TEMA 3

PLANIFICACIÓN O REORDENAMIENTO *(SCHEDULING)* DE INSTRUCCIONES

ÍNDICE

3.1. CONCEPTOS FUNDAMENTALES

- Definición de Planificación, secuenciamiento o reordenamiento (*scheduling*): cambiar el orden (reordenar) instrucciones para evitar bloqueos.
- Objetivo: eliminar riesgos (y reducir bloqueos). Típicamente se pueden reducir los bloqueos de datos. Bq de control se vieron 2.8,2.9. Bq estructurales dependen del hw
  - Recordar: dependencias WAR, WAW ficticias, se pueden evitar por renombrado de registros. Luego es importante tener un conjunto amplio de registros para renombrar y reordenar (ventaja RISC).
  - Pero dependencias reales RAW no pueden eliminarse, son intrínsecas al algoritmo (habrá algoritmos con más RAW y otros con menos). Eliminarlas no es tema relativo a la Arquitectura sino a la Algorítmica.
- **Conclusión**: realmente un programa podría escribirse atendiendo exclusivamente a las dependencias reales, sin ser necesarios en un primer momento los registros.
- Claro que cuando se diseña un procesador como máquina síncrona, aparecen obligatoriamente los registros como el estado de la máquina almacenado tras cada periodo.

  $\Rightarrow$ Objetivo: intentar que el programa se ejecute como el grafo siguiente.

Renombrado. Ejemplo

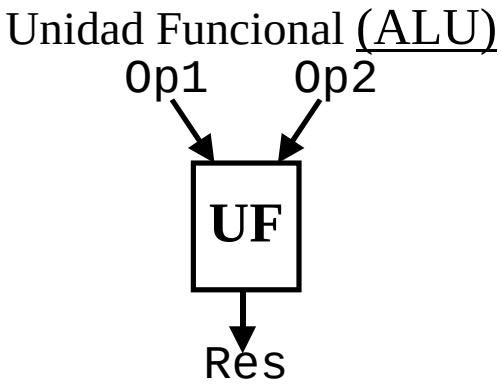
```
MULTF   F1, F2, F3
DIVF    F2, F1, F4
ADDF    F1, F2, F3
SF      0(R1), F1
```

```
MULTF   F1, F2, F3
DIVF    F20, F1, F4
ADDF    F10, F20, F3
SF      0(R1), F10
```

"Ensamblador" con grafo de dependencias

Ensamblador clásico

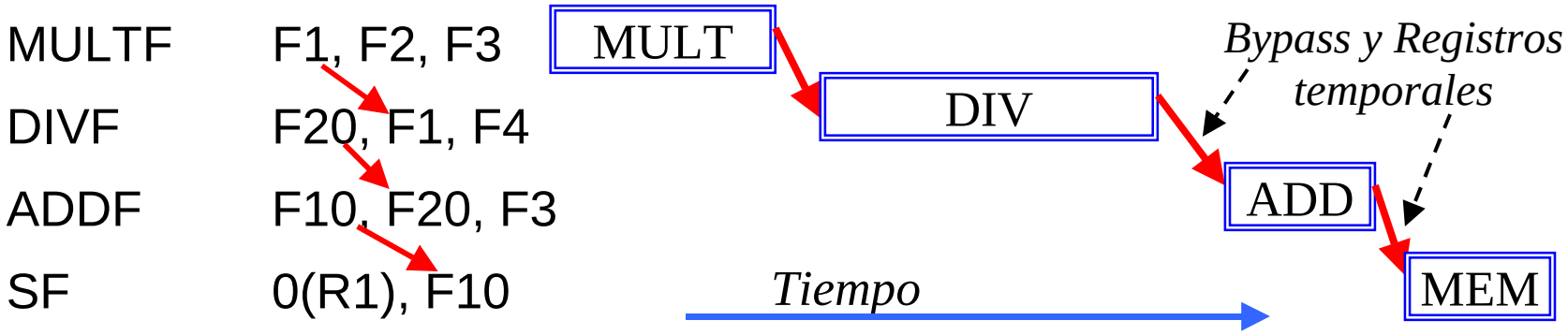
```
MULTF   F1, F2, F3
DIVF    F20, F1, F4
ADDF    F10, F20, F3
SF      0(R1), F10
```



Los registros (celeste) son prescindibles

Límite de las prestaciones

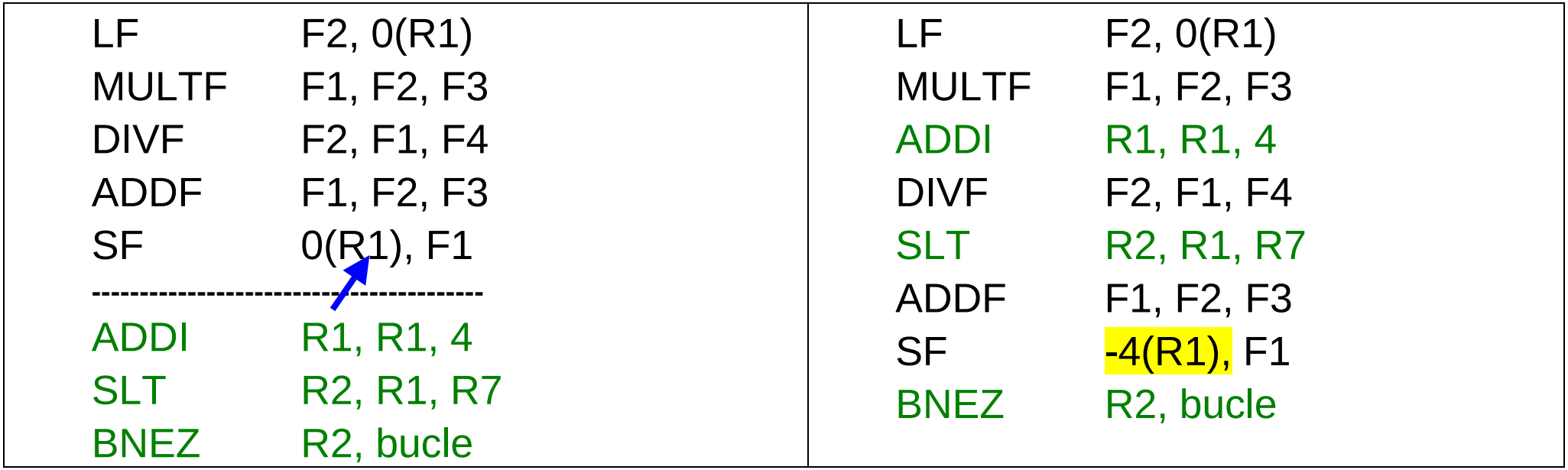
- En encadenamiento básico, un bloqueo por cualquier dependencia bloquea totalmente la cadena (máquina): aumenta el CPI
- Luego, reordenando podemos eliminar c. bq. (típicamente de datos).
- Pero existe un límite (en la aceleración alcanzable...):
  - ⇒ límite de CPI (*data flow limit* o límite de flujo de datos)
- Ejemplo: Supongamos que las fases "accesorias" de una instrucción (en las que la instrucción se busca, decodifica, etc.) duran cero ciclos, y que únicamente duran un tiempo apreciable las operaciones (entre ellas el acceso a la memoria).
  - Realmente los procesadores actuales han llegado prácticamente a esto.
- El grafo dependencias se convertiría en un cronograma (sólo fases EX, MEM)



Planificación instrucciones FP - INT

- Generalmente la duración de las unidades funcionales U.F. FP es mayor ⇒ más c.bq. datos: problema clásico de los años ‘60 (código científico, supercomputadores electrónica discreta).
- Típicamente un bucle (más del 90% t. de ejec.) de código FP se puede dividir en instr. INT e instr. FP. Cada grupo de instrucciones tendría su grafo de dependencias por separado; cada grupo usa registros diferentes y propios (excepto los Load-Store que mezclan un reg. INT con otro FP) ⇒ no hay dependencias entre ambos grupos.

⇒ Reordenación estática de instr. INT con FP muy evidente



```
LF      F2, 0(R1)
MULTF   F1, F2, F3
DIVF    F2, F1, F4
ADDF    F1, F2, F3
SF      0(R1), F1
-----------------------------------
ADDI    R1, R1, 4
SLT     R2, R1, R7
BNEZ    R2, bucle
```

```
LF      F2, 0(R1)
MULTF   F1, F2, F3
ADDI    R1, R1, 4
DIVF    F2, F1, F4
SLT     R2, R1, R7
ADDF    F1, F2, F3
SF      -4(R1), F1
BNEZ    R2, bucle
```

• Planificación estática

- PERO PROBLEMA: el compilador debe conocer la arquitectura (la duración de las operaciones).
- ¡Y esto puede cambiar con la versión del procesador! El programa reordenado estáticamente que hoy es rápido, mañana puede ser lento. Luego habría que recompilar todos los programas para conseguir el ajuste ideal ("sintonización") entre código y procesador (técnicas dinámicas no sufren de este defecto). La sintonización entre aplicación (en este caso, su rendimiento), compilador y hw. no gusta al usuario.
- Lo que siempre es más ventajoso en el uso de técnicas estáticas es que el hardware puede simplificarse (o usar estos recursos –transistores- para otras partes, como más cachés, más registros, más unidades funcionales –ALU's-, etc.).

