendo al uso de registros adicionales u otro tipo de almacenamiento.

En esta primera explicación del renombramiento de registros, no se tendrá en cuenta el número de registros físicos, como se consiguen, ni como se liberan. Llamaremos registros *lógicos* a los registros que aparecen en la especificación del conjunto de instrucciones y registro *físicos* a todo otro almacenamiento que pueda ser usado como registro. Los registros lógicos de una instrucción serán renombrados con los registros físicos cuando

la instrucción alcance la última etapa del extremo inicial. Los nombres de los operandos fuente serán reemplazados por los registros físicos correspondientes y el registro destino se cambiará por el nombre de un registro físico disponible. En otras palabras, si $R_i \leftarrow R_j Op R_k$ es la siguiente instrucción cuyos registros deben ser renombrados, donde $Renombrar(R_a)$ es la tabla actual de correspondencia y *ListaLibre* es una lista de los registros físicos que no han sido asignados, con *primero* apuntando al próximo disponible, entonces el renombramiento de registros procede como sigue:

Etapa de renombramiento.

$Inst_m : R_i \leftarrow R_j Op R_k$ se convierte en $R_a \leftarrow R_b Op R_c$ donde
R_b = Renombrar(R_j);
R_c = Renombrar(R_k);
R_a = listalibre(primero);
Renombrar(R_i) = listalibre(primero);
primero ← próximo(primero).

Ejemplo 2. *Continuación del Ejemplo 1. Se asumió que al comienzo de la secuencia de instrucciones todos los registros lógicos son renombrados con su propio nombre. listalibre contiene los registros físicos en orden numérico: R32, R33 y así siguiendo. Cuando las instrucciones i*1*, i*2*, i*3 *alcanzan juntas la etapa de renombramiento, asumiendo triple despacho, entonces:*

- *Para la instrucción i1, los registros fuente R2 y R3 conservan sus nombres. El registro destino R1 es renombrado como R32.*
- *Para la instrucción i2, el registro fuente R1 se ha convertido en R32 y el registro fuente R5 mantiene su nombre. El registro destino R4 se renombra como R33.*
- *Para la instrucción i3, los registros fuente R6 y R7 conservan sus nombres. El registro destino R5 es renombrado como R34.*

Después del primer ciclo, la secuencia se convirtió en:

```
i1:  R32 ← R2/R3     # la división toma varios ciclos en finalizar
i2:  R33 ← R32+R5    # Se mantiene aún el riesgo RAW con i1
i3:  R34 ← R6+R7     # ya no hay riesgo WAR con i2
i4:  R1 ← R8+R9      # Aun sin cambiar
```

En el siguiente ciclo las dos instrucciones i1 e i3 pueden comenzar la ejecución (la instrucción i2 no puede comenzar debido al riesgo RAW), y el renombramiento se puede hacer para i4. R1 será renombrado como R35, con lo que la instrucción i4 quedará como:

```
i4:  R35 ← R8+R9   # Ya no hay riesgo con i1
```

Notar que R1 ha sido renombrada dos veces. Es claro que las instrucciones que siguen a i4 y usen R1 como un operando fuente esperan el resultado correcto de i4 y el renombramiento de registros debe asegurar esta última correspondencia.

3.2. Buffer de reordenamiento. Continuando con el Ejemplo 2, es muy probable que las instrucciones i3 e i4 calculen sus resultados antes que i1 e i2. Se debe prevenir que los resultados almacenados en R34 y R35 sean puestos en sus correspondientes registros lógicos (R1 y R5 respectivamente) antes que los contenidos de R32 y R33 sean almacenados en R1 y R2 respectivamente. En otro caso cuando $i1$ almacene su resultado en R1, podría sobreescribir el valor ya almacenado en R1 como resultado de la instrucción $i4$. Como otra posibilidad, si $i1$ generó una excepción (por ejemplo si ocurre una división por cero) podría no ser aceptable que el resultado de $i3$ sea parte del estado del procesador. Almacenar los resultados en el orden del programa se consigue con la ayuda de un bufer de reordenamiento (en inglés reorder buffer o ROB). Conceptualmente ROB es una cola FIFO donde cada entrada es una cuádrupla (bandera, valor, nombre del registro resultado, tipo de instrucción) con los siguientes significados para los componentes de la tupla (por el momento restringido solo a operaciones aritméticas)

- La bandera indica si la instrucción ha completado su ejecución.
- El valor es el valor calculado por la instrucción.
- El nombre del registro resultado es el nombre del registro lógico donde el resultado debe ser almacenado.
- El tipo de instrucción muestra el tipo de instrucción como aritmética, load, store, salto, etc.

