

## Capítulo 7

---

# EJECUCIÓN CON CAUCE SEGMENTADO: *PIPELINING*

---

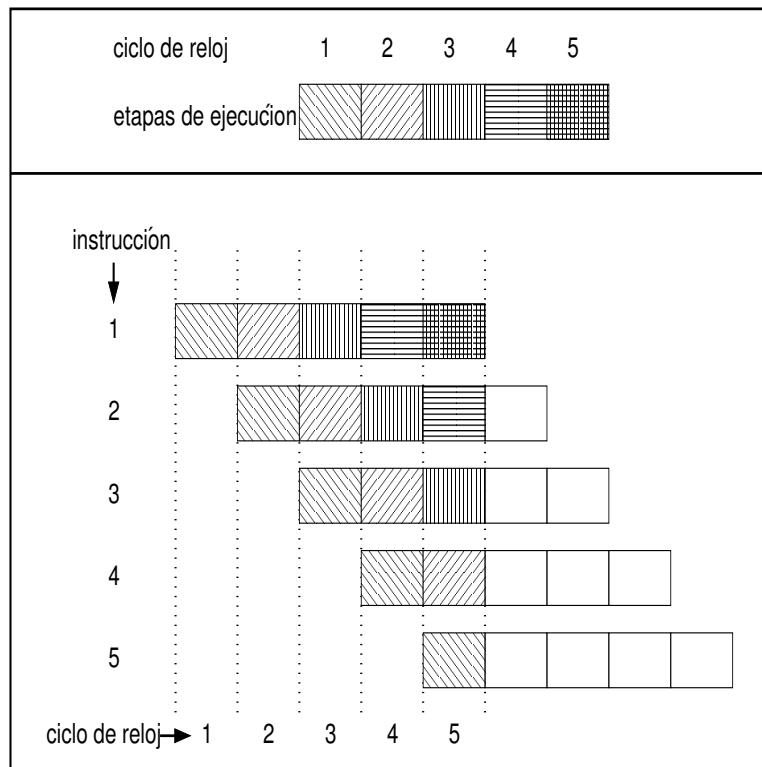
### 7.1 EL CONCEPTO DE CAUCE SEGMENTADO

El concepto de cauce segmentado, al que denominaremos indistintamente como *pipeline*<sup>1</sup> es realmente anterior a las computadoras digitales. En realidad es el mismo concepto que el de línea de producción que Henry Ford puso en práctica durante la fabricación del modelo T a principios del siglo XX.

En esencia pipeline significa que se comienza a ejecutar una instrucción antes de que la inmediata anterior termine de ejecutarse. En el capítulo pasado planteamos la ejecución de una instrucción como el envío de diversas señales de control y datos a través de la ruta de datos de la computadora. No es posible enviar todas las señales de una sola vez sino que hay que hacerlo por etapas. Cada instrucción de lenguaje de máquina es, en realidad, la ejecución de un pequeño “programa”<sup>2</sup>. Como vimos también cada uno de estos pasos puede ser hecho en un solo ciclo de reloj, así que podríamos pensar en ejecutar el paso  $i$  de la instrucción  $j$  al mismo tiempo que el paso  $i - 1$  de la  $j + 1$  y que el  $i - 2$  de la  $j + 2$  y así hasta tener tantas instrucciones ejecutándose como ciclos de reloj son necesarios para ejecutarlas. Esto se muestra esquemáticamente en la figura 7.1.

<sup>1</sup>La traducción de *pipeline* es “tubería” y entonces *pipelining* sería “entubamiento”, nada que ver con el concepto que necesitamos. Así que en general usaremos el término en inglés.

<sup>2</sup>Usamos aquí la palabra programa para hacer claro el proceso secuencial, paso a paso. Como ya sabemos, en realidad este “programa” puede estar alambrado.



**Figura 7.1.** En una máquina X normalmente una instrucción toma 5 ciclos de reloj en su ejecución y hasta que se termina de ejecutar es posible ejecutar la siguiente. Si la máquina en cuestión tuviera pipeline entonces es posible ejecutar el mismo tiempo diferentes trozos de diferentes instrucciones.

Hay que hacer énfasis en que el pipeline no disminuye el tiempo necesario para ejecutar una instrucción dada. En el caso de la figura 7.1, la instrucción 1 sale del procesador luego de cinco ciclos de reloj, haya o no pipeline. La mejora del pipeline consiste en aumentar el número de instrucciones que salen del procesador por unidad de tiempo, el *throughput*. Para ejecutar 5 instrucciones en la máquina de la figura, sin pipeline, se requiere de 25 ciclos de reloj, 5 por cada instrucción. En la misma máquina con pipeline las cinco instrucciones requieren 5 ciclos de reloj para ejecutarse con el pipeline completamente lleno o 9 en el peor de los casos, cuando el pipeline empieza a llenarse, como se muestra en la figura 7.1.