• Planificación dinámica

- Veremos que la reordenación se puede hacer dinámicamente (3.3).
- *IDEA*: la CPU puede anotar en tiempo de ejecución las dependencias que van surgiendo sin bloquear la máquina (ya no se ve tan clara la ejecución en cadena). La instr. que necesita un registro (debido a una RAW) se espera, sin ejecutar. Cuando los operandos de tal instrucción estén disponibles realiza su fase EX. Mientras, otras instrucciones (sin RAW) seguirán adelante con todas sus fases.

3.2, 3.4. PLANIFICACIÓN ESTÁTICA. DESENROLLADO DE BUCLES.

- La planificación estática está limitada por los saltos (es más difícil "cruzarlos" y existen pocos casos en que se pueda).
- Siempre se puede apostar por un comportamiento T/NT en tiempo de compilación (predicción estática) y reordenar usando también instrucciones de la rama predicha (*especulación software*). Pero esto conlleva a complicaciones y en general no es muy eficaz (se verá en ASP2).
- NOTA: la predicción dinámica es mucho más certera que la estática $\Rightarrow$ gracias a esto la planificación dinámica puede sacar más partido (las reordenamientos acertarán más veces). La *especulación hardware* es más poderosa.
- Sin embargo, el desenrollado de bucles (*loop-unrolling*) va algo más allá que la simple reordenación, pues elimina instrucciones (de *overhead* o sobrecarga), con lo cual elimina también ciertas dependencias. Por tanto, en principio superará todas las técnicas dinámicas, con lo que respecta a la eliminación de esas instrucciones.
- Veremos que con planificación dinámica también se hace una especie de desenrollado (simplemente usando el código original).

Resumen de desenrollado (ver práctica 4)

Suponemos bucle paralelizable (iteraciones independientes).

Ejemplo:

```
double s, x[M] , y[M];  int i;
for (i=0 ; i<M ; i++ )  y[i]= x[i] * s ;
```

Duración U.F. MULT (la dependencia que dará más c. bq.) = **4 ciclos**.

Nº c. bq. entre MULT y Store = Duración – 1 (–1) = 4 –1 –1 = 2

(el último –1 porque el Store de DLX usa el dato en la fase MEM)

Como regla general, necesito desenrollar **1+(Nº c. bq. en la dependencia más larga)**= 3+1 iteraciones. Si Nº c.bq.=0 habría que desenrollar 1 iter., es decir dejar el bucle como está. Nos concentramos en un desenrollado sistemático en tal dependencia.

```
for (i=0 ; i<M%3 ; i++)  y[i]= x[i] * s; //star-up (arranque)
for (    ; i<M ; i+=3 ) {
   y[i+0]= x[i+0] * s ;
   y[i+1]= x[i+1] * s ;
   y[i+2]= x[i+2] * s ;
  }
```

- En ensamblador, se distingue entre **instrucciones útiles** (de cómputo y de acceso a memoria de los datos y resultados) e instrucciones de sobrecarga (*overhead*). Estas últimas son las que regulan el bucle (por estar escrito en lenguaje iterativo imperativo) y pueden eliminarse si se desenrollara "infinitas" veces el bucle. P. ej. en un lenguaje orientado a vectores, no existiría tal bucle.

```
bucle_orig:
    LD     F2, 0(R1)
    MULTD  F4, F2, F24  ; F24 contiene el valor de s
    SD     (R3)0, F4
    ;-----------------------
    ADDI   R1, R1, 8
    ADDI   R3, R3, 8
    SLTI   R7, R1, fin_array_x; constante apunta al final de x[M]
    BNEZ   R7, bucle_orig
```

Desenrollado sistemático

1. Se despliegan varias iteraciones

```
  bucle_intermedio1:
    LD     F2, 0(R1)
    MULTD  F4, F2, F24 ; F24 contiene el valor de s
    SD     (R3)0, F4
    ADDI   R1, R1, 8
    ADDI   R3, R3, 8
    SLTI   R7, R1, fin_array_x;
```

```
      ; el salto sobra, luego la instr. anterior de comp. tb. sobra
   LD     F2, 0(R1)
   MULTD  F4, F2, F24
   SD     (R3)0, F4
   ADDI   R1, R1, 8
   ADDI   R3, R3, 8
   SLTI   R7, R1, fin_array_x;
      ; el salto sobra, luego la instr. anterior de comp. tb. sobra

   LD     F2, 0(R1)
   MULTD  F4, F2, F24
   SD     (R3)0, F4
   ADDI   R1, R1, 8
   ADDI   R3, R3, 8
   SLTI   R7, R1, fin_array_x
   BNEZ   R7, bucle_intermedio1
```

2. Se eliminan las instrucciones inútiles, modificando direccionamientos y otros operandos inmediatos. Notar como ciertas dependencias reales entre instrucciones *overhead* van desapareciendo. Y que la cantidad de instrucciones *overhead* que se eliminan guarda una proporción directa con el número de iteraciones desenrolladas.

```
 bucle_intermedio2:
   LD     F2, 0(R1)
   MULTD  F4, F2, F24 ; F24 contiene el valor de s
   SD     (R3)0, F4
```

```
    LD     F2, 8(R1)
    MULTD  F4, F2, F24
    SD     (R3)8, F4

    LD     F2, 16(R1)
    MULTD  F4, F2, F24
    SD     (R3)16, F4
    ;----------------------
    ADDI   R1, R1, 8*3
    ADDI   R3, R3, 8*3
    SLTI   R7, R1, fin_array_x
    BNEZ   R7, bucle_intermedio2
```

3. Se renombran registros para evitar dependencias ficticias WAR, WAW. Usamos notación con primas (‘) por simplicidad (el DLX utilizaría registros libres de F0-F30)

```
 bucle_intermedio3:
    LD     F2, 0(R1)
    MULTD  F4, F2, F24 ; F24 contiene el valor de s
    SD     (R3)0, F4

    LD     F2’, 8(R1)
    MULTD  F4’, F2’, F24
    SD     (R3)8, F4’
```

```
    LD     F2'', 16(R1)
    MULTD  F4'', F2'', F24
    SD     (R3)16, F4''
    ;-----------------------
    ADDI   R1, R1, 8*3
    ADDI   R3, R3, 8*3
    SLTI   R7, R1, fin_array_x
    BNEZ   R7, bucle_intermedio3
```

4. Se <u>entrelazan sistemáticamente</u> las instrucciones de las distintas iteraciones

```
  bucle_desenrollado:
    LD     F2, 0(R1)
    LD     F2', 8(R1)
    LD     F2'', 16(R1)

    MULTD  F4, F2, F24 ;   F24 contiene el valor de s
    MULTD  F4', F2', F24
    MULTD  F4'', F2'', F24

    SD     (R3)0, F4
    SD     (R3)8, F4'
    SD     (R3)16, F4''
    ;-----------------------
    ADDI   R1, R1, 8*3
```

```
    ADDI   R3, R3, 8*3
    SLTI   R7, R1, fin_array_x
    BNEZ   R7, bucle_desenrollado
```

Ahora se ve que las dependencias más largas no producen c.bq. (de ahí que se hayan desenrollado 3 iteraciones)

5. Opcionalmente se reordenan las instrucciones para reducir algún c. Bq. (que no era el de la dependencia más larga). Típicamente se mezclan las instr. FP con las INT.

```
bucle_desenrollado_opc:
    LD     F2, 0(R1)
    LD     F2', 8(R1)
    LD     F2'', 16(R1)
    MULTD  F4, F2, F24 ;   F24 contiene el valor de s
    MULTD  F4', F2', F24
    MULTD  F4'', F2'', F24
    ADDI   R1, R1, 8*3
    SD     (R3)0, F4
    SD     (R3)8, F4'
    SD     (R3)16, F4''
    SLTI   R7, R1, fin_array_x ;  Quito c.bq. entre SLTI y BNEZ
    ADDI   R3, R3, 8*3
    BNEZ   R7, bucle_desenrollado_opc
```

Nota: gracias a utilizar ADDI R1,... de "colchón" entre la dependencia más larga, el nº de iter. desenrolladas podría haber sido menor.

- **Aceleración**: ¡cuidado! al reducir el número de instrucciones, el CPI no es buena medida de rendimiento. Usar N_{instr}*CPI o tb **Ciclos / elem. procesado del array**.

