UT 2. Computadores segmentados
Tema 2.4 Gestión dinámica de instrucciones y especulación

J. Flich, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Departamento de Informática de Sistemas y Computadores
Universitat Politècnica de València

UNIVERSIDAD POLITECNICA DE VALENCIA

DISCA

Índice

Bibliografía

John L. Hennessy and David A. Patterson.
Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.

Índice

Principios básicos

El *hardware* aumenta el ILP reordenando las instrucciones en tiempo de ejecución:

- Las instrucciones independientes se ejecutan simultáneamente en la unidad segmentada.
- Las instrucciones dependientes se ejecutan secuencialmente.

Hasta ahora, si una instrucción `i` se queda parada, ninguna instrucción `j` posterior puede continuar, incluso si `j` es *independiente* de las que están en ejecución, y el *operador* que `j` necesita está *libre*.

Ejemplo:

```
  DIV.D F0,F2,F4     IF ID DIV DIV ... DIV WB
i ADD.D F10,F0,F8       IF ID  ID  ... ID  A1  A2 ...
j MUL.D F12,F8,F14         IF  IF  ... IF  ID  M1 ...
```

Principios básicos (cont.)

Idea clave

El *hardware* debe poder lanzar a ejecución instrucciones posteriores a la que se ha parado → se *altera* dinámicamente el orden de la ejecución, evitando que el hecho de parar una instrucción afecte a las que le siguen.

Ventajas e inconvenientes

Ventajas:

- Simplifica el diseño del compilador.
- Soluciona eficientemente dependencias desconocidas en tiempo de compilación.
  Por ejemplo, originadas entre instrucciones solo cuando cierta condición de salto se produce, que involucran una referencia a memoria (`S.D ...,20(R1)` y `L.D .., 30(R2)` presentan dependencia para R1=R2+10), ...
- Favorece la compatibilidad binaria entre procesadores con diferente organización (diferentes operadores, diferentes latencias, etc).

Inconvenientes:

- Complica el diseño del *hardware*.

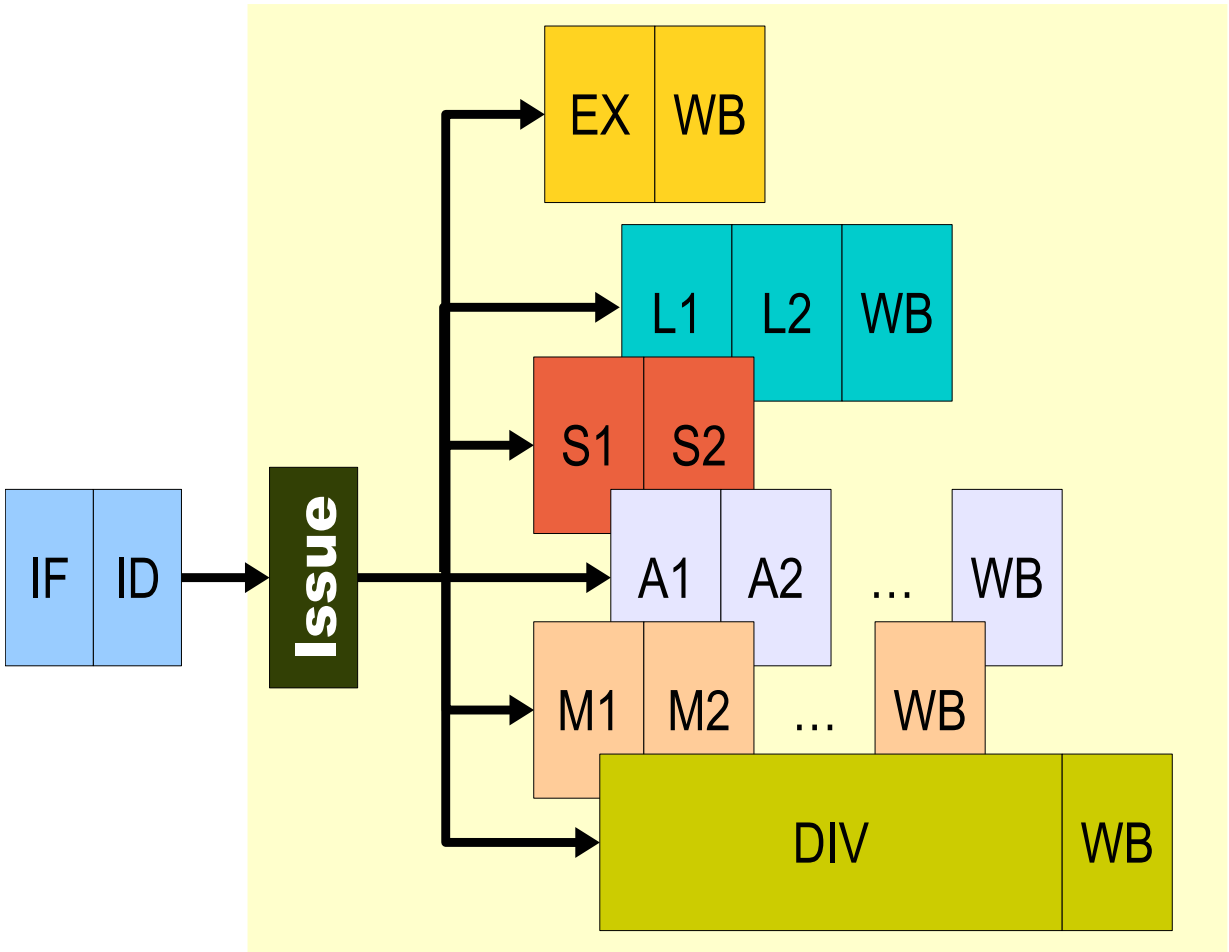
Índice

Objetivos

- Evitar ciclos de parada en ID $\rightarrow$ CPI $\approx$ 1.
- Ejecutando inmediatamente las instrucciones independientes.
- Detectando las instrucciones dependientes y gestionándolas correctamente.
- Permitiendo que las instrucciones independientes adelanten a instrucciones que estén en espera.

$\Rightarrow$ **Algoritmo de Tomasulo**
(desarrollado por Robert M. Tomasulo en 1967 para el IBM 360/91)

Modificación de la unidad de instrucción segmentada



Comentarios

1. Etapa *Issue*. Cuando la etapa ID decodifica una instrucción, se la pasa a la etapa *Issue*.
   - Si el operador implicado está disponible y la instrucción tiene todos sus operandos disponibles, la lanza a ejecución.
   - Si el operador implicado no está disponible, la instrucción espera.
   - Si algún operando no está disponible (por que hay una dependencia de datos con otra instrucción en ejecución), la instrucción debe esperar.

- ¿Dónde se esperan las instrucciones dependientes?
- ¿Cómo continúa la ejecución cuando desaparece la dependencia?

Comentarios (cont.)

2. Para mayor generalidad, se añade una unidad especializada en las operaciones de carga/almacenamiento.
   - El tiempo de acceso a la caché puede ser mayor que un ciclo.
   - Los fallos de caché afectan sólo a esta unidad, sin parar todas las instrucciones posteriores. Además, cuando se produce un fallo de caché → mayor tiempo de acceso.
   - Detección de dependencias en instrucciones de acceso a memoria. Ejemplo: La secuencia

     ```
     S.D F2, 30(R2)
     ...
     L.D F0, 20(R1)
     ```

     presenta una dependencia de datos $\forall$ R1=R2+10

¿Dónde se esperan las instrucciones dependientes?

- En la propia etapa *Issue* → se detiene la decodificación de instrucciones.

```
DIV.D F0,F2,F4     IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8       IF ID I   I   ... I   A1  A2 ...
MUL.D F12,F8,F14         IF ID  ID  ... ID  I   M1 ...
```

  → NO es gestión dinámica de instrucciones.

- En el operador correspondiente.

```
DIV.D F0,F2,F4     IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8       IF ID I   A1  ... A1  A1  A2 ...
MUL.D F12,F8,F14         IF ID  I   M1  ...
```

  → OK

¿Dónde se esperan las instrucciones dependientes? (cont.)

- ¿Y si la siguiente instrucción demanda el *mismo* operador?

```
DIV.D F0,F2,F4     IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8       IF ID I   A1  ... A1  A1  A2 ...
ADD.D F12,F8,F14         IF ID  I   ... I   I   A1 ...
```

  → NO es gestión dinámica de instrucciones.
  → `ADD.D F10,F0,F8` está ocupando un operador sin utilizarlo.

- En una estructura de datos asociada al operador correspondiente (**Estación de reserva**):

```
DIV.D F0,F2,F4   IF ID I  DIV DIV ... DIV WB
ADD.D F10,F0,F8     IF ID I   a1  ... a1  A1  A2 ...
ADD.D F12,F8,F14       IF ID  I   A1  ...
```

  → OK
  Gracias a las estaciones de reserva, un operador ofrece varios *operadores virtuales*.

¿Cómo continúa la ejecución cuando desaparece la dependencia?

→ Generando un grafo de dependencias de las instrucciones:

- Etapa *Issue*: añade nuevas dependencias al grafo.
- Etapa *Writeback*: resuelve dependencias que son eliminadas del grafo.

Índice

Consideraciones

- La gestión dinámica implica mantener, durante la ejecución de las instrucciones, una representación de las dependencias de datos que aún no se han resuelto.
- Estas dependencias ligan los operadores virtuales entre ellos y los registros del banco.
- Cada vez que una instrucción pasa por *Issue*, se añaden nuevas dependencias al grafo.
- Cada vez que una instrucción pasa por *Writeback* se difunde el resultado de la instrucción, resolviendo dependencias que son eliminadas del grafo.