Cuando una instrucción está en la etapa de renombramiento, una entrada para ella es insertada al final del ROB con la bandera *falso* indicando que el resultado aún no ha sido calculado. Es decir que en el paso de renombramiento se deben hacer los siguientes pasos adicionales:

Etapa de renombramiento (pasos adicionales)

ROB(cola) = (falso, NA, R_i, op);

cola ← próximo(cola)

Cuando la ejecución de la instrucción es completada, el valor es colocado en la entrada correspondiente del ROB y la bandera indica que el resultado puede ser colocado en el registro correspondiente. Sin embargo, esta última acción llamada *consolidación* (en inglés commit) será realizada solamente cuando la instrucción llegue a la cabeza del ROB. Esto es:

Etapa de consolidación

si ((ROB(cabeza) = $Inst_m$ ∧ bandera(ROB(cabeza)))
then begin R_i = valor; cabeza ← próximo(cabeza) end
else repetir el mismo control el próximo ciclo

Notar que o bien una instrucción debe llevar el índice de la entrada en ROB o algún esquema similar debe ser provisto para que las instrucciones y la correspondiente entrada en ROB sean identificadas cuando la instrucción completa la etapa de ejecución.

Después del renombramiento de las primeras tres instrucciones, el ROB se ve como se observa en la Tabla 5(a). Las tres instrucciones que tienen sus registros destinos renombrados se ingresan en el ROB en el orden del programa. En el siguiente ciclo la instrucción $i4$ y las dos siguientes (que no son mostradas en el ejemplo), serán ingresadas en la cola del ROB.

Algunos ciclos después, las instrucciones $i3$ e $i4$ habrán calculado sus resultados antes que $i1$ haya calculado los suyos. Estos resultados serán ingresados en el ROB, como se

observa en la Tabla 5(b), con las banderas de las respectivas entradas indicando que los resultados están listos para ser almacenados. Sin embargo, los resultados no serán almacenados en los registros lógicos de resultados debido a que estas instrucciones no están en la cabeza del ROB.

	Bandera	Valor	Nom.Reg	Tipo	
(a) Después de renombrar las primeras 3 instrucciones					
$i1$	no_lista	ninguno	R1	Aritm	←Cabeza
$i2$	no_lista	ninguno	R4	Aritm	
$i3$	no_lista	ninguno	R5	Aritm	
					←Cola
(b) Después de la ejecución de $i3$ e $i4$					
$i1$	no_lista	ninguno	R1	Aritm	←Cabeza
$i2$	no_lista	ninguno	R4	Aritm	
$i3$	lista	alguno	R5	Aritm	
$i4$	lista	alguno	R1	Aritm	
					←Cola
(c) Después de finalizar $i1$					
$i1$	lista	alguno	R1	Aritm	
$i2$	no_lista	ninguno	R4	Aritm	←Cabeza
$i3$	lista	alguno	R5	Aritm	
$i4$	lista	alguno	R1	Aritm	
					←Cola

Tabla 5. Instantáneas del buffer de reordenamiento (ROB)

Cuando la instrucción $i1$ termina, se realizan acciones idénticas. Ahora, en el siguiente ciclo, el resultado de $i1$ puede ser escrito en el registro lógico R1, porque $i1$ está a la cabeza del ROB. Esta operación de consolidación, realizada por la *unidad de retiro*, puede ser considerada la última etapa del pipeline de instrucción. El contenido del ROB es el que corresponde a la Tabla 5(c).
Después del ciclo en el cual la instrucción $i2$ termina, las tres instrucciones restantes están listas para terminar. En general es buena idea balancear el número de instrucciones capaces de finalizar en el mismo ciclo con el número de instrucciones que pueden ser ingresadas en el ROB en la etapa de renombramiento, así en un procesador de tres vías, las tres instrucciones $i2$, $i3$, $i4$ podrían finalizar en el mismo ciclo. De la misma forma que el renombramiento de registros debe identificar la última instancia de renombramiento de un registro, las instrucciones que están finalizando se deben asegurar que la entrada más joven en el ROB puedan sobrescribir las entradas más antiguas. Si, por ejemplo $i2$ e $i4$ trataran de escribir en el mismo ciclo sobre el mismo registro lógico, solamente $i4$ estaría habilitada para hacerlo.