	Bucle original	Bucle desenrollado	Desenrollado"infinitas" iteraciones (K→∞)
Instr./elem array	(3 instr útiles + 4 instr overhead) / elem	(3*3 instr útiles + 4 instr overhead) / 3 elem array = = (3 instr útiles + 4/3 instr overhead) /elem	(3*K instr útiles + 4 instr overhead) / K elem array = (K→∞) = = 3 instr útiles/elem
F_{saltos}	1/7 = 14.3%	1/13 = 7.7%	0%
Estimación ciclos / elem array	7 instr+(1 LD-MULTD + 2 MULTD-SD + 1 SLTI-BNEZ) c.bq. datos + casi 1 c.bq. control = = casi 12 Ciclos	(13 instr + casi 1 c.bq. control) / 3 elem array = = casi 14/3 ciclos / elem = = 4.66^	(3*K instr + casi 1 c.bq. control) / K elem array = = 3 ciclos / elem
Tam código estático (bytes)	7 * 4	13 * 4	3 * K * 4
Notas	Tal vez surja un bloqueo estructural. Se pueden eliminar algunos c.bq. datos reordenando	Tal vez surjan bq estr.; Desprecio iteraciones de arranque	Prestaciones máximas. Tamaño de código estático máximo (más fallos de caché)

- *Ejercicio*: eliminar c.bq. control con diversas técnicas de saltos retrasados. Pensar en uso de BTB.
- *Ejercicio*: se podrían haber eliminado algunos c.bq. datos reordenando.
- Prestaciones máximas si desenrollo “infinitas” veces

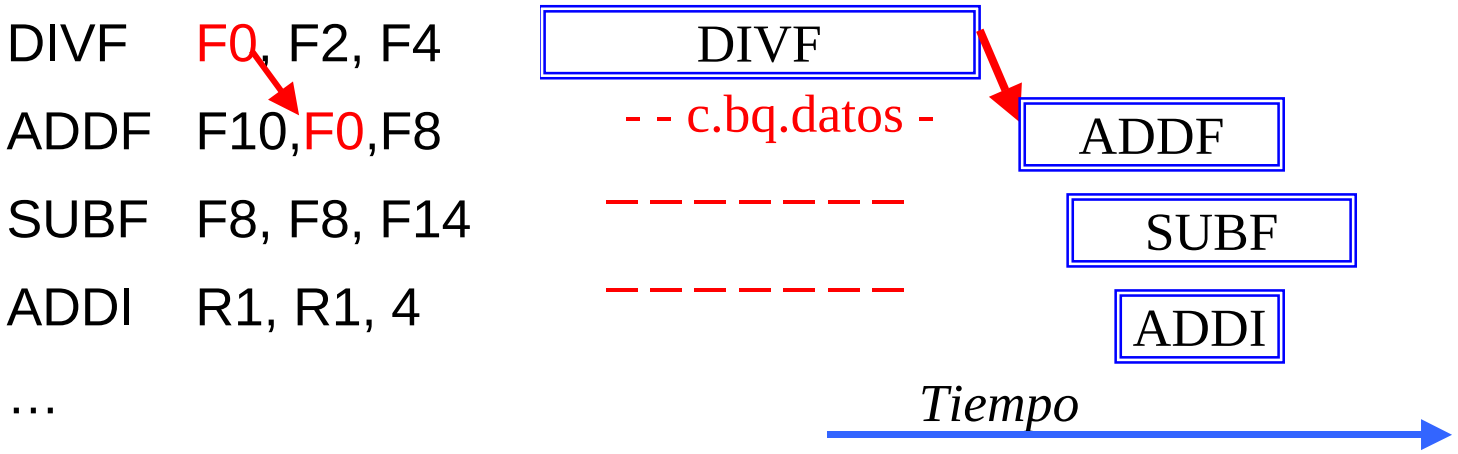
```
x[0]= x[0] * s ;
x[1]= x[1] * s ;
…
x[M-1]= x[M-1] * s ;
```

- CONCLUSIONES:
  - Se han reducido las instr. overhead (estamos “rescribiendo” el código fuente de leng. iterativo para hacerlo más rápido)
  - Se aumenta las posibilidades de planif. estática, en cuanto se reducen el porcentaje de saltos (por cada salto, hay más instr. de otro tipo)
  - Se necesitan más registros
  - Se ha de conocer la endoarquitectura de la CPU (duración de U.F., existencia de bypass, etc.)

3.3. PLANIFICACIÓN DINÁMICA.

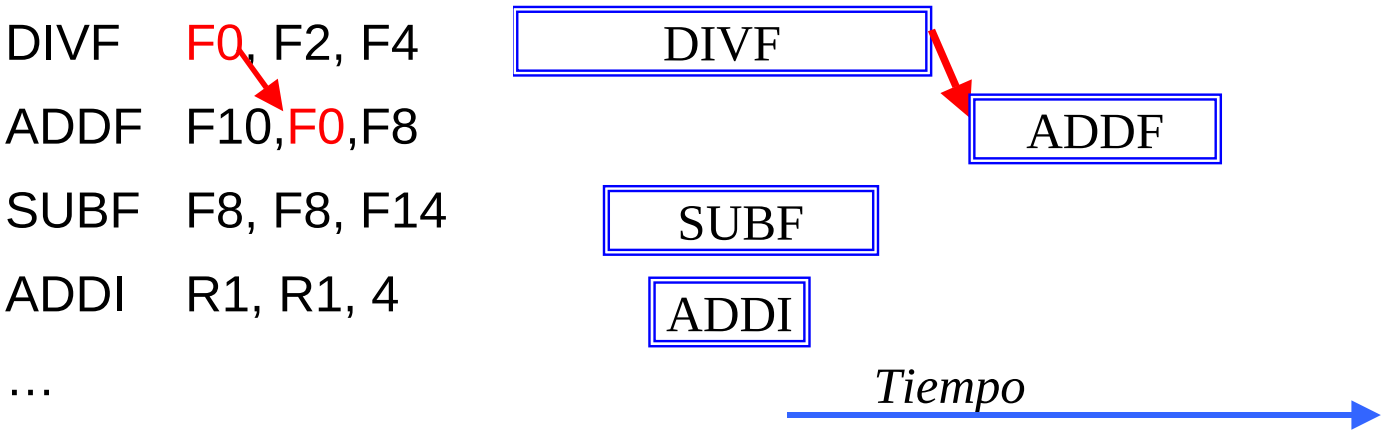
- Hasta ahora todas las técnicas de segmentación ejecutan las instrucciones en orden. Si una instrucción no puede ejecutarse, la cadena se para completamente:

Cronograma ***sin*** *planificación dinámica*



- La espera que ADDF introduce en SUBF, ADDI, ..., podría eliminarse si se permitiera que la ejecución (EX) no se hiciera en orden (*out-of-order execution*), permitiendo que SUBF y ADDI comiencen antes su ejecución (EX) que ADDF.
- Es decir, estamos reordenando la fase EX en tiempo de ejecución (dinámicamente)

Cronograma usando Planificación dinámica



- Recordar grafo de dependencias ⇒ límite de CPI (*data flow limit* o límite de flujo de datos). Esto se intenta conseguir dinámicamente como en el cronograma anterior.

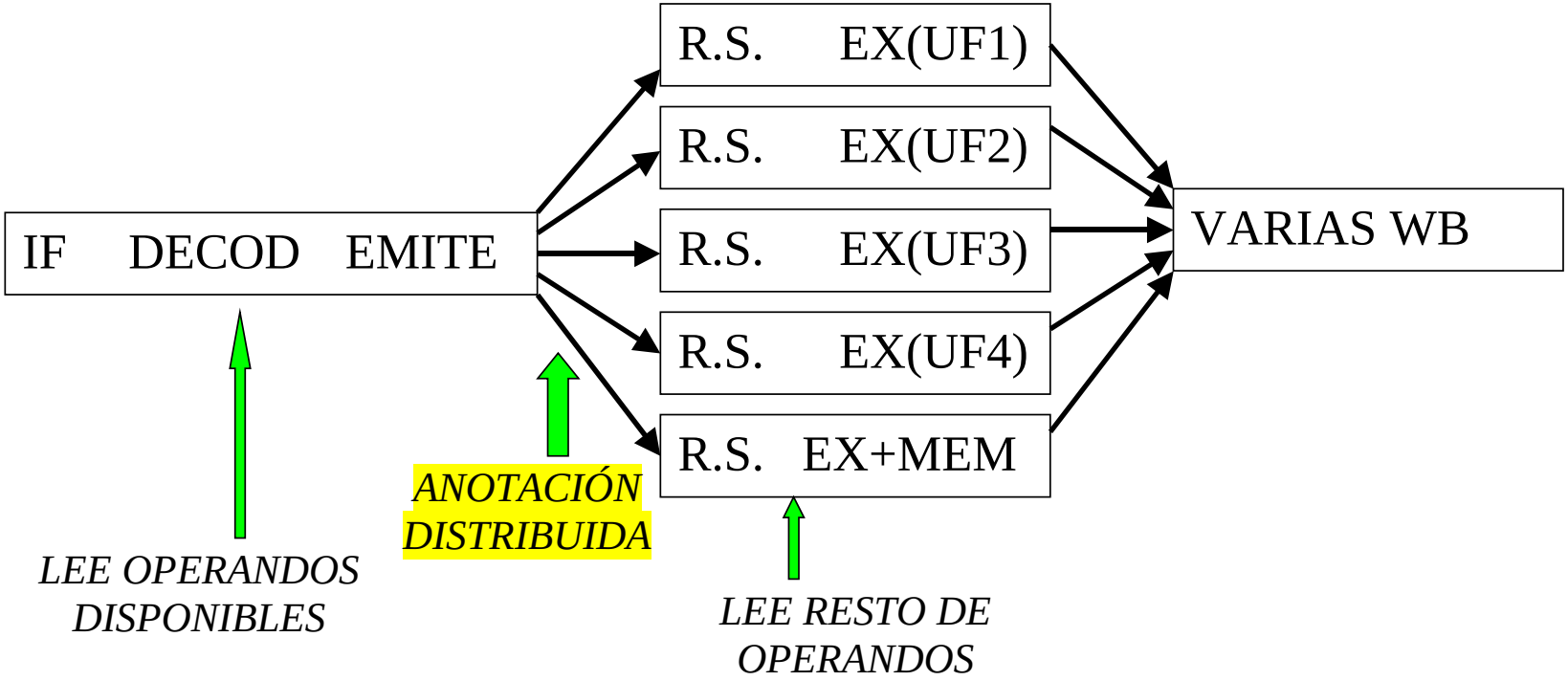
➢ La fase de decodificación (ID) siempre debe ir en orden (para comprobar dependencias).

