Capítulo 7

PARALELISMO A NIVEL DE INSTRUCCIÓN

7.1 EL CONCEPTO DE CAUCE SEGMENTADO

El concepto de cauce segmentado, al que denominaremos indistintamente como *pipeline*[1] es realmente anterior a las computadoras digitales. En realidad es el mismo concepto que el de linea 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"[2]. 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.

[1]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.
[2]Usamos aquí la palabra programa para hacer claro el proceso secuencial, paso a paso. Como ya sabemos, en realidad este "programa" puede estar alambrado.

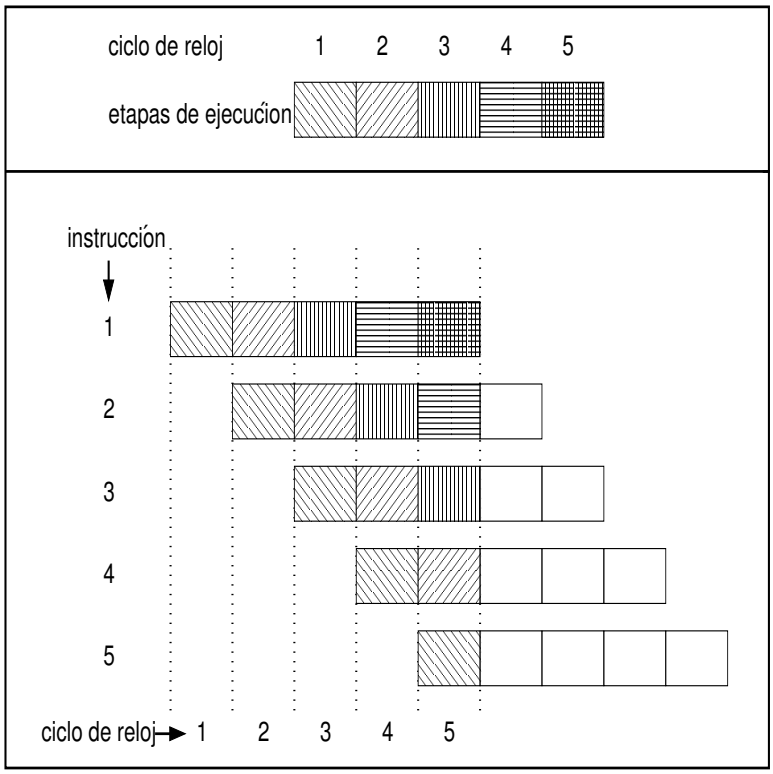

Figura 7.1. En una máquina X normalmente una instrucción toma 5 ciclos de reloj en su ejecución y hasta que se termina de ejecutar es posible ejecutar la siguiente. Si la máquina en cuestión tuviera pipeline entonces es posible ejecutar el mismo tiempo diferentes trozos de diferentes instrucciones.

Pipeline y rendimiento (*throughput*).

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.

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:

$$T_{ins}^{pipeline} = \frac{T_{ins}}{N} + \epsilon \tag{7.1}$$

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 ϵ 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*)
$\mathsf{IR} = \mathsf{mem}[\mathsf{PC}]$, $\mathsf{NPC} = \mathsf{PC} + 4$

ID (*Instruction Decode*)
$\mathsf{A} = \mathsf{regs}[\mathsf{IR}_{6,\ldots,10}]$, $\mathsf{B} = \mathsf{regs}[\mathsf{IR}_{11,\ldots,15}]$, $\mathsf{Imm} = ((\mathsf{IR}_{16})^{16} \uplus \mathsf{IR}_{16,\ldots,31})$.
(recuérdese los formatos de la figura 5.8, A es `rs1` y B es `rd`).

EX (*Execution*).

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

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

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

WB (*Write Back*).

- instrucción aritmético-lógica, registro-registro.
  $\mathsf{regs}[\mathsf{IR}_{16,\ldots,20}] = \mathsf{ALUOutput}$
- Instrucción aritmético-lógica, registro-inmediato.
  $\mathsf{regs}[\mathsf{IR}_{11,\ldots,15}] = \mathsf{ALUOutput}$
- Instrucción de carga. $\mathsf{regs}[\mathsf{IR}_{11,\ldots,15}] = \mathsf{LMD}$

Ejemplos.