**Pipeline y rendimiento**  
(*throughput*).

Para que el pipeline funcione bien es necesario hacer que ninguna de las etapas de la ejecución de una instrucción sea más tardada que las demás. Como no podemos ir amontonando instrucciones a la entrada de esa etapa tendríamos que retardar todas las etapas anteriores. Este es el reto más importante para diseñar un pipeline, balancear lo mejor posible la duración de las etapas. Si todo está balanceado entonces:

$$T_{ins}^{pipeline} = \frac{T_{ins}}{N} + \epsilon \quad (7.1)$$

donde  $T_{ins}^{pipeline}$  es el tiempo promedio requerido por instrucción en la máquina con pipeline: si me paro con un reloj junto al procesador, cada cuanto tiempo veo que sale una instrucción.  $T_{ins}$  es el tiempo promedio por instrucción sin pipeline y  $N$  es el número de etapas en el pipeline. El pipeline tiene el efecto de reducir *CPI*. El término  $\epsilon$  de la ecuación 7.1 representa una pequeña sobrecarga, en general al implementar el pipeline de una arquitectura se requiere alargar un poco la duración del ciclo de reloj.

## 7.2 REVISIÓN DEL CICLO DE EJECUCIÓN

En DLX podemos dividir naturalmente el ciclo total de ejecución de una instrucción en cinco etapas:

1. Fetch. Se carga la instrucción a ejecutar y se incrementa el PC .
2. Decodificación. Se determina la instrucción a ejecutar, se transfieren los operandos a la ALU.
3. Ejecución. Se efectúa la operación requerida en la ALU.
4. Acceso a memoria. Si se trata de un load o un store se transfiere el resultado de la ALU a la memoria como dirección.

5. Escritura de resultados. El resultado de la ALU o el valor regresado por la memoria se escribe en los registros del procesador.

Para ser más precisos lo que se hace en cada etapa es:

**IF** (*Instruction Fetch*)

$IR = \text{mem}[PC], NPC = PC + 4$

**ID** (*Instruction Decode*)

$A = \text{regs}[IR_{6,\dots,10}], B = \text{regs}[IR_{11,\dots,15}], \text{Imm} = ((IR_{16})^{16} \uplus IR_{16,\dots,31})$ .  
(recuérdese los formatos de la figura 5.8, A es **rs1** y B es **rd**).

**EX** (*Execution*).

- Referencia a memoria.  $\text{ALUOutput} = A + \text{Imm}$ , luego de esto  $\text{ALUOutput}$  tiene la dirección efectiva del load o store.
- Instrucción aritmético-lógica registro-registro.  
 $\text{ALUOutput} = A \text{ func } B$ .
- Operación aritmético-lógica registro-inmediato.  
 $\text{ALUOutput} = A \text{ oper } \text{Imm}$ .
- Salto.  $\text{ALUOutput} = NPC + \text{Imm}, \text{Cond} = (A \text{ comp } 0)$ .

**MEM** (*Memory*) Acceso a memoria o finalización de salto.  $PC = NPC$

- Referencia a memoria.  
 $\text{LMD} = \text{mem}[\text{ALUOutput}]$  o bien  
 $\text{mem}[\text{ALUOutput}] = B$
- Salto. if (cond)  $PC = \text{ALUOutput}$ .

**WB** (*Write Back*).

- instrucción aritmético-lógica, registro-registro.  
 $\text{regs}[IR_{16,\dots,20}] = \text{ALUOutput}$
- Instrucción aritmético-lógica, registro-inmediato.  
 $\text{regs}[IR_{11,\dots,15}] = \text{ALUOutput}$
- Instrucción de carga.  $\text{regs}[IR_{11,\dots,15}] = \text{LMD}$

*Ejemplos.*

- **LW** R1, 30(R2)

**IF**  $IR = \text{mem}[PC], NPC = PC + 4$

**ID**  $A = \text{regs}[IR_{6,\dots,10}] = \text{regs}[2] \text{ (rs1)}, B = \text{regs}[IR_{11,\dots,15}] = \text{regs}[1] \text{ (rd)}, \text{Imm} = 30$

**EX**  $ALUOutput = A + Imm = 30 + regs[2]$

**MEM**  $PC = NPC, LMD = mem[ALUOutput] = mem[30 + regs[2]]$

**WB**  $regs[IR_{11,...,15}] = regs[1] = LMD$

- **SW R1, 30(R2)**

**IF**  $IR = mem[PC], NPC = PC + 4$

**ID**  $A = regs[IR_{6,...,10}] = regs[2] \text{ (rs1)}, B = regs[IR_{11,...,15}] = regs[1] \text{ (rd)}, Imm = 30$

**EX**  $ALUOutput = A + Imm = 30 + regs[2]$

**MEM**  $PC = NPC, mem[ALUOutput] = mem[30 + regs[2]] = B = regs[1]$

**WB** No se usa

- **J etiqueta**

**IF**  $IR = mem[PC], NPC = PC + 4$

**ID**  $Imm = dist(PC + 4, etiqueta)$

**EX**  $ALUOutput = NPC + Imm$

**MEM**  $PC = ALUOutput$

**WB** No se usa

- **BEQZ R4, etiqueta**

**IF**  $IR = mem[PC], NPC = PC + 4$

**ID**  $A = regs[IR_{6,...,10}] = regs[4], B = regs[0], Imm = dist(PC + 4, etiqueta)$

**EX**  $ALUOutput = NPC + Imm$

**MEM**  $PC = NPC, \text{if } (regs[4] == 0) PC = ALUOutput$

**WB** No se usa

- **ADD R1, R2, R3**

**IF**  $IR = mem[PC], NPC = PC + 4$

**ID**  $A = regs[IR_{6,...,10}] = regs[2], B = regs[3], Imm = regs[1] + func,$   
esto último no tiene sentido, pero no importa, no se usará.

