ARQUITECTURA E INGENIERÍA DE COMPUTADORES

Tema 2.4

Miquel Gómez Corral

Ing. Informática ETSINF, UPV

Tema 2.4

Gestión dinámica de instrucciones y especulación

El objetivo es que el ILP se aumente, pero este caso lo hará el hardware reordenando las instrucciones en tiempo de ejecución. 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.

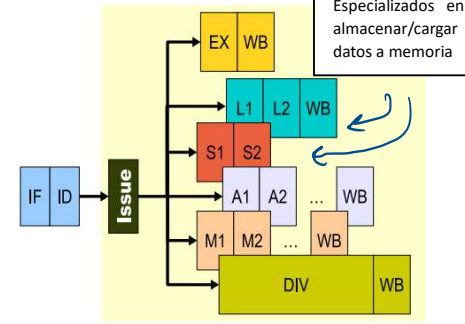
Se quiere que el hardware pueda lanzar a ejecución instrucciones que van después a la que está detenida.

Conseguimos: CPI ≈ 1. Soluciona dependencias desconocidas en tiempo de compilación. Favorece la compatibilidad binaria entre procesadores (latencias, operadores…). Simplifica el compilador. *COMPLICA EL DISEÑO HARWARE*.

GESTIÓN DINÁMICA DE INSTRUCCIONES

Evitar ciclos de parada en ID → CPI ≈ 1. **Ejecutar inmediatamente las instrucciones independientes**, Detectando las instrucciones dependientes y gestionando las correctamente: HACER que las **instrucciones independientes** adelanten a instrucciones que estén en espera. *Que se ejecuten por su cuenta las instrucciones que no tienen dependencia y cuando las dependientes dejen de tener problemas las ejecutas.* → **Algoritmo de Tomasulo**.

Nueva etapa Issue



Cuando la etapa ID decodifica una instrucción, se la pasa a la etapa Issue. Hace cosas y ya ve donde a que operador le toca mandar:

- Si el **operador** implicado **está disponible** y la instrucción tiene todos sus operandos disponibles, la lanza a **ejecución**. **NO DEPENDENCIA**.
- Si el **operador** implicado **no está disponible**, la instrucción **espera**. **DEPENDENCIA**.
- Si algún **operando no está disponible**, porque hay una dependencia de datos con otra instrucción en ejecución, la instrucción **debe esperar**. **DEPENDENCIA**.

Las instrucciones que dependen de algo se esperan. Antes se esperaban en ID lo que hacía que no pudiera avanzar ninguna otra instrucción. Ahora podríamos hacerlo que se espere en el operador que le toque, pero si se ejecuta otra que también le hace falta el operador vuelves a tener el mismo problema.

Nuevas etapas: La solución es crear otro **sitio** donde hacer que las **instrucciones se esperen**: Reservation station.

```
fdiv.d f0,f2,f4    IF ID I  DIV DIV ... DIV WB
fadd.d f10,f0,f8      IF ID I   I   ... I   A1  A2 ...
fmul.d f12,f8,f14        IF ID  ID  ... ID  I   M1 ...
```

→

```
fdiv.d f0,f2,f4   IF ID I  DIV DIV ... DIV WB
fadd.d f10,f0,f8     IF ID I   a1  ... a1  A1  A2 ...
fadd.d f12,f8,f14       IF ID  I   A1  ...
```

Se puede poner como una "-->" o como la letra del operador que espera: M1 → m1, A1 → a1, DIV → div…

Todo esto nos crea un "grafo de dependencias". Las dependencias se añaden en ISSUE y se resuelven en WrtieBack. *No se usan como tal cortocircuitos, se usa más como un buffer que le lleva el valor a todos los que esperaban el valor del registro (xq ya ha pasado de ID, tonces tiene que darle el valor de alguna manera).*

GRAFO DE DEPENDENCIAS

En el grafo se **creará un nodo por cada instrucción** que pase por "ID". Ahí, cada instrucción será asignada a la cola de espera del operador que necesita con su número de instrucción, pillando una copia de los datos que tenga ya disponibles. Cuando tenga todo y el operador esté disponible entrará y se ejecutará.