- `LW R1, 30(R2)`

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

  ID $\mathsf{A} = \mathsf{regs}[\mathsf{IR}_{6,\ldots,10}] = \mathsf{regs}[2]$ (`rs1`), $\mathsf{B} = \mathsf{regs}[\mathsf{IR}_{11,\ldots,15}] = \mathsf{regs}[1]$ (`rd`), $\mathsf{Imm} = 30$

EX $\mathsf{ALUOutput} = \mathsf{A} + \mathsf{Imm} = 30 + \mathsf{regs}[2]$

MEM $\mathsf{PC} = \mathsf{NPC}, \mathsf{LMD} = \mathsf{mem}[\mathsf{ALUOutput}] = \mathsf{mem}[30 + \mathsf{regs}[2]]$

WB $\mathsf{regs}[\mathsf{IR}_{11,\ldots,15}] = \mathsf{regs}[1] = \mathsf{LMD}$

- `SW R1, 30(R2)`

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

  ID $\mathsf{A} = \mathsf{regs}[\mathsf{IR}_{6,\ldots,10}] = \mathsf{regs}[2]$ (`rs1`), $\mathsf{B} = \mathsf{regs}[\mathsf{IR}_{11,\ldots,15}] = \mathsf{regs}[1]$ (`rd`), $\mathsf{Imm} = 30$

  EX $\mathsf{ALUOutput} = \mathsf{A} + \mathsf{Imm} = 30 + \mathsf{regs}[2]$

  MEM $\mathsf{PC} = \mathsf{NPC}, \mathsf{mem}[\mathsf{ALUOutput}] = \mathsf{mem}[30 + \mathsf{regs}[2]] = \mathsf{B} = \mathsf{regs}[1]$

  WB No se usa

- `J etiqueta`

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

  ID $\mathsf{Imm} = \mathsf{dist}(\mathsf{PC} + 4, \mathsf{etiqueta})$

  EX $\mathsf{ALUOutput} = \mathsf{NPC} + \mathsf{Imm}$

  MEM $\mathsf{PC} = \mathsf{ALUOutput}$

  WB No se usa

- `BEQZ R4, etiqueta`

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

  ID $\mathsf{A} = \mathsf{regs}[\mathsf{IR}_{6,\ldots,10}] = \mathsf{regs}[4], \mathsf{B} = \mathsf{regs}[0], \mathsf{Imm} = \mathsf{dist}(\mathsf{PC} + 4, \mathsf{etiqueta})$

  EX $\mathsf{ALUOutput} = \mathsf{NPC} + \mathsf{Imm}$

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

  WB No se usa

- `ADD R1, R2, R3`

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

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

  EX $\mathsf{ALUOutput} = \mathsf{A} + \mathsf{B}$

  MEM $\mathsf{PC} = \mathsf{NPC}$

  WB $\mathsf{regs}[\mathsf{IR}_{16,\ldots,20}] = \mathsf{regs}[1] = \mathsf{ALUOutput}$

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.$IR_{6,\ldots,10}$]
ID/EX.B = regs[IF/ID.$IR_{11,\ldots,15}$]
ID/EX.NPC = IF/ID.NPC
ID/EX.IR = IF/ID.IR
ID/EX.Imm = (IF/ID.IR_{16})^{16} $\uplus$ IF/ID.$IR_{16,\ldots,31}$