**EX**  $ALUOutput = A + B$

**MEM**  $PC = NPC$

**WB**  $regs[IR_{16,...,20}] = regs[1] = ALUOutput$

### 7.3 REGISTROS DE PIPELINING

Con este esquema de división de tareas a realizar para la ejecución de una instrucción, podemos pensar en ponerle un pipeline a DLX. Cada etapa: IF, ID, EX, MEM y WB puede verse como una etapa de pipeline, cada una llevándose a cabo en un ciclo de reloj. Esto significa que cuando el pipeline este completamente lleno, luego de que se estén ejecutando las primeras 5 instrucciones, tendremos, en cada ciclo de reloj, una instrucción en cada etapa. Todas las etapas ocurren en cada ciclo de reloj, cada una sobre diferente instrucción. Esto también trae algunos problemas.

En el ciclo ID se leen dos registros del banco de registros, en WB se escribe uno de ellos. Hay que habilitar entonces el banco de registros para que pueda leer dos registros y escribir uno en un solo ciclo de reloj, no hay problema, esto se puede hacer, pero, ¿qué tal si una escritura y la lectura involucran al mismo registro?. Hablaremos de esto un poco más adelante.

En cada ciclo hay que traer una nueva instrucción, pero el PC se actualiza hasta el ciclo cuatro, MEM, ¿qué tal si hay un salto? ¿como direccionamos la siguiente instrucción? Esto lo vamos a medio solucionar permitiendo que en la etapa IF misma se actualice el valor de PC con  $PC + 4$ . Si la instrucción es un salto ya veremos como hacer mas adelante.

Imaginemos que en un solo ciclo de reloj se ejecuta la etapa ID de un ADD registro-registro y la etapa EX de un load. En el caso del ADD en A y B se guardarán los operandos. En el caso de el load en ese mismo ciclo la ALU calcula la suma de A e Imm . ¿Cual valor de A se toma?.

Para resolver el último problema mencionado en particular y en general, los problemas que involucran el acceso a registros temporales (A , B , Imm , ALUOutput, IR , NPC , Cond y LMD ), se añadirán registros grandes que permitan mantener el estado de los registros temporales entre etapas del pipeline. Estos registros se nombrarán usando el identificador de las etapas entre las que se encuentran: IF/ID, ID/EX, EX/MEM y MEM/WB. Cualquier valor temporal almacenado en uno de estos registros y que sea necesario acceder en etapas posteriores de ejecución de la instrucción actual, deberá ser copiado al registro siguiente acarreandolo hasta que ya no sea necesario.

En el caso mencionado, por ejemplo, suponiendo que el load es la instrucción más avanzada en su ejecución y que el ADD es más reciente: cuando el load pasó por la etapa ID, se cargó al principio de ella, en el campo correspondiente al registro A de IF/ID, el registro que

será usado como parámetro, cuando load pasa a la siguiente etapa, se copia el registro IF/ID completo al registro ID/EX donde se guarda el valor útil de A para el load, mientras que en ese mismo ciclo se escribe también el registro A o mejor dicho el campo A, pero del registro IF/ID, con el valor que le será útil al ADD. Podría objetarse que queremos leer y escribir en un solo ciclo de reloj en el mismo registro, en este caso, en el campo A del registro IF/ID, pero esto se puede, siempre queremos leerlo en ID/EX antes de escribirlo, podemos hacerlo con unos flip-flops de disparo de flanco, de tal forma que cuando suba el reloj se lea de ellos y cuando baje el reloj, en ese mismo ciclo, se escriban. El resultado de las modificaciones hechas se muestra en la figura 7.2.

Haremos referencia a los distintos campos de estos registros usando la notación *Ee/Es.campo* donde *Ee* es la etapa de entrada, *Es* es la de salida y *campo* es el nombre del campo del registro que nos interesa y que generalmente es el nombre de algunos de los registros temporales que teníamos en el flujo de datos sin pipeline.

Con esto en mente podemos describir lo que ocurre en cada etapa de ejecución:

**IF** IF/ID.IR = mem[PC]  
       IF/ID.NPC, PC = ((EX/MEM.opcode == branch) & EX/MEM.cond)?  
       EX/MEM.ALUOutput : PC + 4

**ID** ID/EX.A = regs[IF/ID.IR<sub>6,...,10</sub>]  
       ID/EX.B = regs[IF/ID.IR<sub>11,...,15</sub>]  
       ID/EX.NPC = IF/ID.NPC  
       ID/EX.IR = IF/ID.IR  
       ID/EX.Imm = (IF/ID.IR<sub>16</sub>)<sup>16</sup>  $\uplus$  IF/ID.IR<sub>16,...,31</sub>

**EX**   • Instrucción aritmético-lógica

- registro-registro  
       EX/MEM.IR = ID/EX.IR  
       EX/MEM.ALUOutput = ID/EX.A func ID/EX.B  
       EX/MEM.cond = 0
- registro-inmediato  
       EX/MEM.IR = ID/EX.IR  
       EX/MEM.ALUOutput = ID/EX.A op ID/EX.Imm  
       EX/MEM.cond = 0