3.3. Estaciones de reservación y/o ventana de instrucciones. En las secciones anteriores hemos identificado acciones que deben ser realizadas en el extremo inicial, como ser renombramiento de registros e insertar instrucciones en el orden del programa en el ROB. También hemos visto como los resultados de los cálculos son almacenados en registros lógicos en el orden del programa por medio de la operación de consolidación. Sin embargo, aparte de algunas consideraciones de implementación, hemos saltado algunas partes importante del proceso, tales como dónde esperan las instrucciones antes de ser ejecutadas y cómo saben las instrucciones que están listas para ejecutar.
Después que en el extremo-inicial se hayan renombrado los registros de una instrucción y se haya ingresado su correspondiente tupla en el ROB, se *despachará* [3] la instrucción a una *estación de reservación*. Una estación de reservación es un lugar que mantiene una instrucción y debe contener de una forma u otra lo siguiente:

- La operación a ser realizada.
- Los valores de los operandos fuente o sus nombres físicos (una bandera lo indicará)
- El nombre físico del registro resultado
- La entrada en el ROB donde el resultado debe ser almacenado

Las estaciones de reservación pueden ser agrupadas en una cola centralizada, a veces llamada *ventana de instrucciones* o pueden ser asociados con (conjuntos de) unidades funcionales de acuerdo a su código de operación. Independientemente si la cola es unificada o no, cuando en un ciclo determinado, hay más de una instrucción lista para ir a una unidad funcional hay contención para esa unidad funcional y la decisión se toma mediante *planificación*. Con frecuencia la instrucción más antigua, es decir la primera en el orden del programa, será la primera emitida (issue). Otras posibilidades, como emitir primero aquellas instrucciones identificadas como *críticas*, también son posibles.

La última tarea que tenemos que describir es la detección de disponibilidad de una instrucción para su emisión. Esto ocurrirá cuando ambos operandos fuente sean valores. Los operandos serán las entradas a la unidad funcional que ejecutará la operación. Para saber cuando hacer la emisión, se puede asociar un bit ready con cada registro físico para indicar si el resultado para ese registro ha sido calculado o no. Cuando un registro es el destino de una instrucción, la correspondencia entre ambos se hace durante la etapa de renombramiento, y el bit ready del registro se coloca en 0. Cuando una instrucción es despachada, si el bit ready de un operando fuente está en 1, se pasa a la estación de reservación el contenido del registro y se coloca *verdadero* en la bandera correspondiente. Si el bit ready de un operando está en 0, se pasa el nombre del registro y se coloca la bandera correspondiente a *falso*. La instrucción será despachada cuando ambos operandos estén en verdadero. Así:

[3] Los autores de libros académicos y los manuales de la industria no son consistentes en la distinción entre *despacho* (*despach* en inglés) y *emisión* (*issue* en inglés). Nosotros diremos que el *despacho* es una tarea del extremo-inicial (llenar una estación de reservación) y la *emisión* es una tarea del extremo-final (envío de una instrucción completa a la unidad funcional). Notar que somos culpables de alguna inconsistencia ya que, por razones históricas, hemos mostrado la etapa de *emisión* antes que la etapa de *despacho* para el scoreboard del CDC6600

Etapa de Despacho

Para la $Inst_m$: $R_a \leftarrow R_b \, op \, R_c$ *(los registro son físicos)*
Si todas las estaciones de reservación para op están ocupadas
then generar atasco y repetir en el próximo ciclo
Llenar (próxima) estación de reservación para op con la tupla
{ *operador = op,*
Si Listo(R_b) *then (valor[*R_b*],verdadero) else (*R_b*, falso),*
Si Listo(R_c) *then (valor[*R_c*],verdadero) else (*R_c*, falso),*
R_a*,*
puntero a una entrada en ROB
}

Etapa de Emisión

Si (bandera(R_b) $\wedge$ *bandera*(R_c)*)*
then enviar (valor[R_b*],valor[*R_c*]) a la unidad funcional*
else repetir el siguiente ciclo

Cuando una instrucción se completa, se difunde el nombre del registro físico y su contenido a todas las estaciones de reservación. Cada estación de reservación que tenga el nombre de este registro físico como uno de sus operandos fuente, copiará su contenido y pondrá la bandera correspondiente en 1. Al mismo tiempo, los contenidos son copiados en los registros físicos destino y su bit ready puesto en 1. Los resultados también son almacenados en el ROB, si éste es el soporte para los registros físicos.

Ejemplo 3. *Hemos asumido que todos los registros están disponibles. Más precisamente, se asumió que los contenidos de los registros lógicos están consolidados y que todos los otros registros físicos están disponibles. Cuando* $R1$ *fue renombrado a* $R32$ *(instrucción* $i1$*), el bit ready de* $R32$ *es apagado. Cuando* $i2$ *es despachada el nombre "*$R32$*" es enviado a la estación de reservación que aloja a* $i2$ *y la bandera correspondiente es desactivada, puesta en 0. Por lo tanto,* $i2$ *no está lista para ser emitida. Cuando* $i1$ *completa su ejecución, difunde el nombre "*$R32$*" conjuntamente con el resultado de la operación que acaba de realizar. La estación de reservación que aloja a* $i2$ *reconoce el nombre, graba el resultado y cambia la bandera correspondiente al valor activa, permitiendo que* $i2$ *sea emitida en el siguiente ciclo.*

La sección sobre la IBM 360/91 y el algoritmo de Tomasulo da una implementación específica de renombramiento, ROB, y emisión de instrucciones. Una implementación moderna y bastante directa del algoritmo de Tomasulo está presente en el procesador Pentium P6.