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

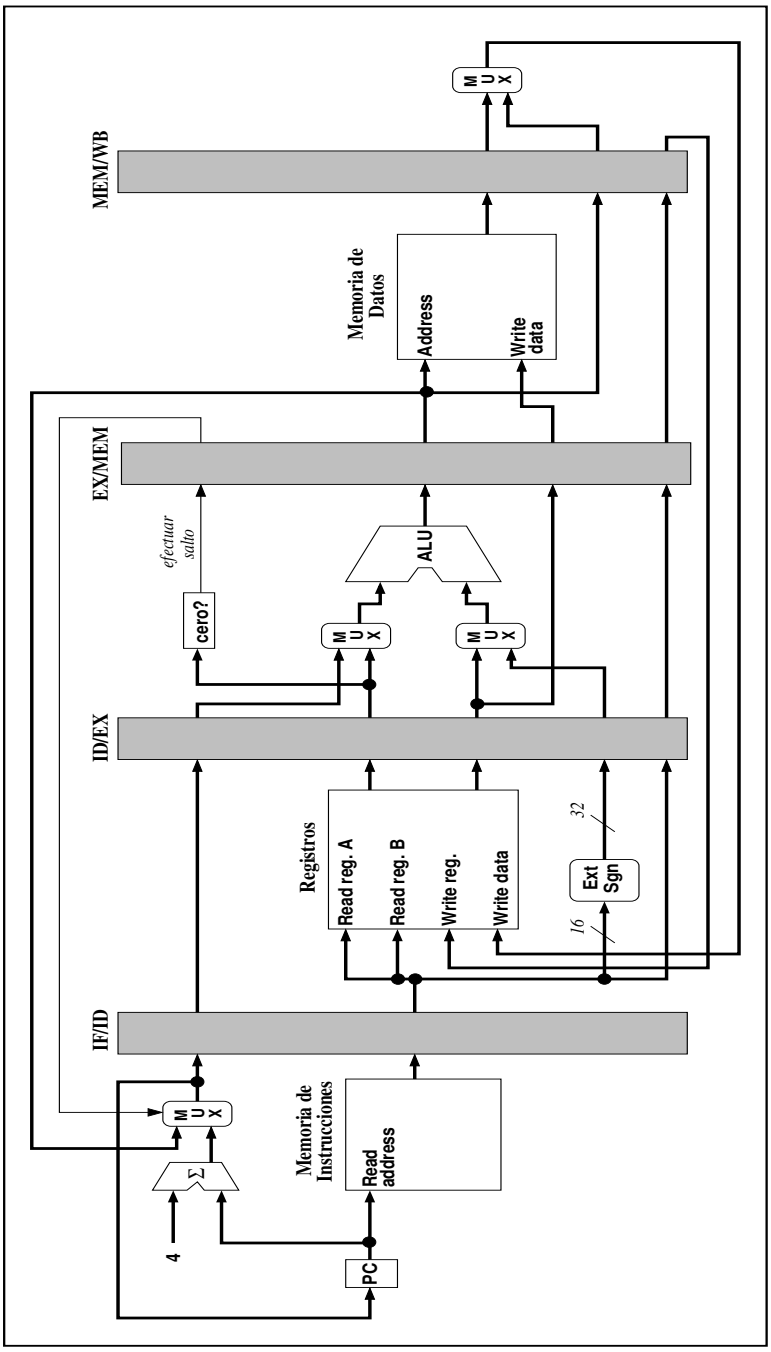

Figura 7.2. DLX preparada para pipeline.

- 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 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.$IR_{16,\ldots,20}$] = MEM/WB.ALUOutput
  - registro-inmediato
    regs[MEM/WB.$IR_{11,\ldots,15}$] = MEM/WB.ALUOutput
- load
  regs[MEM/WB.$IR_{11,\ldots,15}$] = 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:

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

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 $T_{ins}^{pipeline} = 11\ ns$ la *ganancia* es:

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

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

Retomando nuestra ecuaciôn 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 linea 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*.

Clasificación de *hazards*.

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

- 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 linea 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 frecuentamente 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 ejecucuón 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

Forwarding.

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 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[3]. 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.

[3]Por cierto esto lo hace un dispositivo de hardware llamado *pipeline interlock*.

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.

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.

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.

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 instruciones 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 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.

Solución de *hazards* de control.

Hay dos posibilidades para tratar de resolver un hazard 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 actualizción 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 e 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).

Predicción de saltos.

En SPECint el 13% de los saltos son condicionales hacia adelante, el 3% son condicionales hacia atrás y un 4% son incondicionales[4], 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.

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

[4]Nuevamente las medidas se tomaron con SPECint en MIPS.

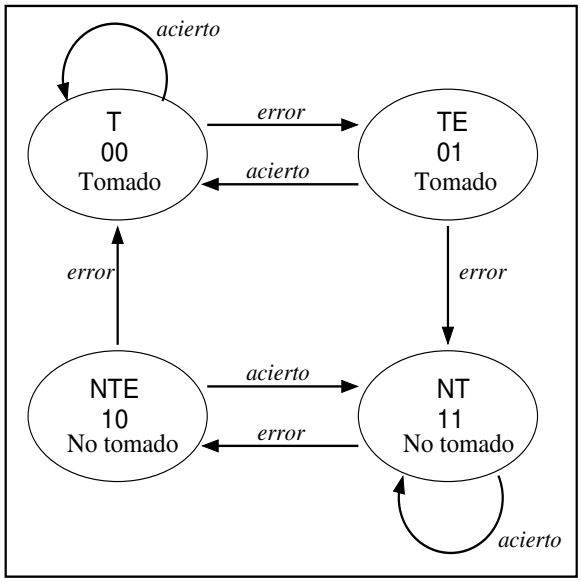

Figura 7.5. Máquina de estados finita para *2-bit branch prediction*

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 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).

Ranuras de retardo y el compilador.

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.