**Figura 7.2.** DLX preparada para pipeline.



- load o store  
 $EX/MEM.IR = ID/EX.IR$   
 $EX/MEM.ALUOutput = ID/EX.A + ID/EX.Imm$   
 $EX/MEM.cond = 0$   
 $EX/MEM.B = ID/EX.B$
- salto  
 $EX/MEM.ALUOutput = ID/EX.NPC + ID/EX.Imm$   
 $EX/MEM.cond = (ID/EX.A \text{ comp } 0)$

**MEM**

- Instrucción aritmético-lógica  
 $MEM/WB.IR = EX/MEM.IR$   
 $MEM/WB.ALUOutput = EX/MEM.ALUOutput$
- load  
 $MEM/WB.IR = EX/MEM.IR$   
 $MEM/WB.LMD = mem[EX/MEM.ALUOutput]$
- store  
 $MEM/WB.IR = EX/MEM.IR$   
 $mem[EX/MEM.ALUOutput] = EX/MEM.B$

**WB**

- Instrucción aritmético-lógica
  - registro-registro  
 $regs[MEM/WB.IR_{16,...,20}] = MEM/WB.ALUOutput$
  - registro-inmediato  
 $regs[MEM/WB.IR_{11,...,15}] = MEM/WB.ALUOutput$
- load  
 $regs[MEM/WB.IR_{11,...,15}] = MEM/WB.LMD$

## 7.4 DESEMPEÑO DE PIPELINE

Ya hemos dicho que el pipeline incrementa el rendimiento (*throughput*) del procesador y que no se reduce el tiempo que una instrucción cualquiera tarda en ejecutarse. De hecho el tiempo que toma la ejecución de una instrucción se incrementa por la carga extra que implica el pipeline. Por ejemplo en nuestra arquitectura sin pipeline no teníamos que copiar e ir cargando los registros temporales de etapa en etapa.

Supongamos, por ejemplo, que tenemos una máquina sin pipeline cuyos ciclos de reloj son de 10 ns. y que la frecuencia de uso y duración de sus instrucciones es la siguiente:

- Operaciones de ALU, 40%, 4 ciclos
- Saltos, 20%, 4 ciclos
- Operaciones de memoria, 40%, 5 ciclos

El tiempo promedio de ejecución de instrucción es:

$$T_{ins} = 10 \times [4 \times (40 + 20) + 5 \times 40] = 44 \text{ ns.}$$

Supongamos que ahora se implementa la misma arquitectura pero con un cauce segmentado de 5 etapas y que eso genera un retraso de 1 ns. por cada ciclo de reloj. Entonces los ciclos quedan de 11 ns. y cada 11 ns. una instrucción se termina de ejecutar. Así que  $T_{ins}^{pipeline} = 11 \text{ ns}$  la ganancia es:

$$S = \frac{44}{11} = 4$$

La máquina con pipeline es 4 veces mejor que la original.

Retomando nuestra ecuación 7.1:

$$T_{ins}^{pipeline} = \frac{T_{ins}}{N} + \epsilon = \frac{44}{5} + 2.2$$

## 7.5 DISEÑO PARA FACILITAR EL PIPELINE

Hay varios factores a considerar para hacer efectivo el pipeline:

1. Si la longitud de instrucción es muy variable es difícil establecer una línea de pipeline fija. Si hay mucha variedad en la longitud de las instrucciones hay, en general, mucha variedad en tiempos de ejecución. En un 80X86, por ejemplo, las instrucciones miden de 1 a 17 bytes de longitud, simplemente para hacer fetch ya hay que haber comenzado a decodificar, tendremos que frenar el proceso de ejecución para traer lo que falta de una instrucción larga.
2. Si la arquitectura no es load-store entonces puede no haber una única etapa del ciclo de ejecución en la que se accede a la memoria, pueden ser varias y puede ser que haya que hacerlas de manera exclusiva. Si es una máquina registro-memoria, en las que a lo más un operando está en memoria, entonces puede ser que exista una sola etapa de acceso a memoria, pero si se usan modos de direccionamiento que no sea el directo de registro, hay

que calcular la dirección y hacer la operación con una sola ALU (si es el caso), las cosas se complican.

3. Si el formato de instrucción es fijo entonces es fácil saber tempranamente cuál es el destino y cuales son los operandos y se pueden cargar en los registros y tenerlos listos para la operación, cualquiera que esta sea. Es decir esto se puede hacer al mismo tiempo que se decodifica la instrucción. De otro modo hay que determinar primero que instrucción es, para determinar donde están los operandos. Se añade una etapa.
4. Si se fuerza a que los datos estén alineados en memoria cada lectura/escritura implica un solo acceso a memoria. Si esto no es así es posible requerir de más de un acceso implícitamente en cada lectura/escritura, esto retarda toda la línea de pipeline.

## 7.6 CONFLICTOS (*HAZARDS*)

Al inicio de este capítulo ya hemos apuntado algunos problemas que pueden ocurrir al implementar un pipeline. Nos avocaremos ahora a analizar estos y otros problemas llamados conflictos o *hazards*.