➢ En el DLX vamos a suponer que se comprueban las dependencias de datos y los riesgos estructurales en una única fase (ID o IS).

➢ La etapa ID (IS), por tanto, se puede considerar que debe hacer dos cosas:

- Decodificación y Emisión (*Issue*): Decodificar la instrucción y comprobar dependencias de datos y riesgos estructurales (anotándolos de alguna forma).
- Lectura de operandos (*Read Operands*): Leer los operandos disponibles (no tienen RAW cercana) y esperar por los operandos que tienen dependencia.

➢Todas las instrucciones pasan en orden por la etapa ID o ISsue, y a partir de aquí pueden ser detenidas o desviadas a la unidad de ejecución correspondiente. (La etapa Issue no puede desordenarse, porque entonces no se sabrían cuáles son las dependencias).



- Apuntar la información de dependencias en algún sitio:
  - Si la anotación es una tabla central, el algoritmo se llama (1964) marcador centralizado (*scoreboard*).
  - Si la anotación es distribuida, el principal algoritmo es el de R. Tomasulo (1967).

Algoritmo de Tomasulo.

- Técnica de planificación dinámica de instrucciones con gestión distribuida. Inicialmente, en el IBM/360 (1967, Robert Tomasulo) era sólo para operaciones de FP. Hoy se aplica a todas las instrucciones, y la mayoría de procesadores avanzados llevan un algoritmo similar al clásico (estudiaremos el clásico pero para todo tipo de instrucciones, porque es el explicado en los libros generalmente).
- Este algoritmo se puede aplicar a otro tipo de sistemas donde hay ciertos recursos compartidos por diversos agentes, y se quieren gestionar los recursos de forma distribuida. Evidentemente la implementación que veremos aquí es hardware.
- Durante su ejecución provoca un reordenamiento dinámico en la ejecución (fase EX) de las instrucciones, aunque la emisión sigue siendo en el orden del código original.
- Permite la ejecución de instrucciones no dependientes de otras anteriores (aunque estas últimas estén bloqueadas esperando por algún operando fuente).
- Va a realizar un renombrado dinámico de registros para evitar riesgos por dependencias (WAR, WAW) entre instrucciones. Todos los registros de destino se renombran y el nuevo nombre que toman se llama **etiqueta** (*tag*). Ver transparencia 26.

Definiciones

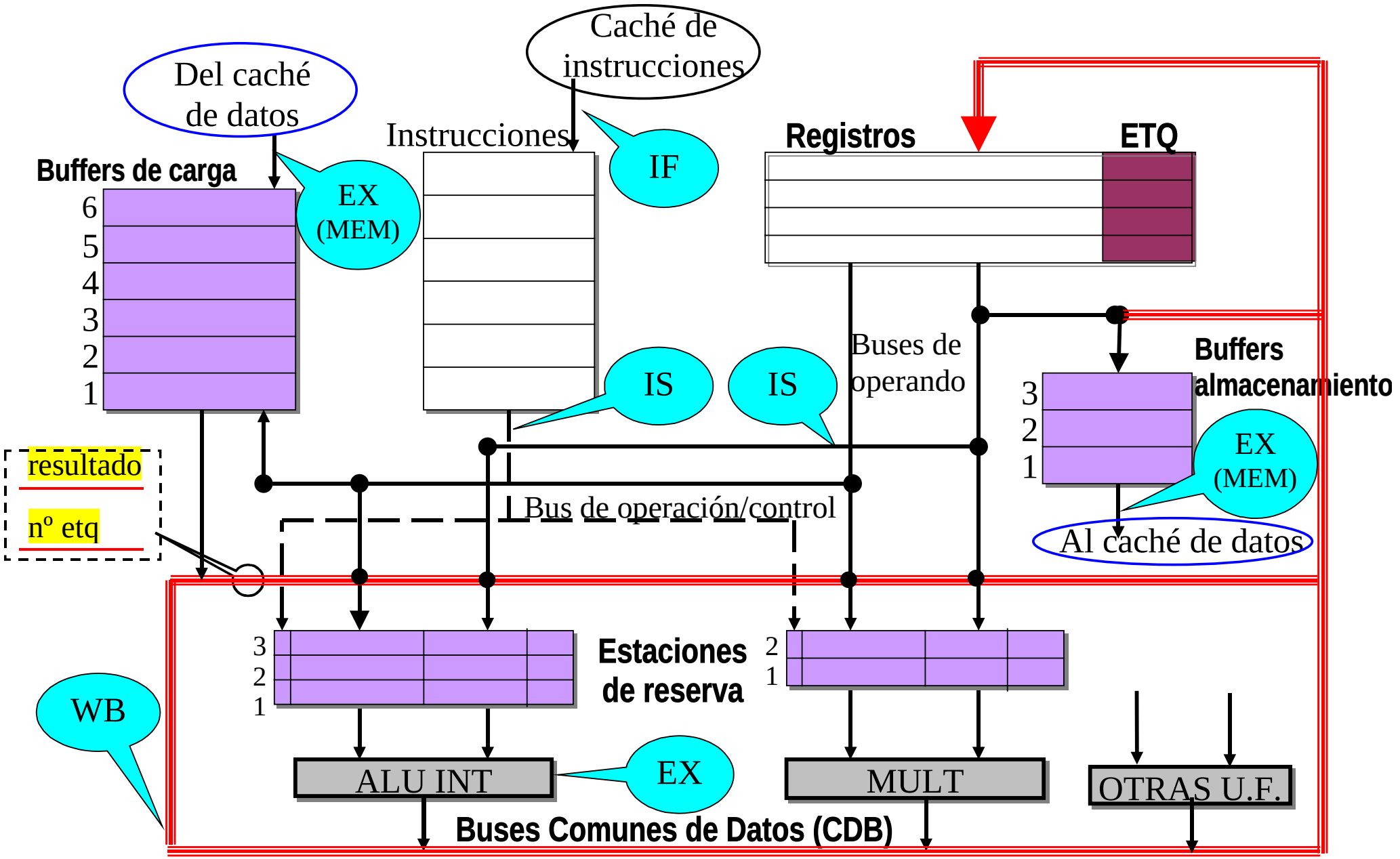
En este algoritmo se ponen una serie de entradas en cada UF, llamadas estaciones de reserva RS. A éstas llegan las emisiones (IS) de instrucciones con toda la información de la misma, también llegan los valores de los operandos fuente, si han podido leerse, o cierta información (etiqueta) que indica que operando falta (no se bloquea la cadena).

- ***Estación de Reserva (Reservation Stations, R.S.)***: Lugar dónde esperan las instrucciones emitidas junto a las ALU's (U.F.), hasta que pueden ejecutarse en la U.F. Esta espera se debe a la no disponibilidad de todos los operandos fuente (hay RAW: una instrucción anterior aún no ha generado algún operando). La R.S. tendrá una serie de campos para anotar el valor de los operandos fuente (y si es necesario el valor del resultado de la operación y la etiqueta asociada al registro destino).

  En la implementación del alg. Tomasulo, las estaciones de reserva (RS) son registros asociados a las unidades funcionales. De manera que forman una pequeña cola de instrucciones a la entrada de la unidad funcional (de unas pocas por UF si la operación no es muy común, como DIV, a muchas si es muy frecuente). Cada registro contendrá la información necesaria para la ejecución de la instrucción y para la llegada de los operandos que le falten (en caso de que la instrucción esté en espera).