3.4. Resumen de la sección. La Tabla 6 resume como fluye una instrucción en un procesador fuera-de-orden. Se ha elegido dividir los pasos de decodificación-renombrado-despacho en dos pasos (decodificación-renombrado [4] y despacho) porque en muchas microarquitecturas hay una cola de instrucciones listas para ser despachadas que están en una representación interna diferente de aquella con la cual fueron almacenadas en el buffer de instrucciones. De la misma manera se ha dividido el paso emisión-ejecución en dos etapas: emisión y ejecución, a pesar que ambas requieren su ejecución en las unidades

[4] El Mapa de Registros usado en esta etapa es una tabla que a cada registro lógico le asocia su nombre cuando éste ya tiene un valor o la entrada en ROB donde se está calculando el nuevo valor.

	Etapa	Recurso leído	Recursos escritos o utilizados
Extremo inicial	Fetch	PC	PC
		Predictor de salto	Buffer de instrucciones
		Cache de Instrucciones	
	Decodific-	buffer de instrucciones	Buffer de decodificación
	Renombrado	Mapa de registros	Mapa de registros
			ROB
	Despacho	Buffer de decodificación	Estación de reservación
		Mapa de registros	ROB
		Banco registros (lóg y fís)	
Extremo Final	Emisión	Estaciones de reservación	Unidades Funcionales
			Cache de datos
	Ejecución	Unidades funcionales	Estaciones de reservación
			ROB
			Registros Físicos
			Predictor de saltos
			Buffer de almacenamiento
	Consolidación	ROB	ROB
		Registros físicos	Registros lógicos
		Buffer de almacenamiento	Mapa de registros
			Cache de datos

Tabla 6. Flujo de instrucciones y recursos de un procesador fuera de orden

funcionales o en la cache D (de datos). El motivo para esta división es que la emisión remueve las instrucciones de la estación de reservación mientras que la ejecución modifica las estaciones de reservación, el ROB, y el banco de registros físicos como así también otras estructuras de datos, dependiendo del tipo de instrucción.

4. Caso de estudio: Unidad de punto flotante de IBM 360/91 y el algoritmo de Tomasulo

El algoritmo de Tomasulo fue propuesto por Robert Tomasulo [5] , en éste los hazards RAW se resuelven ejecutando una instrucción sólo cuando sus operandos están disponibles. Los hazards WAR y WAW, los cuales surgen por dependencias de nombre, son eliminados mediante el renombramiento de registros. El *Renombramiento de registros* elimina estos hazards cambiando la denominación de los registros destinos y usando esa denominación de manera consistente, logrando que la escritura fuera de orden no afecte a ninguna instrucción.

Renombramiento de Registros. Para entender mejor como el renombramiento de registros elimina los riesgos WAR y WAW considere el siguiente fragmento de código:

[5]Este caso de estudio fue extraído de la sección 2.4 del libro de Hennessy & Patterson [2]

```
div.d   f0 , f2 , f4
add.d   f6 , f0 , f8
s.d     f6 , 0(r1)
sub.d   f8 , f10, f14
mul.d   f6 , f10, f8
```

Existe una antidependencia entre la instrucción de suma y la de resta y una dependencia de salida entre la intrucción de suma y la multiplicación, indicando que es posibles que ocurran dos riesgos: un hazard WAR por el uso de F8 por la suma y un hazard WAW ya que la suma podría finalizar después que la multiplicación. Existen también algunas dependencias de datos reales. Una de ellas se dá entre la división y la suma, otra entre la resta y la multiplicación y finalmente una entre la suma y el store.

Los conflictos por dependencias de nombre pueden ser eliminados mediante renombramiento de registros. Asumimos la existencia de dos registros temporarios, S y T. Usando S y T, el fragmento de código anterior puede ser reescrito de la siguiente manera:

```
div.d   f0 , f2 , f4
add.d   S  , f0 , f8
s.d     S  , 0(r1)
sub.d   T  , f10, f14
mul.d   f6 , f10, T
```

Cualquier uso del registro F8 posterior a la resta debe ser reemplazado por el registro T lo mismo ocurre con F6, que después de la suma, es reemplazado por S. En el caso de este ejemplo el proceso de renombramiento es realizado estáticamente por un compilador. Para conseguir esto, se requiere de un compilador sofisticado. El algoritmo de Tomasulo permite manejar el renombramiento en ejecución mediante hardware especialmente diseñado para ello.