7.12 PREDICACIÓN Y ESPECULACIÓN

Dado que los saltos son el problema fundamental en el caso de los *hazards* de control, convendría evitarlos en la medida de lo posible. Si pudieramos eliminarlos, o al menos algunos de ellos resultaría provechoso. Ese es el objetivo de una técnica llamada *predicación*. Consiste en preceder algunas instrucciones del nombre de un registro de predicado, que no es otra cosa más que un valor de verdad. Si el registro de predicado contiene un 1 (verdadero) la instrucción que precede se ejecuta, si contiene un 0 (falso) la instrucción no se ejecuta. El valor del registro de predicado se establece en una instrucción de comparación, como las que solemos usar para establecer las condiciones de un salto. El resultado de la comparación se almacena en un registro de predicado y el complemento en otro. El objetivo de la predicación es que, desde que se hace fetch de la instrucción se consulta el registro de predicado que la precede y sólo si es verdadero se pasa a decodificación, de otro modo se descarta. Se pueden entoces intercalar las instrucciones que requieren el predicado como verdadero con las que lo requieren como falso, es decir, las que están precedidas por el predicado complementario.

El siguiente segmento de código, por ejemplo:

```
 if (a) { /* si a no es cero */
   c = d | e;
   b = c + d;
}
else /* a es cero */
   e = d;
```

En una arquitectura IA-64 de Intel, usando predicación, se traduciría en:

```
cmp.ne p1,p2 = r6, r0  // p1 = (a != 0)
(p1) or  r8 = r9, r10
(p1) add r7 = r8, r9
(p2) mov r10 = r9
```

Otro concepto que contribuye a mantener el pipeline lleno, es el conocido como *especulación*. Este consiste en tratar de predecir el futuro, específicamente en el caso de una carga (load). Si suponemos que una cierta instrucción de carga realmente se va a llevar a cabo podríamos pensar en hacerla cuanto antes, en vez de esperar a que se lleve a cabo cuando lo indica el código, de ese modo podríamos

evitar algunos hazards RAW con las instrucciones que requieren el dato cargado inmediatamente después de la instrucción en cuestión y que deben retrasarse por la latencia de memoria.

Hay dos tipos de carga especulativa:

- Especulación de datos. Cuando al adelantar la instrucción de carga se "pasa por encima" de una instrucción de *store*.
- Especulación de control. Cuando al adelantar la instrucción de carga se "pasa por encima" de una instrucción de *salto*.

En el primer caso la *especulación* consiste en suponer que la carga tiene lugar sobre una dirección de memoria diferente de la instrucción de *store*. Si esta suposición no es correcta significa que la dirección de carga y la de *store* son la misma, un caso de lo que se conoce como *aliasing*. En el caso de que la dirección de carga sea un "alias" de la de *store* entonces, al adelantar la carga, estamos leyendo un dato más viejo de lo que realmente deberíamos ya que el *store* actualiza el contenido de esa direción. En ese caso la especulación nos lleva a un error y habría que corregirlo. Para detectar el fallo lo que se hace es registrar las direcciones accedidas en las cargas especulativas en una tabla, en el caso de la arquitectura IA-64 de Intel esta tiene el nombre de *Advanced Load Address Table* o ALAT. Cuando se hace un *store* se busca la dirección accedida en la ALAT y si se encuentra allí se borra. Cuando, más adelante, se llega al sitio donde antes estaba la instrucción de carga se hace uina verificación buscando la dirección en la ALAT, si está allí significa que ningún *store* la accedió y entonces la versión de la carga que tenemos es correcta; si, por el contrario, la dirección ya no está, significa que algún *store* enmedio la accedió y la version de dato que tenemos es incorrecta, por los que la carga deberá hacerse de nuevo, como en el c'odigo original. A continuación se muestra un ejemplo de especulación de datos:

```
add r9  = r13,r4
shr r18 = r5,r1
sub r2 3 = r13,r1
st8 [r55] = r23
ld8 r5  = [r8]
shr r7  = r5,r12  // RAW con r5
```

Este código se transforma, haciendo la carga especulativa al principio, en:

```
ld8.a r5 = [r8]  // carga avanzada r8 a ALAT
```

```
add r9  = r13,r4
shr r18 = r5,r1
sub r23 = r13,r1
st8 [r55] = r23 // r55 se borra de ALAT
ld8.c r5 = [r8] // Sólo si r8 no está en ALAT
shr r7  = r5,r12
```