Un ejemplo

Diagrama de instrucciones con gestión dinámica

	1	2	3	4	5	6	7	8	9	10	
`ADD.D F4,F0,F2`	IF	ID	I	A1	A2	A3	WB				
`MUL.D F6,F4,F0`		IF	ID	I	→	→	→	M1	M2	M3	...
`L.D F4,x`			IF	ID	I	L1	L2	WB			
`MUL.D F6,F6,F4`				IF	ID	I	→	→	→	→	...

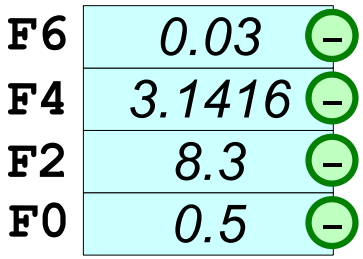
- El símbolo → expresa la espera que no bloquea la decodificación ni el lanzamiento de las instrucciones.
- La espera → se realiza en un operador virtual.
- Cuando una instrucción llega a la etapa WB, las instrucciones que esperaban su resultado continúan su ejecución.

Ejemplo de gestión dinámica de instrucciones– Estado inicial

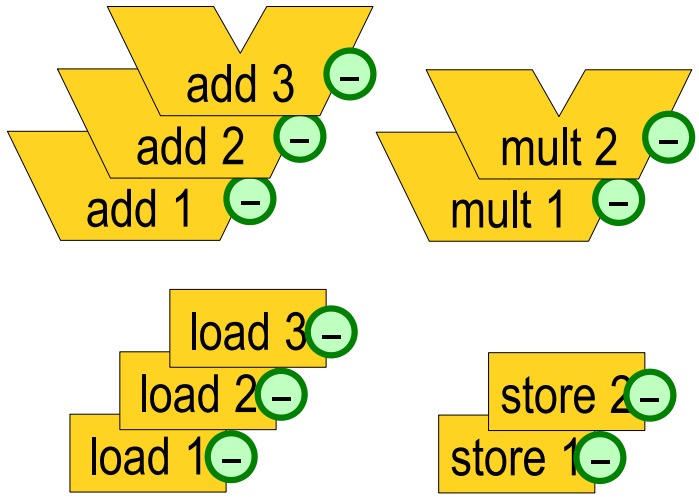
Disponemos de...

- un conjunto de instrucciones identificadas

```
① ADD.D F4,F0,F2
② MUL.D F6,F4,F0
③ L.D F4,x
④ MUL.D F6,F6,F4
```



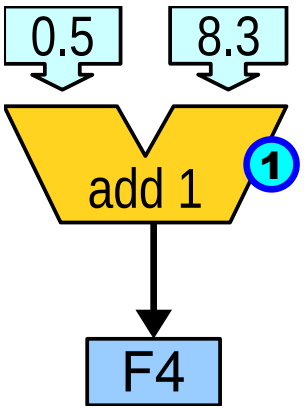
- un sistema de marcas (①, ②, ③ y ④) que permite asociar operadores y registros a las instrucciones y liberarlos (–)
- un conjunto de registros libres con su valor inicial
- un conjunto de operadores (virtuales) libres disponibles



Ejemplo de gestión dinámica de instrucciones– Ciclo 3

		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F0,F2`	IF	ID	I							
②	`MUL.D F6,F4,F0`		IF	ID							
③	`L.D F4,x`			IF							
④	`MUL.D F6,F6,F4`										

Grafo:



Banco:

F6	0.03	–
F4	3.1416	1
F2	8.3	–
F0	0.5	–

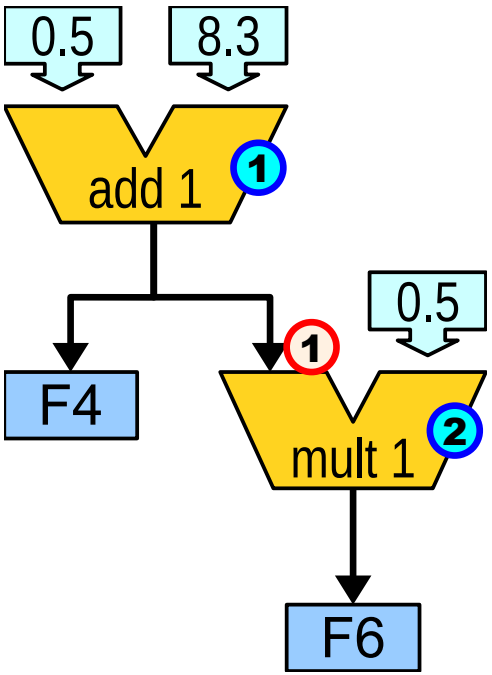
Marcas

- El lanzamiento de la instrucción ① ocupa un operador.
- Los registros fuente F2 y F0 están libres (–) y su contenido se transfiere al operador.
- El registro destino F4 queda ocupado. Su contenido está pendiente de ser actualizado.
- Nótese que el operador físico está libre, luego podrá comenzar.

Ejemplo de gestión dinámica de instrucciones– Ciclo 4

		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F0,F2`	IF	ID	I	A1						
②	`MUL.D F6,F4,F0`		IF	ID	I						
③	`L.D F4,x`			IF	ID						
④	`MUL.D F6,F6,F4`				IF						

Grafo:



Banco:

F6	*0.03*	2
F4	*3.1416*	1
F2	*8.3*	–
F0	*0.5*	–

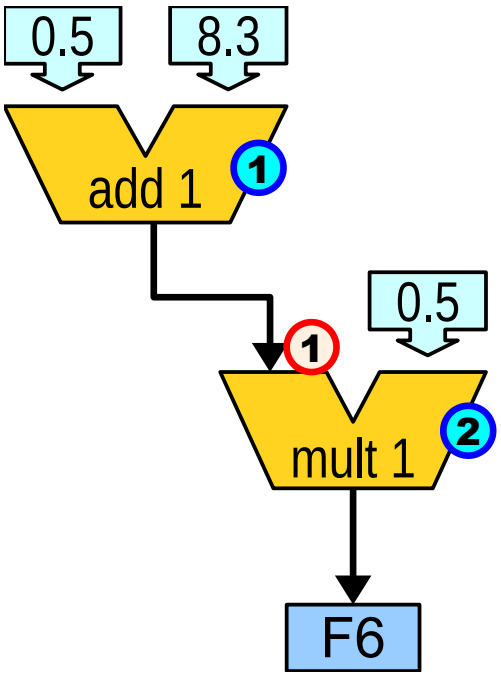
Detección de la dependencia

- Al lanzarse la instrucción ② se selecciona un nuevo operador, el registro fuente F0 está libre y se puede usar su contenido...
- ... pero la marca ① en F4 indica la dependencia entre ① y ②.
- El operador virtual asociado a ② queda a la espera de que se difunda el valor que falta.

Ejemplo de gestión dinámica de instrucciones– Ciclo 5

		1	2	3	4	5	6	7	8	9	10
1	`ADD.D F4,F0,F2`	IF	ID	I	A1	A2					
2	`MUL.D F6,F4,F0`		IF	ID	I	→					
3	`L.D F4,x`			IF	ID	I					
4	`MUL.D F6,F6,F4`				IF	ID					

Grafo:



Banco:

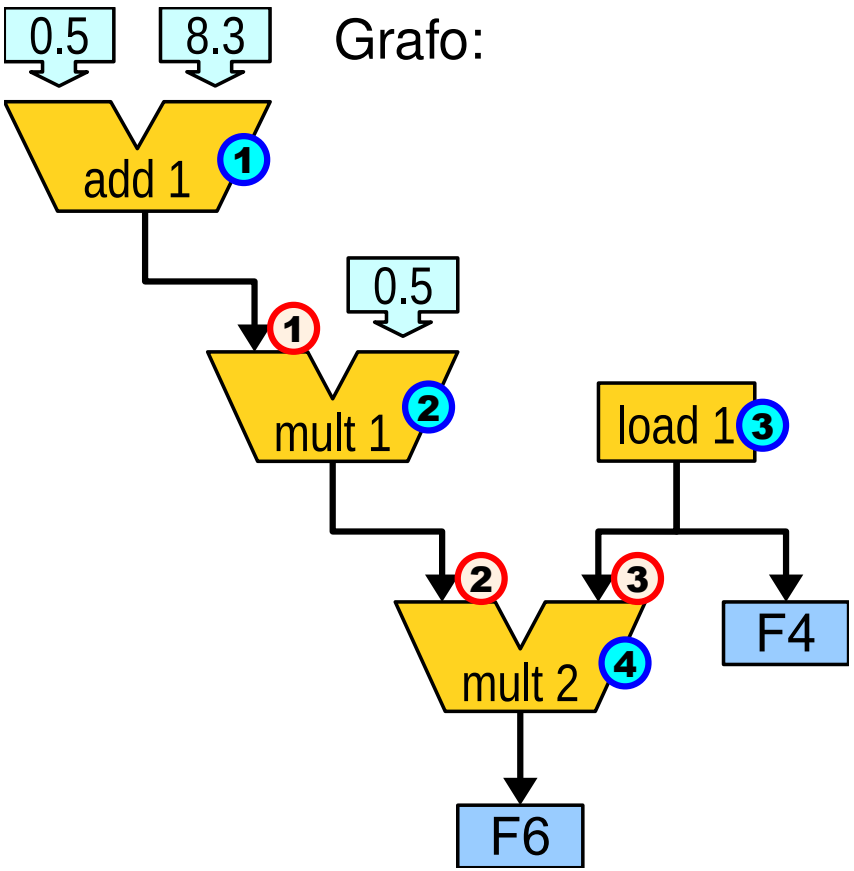
F6	*0.03*	2
F4	*3.1416*	3
F2	*8.3*	–
F0	*0.5*	–