Arquitectura del MIPS con Tomasulo. La Figura 7 muestra la estructura básica de un procesador MIPS basado en Tomasulo, incluyendo la unidad de punto flotante y la unidad load-store. Las instrucciones son enviadas desde la unidad de instrucción hacia la cola de instrucción desde la cual son despachadas en orden FIFO. Cada estación de reservación mantiene:

- una instrucción que ha sido despachada y está esperando por una unidad funcional;
- los valores de los operandos, si se encuentran disponibles;
- los nombres de las estaciones de reservación que producen los operandos;
- otra información necesaria para la detección y resolución de hazards.

Los buffers de load poseen tres funciones: **mantener** los componentes para el cálculo de la dirección efectiva, **actuar** como referencia de los requerimientos a memoria que aún no han sido completados, y **mantener** los resultados de los loads que ya han finalizado. Similarmente, los buffers de store también poseen tres funciones: **mantener** los componentes para el cálculo de la dirección efectiva, **mantener** las direcciones destino de stores que se encuentran esperando por los datos a almacenar, y **mantener** las direcciones y valores a almacenar hasta que la memoria complete la escritura.

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 store. Todos los resultados de las unidades funcionales y de la memoria son eviados a través del *Common Data Bus* (CDB por sus

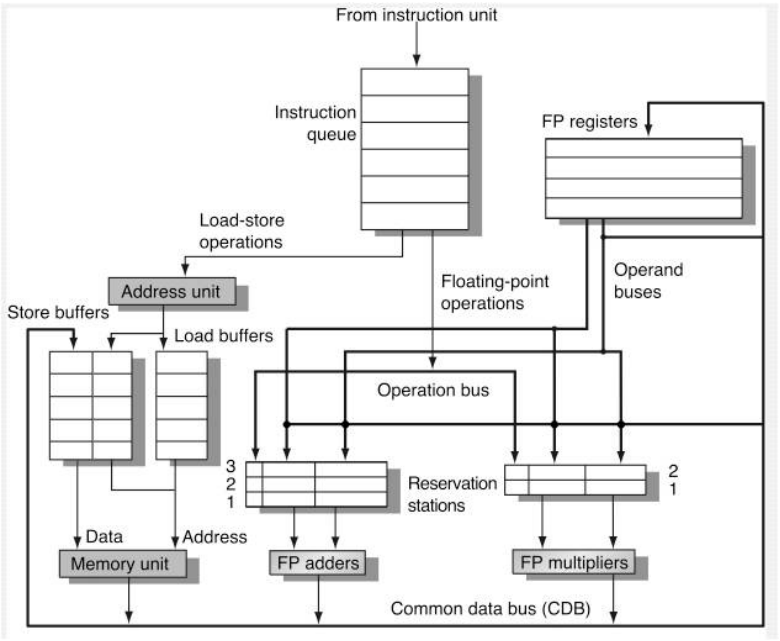


FIGURA 7. Estructura básica de una unidad de punto flotante de MIPS usando el algoritmo de Tomasulo

siglas en inglés), el cual llega a todas partes excepto al buffer de load. Todas las estaciones de reservación poseen campos *tag*, empleados para el control de pipeline.

Algoritmo de Tomasulo. En este algoritmo el renombramiento de registros es realizado por la lógica de emisión mediante las *Estaciones de Reservación*, en las cuales se almacenan los operandos de las instrucciones. La idea básica es que una estación de reservación busca y almacena un operando tan pronto como esté disponible, eliminando la necesidad de obtener los operandos desde un registro. Además, las instrucciones pendientes emplean la denominación (tag) de la/s estación/es de reservación que proveerá/n el/los operando/s. Finalmente, si ocurren escrituras sucesivas a un mismo registro sólo la última es considerada para actualizar el registro.

El uso de estaciones de reservación, en lugar de un banco de registros centralizado, permite otras dos importantes propiedades: Primero, tanto la detección de hazards como el control de la ejecución están distribuidos entre los distintos recursos. La información mantenida en las estaciones de reservación de cada unidad funcional determina cuando una instrucción puede comenzar su ejecución en esa unidad. Segundo, los resultados son pasados directamente a las unidades funcionales desde las estaciones de reservación donde ellos estaban almacenados, en lugar de ir a través de los registros. Esto es realizado mediante un bus común de resultado, denominado *Common Data Bus* (CDB) en la IBM 360/91 el que permite que un operando sea cargado simultáneamente en todas las unidades que lo requieran mientras se escribe en el banco de registros.

A continuación se detallan las etapas por las que pasa una instrucción.