- ***Buffers de Memoria***: Son las R.S. para instrucciones de carga/almacenamiento. Es el lugar dónde esperan los acceso a memoria emitidos hasta que disponen de todos sus operandos y el bus de memoria (caché) está libre. En el alg. clásico y en general, hay buffers separados para instrucciones de carga y para los de almacenamiento.
- ***Bus Común de Datos (Common Data Bus, CDB)***: Une la salida de las unidades funcionales y la carga de memoria con el fichero de registros, las estaciones de reserva (y los buffers). Las UF mandan el dato de salida a través de él para que lo lean el resto de elementos de la CPU (es decir, se usa en la fase WB, produciéndose un bypass). Se trata de un recurso común por el que hay que competir. En general, las máquinas suelen tener más de un CDB (p.ej. uno para enteros y otro para FP) puesto que el número de WB por cada ciclo puede ser mayor que 1. Cualquier CDB tiene dos buses: uno para el resultado de una operación (32 líneas para INT/float, 64 para double) y otro para la etiqueta asociada al mismo (tamaño $\log_2$ del número de etiquetas).
- ***Etiqueta (tag)***: Son identificadores que se asocian a los registros destino cada vez que se emite una instrucción. A partir de renombrar un registro destino con una etiqueta, es ésta la que circula por toda la CPU (estaciones de reserva, buffers, el fichero de registros, CDB, etc.). Una etiqueta estará en uso desde que la instrucción se emite hasta que su resultado es puesto en el CDB. En el alg. clásico el nombre de la etiqueta coincide con el de la R.S. Por ej. si la UF de MULT tiene 3 R.S., entonces las etiquetas serán MULT1, MULT2, MULT3 (codificadas en binario).

Esquemático Algoritmo Tomasulo clásico



Etapas Pipeline

- Se puede definir la especificación exacta (en lenguaje C o con simuladores) del algoritmo de Tomasulo. Esto se implementa luego en hardware.

  Dado que con la arquitectura hardware del algoritmo de Tomasulo la cadena no se bloqueará cuando haya RAW, la cadena (unidad de control) debe cambiar.

  - **Fetch (IF):** Acceso a caché de instrucciones para leer instrucción apuntada por PC
  - **Issue (IS):** Emisión de la instrucción. En el alg. clásico en esta etapa se hace en un solo ciclo (hoy en día esta etapa dura varios ciclos de reloj). Realizaría todo esto:
    - Decodificación
    - Lectura de operandos ya disponibles
    - Envío hacia la R.S. (esta sería la auténtica emisión).

    Al final de IS, se intenta enviar una instrucción a una R.S. (o buffer de carga/almacenamiento), asignando una etiqueta al reg. destino.

    - Si hay una RS. (o buffer) libre en la UF requerida por la operación: Se emite la instrucción mandando a la RS los operandos disponibles. Para los que no estén disponibles, se anota la etiqueta por la que esperan.
    - Si no hay RS (o buffer) libre o no quedan etiquetas libres, la etapa IS se bloqueará hasta que una RS de la UF afectada se libere. En el alg. clásico (y siempre que todo se realice en una única fase IS) esto provoca una detención de la cadena de ejecución.

- **Execution (EX):** Cuando una RS (buffer) obtuvo todos sus operandos, empieza a ejecutarse. La RS puede quedar libre:
  - Al comenzar a ejecutarse en la UF (primer EX1). Entonces, esta etapa conservará en algún otro sitio dos campos: uno para el resultado y otro para la etiqueta asociada al valor de salida (se envía en WB junto al resultado de la operación).
  - O al final de WB. La R.S. tendrá todos los campos necesarios para operandos fuente, resultado y etiqueta. Por motivos que veremos en ASP2, esto suele ser lo normal y será el criterio para ASP1 (mientras no se indique lo contrario).
- **Write (WB):** El resultado de una operación y su etiqueta se escriben en el CDB, de manera que las RS o buffers que estén esperando por este valor (tienen la misma etiqueta en alguno de sus operandos) lo toman, es decir, se produce un bypass o camino de desvío a través del CDB. Igualmente el valor del CDB se escribe en el fichero de registros FP si su etiqueta coincide con alguno. Tras la escritura la etiqueta se libera. El fichero de registros deberá, por tanto, tener información sobre la última etiqueta que va asociada a cada registro.

NOTA: fase MEM no existe (se considera dentro del EX para los buffers de memoria).

Especificación de Arquitectura de Tomasulo

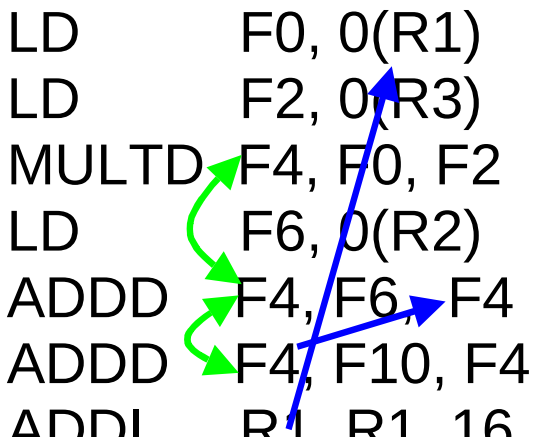
- Campos de cada Estación de Reserva (según versión, algunos son prescindibles):
  - Op: operación
  - Qj, Qk: Etiqueta de los operandos j y k. Si está vacío, el operando está disponible.
  - Vj, Vk: Valores de los operandos.
  - Ocupado: Indica que la RS está en uso.
- Buffer de LD:
  - Dirección: para el acceso a memoria (etiqueta si oper. no disponible ó valor oper.)
  - Ocupado: Indica que el buffer está en uso.
- Buffer de ST:
  - Qi: Etiqueta de la RS que producirá el valor a almacenar en memoria.
  - Dirección: para el acceso a memoria (etiqueta si oper. no disponible ó valor oper.)
  - Vi: valor a almacenar. Estará vacio cuando Qi esté lleno.
  - Ocupado: Indica que el buffer está en uso.
- Fichero de Registros FP. Cada registro tendrá la estructura:
  - Qi: Etiqueta de la RS que producirá el valor a almacenar en el registro.
  - Vi: valor del registro.
  - Ocupado: Indica que el valor actual del registro no es válido (espera por el nuevo).

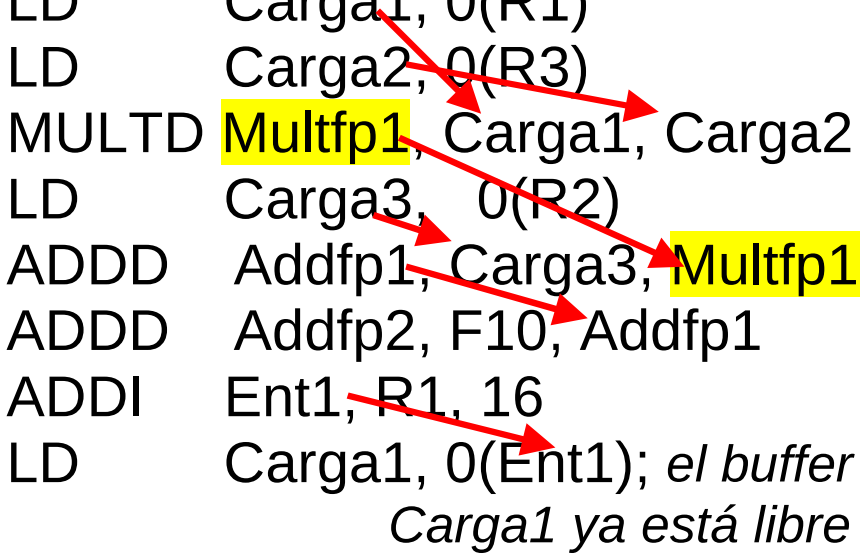
Resolución de riesgos

Con el alg. Tomasulo, se eliminan los riesgos por dependencias de datos. Sólo existe bloqueo (de tipo estructural) cuando se agotan las R.S. o las etq. $CPI_{datos} = 0$

- RAW: las anota en las estaciones de reserva y espera por las etiquetas.
- WAW: al renombrar registros, nunca se les asocia la misma etiqueta a dos registros de destino que aún estén "circulando" por la CPU.
- WAR: al renombrar registros para evitar la WAW, la WAR desaparece.

Ejemplo de renombrado: etiquetas se llaman como R.S: (de las cuales hay infinitas). Se supone que los reg. R1, R2, F0,... no estaban "en uso" antes de este código. Notar que sólo quedan las RAW reales, que van formando el grafo de dependencias.





Ejemplo 1

Cronograma del fragmento: Suponemos 3 RS suma, 2 RS multiplicación y sólo 2 de carga. (Carga3 no existe, deberá usar Carga1 otra vez y esperar a que esté libre)

Duración Ld/St : 2ciclos (como EX+MEM). MULTD: 4 ciclos, ADDD: 2 ciclos

```
LD      F0, 0(R1)
LD      F2, 0(R3)
MULTD   F4, F0, F2
LD      F6, 0(R2)
ADDD    F4, F6,  F4
ADDD    F4, F10, F4
ADDI    R1, R1, 16
LD      F0, 8(R1)
```

Notar el reordenamiento dinámico de las fases EX. Típicamente las INT se anticipan a las FP.