Cuando una **instrucción depende de otra** para recibir un dato, se pone como que desciende de esa operación (*espera resultado de esta*). Cuando la operación de la que depende termine (*pase por wb*), la dependencia se transforma en el valor resultado y se le pasa al que esperaba. Cuando tenga todo (valores y operando) se ejecutará.

Diagrama de instrucciones con gestión dinámica

	1	2	3	4	5	6	7	8	9	10	
fadd.d f4,f2,f0	IF	ID	I	A1	A2	A3	WB				
fmul.d f6,f4,f0		IF	ID	I	→	→	→	M1	M2	M3	...
fld f4,t1			IF	ID	I	L1	L2	WB			
fmul.d f6,f6,f4				IF	ID	I	→	→	→	...	

- El símbolo "→" expresa **espera no bloqueante**.
- Cuando una instrucción llega a la etapa WB, las instrucciones que esperaban su resultado continúan.
    - Cuando una instrucción termina es como que dice por un bus: "Soy el número x, y tengo el valor 3". Entonces loque escuchen dirán "buah, yo estaba esperando a 1, pásame el dato bro…"
- Cuando una instrucción se mete a esperar a otra, **se lleva consigo los datos que ya están**, por lo que **da igual que mientras espere le cambien el registro**, como ya lo tiene no es problem. Ya cuando termine por la que estaba esperando está le hará llegar el valor y esta lo pilla. Cuando los tenga todos se mete a ejecutar.

Resolución de riesgos con Tomasulo

Estructurales: Se usan los **operadores virtuales** 'esperando'. Se ejecuta cuando el operador físico este libre.

SIN Gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8
fdiv.d f6,f4,f2	IF	ID	D1	D2	D3	D4	D5	WB
fdiv.d f12,f10,f8		IF	ID	*ID*	*ID*	*ID*	*ID*	D1
fld f14,t1			IF	*IF*	*IF*	*IF*	*IF*	ID

Gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8	9
fdiv.d f6,f4,f2	IF	ID	I	D1	D2	D3	D4	D5	WB
fdiv.d f12,f10,f8		IF	ID	I	–	–	–	–	D1
fld f14,t1			IF	ID	I	L1	L2	WB	

Ahora espera en el sitio de esperar, por lo que te mejora los riesgos estructurales haciendo que mientras una espera, las demás vayan.

RAW: Se resuelven **encadenando operadores implicados**. *Si 2 depende de 1, cuando termine 1 se le da el valor a 2.*

SIN Gestión dinámica de instrucciones:

	1	2	3	4	5	6	7
fadd.d **f4**,f2,f0	IF	ID	A1	A2	A3	WB	
fmul.d f6,**f4**,f0		IF	ID	*ID*	*ID*	M1	M2
fld f4,t1			IF	*IF*	*IF*	ID	L1

Gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8	9
fadd.d **f4**,f2,f0	IF	ID	I	**A1**	**A2**	**A3**	WB		
fmul.d f6,**f4**,f0		IF	ID	I	–	–	**–**	**M1**	**M2**
fld f4,t1			IF	ID	I	L1	L2	WB	

WAW: Se resuelven haciendo que la última instrucción lanzada sea la que escribe de un modo efectivo en el registro implicado. *Lo que quieres es que la segunda escriba → a tomar por culo la primera y solo escribe la 2nd.*

SIN Gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8	9
fmul.d **f2**,f4,f0	IF	ID	M1	M2	M3	M4	M5	WB	
fadd.d **f2**,f6,f0		IF	ID	*ID*	*ID*	A1	A2	A3	WB
fdiv.d f4,f8,f10			IF	*IF*	*IF*	ID	D1	D2	D3

Gestión dinámica de instrucciones:

	1	2	3	4	5	6	7	8	9
fmul.d **f2**,f4,f0	IF	ID	I	M1	M2	M3	M4	M5	WB
fadd.d **f2**,f6,f0		IF	ID	I	A1	A2	A3	**WB**	
fdiv.d f4,f8,f10			IF	ID	I	D1	D2	D3	D4

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

	1	2	3	4	5	6	7	8	9	10
fmul.d f2,f8,f0	IF	ID	I	M1	M2	M3	M4	M5	WB	
fmul.d f6,**f4**,f2		IF	ID	**I**	–	–	–	–	–	M1
fld **f4**,t1				IF	ID	I	L1	L2	**WB**	

Las **instrucciones pasan por Issue** en orden y Se **leen los operandos que ya estén disponibles** en ese momento (*f4 en el ejemplo*), **SE LOS GUARDA MIENTRAS ESPERA**, incluso cuando otros operandos tengan dependencia de datos con otras instrucciones previas (*f2 en el ejemplo*).