Etapas. Cada instrucción pasa sólo por tres etapas, cada una de las cuales puede insumir un número arbitrario de ciclos de reloj. Estas etapas son:

1. **Despacho**: Obtiene la próxima instrucción desde la cola de instrucciones, que se mentiene en un orden FIFO. Si una estación de reservación adecuada para la siguiente instrucción está disponible, la instrucción se despacha a esa estación de reservación ya con los valores de sus operandos, si ellos se encuentran en los registros. Si no hay una estación de reservación libre, entonces hay un hazard estructural y la instrucción es detenida hasta que una estación sea liberada. Si los operandos no se encuentran en los registros, mantiene información de que unidad funcional producirá los operandos deseados. Este paso renombra los registros, eliminando los hazards WAR y WAW.
2. **Ejecución**: Si uno o más de los operandos no se encuentran aún disponibles, la estación de reservación monitorea el CDB mientras espera a que sea calculado. Cuando un operando está disponble, es ubicado en las estaciones de reservación que lo aguardaban. Cuando todos los operandos están disponibles, la operación puede ser ejecutada. Esta demora de la ejecución hasta que los operandos estén disponibles resuelve los hazards RAW.
3. **Escritura de Resultados**: Cuando un resultado está disponible, se escribe sobre el CDB y desde allí a los registros y en cualquier estación de reservación que estuviese esperando por ese resultado. Los stores también escriben los datos en la memoria durante este paso: una vez que la dirección y el dato estén disponibles, se los envía a la unidad de memoria.

Las etiquetas son esencialmente nombres para un conjunto extendido de registros virtuales usados durante el renombramiento. En nuestro caso, la etiqueta es de 4-bits lo que es suficiente para denotar una de las cinco estaciones de reservación o uno de los seis buffers de carga. Esto es equivalente a once registros que se pueden usar como resultados intermedios. La etiqueta identifica aquella estación de reservación con la instrucción que produce un resultado que será empleado como operando. Es decir, las instrucciones en las estaciones de reservación se refieren a los operandos a través de la etiqueta correspondiente. Los valores de etiqueta que nunca podrían ser usados, como cero, indican que el operando se encuentra disponible en los registros.

Cada estación de reservación posee siete campos:

- Op: La operación que se ejecutará.
- Q_j, Q_k: Las estaciones de reservación que producirán los correspondientes operandos fuentes; un valor de cero indica que el operando fuente se encuentra disponible en V_j o V_k o no es necesario.
- V_j, V_k: Los valores de los operandos fuentes. Note que para cada uno de los operandos, o bien el valor está en el campo V correspondiente, o bien el campo Q contiene la etiqueta de la unidad que producirá el valor. Para los loads, el campo V_k es usado para mantener el desplazamiento.
- A: Usado para mantener información para el cálculo de la dirección de memoria para un load o store. Inicialmente el campo inmediato es almacenado aquí y después, la dirección efectiva una vez que ha sido calculada.
- $Busy$: Indica que la estación de reservación y su correspondiente unidad funcional están ocupadas.

Los registros poseen un campo, Q_i, el cual es la etiqueta de la estación de reservación que contiene la operación cuyo resultado debe ser almacenado en este registro. Si Q_i está en blanco (o cero), no hay instrucción activa que esté computando un resultado para ese registro, esto significa que el valor es simplemente el contenido del registro. Los buffers de load y store tienen cada uno un campo, A, el cual mantiene el resultado de la dirección efectiva una vez que el primer paso de la ejecución ha sido completado (los loads y store toman dos ciclos en ejecutar, en el primero calculan la dirección y en el segundo acceden a memoria).

4.1. Ejemplo Tomasulo. A continuación se considera la ejecución de una secuencia de instrucciones en una arquitectura con planificación dinámica controlada por medio del algoritmo de Tomasulo. Se requieren tres tablas principales: Estado de las Instrucciones (EI), Estaciones de Reservación (ER) y Estado de los Registros donde quedan los Resultados (RR). A medida que progresa la ejecución del programa las tablas llevan el control de uso de los diferentes recursos, siempre procurando que las instrucciones finalicen en el menor número de ciclos posible. En cada una de las siguientes figuras se consideran situaciones particulares de la ejecución en un determinado ciclo de reloj.

Antes de analizar el estado de las tablas, es necesario considerar las dependencias que hay en el código que se ejecutará. Por ejemplo, las instrucciones 3, 4 y 6 mantienen dependencias reales con la instrucción 2. Por su parte las instrucciones 4 y 5 dependen de la instrucción 1. También existen otras dependencias de nombres. Es muy importante que antes de continuar con el ejemplo el lector clasifique los distintos tipos de riesgos que surgen durante la ejecución del código.