Actualización de las marcas

- Al lanzar 3, la marca de F4 cambia a 3.
- Eso no afecta a la dependencia entre 1 y 2.
- Ahora tenemos dos grafos disjuntos

Ejemplo de gestión dinámica de instrucciones– Ciclo 6

		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F0,F2`	IF	ID	I	A1	A2	A3				
②	`MUL.D F6,F4,F0`		IF	ID	I	→	→				
③	`L.D F4,x`			IF	ID	I	L1				
④	`MUL.D F6,F6,F4`				IF	ID	I				



Banco:

F6	*0.03*	④
F4	*3.1416*	③
F2	*8.3*	–
F0	*0.5*	–

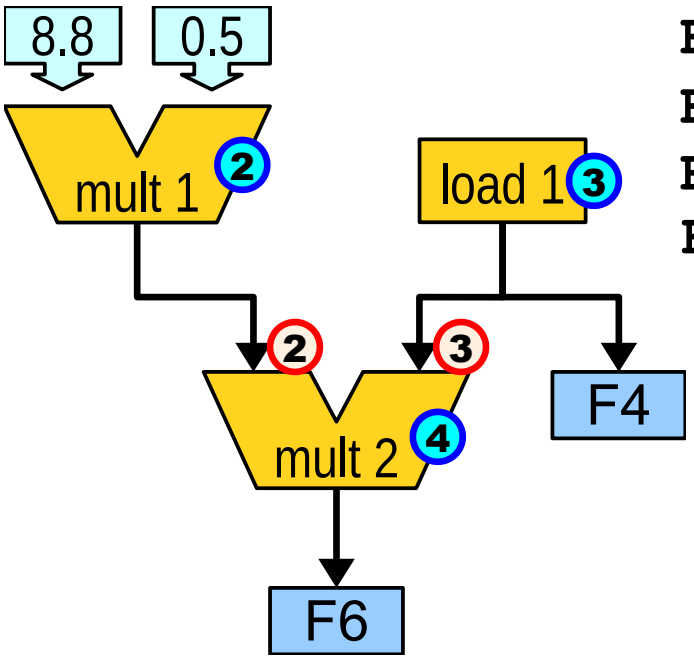
Actualización de las marcas

- Al lanzar ④, la marca ③ de F4 conecta correctamente el nuevo operador en el grafo.

Ejemplo de gestión dinámica de instrucciones– Ciclo 7

		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F0,F2`	IF	ID	I	A1	A2	A3	**WB**			
②	`MUL.D F6,F4,F0`		IF	ID	I	→	→	→			
③	`L.D F4,x`			IF	ID	I	L1	L2			
④	`MUL.D F6,F6,F4`				IF	ID	I	→			

Grafo:



Banco:

F6	*0.03*	4
F4	*3.1416*	3
F2	*8.3*	–
F0	*0.5*	–

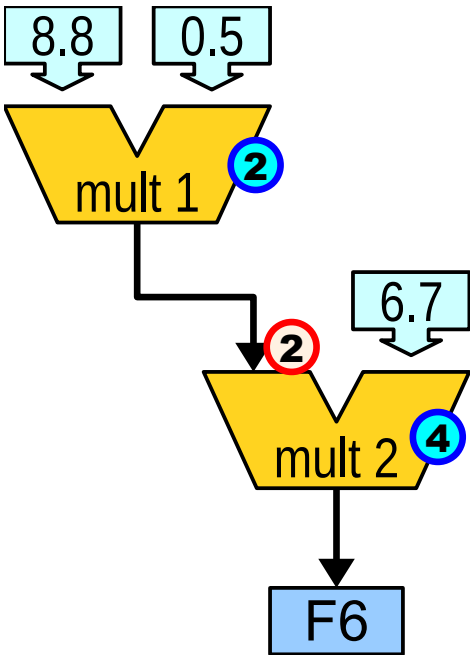
Transmisión de resultados

- Cuando ① llega a la etapa WB, se difunde el resultado a los operadores dependientes y el operador virtual `add 1` queda libre.
- Ahora las instrucciones que esperaban tienen la oportunidad de comenzar su operación.

Ejemplo de gestión dinámica de instrucciones– Ciclo 8

		1	2	3	4	5	6	7	8	9	10
①	`ADD.D F4,F0,F2`	IF	ID	I	A1	A2	A3	WB			
②	`MUL.D F6,F4,F0`		IF	ID	I	→	→	→	M1		
③	`L.D F4,x`			IF	ID	I	L1	L2	**WB**		
④	`MUL.D F6,F6,F4`				IF	ID	I	→	→		

Grafo:



Banco:

F6	*0.03*	4
F4	*6.7*	–
F2	*8.3*	–
F0	*0.5*	–

Escritura de registros

- Al terminar ③, se actualiza y libera F4 porque su marca ③ coincide.
- Además, se transmite el resultado a ④.

Resolución de los riesgos

Estructurales: Se resuelven con los operadores virtuales, iniciando la ejecución cuando el operador físico esté libre.

Sin gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8
DIV.D F6,F4,F2	IF	ID	D1	D2	D3	D4	D5	WB
DIV.D F12,F10,F8		IF	ID	*ID*	*ID*	*ID*	*ID*	D1
L.D F14,x			IF	*IF*	*IF*	*IF*	*IF*	ID

Con gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8	9
DIV.D F6,F4,F2	IF	ID	I	D1	D2	D3	D4	D5	WB
DIV.D F12,F10,F8		IF	ID	I	–	–	–	–	D1
L.D F14,x			IF	ID	I	L1	L2	WB	

Resolución de los riesgos (cont.)

RAW: Se resuelven encadenando los operadores implicados.

Sin gestión dinámica de instrucciones:

	1	2	3	4	5	6	7
ADD.D **F4**,F0,F2	IF	ID	A1	A2	A3	WB	
MUL.D F6,**F4**,F0		IF	ID	*ID*	*ID*	M1	M2
L.D F4,x			IF	*IF*	*IF*	ID	L1

Con gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8	9
ADD.D **F4**,F0,F2	IF	ID	I	**A1**	**A2**	**A3**	WB		
MUL.D F6,**F4**,F0		IF	ID	I	–	–	**–**	**M1**	**M2**
L.D F4,x			IF	ID	I	L1	L2	WB	

Resolución de los riesgos (cont.)

WAW: Se resuelven haciendo que la última instrucción lanzada sea la que escribe de un modo efectivo en el registro implicado.

Sin gestión dinámica de instrucciones:

```
                  1  2  3  4  5  6  7  8  9
MUL.D F2,F4,F0    IF ID M1 M2 M3 M4 M5 WB
ADD.D F2,F6,F0       IF ID ID ID A1 A2 A3 WB
DIV.D F4,F8,F10         IF IF IF ID D1 D2 D3
```

Con gestión dinámica de instrucciones:

```
                  1  2  3  4  5  6  7  8  9
MUL.D F2,F4,F0    IF ID I  M1 M2 M3 M4 M5 WB
ADD.D F2,F6,F0       IF ID I  A1 A2 A3 WB
DIV.D F4,F8,F10         IF ID I  D1 D2 D3 D4
```

El grafo dinámico de dependencias se construye asegurando que solo la última instrucción escribe en F2.

Resolución de los riesgos (cont.)

WAR: Se evitan construyendo el grafo en la etapa *Issue*:

Ejemplo:

```
                    1   2   3   4   5   6   7   8   9   10
MUL.D F2,F8,F0      IF  ID  I   M1  M2  M3  M4  M5  WB
MUL.D F6,F4,F2          IF  ID  I   -   -   -   -   -   M1
L.D F4,x                    IF  ID  I   L1  L2  WB
```

- Las intrucciones pasan por *Issue* en orden
- Se leen los operandos que ya estén disponibles en ese momento (F4 en el ejemplo),
- ... incluso cuando otros operandos tengan dependencia de datos con otras instrucciones previas (F2 en el ejemplo).

Índice

Motivación

Técnicas de **predicción de saltos**:

- Pretenden conocer cuanto antes la dirección efectiva del salto, comenzando a buscar instrucciones en esa dirección.
- Tras ejecutar la instrucción de salto, si no se cumple la predicción, se abortan las instrucciones *lanzadas*.

Problema:

La **condición de salto tarda mucho tiempo** en conocerse → instrucciones posteriores al salto lanzadas a ejecución ya han finalizado y **no se pueden cancelar**.

Ejemplo:

```
loop: ...
      DIV.D F2,F0,F4 ; Operación larga
      C.GE.D F2,F12  ; ¿F2>=F12?
                     ; Resultado en FP status (FPSR) reg.
                     ; Debe esperar a que termine DIV.D
      BC1T  loop     ; Si ((FPSR)=true) ... loop
                     ; Debe esperar a que termine C.GE.D
      ...
Posible ejecución:
DIV.D F2,F0,F4       IF ID I  D1 D2 D3 D4 WB
C.GE.D F2,F12           IF ID I              A1 A2 WB
BC1T loop                  IF ID I                    EX
ADD.D F6,F4,F4                IF ID I  A1 A2 WB
```

→ cuando `BC1T loop` resuelve la condición, `ADD.D F6,F4,F4` ya ha terminado.