Mostrar el estado de las RS y de los registros, tras emitir la última suma flotante.

Ejemplo 2: Bucles con Tomasulo

Sea el procesador DLX con algoritmo de Tomasulo con las siguientes características:

- 1 UF de operaciones enteras simples de un ciclo de duración, con 2 RS.
- 1 UF de Ld/St duración 2 ciclos, con buffers indep. de LD y ST, de 3 entradas.
- 1 UF de operaciones FP, 4 ciclos para sumas y mult., con 3 RS.
- 1 UF de saltos de un ciclo de duración, con 1 RS.
- Existen 2 CDB, uno para operaciones enteras, y otro para punto flotante.

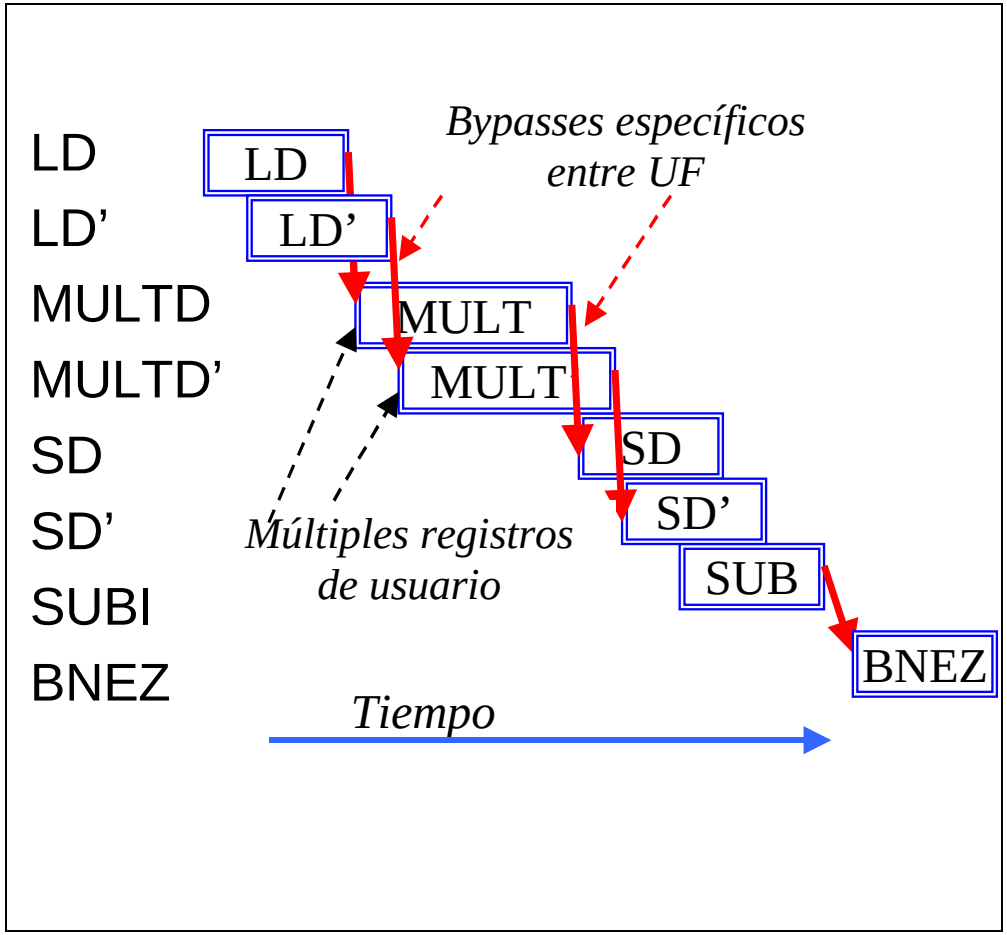
Cronograma del código para tal DLX. Suponer BTB accede en IF y siempre acierta.

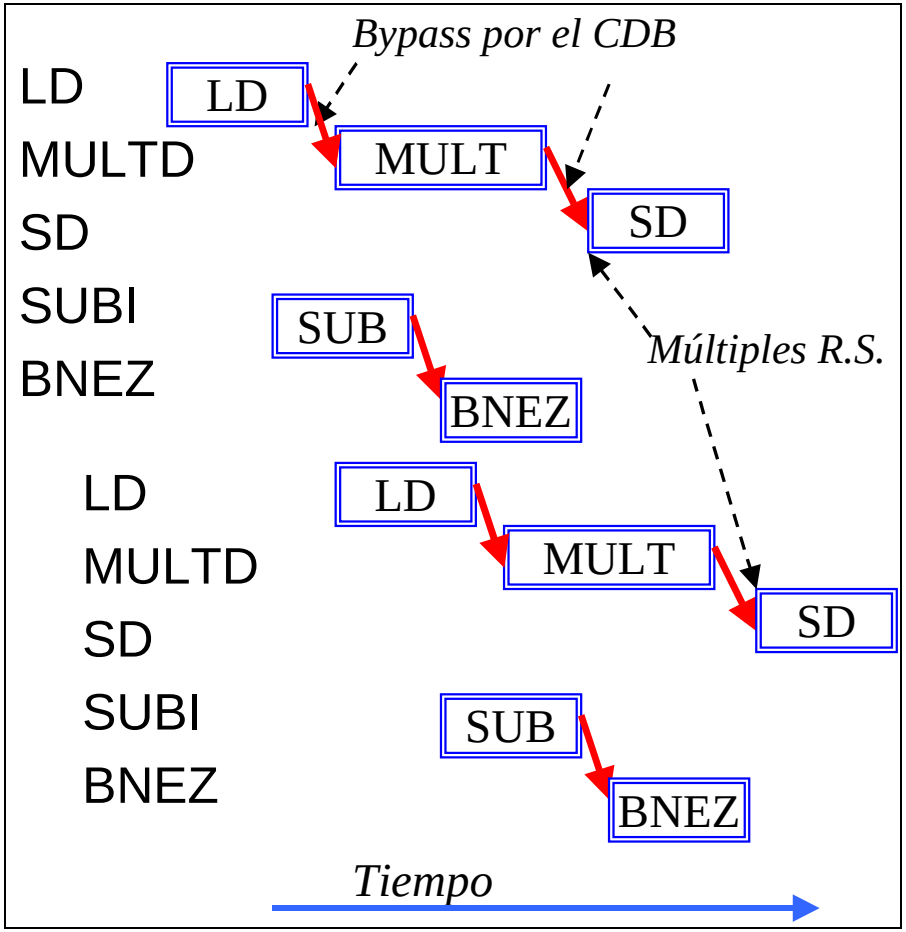
```
Loop:  LD     F0, 0(R1)
       MULTD  F4, F0, F2
       SD     0(R1), F4
       SUBI   R1, R1, 8
       BNEZ   R1, Loop
```

Ejecutar dos iteraciones seguidas y comprobar el desenrollado dinámico.

NOTA: Desenrollar para una máquina con alg. Tomasulo puede provocar bloqueos por agotarse las RS por lo que la cadena se bloquearía y se empeoraría el CPI.

Croquis: diferencia entre planificación estática y dinámica para bucle paralelizable (sólo flujo de datos)





Desenrollado estático

Desenrollado dinámico:

- Tb. conviven varias iteraciones a la vez
- Adelantar emisión de instr de 2ª iter ⇒ mejor

Variaciones del Alg. Tomasulo clásico

La implementación del algoritmo de Tomasulo en las máquinas actuales sigue la misma idea principal del algoritmo visto, pero se puede encontrar con distintas modificaciones o adaptaciones:

- Antiguamente se aplicaba el algoritmo de Tomasulo sólo para instrucciones largas (típicamente FP, código para supercomputadores científicos). De forma que la cadena era distinta para FP que para enteros. Actualmente las operaciones enteras también llevan planif dinámica similar al algoritmo de Tomasulo. A veces las secciones FP e INT están separadas y otras en común.
- En el algoritmo clásico etiqueta y R.S. son lo mismo. Se supone una etiqueta por RS, luego para poder emitir, el requisito es único: ¿hay RS disponible? Cuando la UF termina la ejecución, se devuelve el resultado a la RS. La RS pondrá el resultado en el CDB cuando este está libre, quedando libre la R.S. (y la etiqueta). Nosotros usaremos el algoritmo clásico **(libro Henn-Patt)** mientras no se diga lo contrario.
- Arquitectura con RS que se liberan al empezar la fase EX (cuando todos los operandos fuente están disponibles). Cuando la UF termina la ejecución, se recoge el resultado de la operación en otro registro interno. En cuanto haya un CDB libre se libera tal registro interno.