En la especulación de control la carga se hace antes de un salto que la determina, así que si el salto hace que no se pase por la línea de código donde se hace la carga, esta relmente no ocurre. Si la adelantamos corremos el riesgo de ejecutar código que realmente no debería ocurrir. Esto no es un problema en sí, el dato cargado, en principio no se usa si no se lleva a cabo la carga y el contenido del registro seguramente será asignado a un dato útil. El problema es si se accede a una localidad de memoria inválida (*null pointer exception* o *segmentation fault*): si el código no debe ejecutarse entonces el acceso a dicha localidad no debe generar una excepción, pero si sí se debe ejecutar entonces la excepción realmente ocurre y debe ejecutarse. La carga especulativa se realiza y si se accede auna localidad inválida, no se debe lanzar la excepción hasta estar seguros de que en verdad la carga ocurrirá, se debe poder diferir al lanzamiento de la excepción, para ello será necesario prevenir al procesador para que no sea lanzada e introducir una instrucción de verificación en donde antes estaba la carga para poder determinar si realmente debe ejecutarse o no.

El código mostrado abajo:

```
      add r9  = r13,r4
      shr r18 = r5,r1
      sub r23 = r13,r16
      cmp.eq p1,p2 = r23, 0
      br.cond.dptk Etiq1
      ld8 r6  = [r2]
      shr r7  = r6,r12  // RAW con r6
      add r17 = r7,r15  // RAW con r7
Etiq1: and r4  = r23,r1
```

Se transforma, haciendo carga especulativa de control, en:

```
      ld8.s r6 = [r2]   // carga especulativa
      add r9  = r13,r4
      shr r18 = r5,r1
      sub r23 = r13,r16
      shr r7  = r6,r12  // uso de r6
```

```
        cmp.eq p1,p2 = r23, 0
        br.cond.dptk Etiq1
        chk.s r6, Oops   // verificación
        add r17 = r7,r15
Etiq1:  and r4  = r23,r1
   ...
Oops:   // manejo de excepción
```

7.13 EJECUCIÓN SUPERESCALAR

Empezamos estudiando un procesador capaz de ejecutar instrucciones a una tasa de una instrucción por cada cinco ciclos de reloj, es decir la tasa de ciclos de reloj por instrucción, definida en la expresión 1.3, resulta ser un número estrictamente mayor que uno. Pensamos luego en incrementar el desempeño del procesador aumentando el rendimiento (*troughput*) al implementar un cauce segmentado (*pipeline*). Con esto logramos que, prácticamente en cada ciclo de reloj termine de ejecutarse una instrucción. La tasa de ciclos por instrucción es, entoces, casi uno. Esto mejora en general el desempeño del sistema descrito por la ecuación 1.4 por un factor cercano al número de etapas en el pipeline. Revisaremos ahora estratégias cuyo propósito es incrementar aún más el paralelismo en la ejecución del flujo de instrucciones, lo que llamamos el *paralelismo a nivel de instrucción* o ILP. La intención es ahora terminar más de una instrucción en cada ciclo; hacer menor que uno la tasa de ciclos por instrucción.

Para lograr nuestro propósito debemos repensar las labores para ejecutar las instrucciones de acuerdo con el tipo de instrucción: es fácil por ejemplo, darse cuenta que totas las instrucciones de load o store realmente no necesitan una ALU completa, bastaría con un sumador. Lo mismo ocurre con las instrucciones de salto, necesitan un sumador para calcular la dirección de salto y un comparador para determinar si se salta o no, adicionalmente necesitarán una tabla de direcciones de salto previamente calculadas y las tablas de historia de los saltos. Las instrucciones de punto flotante necesitan una ALU cualitativamente diferente de la necesaria para aritmética entera.

Unidades funcionales

Podemos entonces pensar en dotar a nustro procesador de recursos de hardware especificos para cada tipo diferente de instrucción: una ALU entera, una para punto flotante, una unidad de cálculo de direcciones para load-store y otra para procesar saltos. A estas unidades especializadas se les denomina *unidades funcionales*. Podríamos entonces pensar en despachar varias instrucciones en cada ciclo de reloj. En

principio, una aritmético-lógica entera (ALU), una de punto flotante (FPU), una de load o store (LSU) y una de salto (BU).

Superescalar

El problema fundamental ahora es procurar, en lo posible, mantener ocupadas todas las unidades funcionales todo el tiempo. Cada una de ellas, en principio, con su propio pipeline. Del éxito en esta tarea depende el desempeño total del sistema y, claro está, la conveniencia de la relación costo-beneficio entre el costo del hardware adicional incluido y la tasa real de ciclos por instrucción (o su inverso: instrucciones terminadas por ciclo, que frecuentemente preferiremos usar). Se denomina *superescalar* a una arquitectura en la que varias unidades funcionales operan en paralelo ejecutando diferentes instrucciones en los mismos ciclos de reloj. Cada unidad puede o no poseer un pipeline, eso no forma parte formal de la definción de superescalar, pero normalmente lo poseen. La primera arquitectura considerada superescalar fue la CDC 6600 diseñada por Seymour Cray en 1965, la que es tambi[en considerada la primera supercomputadora.