Especulación

Técnica que pretende, no solo lanzar a ejecución, sino ejecutar **completamente** (hasta WB inclusive), si procede, las instrucciones dependientes de un salto antes de conocer su comportamiento:

- No esperar a que termine la instrucción de salto.
- Predecir el comportamiento de la instrucción de salto.
- Ejecutar las instrucciones atendiendo a la predicción → ejecución "especulativa".
- Si finalmente la predicción no es correcta, las acciones realizadas *no deben tener efectos* sobre los resultados de la ejecución del programa.
- Pero, si se confirma la predicción, las acciones realizadas *deben consolidarse.*

Idea clave:

Las instrucciones ejecutadas especulativamente no deben alterar el resultado de la ejecución del programa → el resultado debe ser el mismo que el obtenido sin especulación:

- Se deben respetar las dependencias de datos entre las instrucciones.
- **Hasta que no se confirme la predicción del salto** (hasta que las instrucciones "dejen de ser especulativas"):
  - No se modifican los registros ni la memoria.
    Por tanto, el resultado de las instrucciones ejecutadas especulativamente se almacena *temporalmente*.
  - Las instrucciones ejecutadas especulativamente no deben generar *excepciones*.

Índice

Idea básica

- En tiempo de ejecución se realiza la predicción de la instrucción de salto, buscando y ejecutando "provisionalmente" las instrucciones correspondientes.
- . . . pero no se modifica el estado de la máquina (escritura sobre registros o posiciones de memoria y generación de excepciones) **hasta que se confirma** la predicción efectuada.

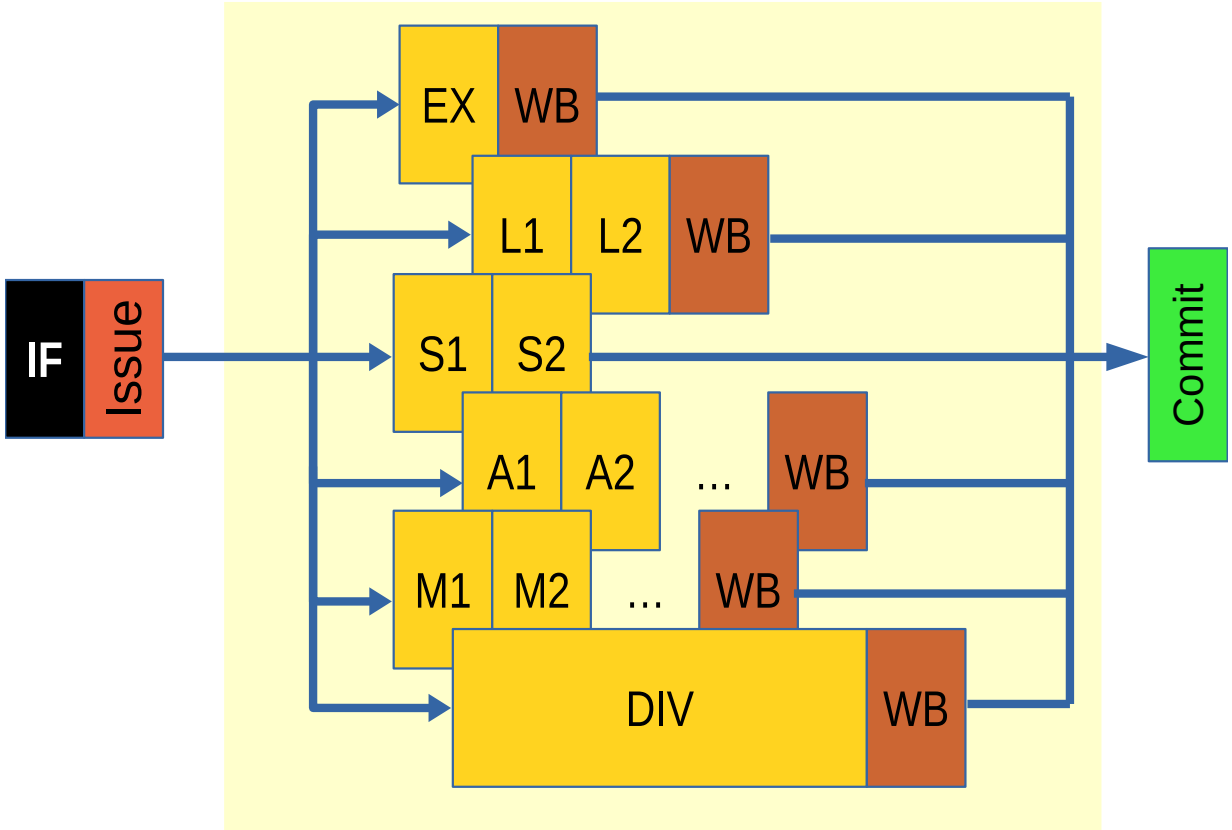
Realización:

1. Predicción dinámica de saltos para seleccionar qué instrucciones hay que ejecutar.
2. Búsqueda de las instrucciones en orden (*IF*).
3. Decodificación y lanzamiento de las instrucciones en orden (*ID+Issue* $\rightarrow$ *I*).
4. Ejecución de las instrucciones fuera de orden (EX, WB).
5. **Finalización** de las instrucciones **en orden** $\rightarrow$ *Commit*.

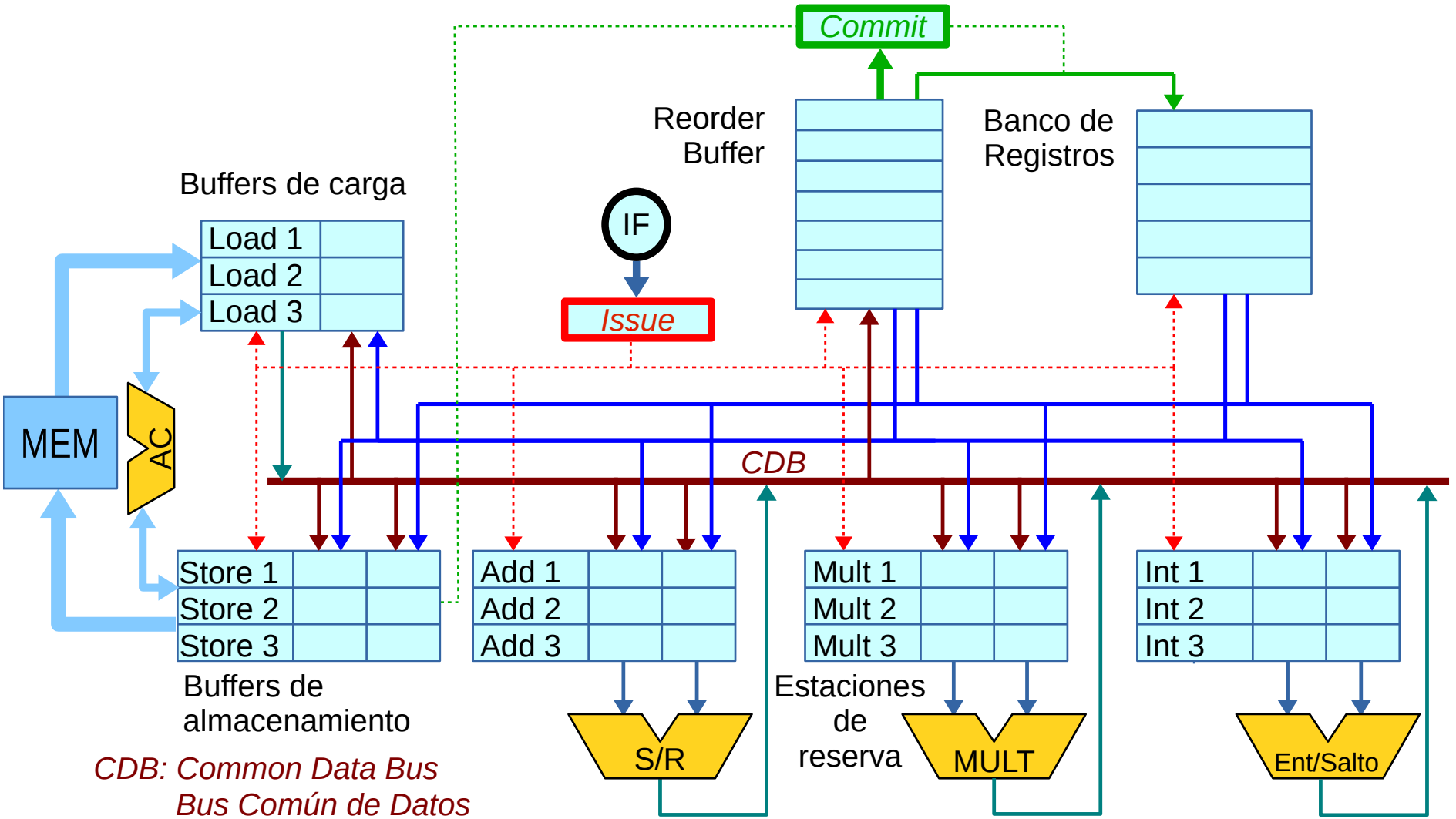
⇒ Etapa adicional para finalizar las instrucciones: **Etapa *Commit***

- Las instrucciones pasan en orden por esta etapa.
- En esta etapa se actualizan los registros o la memoria y se reconocen las excepciones.
- Si una instrucción llega a *Commit*, es la más antigua del procesador:
  → todos los saltos previos están resueltos y ninguna instrucción previa ha producido excepciones.
  → si es un salto incorrectamente predicho, como ninguna instrucción posterior ha modificado el estado de la máquina, se pueden cancelar.
- A la etapa *Commit* sólo llegan instrucciones especuladas correctamente.

Ruta de datos



Ruta de datos



Elementos de la ruta de datos

Registros de uso general

Estaciones de reserva Cada operador tiene asociado una estructura de datos de varias entradas ("estaciones de reserva"). Cada entrada contiene una instrucción en espera o en ejecución.
Un operador ofrece, por tanto, varios "operadores virtuales".

