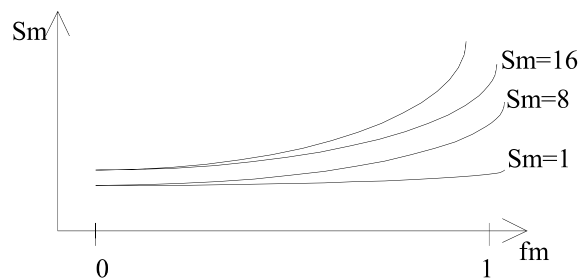
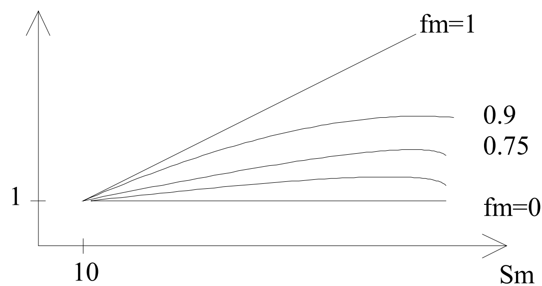
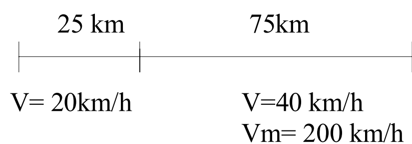
fm -->	0	0.25	0.5	0.75	1
Sm					
1	1	1	1	1	1
2	1	1.14	1.3	1.6	2
4	1	1.23	1.6	2.29	4
8	1	1.28	1.78	2.9	8
16	1	1.31	1.88	3.37	16
32	1	1.32	1.94	3.66	32
128	1	1.33	1.98	3.91	128
1024	1	1.33	1.99	3.99	1024
∞	1	1.3	2	4	∞





Ejemplo: Calcular la aceleración global lograda al incorporar la mejora de:



$$\text{Toriginal} = \frac{25km}{20km/h} + \frac{75km}{40km/h} = 1.25 + 1.875 = 3.125h.$$

$$\text{Tmejora} = \frac{25km}{20km/h} + \frac{75km}{200km/h} = 1.625h.$$

Fracción de mejora: $fm = \frac{1.875}{3.125} = 0.6$

Aceleración (speed-up) : $S = \frac{Toriginal}{Tmejora} = \frac{3.125}{1.625} = 1.92$

- Ademas:

Aceleración máxima o límite:

- se produce cuando la Vm = ∞ ---> $Tmin = \frac{25}{20} + 0 = 1.25$

por tanto: $S_{limite} = \frac{3.125}{1.25} = 2.5$

Si aplicamos la ley de Amdahl:

Sm = 200 / 40 = 5

$$Stotal = \frac{1}{(1 - fm) + \frac{fm}{Sm}} = \frac{1}{(1 - 0.6) + \frac{0.6}{5}} = 1.92$$

Ejemplo:

- El CPI para todas las instrucciones de una máquina es de 1.4, excepto para las de salto que tardan 3 ciclos en ejecutarse. Teniendo en cuenta que hay un 20 % de instrucciones de salto, calcular:

a) CPI de la máquina.

b) Si modificamos las instrucciones de salto para que tarden 2 ciclos, ¿cual es el % de mejora?.

c) ¿Cual es el Speed-up de la mejora?

d) Aceleración global usando la ley de Amdahl.

a) CPI = CPI normal * 0.8 + CPI salto * 0.2 = 1.4 * 0.8 + 3 * 0.2 = 1.72

b) En este caso la nueva arquitectura es X % mejor (más rápida) que la original.

CPI = 1.4 * 0.8 + 2 * 0.2 = 1.52

$$X = \left(\frac{Toriginal - Tmejora}{Tmejora} \right).100 = \left(\frac{CPI - CP\mathrm{Im}ejora}{CP\mathrm{Im}ejora} \right).100 =$$

$$= \left(\frac{1.72 - 1.52}{1.52} \right).100 = 13.15\% mejor$$

c) $$S = \frac{CPIoriginal}{CP\mathrm{Im}ejora} = \frac{1.72}{1.52} = 1.13$$

d) $$Sm = \frac{CPIsalt.orig.}{CPIsalt.mej.} = 3/2 = 1.5$$

$$fm = \frac{CP\mathrm{Im}ejora}{CPI} = \frac{0.2 * 3}{1.72} = 0.349$$

$$Stotal = \frac{1}{(1 - fm) + \frac{fm}{Sm}} = \frac{1}{(1 - 0.349) + \frac{0.349}{1.5}} = 1.13$$

<u>PROBLEMAS DE LA SEGMENTACIÓN</u>

- Lo que pretende la segmentación es conseguir el máximo rendimiento, 1 operación por ciclo <---> CPI = 1. Pero como ya hemos visto existen una serie de problemas que nos impiden conseguirlo:

A) Riesgos estructurales

B) Riesgos de datos

C) Riesgos de control

- Estos problemas tendremos que mirarlos e intentar solventarlos en cada tipo de procesador que veamos.