ESPECUALCIÓN HARDWARE

Ejecución especulativa de instrucciones

Los saltos pueden tardar bastante más que 1 o 2 ciclos si dependen de otra instrucción. Como ahora las instrucciones siguientes se van a seguir ejecutando, puede ser que **haciendo un predict**, ejecuten cosas, **cambien el estado**, pero resulte que NO tenías que ejecutarlas xq la predicción era errónea :O -> WRONG ANSWER. -> Especulación

Especulación

Ejecutar completamente (*hasta WB inclusive*), si procede, las **instrucciones dependientes de un salto antes de conocer su comportamiento, pero pudiendo cancelarlas**: *No dejar que se marchen "terminen/modifiquen el estado" las instrucciones después del sato hasta que lo sepamos, así podemos cancelarlas.*

Excepciones: Esto incluye las excepciones, que **no se puedan emitir** excepciones hasta que se confirme que la intrucción se ejecuta, NO EXCEPCIONES MIENTRAS SON ESPECULATIVAS.

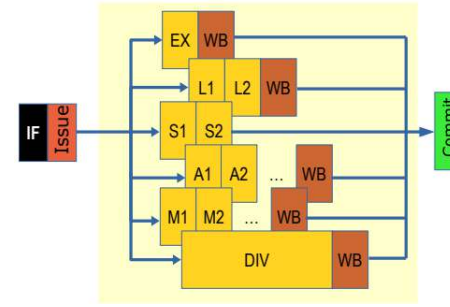
Especulación Hardware *Nueva ruta de datos con Commit (C)*

1. **Predicción dinámica de saltos** para seleccionar que instrucciones hay que ejecutar.
2. Búsqueda de las instrucciones en orden (IF).
3. Etapa "I": Decodificación y lanzamiento de las instrucciones en orden: Se junta ID y ISSUE (ID+Issue → I).
4. Ejecución de las instrucciones fuera de orden (EX, M1,M2, A1, A2… WB).
5. Finalización de las instrucciones en orden → Commit.

IF	I	A1	A2	A3	WB	C					
	IF	I	EX	WB	-	-	C				
		IF	I	M1	M2	M3	M4	M5	WB	C	
			IF	I	EX	WB	-	-	-	-	C

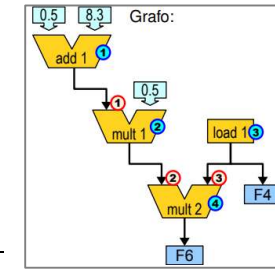
Commit: Ahora **escribimos registro y memoria en COMMIT**, no en WB. WB solo sirve para resolver las dependencias del grafo. A Commit **solo llegan las instrucciones ejecutadas correctamente**. *Si se ha de cancelar una instrucción posterior a un salto xq no debe de ejecutarse*, al no haber llegado a commit no habrá cambiado el estado de la memoria, por lo que está correcto. *Las excepciones ahora también se reconocen aquí. Esto xq llegan en orden.*

ReoderBuffer: Ahora tenemos un buffer en el que se guardarán las instrucciones que NO han llegado a commit, de manera que sabemos donde escribe cada una, que datos tiene, cuales le falta, en qué orden va…



Estación de reserva: Son los **Buffers antes de cada operador**. Cuando están libres pillan una de las instrucciones que haya esperado.

Grafo de dependencia: Se representa con un **símbolo de ALU** a cada operación, se le **asigna la "marca"** de la instrucción que lo ejecuta al lado. Si uno de los valores que va a usar depende de otra operación se le pone una flecha que salga del otro operador. Si no, coge el valor y se lo guarda, lo lee de registros, de forma que ya lo tiene y luego se pueden cambiar los registros sin problema