Estos problemas que, de hecho disminuyen el desempeño del pipeline pueden clasificarse de acuerdo a su origen:

**Clasificación de hazards.**

- Hazards estructurales. Son aquellos ocasionados por conflictos en el uso de hardware. Por ejemplo, si no tenemos dividida de alguna manera la memoria de almacenamiento de datos de la de almacenamiento de código, entonces en un solo ciclo de reloj trataríamos de ejecutar el ciclo IF y el MEM de diferentes instrucciones y eso ocasionaría un conflicto irresoluble, no es posible acceder dos veces a la memoria en un solo ciclo de reloj. La única manera de solucionar esto es retrasando toda la línea de pipeline, todas las instrucciones que siguen a la que se encuentra con el conflicto. A esto se le conoce como retrasar o “atorar” (*stall*) el pipeline. A los ciclos de reloj desocupados se les llama burbujas (*bubbles*).
- Hazards de control. Se toma una decisión que cambia el control de flujo del programa, esta decisión la toma una instrucción mientras las que, en principio le seguirían están en proceso de ejecución (ver figura 7.3).
- Hazards de datos. Un resultado anterior afecta, de alguna manera, a uno posterior. (ver figura 7.4).

```

; for (i=0 i<num; i++) {
    addi R1, R0, 0
    lw R2, num(R0)
repite: slt R3, R1, R2
        beqz R3, fuera
        ; instrucciones del ciclo
        ; ....
        addi R1, R1, 1
        j repite
fuera:  sw R7, resultado(R0)
        ;.....

```

**Figura 7.3.** Un ejemplo de hazard de control. Si es tomado (es decir, ocurre) el salto de la línea 4, entonces la instrucción que sigue es la de la línea 9, sin embargo las instrucciones que se han traído y que están en proceso de ejecución son las que se encuentren después de la línea 4, las del ciclo.

```

add R1, R2, R3 ; establece R1=R2+R3
sub R6, R1, R4 ; requiere R1

```

**Figura 7.4.** Un ejemplo de hazard de datos. La primera instrucción establece el valor de R1, la segunda lo utiliza. En una DLX sin pipeline no hay problema. En DLX con pipeline R1 es establecido por la primera instrucción en la etapa WB, pero la segunda instrucción lo requiere en la etapa ID. Cuando la primera esté en WB la segunda ya debería estar en EX, una etapa adelante de ID.

## 7.7 CLASIFICACIÓN DE HAZARDS DE DATOS

### Clasificación de hazards de datos.

Los hazards de datos pueden clasificarse de acuerdo con el orden de las escrituras y lecturas de datos que tienen lugar. Supongamos que la instrucción  $j$  es posterior a la  $i$ :

**RAW.** *Read After Write.* Es generado por una dependencia real de un resultado que debe ser generado para poder continuar:  $j$  quiere requiere un dato que aun no ha sido escrito por  $i$ . Este es el hazard más usual, pero afortunadamente puede ser resuelto con *forwarding*. A la dependencia que lo causa se le denomina *dependencia verdadera* o *true dependency*.

**WAW.** *Write After Write.*  $j$  trata de escribir un dato que antes de que haya sido escrito por  $i$ . Esto ocurre en pipelines en los que no hay una única etapa de acceso a memoria o si esta etapa es muy lenta y tarda más de un ciclo. En DLX no ocurre, ocurriría y en casos extraños, si

se adelantara la etapa WB de algunas operaciones y de hecho sólo se puede adelantar el WB de las aritmético-lógicas (ya que no acceden a memoria se puede omitir la etapa MEM) y además los accesos a memoria tardaran dos ciclos de reloj.

```
lw  R1, (R2)    ; IF ID EX MEM1 MEM2 WB
add R1, R3, R6 ; IF ID EX WB
```

En principio estas instrucciones no tienen sentido, pero son válidas. Este tipo de conflictos ocurren frecuentemente en arquitecturas en las que las instrucciones se ejecutan fuera de orden (*out of order*) a la dependencia que lo causa se le llama *dependencia de salida* u *output dependency*.

**WAR. Write After Read.** La instrucción *j* trata de escribir un dato que aún no ha sido leído por *i* (i leería un dato “demasiado nuevo”). Nuevamente, este tipo de hazard no ocurre en DLX porque se lee en la etapa ID y se escribe en WB que es posterior. Estos hazards ocurren en máquinas en las que algunas instrucciones pueden escribir muy temprano en la ejecución y otras leen muy tarde. Como en general es natural leer antes de escribir resultados, estos hazards son precedidos de otros y son frecuentes también en arquitecturas con ejecución fuera de orden.

```
sw R2, (R1) ; guarda R2 en mem. IF ID EX MEM1 MEM2 WB
add R2, R3, R5 ; IF ID EX WB
```

Si el store guarda en la segunda mitad del ciclo de MEM2 y la suma guarda el resultado en la primera mitad de su WB estamos en problemas. Al conflicto que los genera se le denomina *antidependencia* o *antidependency*.

Hay que notar que RAR *Read After Read* no es un hazard.

## 7.8 HAZARDS DE DATOS

Para resolver un hazard de datos como el de la figura 7.4 necesitaríamos que el valor que se obtuvo de la ALU en la etapa EX sea capaz de regresar a la ALU sin tener que esperar hasta que en WB se escriba en los registros. Cuando la instrucción de la línea 2 lo necesita es en ID, que es la etapa inmediata anterior a EX. Es decir, formalmente hablando el dato necesario para la instrucción de la línea 2 ya está calculado al final de su ciclo ID, porque para ese entonces ya se terminó también el ciclo EX de la instrucción anterior. Entonces todo lo que

*Forwarding.*

hay que hacer es retroalimentar a la ALU con su salida, podemos hacer que la salida de la ALU sea obtenida en la subida del reloj y que los operandos de la ALU sean establecidos a la bajada del reloj, así todo está bien. A esta técnica se le llama *forwarding*, *bypassing* o *short circuiting*.