7.14 SCOREBOARDING

Se requiere entonces de tomar una secuencia de varias instrucciones del flujo del programa y repartirlas entre las diversas unidades funcionales que pueden ejecutarlas. Pero para que esto tenga el efecto deseado: mantener ocupadas a la mayor cantidad de unidades funcionales que sea posible, es necesario desacomodar las instrucciones, ejecutarlas en un orden probablemente distinto del que tienen en el código del programa. Si hay, por ejemplo, dos instrucciones de aritmética entera consecutivas y luego una de punto flotante, pero se posee sólo una unidad aritmético lógica y una de punto flotante, sería conveniente ejecutar la primera de las instrucciones de aritmética entera y luego la de punto flotante, retrasando un lugar la otra de aritmética entera hasta que la unidad se desocupe. Se requiere hacer lo que se denomina una *planificación dinámica* (*Dynamic Scheduling*) de instrucciones y ejecutarlas fuera de orden (*Out Of Order Execution*). Para hacer esto la CDC 6600 implementó una estrategia llamada *scoreboarding*, cuyas etapas de ejecución representan un cambio sobre las conocidas de nuestro pipeline, aunque en principio este podría adaptarse para implementarla.

Las etapas de ejecución son las siguientes:

Fetch Se cargan en el procesador varias instrucciones.

Issue Se lee el código de operación, se determina el tipo de operación y sí hay o no una unidad funcional disponible que pueda

ejecutarla. Se verifica que el registro en el que escribirá no sea destino de alguna instrucción en ejecución. Se evitan entonces los hazards estructurales WAW. Si existe un hazard estructural o WAW entonces la instrucción se atora en esta etapa.

Read Operands Se verifica si los operandos están disponibles en los registros del procesador, si es así, se enruta la instrucción a la unidad funcional, si no, se atora. Se evitan así los hazards RAW.

Ejecución Cuando la unidad funcional recibe la instrucciön y los operandos están disponibles en los registros del procesador, la ejecuta.

Resultados. Una vez que la instrucción termina de ejecutarse, se verifica que no pretenda escribir en algún operando origen de otra instrucción, si es así se atora, si no se procede a escribir resultados. Se evitan entonces los hazards WAR.

La técnica se llama *scoreboard* porque para implementarla, la CDC usó varias tablas: una para registrar el estado de ejecución de cada instrucción, otra para registrar el estado de cada unidad de ejecución y una última para el estado de los registros.

En el esquema de scoreboarding la tabla de registro de estado de las unidades funcionales contiene los siguientes campos:

Ocupada Bandera para indicar si está ocupada o no.

Op La operación en ejecución.

Dst Registro destino.

Src1, Src2 Registros origen.

FU1, FU2 Unidades funcionales que deben producir Src1 y Src2.

R1, R2 Banderas para indicar si están listos o no los operandos origen. Luego de leidos se pone en falso.

Cada renglón de la tabla corresponde a una unidad funcional.

La tabla de registro de estado de las instrucciones contiene:

Instrucción. La instrucción completa: opcode, operandos origen, operando destino.

Issue. Verdadero si la instrucción está o ya pasó por esa etapa.

		Estado de instrucción			
No.	Instrucción	Issue	ReadOps	ExecComp	WrRes
1	lw R2, 132(R17)	1	2	3	4
2	lw R3, 12(R19)	1	6	7	8
3	add R5, R6, R2	1	5	6	7
4	beqz R3, salto1	1	2		
5	sub R8, R5, R2	1			

Tabla 7.2. Tabla para guardar el estado de cada instrucción. Suponemos que se está en el ciclo de reloj número 7.

Read operands. Verdadero si la instrucción ya ha podido leer todos sus operandos origen.

Execution complete. Si la operación especificada por la instrucción ya se llevó a cabo.

Write result. Indica si ya fue posible escribir en los registros del procesador el resultado de la instrucción.

Por último, la tabla de estado de los registros del procesador indica, para cada uno de ellos, la unidad funcional que debe producir el que será su contenido.