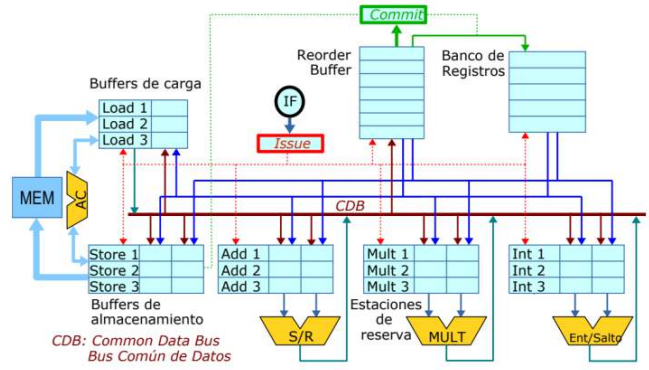


En el grafo, la instrucción 1 ha pillado los datos del banco de registros. La 2 ha pillado 0.5 y se ha guardado que el dato que salga de 1 lo usa. La 3 ha empezado a ejecutarse. Y cuando ha llegado la 4. A esta le hacen falta los datos de 2 y 3.

Ejecución de instrucciones: Ejecutas las instrucciones de esta manera: *A commit solo llegan las que se han de ejecutar. Si se han de cancelar se cancelan antes de hacer commit, por lo que no modifican nada.*

$$IF\ +\ I\ (ID + Issue)\ +\ FASE\ DE\ OPERAR\ +\ WB\ (resolver\ dependencias\ grafo)\ +\ COMMIT$$

Objetivo es ir ejecutando todo lo que podamos y cuando confirmemos que esa instrucción esté oke, la llevamos a commit para poder confirmar. Queremos que acaben en orden por las instrucciones de salto y las excepciones.



CBD: Permite comunicarse con "Difusión", todos los que **producen datos** están conectados a él para que cuando saquen un resultado lo **puedan pasar a los operandos** que les haga falta y **resolver las dependencias**. *También TODOS aquellos que puedan recibir/leer un dato están conectados a ellos para poder recibirlos cuando sean escrito.*

Linea azul: Sirve para **pasar los valores de los bancos de registros**. No solo los que estén físicamente en él, también los que **han sido calculados por alguna instrucción,** pero aún no ha llegado a commit. Si una instrucción a hecho WB, pasará por el CBD su dato a quien le haga falta. Pero si otra instrucción pide ese dato después de que haga WB llegaría tarde: Para que esta también pueda pillar el dato, este se guarda en el ROB, y por aquí lo accede.

ReorderBuffer (ROB): Es **una cola FIFO** que permite que las instrucciones terminen en orden. Como las instrucciones de almacenamiento y escritura en registro SOLO se pueden lanzar cuando hagan commit, se conecta el commit a memoria y al banco de registros para que "hagan" estas operaciones.

```
11  12  13  14  15  16
WB  C
-   -   C   L1  L2  L3
-   -   -   C
```

Cosas raras de las Store: Esto hace que por ejemplo, los ciclos de acceso a memoria de las operaciones de store se haga DESPUÉS DE COMMIT.

Transferencia a través del bus de datos *el CBD*

Dos fases: Sabes que uno se va a querer comunicar con otro (*darle un dato*), pero aún NO sabes el dato. Aprovechas.

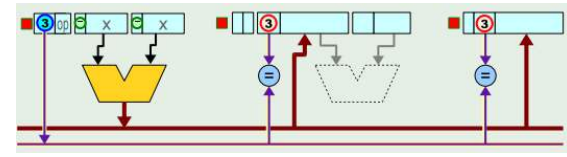
1. **Fase de preparación**: Se **conoce** el origen y el destino de la transferencia, pero todavía **no tiene el dato**.

*fadd.d **f4**,f1,f0 seguida de fmul.d f12,**f4**,f10: Sabes que el sumador escribe en f4, y que este le hace falta al multiplicador. Pues sabes que el mult recibe algo de sum, "te preparas" predices en la fase I los ciclos de parada y eso.*

2. **Fase de transferencia**: El dato ya se ha obtenido y realiza su transferencia. ***Se hace en WB***.

Identificación del dato: Los elementos que escriben en el bus, a cada instrucción que entra se las identificad con un "código", se asigna en Issue según la entrada en el ROB. Posición 0 del ROB código 0, posición 1 código 1…

Los **elementos que leen del bus tienen asignada una "marca"** en Issue. Esta **marca** dice de QUIEN va a recibir el dato dentro de un rato cuando se tenga (*marca es el código de una intrucción que está en algún operando*).