Por supuesto para hacer *forwarding* necesitamos un poco más de hardware. Un dispositivo que sea capaz de notar que se ha producido el hazard de datos y que habilite el que la salida de la ALU regrese a la entrada al mismo tiempo que continúa su camino habitual. Este dispositivo hace parecer que el operando recibido proviene de los registros cuando en realidad es el resultado de la ALU. Además debe ser capaz de retomar el resultado de la ALU luego de la etapa EX o de MEM para hacer posible que no atoren ninguna de las dos instrucciones siguientes.

Una instrucción que pasa de la etapa ID a EX se dice que ha sido despachada (*issued*). En DLX todos los hazards de datos se detectan en ID (donde se obtienen los operandos), así que la instrucción es atorada antes de despacharla<sup>3</sup>. También podemos determinar si hace falta hacer *forwarding* (tomar lo que sale de la ALU por adelantado) y enviar las señales de control que lo hacen posible.

## 7.9 CUANDO HAY QUE ATORAR

Hay situaciones en las que a fuerza hay que atorar el pipeline. Por ejemplo:

```
lw R1, (R2)      ; IF ID EX MEM WB
sub R4, R1, R5   ;      IF ID EX  MEM WB
and R6, R1, R7   ;          IF ID  EX  MEM WB
or R8, R1, R9    ;              IF  ID  EX  MEM WB
```

2 necesita R1 al principio de su ciclo EX

1 lo obtiene hasta el final de ese mismo ciclo en su MEM

sólo es posible hacer *forwarding* con 3 y 4

la instrucción 2 se debe atorar un ciclo, lo que por supuesto retrasa las 3 y 4 también en un ciclo. Cuando se inserta una burbuja en el paso  $i$  de la instrucción  $j$  también hay burbuja en el paso  $i - 1$  de la  $j + 1$ , en el  $i - 2$  de la  $j + 2$  y así sucesivamente hasta llegar a una instrucción que empieza a ejecutarse después de la burbuja.

<sup>3</sup>Por cierto esto lo hace un dispositivo de hardware llamado *pipeline interlock*.

## 7.10 CONTROL DE FORWARDING

Revisemos las posibles situaciones de hazard de datos. Hay dos posibilidades: atorar o hacer *forwarding*. Ejemplifiquemos:

- Atorar.

```
lw  R1, 45(R2)
add R5, R1, R7
sub R8, R6, R7
or  R9, R6, R7
```

R1 se usa en el `add`, esto lo puede notar un comparador (luego veremos con detalle). Atoramos el `add` antes de despachar la instrucción. Desatoramos cuando el `lw` haya hecho WB o, mejor aun, hacemos *forwarding* desde el ciclo MEM y atoramos un ciclo menos.

- *forwarding*

```
lw  R1, 45(R2)
add R5, R6, R7
sub R8, R1, R7
```

`sub` usa R1 que es escrito por el `lw` de dos instrucciones anteriores. El mismo comparador mencionado arriba puede detectar esto.

Los conflictos de tipo WAR y WAW no son generados por una dependencia real de un dato del que depende el algoritmo en ejecución, son sólo derivados de querer usar un lugar que está siendo usado para otra cosa, así que es sencillo resolverlos si se tiene otro lugar disponible para usar.

En un hazard WAR o antidependencia el problema es que una instrucción posterior pretende usar para escribir un lugar en el que una instrucción previa debe leer un dato. El conflicto deja de existir si la instrucción posterior utiliza otro lugar para escribir.

Análogamente en un hazard WAW o dependencia de salida, una instrucción posterior pretende usar para escribir un lugar en el que una instrucción previa debe también escribir un resultado (lo que por cierto significa es que entre estas dos instrucciones seguramente hay alguna que tiene un RAW con la previa). De la misma manera, el conflicto deja de existir si la instrucción posterior utiliza otro lugar para escribir.

Claro que, en ambos casos, si la instrucción posterior escribe en otro sitio, entonces cada vez que el dato que se escribió en él sea usado en el

Instr. en ID	Instr. fuente	Reg. fuente	Reg. dest.	Coinciden
ALU R load, store, branch	ALU R	EX/MEM.	A	EX/MEM.IR(16,20) (rd) ID/EX.IR(6,10) (rs1)
ALU R	ALU R	EX/MEM.	B	EX/MEM.IR(16,20) (rd) ID/EX.IR(11,15) (rs2)
ALU R load,store, branch	ALU R	MEM/WB.	A	MEM/WB.IR(16,20) (rd) ID/EX.IR(6,10) (rs1)
ALU R	ALU R	MEM/WB.	B	MEM/WB.IR(16,20) (rd) ID/EX.IR(11,15) (rs2)
ALU R load, store, branch	ALU I	EX/MEM.	A	EX/MEM.IR(11,15) (rs2) ID/EX.IR(6,10) (rs1)
ALU R	ALU I	EX/MEM.	B	EX/MEM.IR(11,15) (rs2) ID/EX.IR(11,15) (rs2)
ALU R load, store, branch	ALU I	MEM/WB.	A	MEM/WB.IR(11,15) (rs2) ID/EX.IR(6,10) (rs1)
ALU R	ALU I	MEM/WB.	B	MEM/WB.IR(11,15) (rs2) ID/EX.IR(11,15) (rs2)
ALU R load, store, branch	load	MEM/WB.	A	MEM/WB.IR(11,15) (rs2) ID/EX.IR(6,10) (rs1)
ALU R	load	MEM/WB.	B	MEM/WB.IR(11,15) (rs2) ID/EX.IR(11,15) (rs2)