■ **EJEMPLO 7.1**

■

La técnica de scoreboarding es un primer y muy importante paso para la repartición dinámica de instrucciones, condicion necesaria para la ejecución fuera de orden. Pero hay que señalar que no fue diseñada, en principio, con el propósito de ejecutar las instrucciones fuera de orden. La técnica opera, diríamos hoy día, conservadoramente: atora el reparto en cuanto se detecta un hazard y no implementa forwarding o renombrado de registros. Algo que la siguiente estrategia sí hace.

	Estado de Unidad								
Nombre	Ocupada	Op	Dst	Src1	Src2	FU1	FU2	R1	R2
ALU	Sí	add	R5	R6	R2		LSU	Sí	No
FPU	No								
LSU	Sí	load	R2	R17	132			Sí	Sí
BrU	Sí	beqz		R3	R0	LSU		No	Sí

Tabla 7.3. Tabla para guardar el estado de cada unidad funcional. Suponemos que estaríamos en el ciclo de reloj número 3.

R2	R3	R5	R6	R8	R17	R19
LSU	LSU	ALU				

Tabla 7.4. Tabla para guardar el estado de los registros.

7.15 ALGORITMO DE TOMASULO

En 1967 Robert Tomasulo, ideo otro esquema para realizar el reparto dinámico de instrucciones en la unidad de punto flotante de la IBM 360. Las condiciones ante las que las instrucciones se atoran son menos que en scoreboarding, pero se requiere de hardware un poco más complejo. A diferencia de scoreboarding, el algoritmo de Tomasulo sí fue pensado para ejecución fuera de orden y por tanto ya atora en tantos casos.

El algoritmo está basado en los conceptos de:

- Estación de reservación. Entidades asociadas, uno a uno, con las unidades funcionales. Se encargan de gestionar la ejecución de las instrucciones, resolver los hazards, almacenar resultados.

- Canal de datos común. O *Common Data Bus*, CDB por sus siglas. Es, como su nombre lo indica, un canal por el que transitan los resultados de la ejecución de las instrucciones y que serán escritos en los registros y además resultan útiles para las instrucciones en proceso.

Las etapas:

Fetch Se trae una secuencia de varias instrucciones a la *cola de fetch*, una estructura de datos FIFO para mantener el orden de las instrucciones.

Issue Se toma la instrucción siguiente de la cola de fetch.

- Si sus operandos están en los registros.
  - Si hay una unidad funcional disponible, se envía a ejecución.
  - Si no se atora la instrucción hasta que haya unidad funcional.
- Si no están en registros entonces se registra qué unidades funcionales los deben producir para dar seguimiento, pero la instrucción pasa de todos modos a ejecución (donde se atorará hasta que sus operandos estén disponibles).
- Si hay operando destino que coincida con un fuente o destino de instrucción activa previa, entonces se hace renombrado de registros. Esto elimina hazards WAR y WAW.

Execute En cada estación de reservación y por cada instrucción.

- Si todos los operandos están disponibles, se copian sus valores de donde están a la entrada de la tabla de instrucciones de la estación de reservación. El equivalente, digamos al registro de instrucción.
- Si no están disponibles aún se "escucha" el CDB para saber cuando son producidos y cuál es el valor, mismo que se escribe en el registro de las estaciones que lo esperan. Esto es equivalente a hacer forwarding, con lo que los hazards RAW se resuelven tan pronto como se puede.
- Si una instrucción es posterior a un salto que la determina entonces permanece atorada aunque tenga sus operandos a menos que se use especulación.
- Si más de una instrucción ya posee todos sus operandos listos pueden ejecutarse simultáneamente, si sus unidades funcionales son diferentes o en orden arbitrario si están asociadas a la misma unidad.
- Los load y los store requieren trabajo en dos fases: se calcula la dirección de acceso y se guarda en un buffer de load o store según el caso, para ello se espera sólo por los registros necesarios para calcular la dirección. En una segunda fase

las direcciones son tomadas por la unidad de memoria Si la instrucción es un store, probablemente se deba esperar antes también por el registro a escribir en la memoria. Los load y store se mantienen en el orden de la cola de fetch para evitar hazards.

Write result Los resultados se escriben simultáneamente en el CDB y en los registros, los store en memoria se almacenan en un buffer hasta que se haya calculado la dirección y haya unidad de acceso a memoria disponible. Tanto los load como los store son ejecutados en el orden preciso que corresponde.