- Arquitectura con "piscina de etiquetas" (*tagpool)*. Las etiquetas son un recurso común y compartido. Para emitir una instrucción, requisito doble: debe existir una etiqueta disponible y una RS libre (la etiqueta tiene asociado otro elemento hardware). El Reg destino se renombra con la etiqueta (y no coincide con la R.S.).
- Un ejemplo usual de piscina de etiquetas donde las RS se liberan en EX, se da en los micros que renombran, dinámicamente y para todas las instr. emitidas, los registros lógicos (de usuario, de las instr. de ensamblador) por registros físicos (internos, no visibles al programador). Los registros físicos funcionan como etiquetas. Suele haber muchos más registros físicos que lógicos. Ej Power PC, MIPS
- El conjunto de RS puede ser: *a)* común para todas las UF, *b)* separadas en RS INT y RS FP, *c)* cada UF tenga sus RS asociadas y propias (como Tomasulo clásico).
- Todos tienen varios CDB para permitir más de una escritura al mismo tiempo.
- En algunos micros la fase WB no existe, en el último ciclo de ejecución se escribe en el bus común (como se hace en el DLX sin planif dinámica, el desvío de un dato por un bypass no tarda ningún ciclo). Así se acercan al límite flujo de datos.
- La cadena tiene diferente número de etapas (no es la clásica IF IS EX WB). Generalmente la fase IS se subdivide en varias subfases, pues hay que hacer muchas operaciones: decodificación, lectura operandos, renombrado, emisión, etc.

Dependencias de Memoria

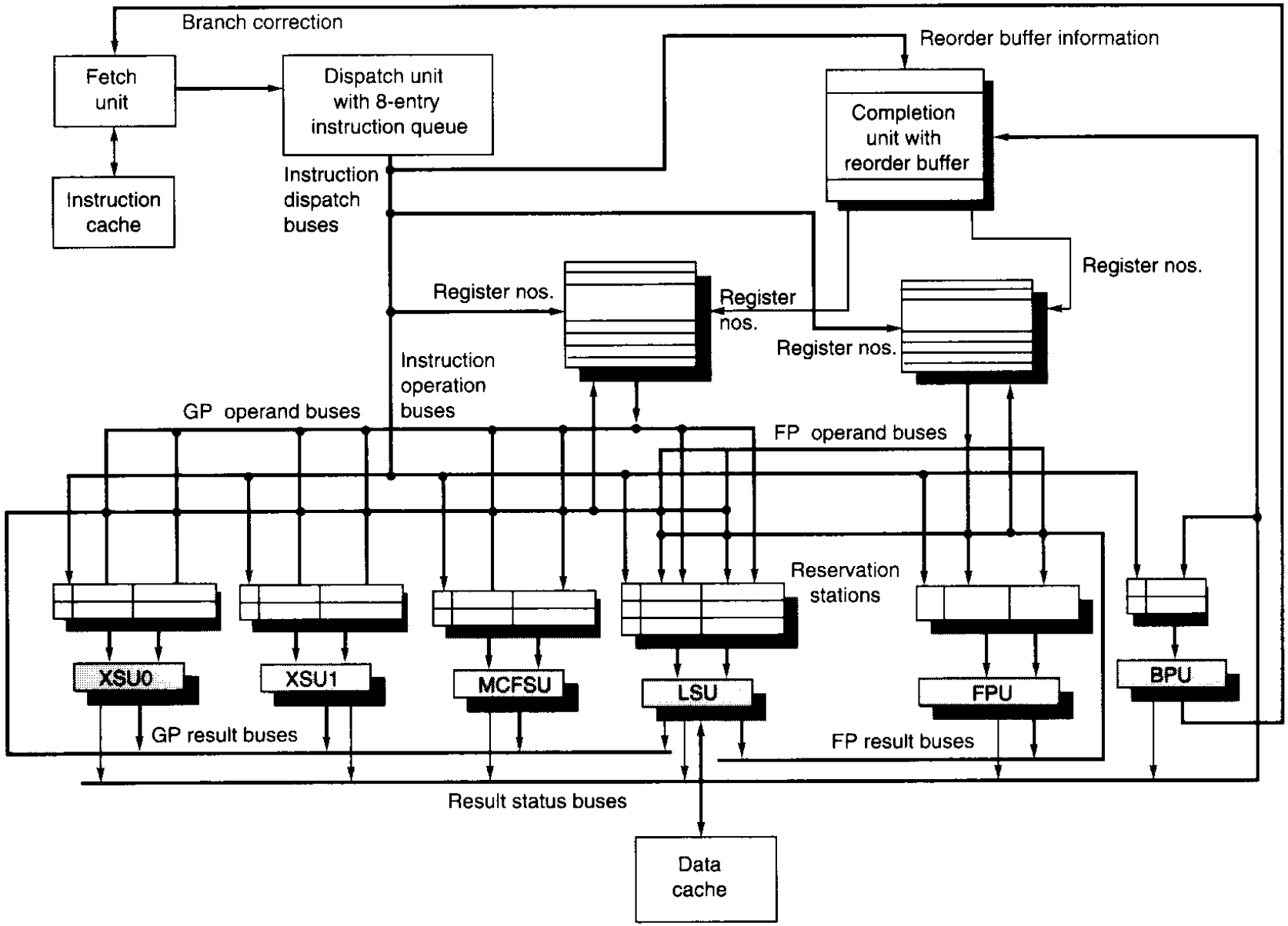
- Los accesos a memoria (ejecutados en búferes de carga y de almacenamiento) pueden tener dependencias entre los datos que leen o escriben en memoria.
- Para ello debe ocurrir que la dirección de un acceso sea igual a la de otro. Evidentemente esta dependencia no se puede detectar en t. de compilación. Ej:
  - `SW  20(R3), R9`
  - `LW  R1,  8(R4)` Si ocurre que: 20+R3 = 8+R4 ⇒ RAW en memoria
- Ej: Antes de que un Load acceda al caché, se debe comparar su dirección efectiva con todas las de los búferes de almacenamiento. Si hay coincidencia ⇒ se detecta una dependencia real en memoria, y hay que esperar a que el Store termine. De no existir tal comparación de direcciones, no se podría ejecutar un Load antes de que acabaran todos los Store (caché actualizado). Y los Store suelen ser las últimas instrucciones en ejecutarse (ver cronogramas, p ej. bucle del Ejemplo 2, pág 29).
- Los búferes de Store son como un *buffer de escritura* (conserva dirección y dato escrito). Los Store se guardan en el buffer y se dan por realizados. Las Lecturas son más prioritarias (hay otras instr. esperando por sus datos).
- En procesadores con pocos registros de usuario (CISC) el código tendrá muchos accesos a memoria. El riesgo es mayor y se necesita más hw para resolverlo.

Tomasulo en procesadores actuales

- En general los micros con planif dinámica son capaces de emitir y ejecutar varias instrucciones a la vez, y su arquitectura es muy potente.
- Uno de los primeros microprocesadores con planif dinámica fue Motorola **PowerPC 620** (1995). Es un ejemplo muy bueno: su cadena (4 fases: IF ID IS EX) y arquitectura es muy similar al algoritmo clásico Tomasulo. Poseía unas pocas R.S. por cada UF.
  - Dos unidades enteras simples (1 ciclo).
  - Una unidad entera compleja (3 a 20 ciclos) para MULT y DIV enteras.
  - Una unidad de carga-almacenamiento (1 ciclo si acierta en caché).
  - Una unidad de punto flotante (31 ciclos DIVFP, 2 ciclos ADDFP y MULTFP).
  - Y una unidad de saltos, que actualice BTB en caso de fallo de predicción.
- **Pentium Pro** (su parte INT es idem que P.II, P.III y similar al P4). Su cadena es de 11 fases. Posee 40 R.S. comunes a todas las UF:
  - Dos unidades enteras (1 ciclo). Una de ellas se usan para saltos (actualiza BTB en caso de fallo de predicción). La otra se usa también para multiplicaciones y divisiones enteras.

  - Dos unidades FP para multiplicaciones y divisiones de punto flotante. Realmente solo puede empezar en el mismo ciclo una de ellas.
  - Una unidad de carga (1 ciclo).
  - Una unidad de almacenamiento más compleja (lleva un buffer MOB especial).
- **MIPS R10000** (similar a los actuales). Posee 16 R.S. para INT, 16 para FP y 16 para Ld-St:
  - Dos unidades enteras (una también sirve para saltos, y la otra para MULT INT).
  - Una unidad para cálculo de direcciones de acceso a memoria (para carga-almacenamiento).
  - Una unidad de punto flotante simple (sólo para ADDFP).
  - Una unidad de punto flotante compleja (DIV, MULT, SQRT-FP).
- Las unidades funcionales suelen ser segmentadas excepto en las divisiones.