***Buffers* de carga y almacenamiento** Contienen los datos proporcionados por la memoria/a escribir en la memoria. Los gestionan las unidades de carga y almacenamiento.

Bus común de datos alimentado por todas las unidades capaces de generar resultados, y conectado a todos los elementos capaces de leer datos.
El acceso a este bus debe arbitrarse cuando hay varias unidades intentando transferir datos al mismo tiempo.

Reorder Buffer

Transferencias a través del bus común de datos

Se realiza en dos fases:

- Fase de preparación.
  Se conoce el origen y el destino de la transferencia, pero todavía no se tiene el dato a transferir.
  → Etapa *Issue*.
  Ejemplos:
  - `add.d f4,f1,f0`.
    Se sabe que la instrucción `add.d` escribirá en el registro f4.
  - `add.d f4,f1,f0` seguida de `mul.d f12,f4,f10`
    Se sabe que la instrucción `mul.d` obtendrá su primer operando del resultado de `add.d`.
- Fase de transferencia. El dato ya se ha obtenido y realiza su transferencia.
  → Etapa *WB*

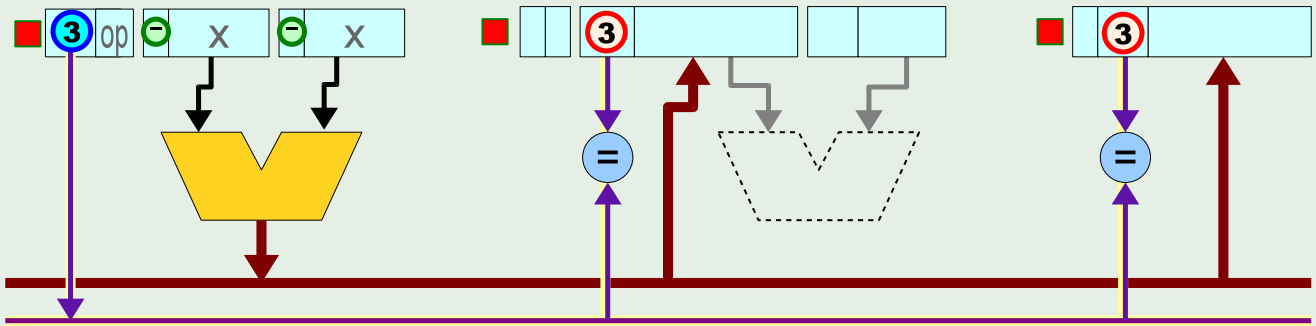
Transferencias a través del bus común de datos (cont.)

- Los elementos que generan un resultado en el bus difunden el resultado junto con un "código" que identifica a la instrucción que produce el resultado:
  1. Operadores virtuales.
  2. *Buffers* de carga.
- Los elementos que leen datos del bus tiene asignada una "marca" que identifica a la instrucción de la que consumen el resultado.
  1. Estaciones de reserva.
  2. *Buffers* de almacenamiento.
- Difusión: Cuando un elemento vuelca un dato en el bus, escribe también su código. Todos los elementos cuyo campo de marca coincida con el código presente en el bus leen el dato.

Transferencias a través del bus común de datos (cont.)

Ejemplo: Transferencia de un dato por el CDB

1. Preparación: Escribir el código del emisor (3) en el campo de marca de los receptores.
2. Transferencia:
   1. Volcar al bus el dato y el código (3).
   2. En cada estación de reserva, si el campo de marca coincide con el código presente en el bus, se lee el dato.



Reorder buffer (ROB)

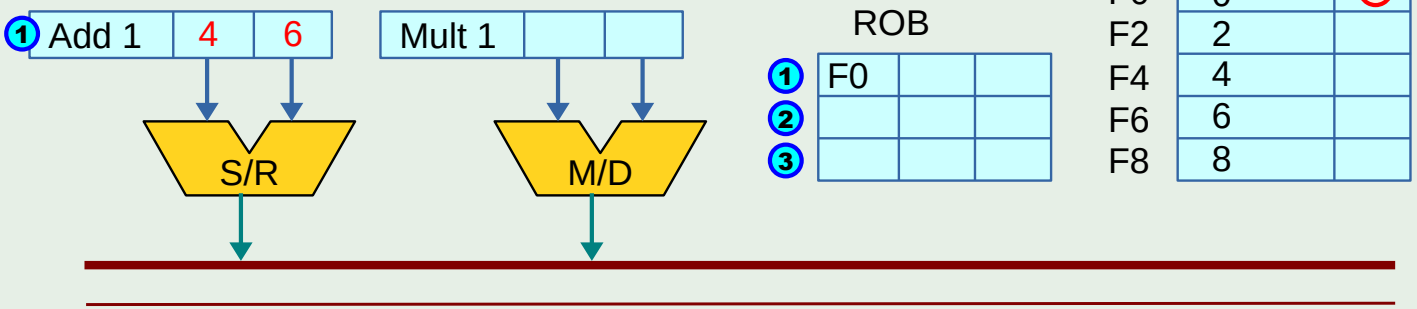
- Mantenimiento del orden de las instrucciones.
  En la etapa *Issue*, se reserva una entrada en el ROB. El número de entrada se utiliza como marca en las estaciones de reserva y el banco de registros.
- Almacenamiento temporal.
  En la etapa *WB*, se difunde el resultado a las estaciones de reserva pero no se escribe en el registro destino, sino en la *entrada correspondiente del ROB*.
  → una instrucción dependiente deberá obtener sus operandos del ROB, y no de los registros.
- Excepciones.
  Si una instrucción origina una excepción, se anota tal evento en la entrada correspondiente del ROB.

Reorder buffer (ROB) (cont.)

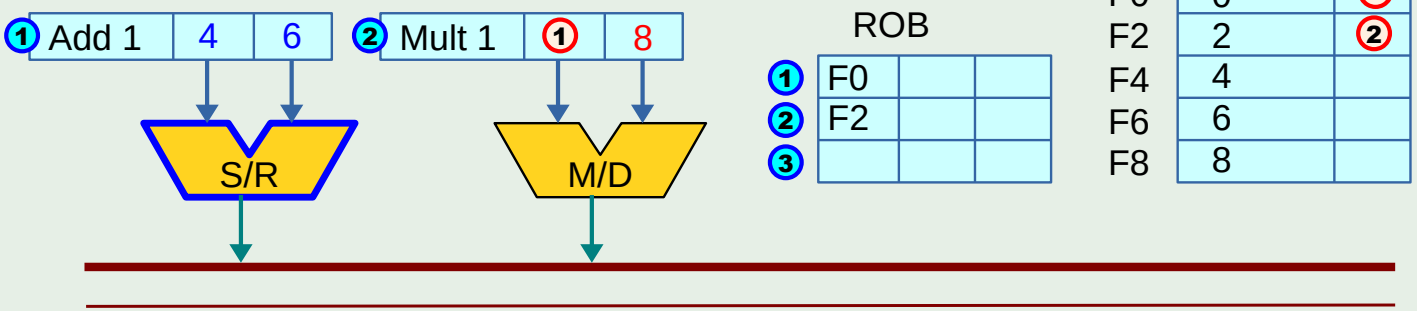
- Cuando la instrucción más antigua del ROB finaliza (etapa *Commit*):
  - Se comprueba si ha producido alguna excepción, lanzando la rutina de servicio, en su caso.
  - Se escribe su resultado desde el ROB al registro destino o se realiza la escritura en la posición de memoria correspondiente.
  - Se libera la entrada del ROB.
- Si un salto es incorrectamente predicho, cuando llega a la etapa *Commit*, borra el contenido del ROB.
  → las instrucciones lanzadas a ejecución especulativamente (y ahora se sabe que incorrectamente) después de la de salto:
  - no escriben sobre el registro destino (no se terminan).
  - no originan excepciones.

Especulación *hardware* a través de un ejemplo:

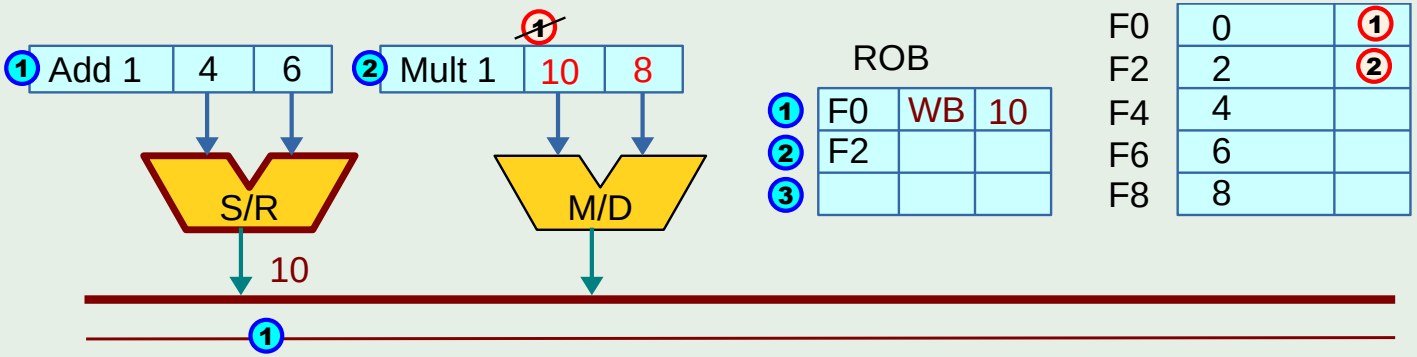
```
ADD.D F0,F4,F6   IF I
```



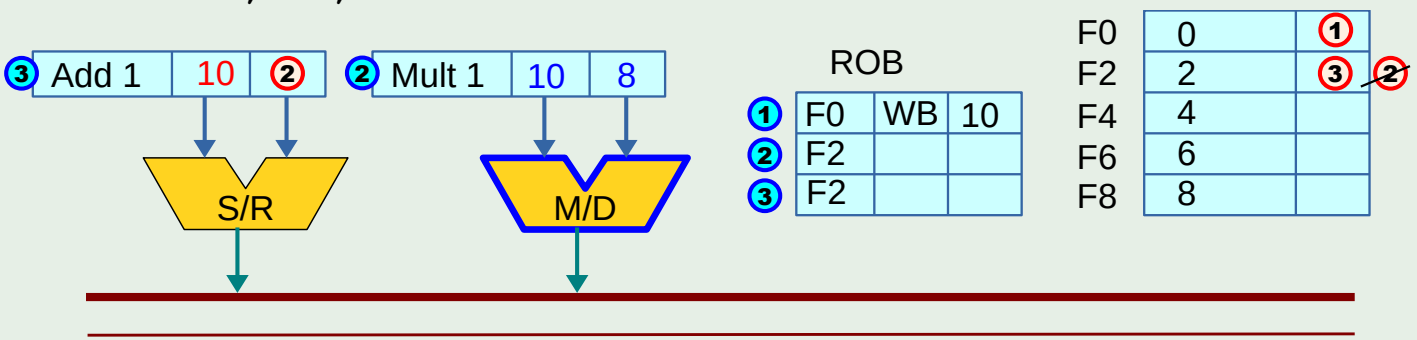

```
ADD.D F0,F4,F6   IF I   A1
MUL.D F2,F0,F8      IF I
```



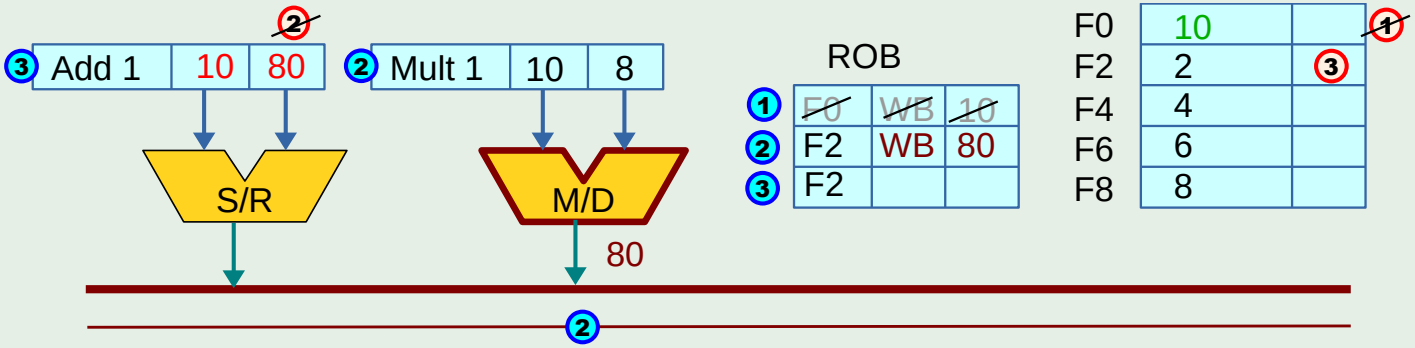
```
ADD.D F0,F4,F6   IF I  A1 A2 A3 WB
MUL.D F2,F0,F8      IF I  -  -  -
```



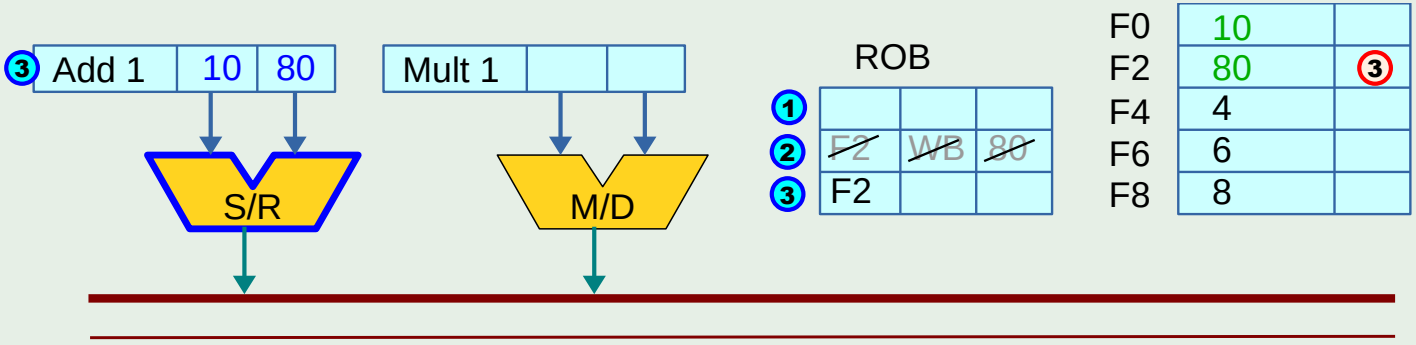

```
ADD.D F0,F4,F6   IF I  A1 A2 A3 WB -
MUL.D F2,F0,F8      IF I  -  -  -  M1
...
ADD.D F2,F0,F2                  IF I
```



```
ADD.D F0,F4,F6   IF I  A1 A2 A3 WB -  -  -  -  -  C
MUL.D F2,F0,F8      IF I  -  -  -  M1 M2 M3 M4 M5 WB
...
ADD.D F2,F0,F2                  IF I  -  -  -  -  -
```




```
MUL.D F2,F0,F8   IF I  -  -  -  M1 M2 M3 M4 M5 WB C
...
ADD.D F2,F0,F2                  IF I  -  -  -  -  -  M1
```



Algoritmo de Tomasulo con especulación: estructuras de datos

Registros (`Regs`):

- Valor (`value`)
- Entrada en el ROB (`rob`)

Operadores, de coma flotante, de enteros y saltos, con varias estaciones de reserva (`RS`). Cada estación de reserva:

- Bit de ocupado (`busy`)
- Operación a realizar (`op`)
- Operando 1 - Valor (`V1`)
- Operando 1 - Marca (`Q1`)
- Operando 2 - Valor (`V2`)
- Operando 2 - Marca (`Q2`)
- Entrada en el ROB (`rob`)
- Resultado (`result`)

Algoritmo de Tomasulo con especulación: estructuras de datos (cont.)

Buffers de carga (`LB`):

- Bit de ocupado (`busy`)
- Dirección (con modo de direccionamiento desplazamiento):
  - Registro base - Valor (`V1`)
  - Registro base - Marca (`Q1`)
  - Desplazamiento (`disp`)
  - Dirección calculada (`addr`)
- Entrada en el ROB (`rob`)
- Resultado (`result`)

Algoritmo de Tomasulo con especulación: estructuras de datos (cont.)

Buffers de almacenamiento (`SB`):

- Bit de ocupado (`busy`)
- Operando - Valor (`V2`)
- Operando - Marca (`Q2`)
- Dirección (con modo de direccionamiento desplazamiento):
  - Registro base - Valor (`V1`)
  - Registro base - Marca (`Q1`)
  - Desplazamiento (`disp`)
  - Dirección calculada (`addr`)
- Bit de confirmación (`conf`)

Algoritmo de Tomasulo con especulación: estructuras de datos (cont.)

Reorder buffer (`ROB`):

- Bit de ocupado (`busy`)
- PC de la instrucción (`PC`).
- Instrucción (`instr`): Salto (sin resultado), almacenamiento (resultado en memoria) ó ALU/carga (resultado en registro).
- Destino (`dest`): Número de registro, posición en el *buffer* de almacenamiento o dirección destino (saltos).
- Valor (`value`): Valor a almacenar o condición (`cond`) (saltos). Identificador de excepción, en su caso.
- Completado (`completed`): Indica si la instrucción ya ha llegado a la etapa WB y ha transferido los datos por el bus común, en su caso.
- Predicción (`pred`): Predicción realizada (saltos).

Algoritmo de Tomasulo con especulación

La ejecución de instrucciones tiene lugar en cuatro etapas:

Issue

```
Decodificar instrucción (instr):
-ALU D,S1,S2
-LOAD D,desplazamiento(S1)
-STORE S2,desplazamiento(S1)
-SALTO S1,dirección,predicción

// ¿Hay un operador libre?
// ¿Hay una entrada en el ROB libre?
Si {s:estación de reserva o buffer de carga/almacen.} libre y
   {b:entrada en el ROB} libre,

  // Estación de reserva o buffer de carga/almacen.
  RS[s].busy ó LB[s].busy ó SB[s].busy := SÍ
  RS[s].op := {op:operación aritmética}
  RS[s].rob ó LB[s].rob := b // Entrada del ROB
```

Issue (cont.)

```
// Operando fuente 1, en su caso: ALU, LOAD, STORE, SALTO
Si (Regs[S1].rob = marca_nula)  // Leer valor
  RS[s].V1 ó LB[s].V1 ó SB[s].V1 := Regs[S1].value
  RS[s].Q1 ó LB[s].Q1 ó SB[s].Q1 := marca_nula
Sino
  Si ROB[Regs[S1].rob].completed  // Leer ROB
    RS[s].V1 ó LB[s].V1 ó SB[s].V1 := ROB[Regs[S1].rob].value
    RS[s].Q1 ó LB[s].Q1 ó SB[s].Q1 := marca_nula
  Sino // Anotar entrada del ROB
    RS[s].Q1 ó LB[s].Q1 ó SB[s].Q1 := Regs[S1].rob

// Operando fuente 2, en su caso: ALU o STORE
Si {instr es ALU o STORE}
  Si (Regs[S2].rob = marca_nula)  // Leer valor
    RS[s].V2 ó SB[s].V2 := Regs[S2].value
    RS[s].Q2 ó SB[s].Q2 := marca_nula
  Sino
    Si ROB[Regs[S2].rob].completed  // Leer ROB
      RS[s].V2 ó SB[s].V2 := ROB[Regs[S2].rob].value
      RS[s].Q2 ó SB[s].Q2 := marca_nula
    Sino // Anotar entrada del ROB
      RS[s].Q2 ó SB[s].Q2 := Regs[S2].rob
```