Igual que en scoreboarding, cada instrucción debe recordar (en este caso a través de su estación de reservación) qué unidad funcional producirá sus operandos. De hecho cada estación de reservación debe tener una tabla dividida en 7 campos:

- **Operación**. La operación que debe realizar.
- **RS1** y **RS2**. Las estaciones de reservación que deben producir los operandos origen de la instrucción. Cero si el operando ya está listo en el campo **Val**X respectivo.
- **Val1** y **Val2**. El valor de los operandos origen.
- **Addr**. Para almacenar la dirección de memoria que se debe acceder en un load o store, temporalmente se usa para guardar el desplazamiento de la dirección.
- **Ocupada**. Inidca sí la unidad funcional asociada con la estación está actualmente ocupada.

Adicionalmente, igual que en scoreboarding, se poseen: una tabla con el estado de los registros. y otra con el estado de cada instrucción. En la primera, para cada registro se indica cuál estacion de reservación producirá su contenido futuro inmediato. En la segunda, para cada instrucción en la cola de fetch, se dice en qué etapa de ejecución se encuentra.

Hay que notar que el algoritmo de Tomasulo hace uso de forwarding y renaming para resolver los hazards que sea posible antes de recurrir a atorar la ejecución de una instrucción.

Es importante señalar que en el algoritmo de Tomasulo se almacenan valores en las tablas, como si todo operando fuera inmediato; en scoreboarding en cambio se colocaba la “dirección” donde encontrarlo.

La estrategia de Tomasulo puede ejecutar las instrucciones en desorden para elevar el rendimiento, sin embargo se deben concluir las instrucciones siguiendo un cierto orden, el definido por el programa; no hacerlo así es impensable dado que se compromete la semántica del programa. A esto se le llama *retiro en orden* y para ello al final debe usarse lo que se denomina un *buffer de reordenamiento* (*Re-Order Buffer* o ROB, por sus siglas).

El ROB consiste esencialmente de una cola circular en la que se forman los resultados de las instrucciones completadas. En esta cola se define un espacio para cada instrucción desde la etapa de issue, por lo que en el ROB están en el orden del programa. Cuando se completa una nueva instrucción sus resultados se escriben en su entrada del ROB y sólo si las entradas asociadas a todas las instrucciones previas formadas ya están listas también, se procede a retirar la instrucción escribiendo efectivamente sus resultados. Este esquema se denomina *retiro* o *terminación* en orden (*In Order Completion*). En este buffer de reordenamiento es donde se mantiene el mapeo del renombrado de registros que se hizo al inicio para así poder hacer garantizar la consistencia con el programa original.

7.16 VLIW

Las siglas corresponde a *Very Large Instruction Word*. Una máquina con una arquitectura de conjunto de instrucciones VLIW recibe instrucciones en un formato largo, de cientos de bits, pero que se encuentran divididos en átomos más pequeños que son ejecutables en paralelo. Por ejemplo en la desaparecida arquitectura VLIW de Transmeta, la palabra de instrucción (llamada molécula) era de 128 bits y estaba constituida por 4 átomos de 32 bits cada uno: el primero era una instrucción para una unidad de punto flotante, el segundo para una de aritmética entera, el tercero para una de load/store y el último para una unidad de salto. Así cada molécula indica explícitamente que instrucciones pueden y deben ser ejecutadas simultáneamente e indica también que el rendimiento de este procesador es de hasta 4 instrucciones por ciclo de reloj. Intel prefiere llamar EPIC (*Explicitly Parallel Instruction Computing*) a este tipo de arquitecturas.

Estas palabras de instrucción largas pueden, como era el caso de Transmeta, ser formadas por el procesador o, como en el caso del IA-64, ser construidas por el compilador. En general es mejor que sea éste último quien construya las palabras ya que puede reacomodar mejor el código y avitar hazadrs de todo tipo. Por supuesto algunas de es-

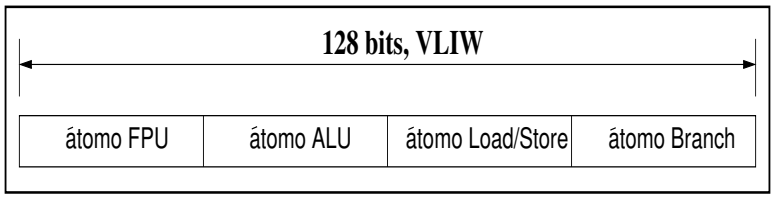

Figura 7.6. Ejemplo de molécula (palabra de instrucción larga) en la arquitectura de Transmeta.

tas palabras no contendrán todos sus átomos, cuando no sea posible encontrar como llenar los huecos. Algunos procesadores, con muchas unidades funcionales idénticas (por ejemplo, más de una ALU o más de una unidad de load/store) tienen varios formatos posibles para optimizar el paralelismo a nivel de instrucción.