Ej: Power PC 620 (similar al Tomasulo clásico)



Rendimiento del Scheduling Dinámico

- Dotar de muchas RS a una UF de corta duración no tiene sentido porque se liberan rápidamente. Aunque cuantas más dependencias reales existan menos se liberarán y más R.S. se necesitan.
- Pero las U.F. de larga duración suelen ser de instr poco comunes (como DIV). Luego tampoco se requieren muchas RS. Hay que calcular el nº RS en función de todo esto y no excederse en RS, puesto que:
- Principal inconveniente del Alg. Tomasulo: alto coste hardware, sobretodo en lógica de comparación de las R.S y búferes acceso a memoria. Cada R.S debe comparar la/s etiqueta/s por la/s que espera con la/s que aparece/n en el/los CDB.
- La aceleración que se consigue para un procesador DLX pipeline con algoritmo de Tomasulo es de 1.58, con respecto al que no tiene planificación dinámica (sí estática del compilador). No es muy alta porque el DLX tiene muchos registros, lo que permite reducir el número de antidependencias y por tanto reordenar mejor.
- La aceleración sería mucho más alta comparando para una máquina con pocos registros.

3.5. TÉCNICAS SOFTWARE AVANZADAS

Existe una gran variedad de casos donde métodos simples de planificación estática no son aplicables directamente. Veamos algunos ejemplos típicos e ilustrativos

- Los bucles de vectores son paralelizables. Pero no todos los bucles son paralelizables. P. ej. cuando hay una dependencia entre la iteración actual `i` y las anteriores, no podrían entrelazarse las instr de las diferentes iteraciones. Típico ej. de dependencia: Serie de Fibonacci (suponemos que la operación suma es larga):

```
for (i=2;i<M; i++) x[i]=x[i-1]+x[i-2]; //(2 Load, 1 Store)/1 elem
// Si desenrollo:
```

```
for (i=2;i<M; i+=3) {
        x[i]=x[i-1]+x[i-2];
        x[i+1]=x[i]+x[i-1];
        x[i+2]=x[i+1]+x[i]; } // En total: (2 Load, 3 Store)/3 elem
```

- Por supuesto siempre se obtiene cierta aceleración porque se eliminan instrucciones de overhead (e incluso algunas de acceso a memoria si el compilador es capaz).
- En general cuando en una serie hay recurrencia: $a_i=f(a_{i-k})$ ó $a_{i+k}=f(a_i)$, para algunos k
- Pero si k es grande puedo desenrollar k veces, con lo cual las depend. RAW se pueden alejar (entrelazando las k iteraciones):

```
for (i=0;i<M; i++) x[i+4]=x[i]*s;  //Si desenrollo 4 iteraciones:
```

```
for (i=0;i<M; i+=4) {
        x[i+4]=x[i+0]*s;
        x[i+5]=x[i+1]*s;
        x[i+6]=x[i+2]*s;
        x[i+7]=x[i+3]*s; }
```

- Si un bucle tiene varias líneas de código con posibles dependencias internas, conviene separar cada línea en bucles diferentes y luego comprobar si existen dependencias. Finalmente desenrollar si ya se puede. Ej:

```
- for (i=0;i<M; i++) {
        x[i]= x[i]* y[i];
        y[i+1]= z[i]* s;
        }   //parece que existe una dependencia entre iteraciones:
```

```
- for (i=0;i<M; i+= 2) {
        x[i]= x[i]* y[i];
        y[i+1]= z[i]* s;
        x[i+1]= x[i+1]* y[i+1];
        y[i+2]= z[i+1]* s;
        } // es ficticia si cambio el orden y separo en dos bucles:
```

```
- for (i=0;i<M; i+=2) {
        y[i+1]= z[i]* s;
        y[i+2]= z[i+1]* s;
        x[i]= x[i]* y[i];
        x[i+1]= x[i+1]* y[i+1];
        } // Luego finalmente (el orden contrario sería incorrecto):
```

```
  for (i=0;i<M; i++) y[i+1]= z[i]* s;
  for (i=0;i<M; i++) x[i]= x[i]* y[i];
```

- Otro ej de bucle muy usual: sumatorio (idem para producto múltiple). En principio existe dependencia entre una iteración y la siguiente a través de la variable acumuladora. Pero tal variable ha surgido de la forma en que se escribe el sumatorio. Si lo rescribo, desaparece. Por ej con varias variable acumuladoras (ejercicio: otra forma sería escribiendo un sumatorio con algoritmo tipo árbol):

```
- for (i=0;i<M;i++) t = t + y[i];
```

```
- for (i=0;i<M; i+=3){
        t0 += y[i+0];
        t1 += y[i+1];
        t2 += y[i+2];
```

```
            }
    t = t0 + t1 + t2;
//Ahora puedo ejecutar estas 3 iteraciones en paralelo (y entrelazarlas)
// quedando las dos sumas de los resultados parciales al final
```

Ejercicio: Desenrollar el Producto escalar de dos vectores

- Encadenamiento software de iteraciones (*software pipelining*). Cuando en un bucle paralelizable no se puede desenrollar porque no hay registros suficientes en el procesador (CISC), o porque no interesa tamaño grande de código, se pueden alejar las dependencias reales rescribiendo el código con esta técnica.

Sea el ejemplo del apartado 3.2, 3.4: `for (i=0 ; i<M ; i++ ) y[i]= x[i]*s;`

```
bucle_orig:
    LD     F2, 0(R1)
    MULTD  F4, F2, F24  ; F24 contiene el valor de s
    SD     (R3)0, F4
    ; instr overhead a continuación (increm. punteros, saltos, comp.)
    ADDI   R1, R1, 8
    ADDI   R3, R3, 8
    SLTI   R7, R1, fin_array_x; constante apunta al final de x[M]
    BNEZ   R7, bucle_orig
```

Se ve claro con un esquema en la siguiente tabla:

iter 0	LD^0	MULTD^0	SD^0	instr overhead^0			
iter 1		LD^1	MULTD^1	SD^1	instr overhead^1		
iter 2			LD^2	MULTD^2	SD^2	instr overhead^2	
iter 3				LD^3	MULTD^3	SD^3	instr overhead^3
iter 4					LD^4	MULTD^4	SD^4

← *instrucciones de arranque* →← *iter. reordenadas*→

De la traza de ejecución normal por filas en la tabla (flechas rojas), cambio a la ejecución por columnas (flechas verdes).

El código resultante tendría unas cuantas de instrucciones de arranque o "start-up" (en este ejemplo las tres primeras columnas) y unas cuantas de instrucciones de clausura o "finish-up" (serían las últimas 4 columnas). El resto de columnas intermedias se pueden ejecutar de arriba abajo, habiendo alejado las dependencias reales (hay que retocar las constantes para que sea correcto).

Sólo quedan antidependencias que no producen ciclos de bloqueo. Para que el salto condicional esté abajo, cambio de posición las instr de overhead (lo que conlleva retocar las constantes de los Ld/St):

;instr overhead^0
SD^1
MULTD^2
LD^3
	bucle_encadenado:
	;instr overhead^0
	SD^1 (R3)0, F4
	MULTD^2 F4, F2, F24
	LD^3 F2, (R1)2*8
	// esta columna no es correcta por que el salto estaría arriba
		// código arranque
		bucle_encadenado:
		SD^1 (R3)8, F4
		MULTD^2 F4, F2, F24
		LD^3 F2, (R1)24
		;instr overhead^0
		// código clausura

Los desplazamientos de los Ld/St se han calculado en función de la iteración a la que pertenece cada Ld/St. En la tercera columna se han cambiado los desplazamientos (y la cte de SLTI, que sería `fin_array_x-24` para que LD F2,(R1)24 no se salga del array) porque las instr. overhead se han movido.

- Notar que las RAW se han convertido en WAR (que no producen bloqueos).
- Notar que no puede haber WAW o WAR en el bucle original (al dar la vuelta al orden de las instr. se podrían convertir en dependencias reales erróneas!)

3.6 CONCLUSIONES: PLANIFICACIÓN ESTÁTICA VS. DINÁMICA. RESUMEN: PROS Y CONTRAS

PLANIFICACIÓN ESTÁTICA	PLANIFICACIÓN DINÁMICA
Menos Hardware	Complicación hardware
Compilador más difícil	Compilador no tiene que optimizar
Posibles problemas de herencia (compilador debe conocer endoarquitectura)	Transparente al usuario
Inconveniente: Dependencia compilación-rendimiento	El hardware extrae el rendimiento que puede en cada versión
Tamaño de código estático puede crecer $\Rightarrow$ más fallos de caché	Tamaño de código estático no se toca
Ventaja: Ventana de instrucciones infinita (análisis global)	Defecto: Ventana de instrucciones limitada (fase IF) (análisis local)
El compilador no puede conocer: - valores de registros (dir. acceso) - predicción dinámica, etc.	En tiempo de ejecución se conoce más: - valores de registros (dir. acceso) - predicción dinámica, etc.

Puede eliminar instrucciones (de overhead u otras)	No puede eliminar instrucciones
Puede necesitar muchos registros de usuario	No necesita muchos registros de usuario (son internos, ocultos al usuario; ej. CISC)

- Conocimiento de programas reales (o los que se van a ejecutar) es muy importante antes de ponerse a diseñar una máquina (o a elegir una)
- Dos Tendencias:

  Hw simple y Compilador complejo vs. Hw complejo y Compilador Simple

  Esta disyuntiva se está dando actualmente con micros avanzados (se verá en ASP2) aunque cada vez se traspasa más funcionalidad al hardware.