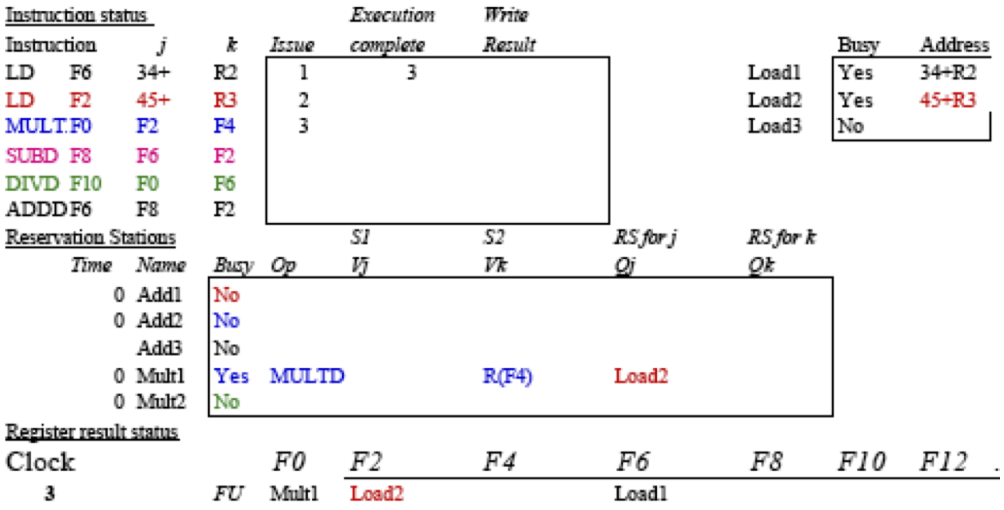


FIGURA 8. Rápido despacho de las instrucciones

Ciclo 3. Como se observa en la Figura 8 en este ciclo ya hay 3 instrucciones despachadas a las estaciones de reservación correspondientes. Se considera que los loads toman 2 ciclos para terminar la etapa de ejecución. La suma requiere 2 ciclos en la etapa de ejecución,

mientras que a la multiplicación le insume 10 ciclos en dicha etapa. Por su parte la división, después de tener sus operandos, tarda 40 ciclos en calcular el resultado.

Al finalizar este tercer ciclo, la primer instrucción de load ha recuperado el contenido de la celda de memoria pero aun no ha difundido dicho valor. Se observa en la tabla ER que hay 3 estaciones de reservación ocupadas (2 de loads y 1 de multiplicación) y que en la tabla RR los tres registros donde quedarán almacenados los valores de las instrucciones indican cual es la estación de reservación que genera el correspondiente valor.

Ciclo 9. La instrucción de resta fue emitida a la unidad de suma al finalizar el ciclo 5 y se mantuvo allí durante los dos ciclos de la etapa ejecución, por eso la tabla de EI indica que SUBD ha completado la ejecución en el ciclo 7. Durante el ciclo 8 se completa la difusión del resultado de la resta hacia el registro F8 y a la segunda estación de reservación de la unidad de suma. La tabla ER de la Figura 9 indica que la instrucción ADDD debe permanecer 1 ciclo más en la etapa de ejecución. Hay que destacar que en cada uno de los 6 primeros ciclos se ha despachado una instrucción.

Instruction status					*Execution*	*Write*				
Instruction		*j*	*k*	*Issue*	*complete*	*Result*			Busy	Address
LD	F6	34+	R2	1	3	4		Load1	No	
LD	F2	45+	R3	2	4	5		Load2	No	
MULT	F0	F2	F4	3				Load3	No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6						
Reservation Stations					*S1*	*S2*	*RS for j*	*RS for k*		
	Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*		
	0	Add1	No							
	1	Add2	Yes	ADDD	M()–M()	M(45+R3)				
	0	Add3	No							
	6	Mult1	Yes	MULTD	M(45+R3)	R(F4)				
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1			
Register result status										
Clock				*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*
9			*FU*	Mult1	M(45+R3)		Add2	M()–M()	Mult2	

FIGURA 9. Finalización de la primer instrucción aritmética corta

Ciclo 12. Ya todas las instrucciones cortas han finalizado su ejecución, la multiplicación tiene 3 ciclos más para finalizar la etapa de ejecución y la división ha sido despachada pero aun no ha sido emitida a la unidad de multiplicación porque se debe esperar a que finalice la multiplicación para conocer el valor del primer operando de la división. En la Figura 10 la tabla ER muestra que las dos estaciones de reservación de la multiplicación están ocupadas: la primera con la instrucción MULT y la segunda con la DIVD. La tabla RR muestra que el registro F0 está esperando que la Unidad de Multiplicación termine de calcular el resultado y difunda el valor, mientras tanto mantiene Mult1 que es donde está MULT. Por su parte el registro F10 indica que la instrucción de la segunda estación de reservación de la multiplicación (Mult2) le proveerá el valor que debe ser almacenado allí. Los otros tres registros involucrados en el código, F8, F6 y F2 ya tienen sus valores