**Tabla 7.1.** Tabla de solución de hazards mediante *forwarding*. La notación (X,Y) indica los bits de índice X a Y del campo especificado.



programa, se debe hacer uso del lugar real en el que el dato fue escrito y no aquel en el que originalmente debía escribirse para garantizar la consistencia semántica del programa. La técnica para llevar a cabo esto es conocida como *renombrado* o *renaming*.

## 7.11 HAZARDS DE CONTROL

En general los hazards de control causan más pérdida de desempeño que los de datos. El problema principal consiste en que la decisión de saltar o no se toma con base en alguna comparación que se efectúa en etapas adelantadas del ciclo de ejecución y la dirección de salto es calculada por la ALU, lo que también significa que se obtiene etapas adelantadas de ejecución, cuando ya se tienen los operandos y, lo que es más importante, ya se ha hecho el fetch de las instrucciones posteriores al salto.

Como siempre el método de solución más simple es el mismo para cualquier hazard, atorar el pipeline hasta que se termine de ejecutar la instrucción de salto. Por supuesto no queremos hacer esto.

Hay dos posibilidades para tratar de resolver un hazard de control:

**Solución de  
hazards de  
control.**

1. Atorar el pipeline, siempre que se traiga una instrucción de salto, hasta determinar si se va a saltar o no.
2. Predecir si el salto será tomado o no. Por ejemplo, en un ciclo como el mostrado en la figura 7.3 las instrucciones del ciclo se ejecutan `num` veces. Es decir: de `num+1` veces que se pasa por la instrucción de la línea 4, `num` veces no se salta, así que si predecimos que no se saltará nuestro pronóstico será acertado casi siempre y entonces casi nunca perdemos los dos ciclos de reloj mencionados. En cambio en el salto de la línea 8 acertaremos la mayor parte del tiempo si predecimos que el salto será tomado, entonces, en lugar de traer y decodificar la instrucción de la línea 9 y siguientes, mejor traemos las de la línea 3 y siguientes.
3. Hacer algo útil mientras se determina si se debe saltar o no. Ejecutar instrucciones que, de todos modos debemos ejecutar sin importar si se salta o no (*delayed branch*).

Para contribuir a resolver un hazard de control, como el del ejemplo 7.3, es que movimos el multiplexor que interviene en la actualización de PC a la izquierda dos etapas (compárense las figuras 6.2 o 6.3 con la 7.2). Con esto la línea de pipeline se entera, lo antes posible, de que se deben ejecutar las instrucciones de salto. En DLX la distancia entre

EX, que determina el nuevo contenido de PC e IF, etapa en la que se encuentra la operación que se da cuenta de que la siguiente instrucción a ejecutar es la de destino del salto, es dos, así que hay que tirar a la basura lo que se ha hecho de dos instrucciones: la que está en ese momento en IF y la que está en ID, dos ciclos de reloj.

En una primera aproximación a una solución razonable tendríamos que atorar el pipeline hasta que ya se tenga información de la dirección de salto y si este se va a tomar o no. En DLX esto es hasta que la instrucción de salto termine su ciclo MEM, que es cuando se pasa el resultado de la ALU al multiplexor de PC .

```
instr. de salto (i) IF ID EX MEM WB
(i+1)                IF bur bur IF ID EX MEM WB
(i+2)                IF ID EX ...
```

En el ciclo WB de la instrucción de salto se procede a cargar la verdadera siguiente instrucción. Pero si no se salta sale sobrando el segundo IF de la instrucción  $i + 1$  ya que es la misma que ya se había cargado, se gastan entonces 3 ciclos de reloj.

Tres ciclos de reloj son muchos, en ese intervalo podrían salir tres instrucciones. Dado que la frecuencia de los saltos es de un 30% del total de instrucciones esto es bastante malo. Hay dos cosas que podemos hacer:

1. Saber si el salto será tomado o no lo más temprano posible.
2. Calcular el valor que deberá tener PC (en caso de que el salto sea tomado) lo más pronto posible.

Para lograr un desempeño óptimo deben hacerse ambas cosas.

En DLX los saltos requieren probar si un registro es cero o no. Esto puede hacerse, lo más temprano, en ID. Pero la ALU, que calcula la dirección de salto, es accedida hasta EX. No podemos adelantar más el conocer si un registro es cero, así que lo único que es posible hacer es poner algún dispositivo que pueda calcular en ID la dirección de salto, si hacemos esto el pipeline sólo tendría que atorarse un ciclo. Pero surge entonces otro problema. Si la instrucción que antecede al salto altera el valor del registro que es probado entonces se incurrirá en un hazard de datos RAW. Que de inmediato pensamos en solucionar haciendo forwarding, pero no podemos porque el valor se obtiene al final del ciclo EX de la instrucción previa al salto y lo necesitamos al principio del ciclo ID de la siguiente, ambas cosas ocurren al mismo tiempo, en el mismo ciclo. Así que tendríamos que viajar hacia atrás en el tiempo. Perdemos un ciclo más en este caso y ni modo.