Cuando alguien escribe en el bus pone su código, los que pueden leer, leen SOLO si la marca coincide con el código.

ReorderBuffer *ROB*

Mantenimiento del orden de las instrucciones. En la etapa Issue, se reserva una entrada en el ROB, *se les da el código*

Datos que guarda: **Bit de Busy** para decir si hay una instrucción que aún no ha hecho commit. **PC** de la instr. Instrucción como tal. **Destino** (pasa saltos). **Valor/Condición** calculada por la intruccón. **Bit Completado** que dice si ya ha dado un valor, se usa cuando una instr va a ver si ya se ha calculado lo que le hace falta. **Predicción** para Saltos.

Almacenamiento temporal del resultado: En la etapa WB se almacena en el ROB el resultado de la operación y se le pasa los datos al resto de las instrucciones que les hagan falta. Se almacena por si alguna lo pide después. *NO SE TOCA NI LA MEMORIA NI EL BANCO DE REGISTRO HASTA QUE LLEGUE A COMMIT.*

Una instrucción dependiente deber a 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. Cuando hagas commit se procesa y se mete la rutina para procesarla correctamente.

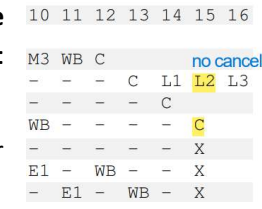
Commit

Solo hay **2 posibilidades** para que se **cancelen instrucciones**: Que haya una excepción o que un salto hay sido mal predicho. Cuando la instrucción más antigua del ROB finaliza (*etapa Commit*):

1. Se **comprueba si ha producido alguna excepción**, lanzando la rutina de servicio, en su caso.
2. Si no, se **escribe su resultado** desde el ROB al registro destino o se realiza la escritura en la posición de memoria correspondiente
3. Se libera la entrada del ROB.

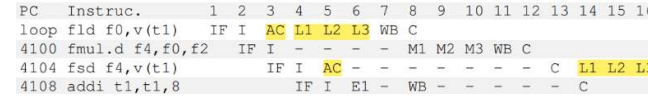
Salto: Si un salto **predicho correctamente** NO HACES NADA. Si es **incorrectamente predicho**, cuando llega a la etapa Commit, borra TODO el contenido del ROB (*cancelas*): Todo lo que haya dentro será posterior al salto porque se habrán puesto en ORDEN tras él.

- **Stores**: Las instrucciones de store hacen cosas después de commit y puede haber alguna aún haciendo sus fases. Esas no las cancenlas.



Cosas importantes

- Ahora las **Instrucciones de memoria hacen una fase AC** para calcular la dirección. Las de Load hacen L1, L2… antes de WB y commit. STORE hace cuando pueda AC y se espera a poder hacer Commit. DESPUÉS DE COMMIT hace L1, L2… El valor de la memoria no se cambiará hasta que terminen los Li… Cuando pasa por commit marca como confirmada la instrucción strore en el buffer de las stores.



- Cuando la **predicción de salto falla** y tienes que cancelar CANCELAS TODO LO QUE HAYA EN LOS BUFFERS, menos los Stores confirmados.
- **No se pueden hacer 2 WB a la vez** xq cuando haces WB usas el CDB, tonces conflicto estructural.
- TODAS las **instrucciones que tengan que escribir en memora/registros lo hacen**, aunque después hay otra que lo haga en el mismo sitio. *Esto para que **si la segunda se cancela**, tengas el dato de la primera.*
- Cuando una instrucción "reserva un registro" xq va a escribir en él, solo lo libera CUANDO HACE COMMIT Y ES SUYO. Si yo reservo f4, y otra también, cuando yo termine NO lo libero xq hay otra después que lo ha reservado.
- Cuando lees un registro del banco de registros el orden es Primero lees marca, luego ROB y sino Lees registro.
- **EJERCICIOS SACAR ROB** en un momento dado: Del Cronograma vas viendo lo que ha terminado y lo que no.
- **EJERCICIOS SACAR REGISTROS**: Mirar rob, quien ha hecho commit y el último que reserva.
- **EJERCICIOS SACAR MEMORIA**: Los que hayan hecho commit son los únicos que la han podido modificar.