Issue (cont.)

```
// Desplazamiento, en su caso: LOAD y STORE
Si {instr es LOAD ó STORE}
 LB[s].disp ó SB[s].disp := desplazamiento

// ROB
ROB[b].busy := SÍ
ROB[b].completed := NO
ROB[b].instr := instr
ROB[b].PC := PC    // PC de la instruccion en Issue
Si {instr es ALU ó LOAD}
  ROB[b].dest := D
Si {instr es STORE}
  ROB[b].dest := s
Si {instr es SALTO}
  ROB[b].dest := dirección  // Por simplicidad, la calcula Issue
  ROB[b].pred := predicción // Lo que indique el predictor

// Reserva registro destino, en su caso: ALU y LOAD
Si {instr es ALU ó LOAD}
  Regs[D].rob := b      // Entrada en el ROB
```

EX (ALU)

```
Si {operador libre} y {hay estaciones de reserva con operandos listos}
  x := Seleccionar_una()
  Operación:
    RS[x].result := RS[x].V1 op RS[x].V2
  Si hay excepción, anotarlo
```

EX (SALTOS)

```
Si {operador libre} y {hay estaciones de reserva con operandos listos}
  x := Seleccionar_una()
  Operación:
    RS[x].result := RS[x].V1 op 0
```

AC, Address Calculation (LOAD y STORE)

```
Si {operador libre} y {hay buffers de carga o almacenamiento
                       con operando para calcular la dirección listo}
x := Seleccionar_una()
Calcular dirección:
  Si (LOAD) LB[x].addr := LB[x].V1 + LB[x].disp
  Si (STORE) SB[x].addr := SB[x].V1 + SB[x].disp
Si hay excepción, anotarlo
```

MEM (LOAD)

```
Si {operador libre} y {hay buffer de carga con dirección calculada
                       y desambiguada}
// Desambiguada: su dirección no coicide con la de STORES anteriores

x := Seleccionar_una()
Acceso a memoria:
  LB[x].result := Mem[LB[x].addr]
```

WB

```
Para {y: operador virtual (ALU, SALTOS) ó buffer de carga (LOAD)}
  Volcar {dato := RS[y].result ó LB[y].result} en el CDB
  Volcar {rb := RS[y].rob ó LB[y].rob} en el CDB
  // También se vuelca la existencia/ausencia de excepción
  // En caso de salto, el dato contiene la condición
  // Si hay excepción, también se almacena en el ROB

  // 1. Copiar al ROB
  ROB[rb].value := dato
  ROB[rb].completed := SÍ   // lista para Commit

  // 2. Transferir dato a instrucciones en espera
  Para {x: op. virtual}
    // Operando 1
    Si RS[x].Q1 = rb   // Marca==#rb
      RS[x].V1 := dato       // leer dato del bus
      RS[x].Q1 := marca_nula // borrar marca

    // Operando 2
    Si RS[x].Q2 = rb   // Marca==#rb
      RS[x].V2 := dato       // leer dato del bus
      RS[x].Q2 := marca_nula // borrar marca
```

WB (cont.)

```
Para {x: buffer de carga}
  // Operando 1
  Si LB[x].Q1 = rb   // Marca==#rb
    LB[x].V1 := dato        // leer dato del bus
    LB[x].Q1 := marca_nula // borrar marca

Para {x: buffer de almacenamiento}
  // Operando 1
  Si SB[x].Q1 = rb   // Marca==#rb
    SB[x].V1 := dato        // leer dato del bus
    SB[x].Q1 := marca_nula // borrar marca

  // Operando 2
  Si SB[x].Q2 = rb   // Marca==#rb
    SB[x].V2 := dato        // leer dato del bus
    SB[x].Q2 := marca_nula // borrar marca

// 3. Liberar estación de reserva o buffer de carga
RS[y].busy ó LB[y].busy := NO
```

Commit

```
Si {instrucción en la cabeza del ROB, entrada h ha terminado}
    // ``ha terminado'':
    // -LOAD, ALU, SALTOS: Han hecho WB
    // -STORE: Tiene la dirección calculada

    // Procesar excepción, en su caso ...
    // Si no hay excepción:

   Si (ROB[h].instr es SALTO) y (ROB[h].pred <> ROB[h].value)
    // Pred. incorrecta
    ROB[*].busy := NO // Borrar ROB
    RS,LB,SB[*].busy := NO // Borrar estaciones de reserva,
                           // excepto las escr. confirmadas
    Regs[*].rob := marca_nula // Liberar registros
    Si (ROB[h].value)    // Buscar instrucciones en el camino correcto
      PC := ROB[h].dest  // Salta
    Sino
      PC := ROB[h].PC+4 // No salta
```

Commit (cont.)

```
Si (ROB[h].instr es STORE)
  SB[ROB[h].dest].conf := SÍ // Confirmar escritura

Si (ROB[h].instr es ALU ó LOAD)
  Regs[ROB[h].dest].value := ROB[h].value // Actualizar registro
  Si (Regs[ROB[h].dest].rob = h) /* Ninguna otra instrucción
                          posterior escribe sobre este registro */
    Regs[ROB[h].dest].rob := marca_nula   // Liberar registro

// Liberar entrada en ROB
ROB[h].busy := NO
```

MEM (STORE)

```
Si {hay buffers de almacenamiento confirmados}
  x := Seleccionar_una()
  Acceso a memoria:
    Mem[SB[x].addr] := SB[x].V2

  // Liberar buffer
  SB[x].busy := NO
  SB[x].conf := NO
```

Comentarios:

- El ROB proporciona un espacio para almacenar el resultado de la instrucción → realiza un **renombrado dinámico** de registros.
- El número de entrada de la instrucción en el ROB permite encadenar aquellas instrucciones con dependencias de datos no resueltas.
- Las estaciones de reserva almacenan las instrucciones desde que se lanzan (*I*) hasta que terminan su ejecución (*WB*)
- Las estaciones de reserva monitorizan el bus común de datos en busca de operandos para instrucciones en espera.
- En la etapa WB, las estaciones de reserva escriben directamente en la entrada del ROB el resultado que han obtenido o la excepción producida, en su caso.
  El ROB no monitoriza el bus común (para un tamaño del ROB grande, se necesitarían demasiados comparadores).

Comentarios: (cont.)

- En la etapa *Commit*, el resultado almacenado en el ROB se copia en el registro destino, independientemente de si hay otra instrucción posterior que ha bloqueado el registro (esa instrucción posterior puede que no se ejecute finalmente).
- Sin embargo, en ese caso, el registro no se libera (ya que su campo `rob` actual se está utilizando para encadenar correctamente las instrucciones con dependencias).

Ejemplo 1

Datos:

- Ud. Carga=2 ciclos, IR=$\frac{1}{2}$
- Sum/Rest=2 ciclos, IR=1
- Mult/Div=7 ciclos, IR=1
- Regs[F4] = 4.0; Regs[R1] = 8; Regs[R2] = 32; Mem[a+8] = *x*; Mem[b+32] = *y*;

Código:

```
l.d f1, a(r1)
l.d f2, b(r2)
mul.d f0,f2,f4
sub.d f3,f2,f1
div.d f5,f0,f1
add.d f0,f3,f2
```

Diagrama i–t

PC	Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	l.d f1,a(r1)	IF	I	AC	L1	L2	WB	C											
1	l.d f2,b(r2)		IF	I	AC	–	L1	L2	WB	C									
2	mul.d f0,f2,f4			IF	I	–	–	–	–	M1	M2	M3	M4	M5	M6	M7	WB	C	
3	sub.d f3,f2,f1				IF	I	–	–	–	A1	A2	WB	–	–	–	–	–	–	C
4	div.d f5,f0,f1					IF	I	–	–	–	–	–	–	–	–	–	–	M1	M2
5	add.d f0,f3,f2						IF	I	–	–	–	–	A1	A2	WB	–	–	–	–

Estado del procesador

Al final del ciclo 16:

ROB

	busy	instr	completed	dest	value	pred	PC
0	NO	*l.d f1,a(r1)*	*SÍ*	*F1*	*x*		*0*
1	NO	*l.d f2,b(r2)*	*SÍ*	*F2*	*y*		*1*
2	SÍ	mul.d f0,f2,f4	SÍ	F0	4*y		2
3	SÍ	sub.d f3,f2,f1	SÍ	F3	y-x		3
4	SÍ	div.d f5,f0,f1		F5			4
5	SÍ	add.d f0,f3,f2	SÍ	F0	y-x+y		5

Estado del procesador (cont.)