Instruction status					Execution	Write				
Instruction		j	k	Issue	complete	Result			Busy	Address
LD	F6	34+	R2	1	3	4		Load1	No	
LD	F2	45+	R3	2	4	5		Load2	No	
MULT	F0	F2	F4	3				Load3	No	
SUBD	F8	F6	F2	4	6	7				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6	10	11				
Reservation Stations					S1	S2	RS for j	RS for k		
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk		
	0	Add1	No							
	0	Add2	No							
	0	Add3	No							
	3	Mult1	Yes	MULTD	M(45+R3)	R(F4)				
	0	Mult2	Yes	DIVD		M(34+R2)	Mult1			
Register result status										
Clock				F0	F2	F4	F6	F8	F10	F12
12			FU	Mult1	M(45+R3)		(M-M)+M()	M()–M()	Mult2	

FIGURA 10. Finalización de todas las instrucciones aritméticas cortas

calculados:
–F2 contiene el valor recuperado desde la celda de memoria de dirección 45 más el contenido del registro entero R3, es decir Memoria(45+R3).
–F8 tendrá el resultado de restar F6 y F2, es decir (Memoria(34+R2) - Memoria(45+R3)).
–F6 mantendrá el resultado de haber sumado F8 y F2, es decir
(Memoria(34+R2) - (Memoria(45+R3)) + Memoria(45+R3).

Se debe tener presente que los registros y la memoria sólo serán modificados cuando las correspondientes instrucciones sean consolidadas (commit). Mientras tanto sus valores serán recuperados desde el buffer de reordenamiento.

Ciclo 16. La instrucción de multiplicación ha finalizado la etapa de ejecución en el ciclo 15 y en el ciclo 16 se ha difundido el resultado por lo que la división se puede emitir inmediatamente. De acuerdo a eso, se puede afirmar que la instrucción MULT fue emitida al finalizar el ciclo 5 y se mantuvo en la etapa de ejecución entre los ciclos 6 y 15 haciendo uso de la Unidad de Multiplicación. En la Figura 11 se observa que la única instrucción que queda pendiente es la división. En la tabla RR el registro F0 ya contiene el valor calculado por MULT, es decir Memoria(45+R3)*F4. La tabla ER indica en la columna Time, que la instrucción de la estación de reservación 2 de la multiplicación (DIVD) ha sido emitida y estará usando la Unidad de Multiplicación durante 40 ciclos.

Instruction status					Execution	Write				
Instruction		j	k	Issue	complete	Result			Busy	Address
LD	F6	34+	R2	1	3	4		Load1	No	
LD	F2	45+	R3	2	4	5		Load2	No	
MULT	F0	F2	F4	3	15	16		Load3	No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5						
ADDD	F6	F8	F2	6	10	11				
Reservation Stations					S1	S2	RS for j	RS for k		
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk		
	0	Add1	No							
	0	Add2	No							
		Add3	No							
	0	Mult1	No							
	40	Mult2	Yes	DIVD	M*F4	M(34+R2)				
Register result status										
Clock				F0	F2	F4	F6	F8	F10	F12
16			FU	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2	

FIGURA 11. Operandos de la división listos

Ciclo 56. El estado de las tablas en el ciclo 56 es mostrado en la Figura 12. Se puede ver que las instrucciones han sido despachadas en orden pero han finalizado su ejecución fuera de orden. Las instrucciones que requieren una menor cantidad de ciclos en la etapa de ejecución no se han visto perjudicadas por las instrucciones que requieren mayor cantidad de ciclos. Al comparar el desempeño alcanzado usando Scoreboarding, Figura 6, con el desempeño usando Tomasulo, Figura 12, se comprueba la ganancia conseguida en el segundo caso.

Instruction status					Execution	Write				
Instruction		j	k	Issue	complete	Result			Busy	Address
LD	F6	34+	R2	1	3	4		Load1	No	
LD	F2	45+	R3	2	4	5		Load2	No	
MULT	F0	F2	F4	3	15	16		Load3	No	
SUBD	F8	F6	F2	4	7	8				
DIVD	F10	F0	F6	5	56					
ADDD	F6	F8	F2	6	10	11				
Reservation Stations					S1	S2	RS for j	RS for k		
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk		
	0	Add1	No							
	0	Add2	No							
		Add3	No							
	0	Mult1	No							
	0	Mult2	Yes	DIVD	M*F4	M(34+R2)				
Register result status										
Clock				F0	F2	F4	F6	F8	F10	F12
56			FU	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2	

FIGURA 12. Termina la ejecución de la división

Referencias

[1] Baer, J.L., Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors, Cambridge University Press, 2010

[2] Hennessy, John L. and Patterson, David A., Computer Architecture, Fourth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., 2006