En otras máquinas el costo por salto (*branch penalty*) es peor, por ejemplo si el ciclo de fetch y el de decodificación tienen un retardo mayor (instrucciones complicadas).

En SPECint el 13% de los saltos son condicionales hacia adelante, el 3% son condicionales hacia atrás y un 4% son incondicionales<sup>4</sup>, en promedio el 67% de los saltos condicionales son tomados así que para reducir la penalización por salto podemos suponer que siempre que se llega a una instrucción de salto este será tomado y en cuanto tengamos acceso a la dirección de salto hagamos fetch de las instrucciones que creemos serán las siguientes. Esto lo podemos hacer si ponemos, como ya habíamos dicho, un sumador que calcule la dirección de salto desde el ciclo de ID de la instrucción de salto. Al mismo tiempo se estaría haciendo el fetch de la instrucción bajo el salto, lo que tiramos a la basura y hacemos fetch del destino de salto.

**Predicción  
de saltos.**

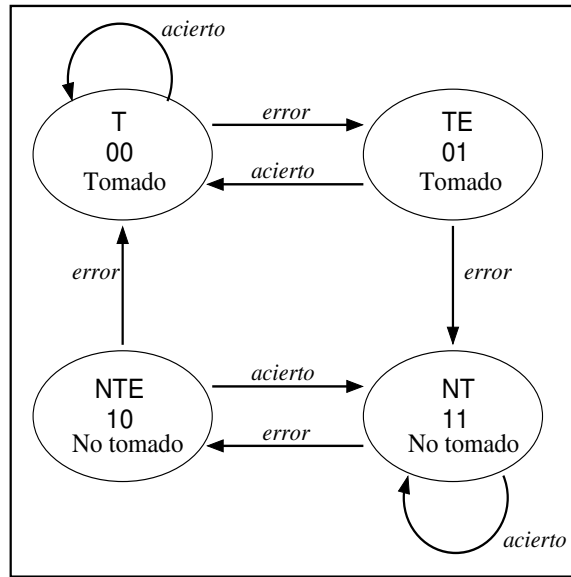
En general tenemos pues las siguientes opciones:

- No predecimos nada, cargamos la instrucción de abajo del salto, Si este es tomado tiramos a la basura las instrucciones que empezaron a ejecutarse (*freeze* o *flush*).
- Predecimos. Hay varias opciones, las más simples son predecir siempre lo mismo: siempre supongo que será tomado o siempre supongo que no será tomado. En ambos casos si atino pierdo un ciclo de reloj, si me equivoco pierdo dos. Hay opciones más interesantes que revisaremos un poco más adelante.

**Predicción de salto de dos bits.** Esta es una alternativa usual. Consiste en manejar una máquina de estados finita con cuatro estados: tomado (T), no tomado (NT), tomado con un error (TE) y no tomado con un error (NTE). Predecimos que el salto será tomado, si acertamos permanecemos en T, si no nos pasamos a TE, la próxima vez que pasemos por ese mismo salto predecimos que lo tomaremos otra vez, si acertamos nos pasamos a T pero si no, pasamos al estado NT. Estando en NT predecimos no tomado, si acertamos permanecemos en ese estado, pero si no pasamos a NTE en el que predecimos, también no tomar el salto si acertamos regresamos a NT y si no, pasamos a T. La gráfica del autómata se muestra en la figura 7.5.

Esto permite, en principio hacer predicciones más precisas considerando, de alguna manera la historia de cada salto, la desventaja de este esquema es que hay que guardar en el CPU una tabla hash con

<sup>4</sup>Nuevamente las medidas se tomaron con SPECint en MIPS.



**Figura 7.5.** Máquina de estados finita para 2-bit branch prediction

una entrada para cada salto que contenga dos bits indicando el estado actual de la máquina de estados finita para ese salto en particular.

#### 7.11.1 Salto retardado

El esquema de salto retardado o *delayed branch* consiste en aprovechar el tiempo mientras se determina si hay que saltar o no, haciendo algo que de todas maneras hay que hacer.

- Salto hacia adelante. Tomamos alguna(s) instrucción(es) que había que hacer antes. Algo como esto:

```
add R1, R2, R3
beqz R2, salto1
```

lo reacomodamos como:

```
beqz R2, salto1
add R1, R2, R3
```

El resultado es el mismo y podemos aprovechar los ciclos de reloj que, de otra manera, habría que tirar a la basura probablemente con un flush. El lugar donde se inserta la instrucción se denomina ranura de retardo (*delayed slot*).

- Salto hacia atrás. Tomamos algo que ya hubo que hacer y hay que repetir. Transformamos algo como esto:

```
salto1: add R4, R5, R6
        ;....
        ;....
        beqz R2, salto1
```

en esto:

```
        add R4, R5, R6
salto1: ;....
        ;....
        beqz R2, salto1
        add R4, R5, R6
```

De esta manera la instrucción siguiente del salto (en la ranura de retardo) también es aprovechable (siempre que se salta).

Evidentemente el más indicado para hacer estas alteraciones del orden del código es el compilador. Es él el indicado para insertar las ranuras de retardo si que se altere la semántica del programa.

**Ranuras de retardo y el compilador.**

## 7.12 PREDICACIÓN Y ESPECULACIÓN