Estaciones de reserva:

	busy	Op	Q1	V1	Q2	V2	rob	result
a1	NO	-		*y*		*x*	*#3*	*y-x*
a2	NO	+		*y-x*		*y*	*#5*	*y-x+y*
m1	NO	*		*y*		*4.00*	*#2*	*4*y*
m2	SÍ	/		4*y		x	#4	

Buffers de carga/almacenamiento:

	busy	Q1	V1	disp	addr	rob	result
l1	NO		*8*	*a*	*8+a*	*#0*	*x*
l2	NO		*32*	*b*	*32+b*	*#1*	*y*

	busy	Q1	V1	disp	addr	rob	Q2	V2	confirm
s1	NO								
s2	NO								

Registros:

	F0	F1	F2	F3	F4	F5	F6	F7
rob	#5			#3		#4		
valor		x	y		4.0			

	R0	R1	R2	R3	R4	R5	R6	R7
rob								
valor	0	8	32					

Ejemplo 2

Datos:

- Carga/almac=3 ciclos, IR=$\frac{1}{3}$
- Mult/Div=3 ciclos, IR=1
- Entero=1 ciclo, IR=1
- Mem[V+72]= $x1$; Mem[V+64]= $x2$; ...
- Regs[F2]= 2,0; Regs[R1]= 72

Código:

```
loop: l.d f0,V(r1)
      mul.d f4,f0,f2
      s.d f4,V(r1)
      dsubi r1,r1,8
      bnez r1,loop
      trap 0
```

Diagrama i–t

PC	Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	l.d f0,V(r1)	IF	I	AC	L1	L2	L3	WB	C								
1	mul.d f4,f0,f2		IF	I	–	–	–	–	M1	M2	M3	WB	C				
2	s.d f4,V(r1)			IF	I	AC	–	–	–	–	–	–	–	C	L1	L2	L3
3	dsubi r1,r1,8				IF	I	E1	–	WB	–	–	–	–	–	C		
4	bnez r1,loop					IF	I	–	–	E1	WB	–	–	–	–	C	
0	l.d f0,V(r1)						IF	I	–	AC	L1	L2	L3	WB	–	–	C
1	mul.d f4,f0,f2							IF	I	–	–	–	–	–	M1	M2	M3
2	s.d f4,V(r1)								IF	I	AC	–	–	–	–	–	–
3	dsubi r1,r1,8									IF	I	E1	WB	–	–	–	–
4	bnez r1,loop										IF	I	–	E1	WB	–	–
0	l.d f0,V(r1)											IF	I	AC	–	–	–
1	mul.d f4,f0,f2												IF	I	–	–	–
2	s.d f4,V(r1)													IF	I	AC	–
3	dsubi r1,r1,8														IF	I	E1
4	bnez r1,loop															IF	I
0	l.d f0,V(r1)																IF

Nota: En el ciclo 7 hay un conflicto en WB por acceso al CDB, que se resuelve dando prioridad a la instrucción más antigua.

Estado del procesador (ciclo 7)

ROB:

	busy	instr	completed	dest	value	pred	PC
0	SÍ	l.d f0,V(r1)	SÍ	F0	x1		0
1	SÍ	mul.d f4,f0,f2		F4			1
2	SÍ	s.d f4,V(r1)		s1			2
3	SÍ	dsubi r1,r1,8		R1			3
4	SÍ	bnez r1,loop		loop		salta	4
5	SÍ	l.d f0,V(r1)		F0			0

Estado del procesador (ciclo 7) (cont.)

Estaciones de reserva:

	busy	Op	Q1	V1	Q2	V2	rob	result
e1	SÍ	-		72		8	#3	64
e2	SÍ	B	#3			0	#4	
m1	SÍ	*		x1		2	#1	
m2	NO							

Buffers de carga/almacenamiento:

	busy	Q1	V1	disp	addr	rob	result
l1	SÍ	#3		V		#5	
l2	NO						
l3	NO						

	busy	Q1	V1	disp	addr	rob	Q2	V2	confirm
s1	SÍ		72	V	V+72	#2	#1		NO
s2	NO								
s3	NO								

Estado del procesador (ciclo 7) (cont.)

Registros:

	F0	F1	F2	F3	F4	F5	F6	F7
rob	#5				#1			
valor			2.00					

	R0	R1	R2	R3	R4	R5	R6	R7
rob		#3						
valor	0	72						

Estado del procesador (ciclo 15)

ROB:

	busy	instr	completed	dest	value	pred	PC
0	NO	*l.d f0,V(r1)*	*SÍ*	*F0*	*x1*		*0*
1	NO	*mul.d f4,f0,f2*	*SÍ*	*F4*	*2*x1*		*1*
2	NO	*s.d f4,V(r1)*		*s1*			*2*
3	NO	*dsubi r1,r1,8*	*SÍ*	*R1*	*64*		*3*
4	NO	*bnez r1,loop*	*SÍ*	*loop*	*Salta*	*Salta*	*4*
5	SÍ	l.d f0,V(r1)	SÍ	F0	x2		0
6	SÍ	mul.d f4,f0,f2		F4			1
7	SÍ	s.d f4,V(r1)		s2			2
8	SÍ	dsubi r1,r1,8	SÍ	R1	56		3
9	SÍ	bnez r1,loop	SÍ	loop	Salta	Salta	4
10	SÍ	l.d f0,V(r1)		F0			0
11	SÍ	mul.d f4,f0,f2		F4			1
12	SÍ	s.d f4,V(r1)		s3			2
13	SÍ	dsubi r1,r1,8		R1			3
14	NO						
15	NO						

Estado del procesador (ciclo 15) (cont.)

Estaciones de reserva:

	busy	Op	Q1	V1	Q2	V2	rob	result
e1	SÍ	-		56		8	#13	
e2	NO	*B*		*56*		*0*	*#9*	*Salta*
m1	SÍ	*	#10			2.0	#11	
m2	SÍ	*		*x2*		2.0	#6	

Buffers de carga/almacenamiento:

	busy	Q1	V1	disp	addr	rob	result
l1	NO		*64*	*V*	*V+64*	*#5*	*x2*
l2	SÍ		56	V	V+56	#10	
l3	NO						

	busy	Q1	V1	disp	addr	rob	Q2	V2	confirm
s1	SÍ		72	V	V+72	#2		2*x1	SÍ
s2	SÍ		64	V	V+64	#7	#6		NO
s3	SÍ		56	V	V+56	#12	#11		NO

Estado del procesador (ciclo 15) (cont.)

Registros:

	F0	F1	F2	F3	F4	F5	F6	F7
rob	#10				#11			
valor	x1		2.00		2*x1			

	R0	R1	R2	R3	R4	R5	R6	R7
rob		#13						
valor	0	64						

Memoria:

Dir	Dato
..	...
V+56	x3
V+64	x2
V+72	x1

Ejemplo 2: El primer salto se predice incorrectamente

PC	Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	l.d f0,V(r1)	IF	I	AC	L1	L2	L3	WB	C								
1	mul.d f4,f0,f2		IF	I	–	–	–	–	M1	M2	M3	WB	C				
2	s.d f4,V(r1)			IF	I	AC	–	–	–	–	–	–	–	C	L1	L2	L3
3	dsubi r1,r1,8				IF	I	E1	–	WB	–	–	–	–	–	C		
4	bnez r1,loop					IF	I	–	–	E1	WB	–	–	–	–	C	
5	trap 0						IF	I	–	–	–	–	–	–	–	x	
6	sigte							IF	I	–	–	–	–	–	–	x	
7	sigte								IF	I	–	–	–	–	–	x	
8	sigte									IF	I	–	–	–	–	x	
9	sigte										IF	I	–	–	–	x	
10	sigte											IF	I	–	–	x	
11	sigte												IF	I	–	x	
12	sigte													IF	I	x	
13	sigte														IF	X	
14	sigte															X	
0	l.d f0,0(r1)																IF

Estado del procesador (ciclo 15)

ROB:

	busy	instr	completed	dest	value	pred	PC
0	NO	*l.d f0,V(r1)*	*SÍ*	*F0*	*x1*		*0*
1	NO	*mul.d f4,f0,f2*	*SÍ*	*F4*	*2*x1*		*1*
2	NO	*s.d f4,V(r1)*		*s1*			*2*
3	NO	*dsubi r1,r1,8*	*SÍ*	*R1*	*64*		*3*
4	NO	*bnez r1,loop*	*SÍ*	*loop*	*Salta*	*No Salta*	*4*
5	NO	*trap 0*	*SÍ*				*5*
6	NO	*sigte*					*6*
7	NO	*sigte*					*7*
8	NO	*sigte*					*8*
9	NO	*sigte*					*9*
10	NO	*sigte*					*10*
11	NO	*sigte*					*11*
12	NO	*sigte*					*12*
13	NO						
14	NO						
15	NO						

Estado del procesador (ciclo 15) (cont.)

Estaciones de reserva:

	busy	Op	Q1	V1	Q2	V2	rob	result
e1	NO	-		72		8	#3	64
e2	NO	B		64		0	#4	Salta
m1	NO	*		x1		2.00	#1	2*x1
m2	NO							

Buffers de carga/almacenamiento:

	busy	Q1	V1	disp	addr	rob	result
l1	NO		72	V	V+72	#0	x1
l2	NO						
l3	NO						

	busy	Q1	V1	disp	addr	rob	Q2	V2	confirm
s1	SÍ		72	V	V+72	#2		2*x1	SÍ
s2	NO								
s3	NO								

Estado del procesador (ciclo 15) (cont.)

Registros:

	F0	F1	F2	F3	F4	F5	F6	F7
rob								
valor	x1		2.0		2*x1			

	R0	R1	R2	R3	R4	R5	R6	R7
rob								
valor	0	64						

Memoria:

Dir	Dato
..	...
V+56	*x*3
V+64	*x*2
V+72	*x*1

Estado del procesador (ciclo 16)

Registros:

	F0	F1	F2	F3	F4	F5	F6	F7
rob								
valor	x1		2.0		2*x1			

	R0	R1	R2	R3	R4	R5	R6	R7
rob								
valor	0	64						

Memoria:

Dir	Dato
..	...
V+56	$x3$
V+64	$x2$
V+72	$2 * x1$