A) RIESGOS ESTRUCTURALES

- Se dan cuando puede haber un conflicto en la utilización de un recurso (UF,BUS,REG,...) por parte de diferentes instrucciones en el mismo ciclo. Tanto puede ser en la misma etapa como un recurso compartido por varias etapas.

- Pueden aparecer:

i) Dos etapas diferentes usan el mismo recurso

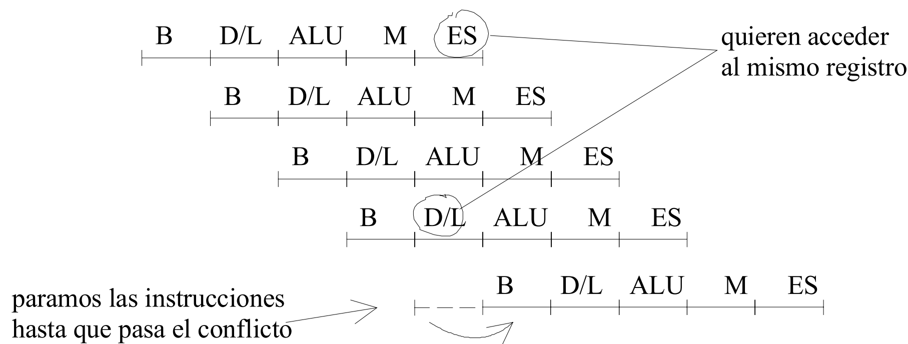
Ejem: - la memoria se usa en 2 etapas B,M.
- En varias etapas se puede escribir en el banco de registros y sólo hay un bus de escritura (o acceder a lec./esc. y un único bus.

ii) Una unidad funcional no está totalmente segmentada ---> no se puede iniciar una operación por ciclo ya que tiene un L>1 y ya está segmentada.

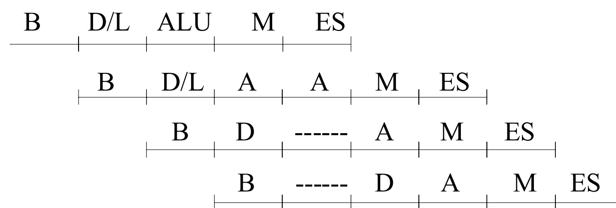
SOLUCIONES

- A) BLOQUEAR el procesador hasta que desaparezca el problema. Esto quiere decir parar las instrucciones siguientes a la del conflicto hasta que desaparezca el problema. Después continuar.

i) Mismo hardware.



ii) UF no segmentada ---> operación en la ALU tarda más.



- B) REPLICAR LOS RECURSOS

Ej: Memoria con 2 partes o cache separada para instrucciones y datos.

- C) SEGMENTAR LAS UFS

B) RIESGOS DE DATOS

- Se producen cuando la orden de acceso a los operandos y de escritura de resultados se modifica por culpa de la segmentación, respecto al orden norma del ejecución secuencial.

- Concretamente habrá riesgo de datos cuando existen <u>dependencias de datos</u> entre instrucciones suficientemente próximas en el tiempo como para que se ejecuten concurrentemente de forma que se modifica la orden correcta de acceso a las variables.

Dada una secuencia de código:

Si
.
.
.
Sj
.
.
.
Sk

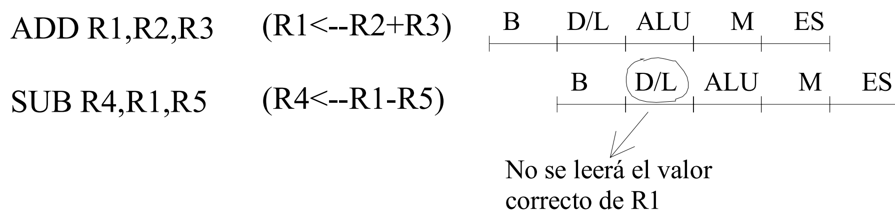
- Hay una dependencia si la ejecución de la instrucción 'Sj' se usa una variable que también se utiliza en la instrucción 'Si'.

- Para toda instrucción distinguiremos:

  - DOMINIO: operandos de la instrucción (Datos de entrada) D(i)

  - RANGO: Resultado de la instrucción R(i).

Ejem.

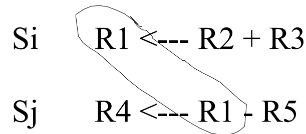


TIPOS DE DEPENDENCIAS DE DATOS

Si
.
.
Sj

- RAW (Read After Write) ; Ri $\cap Dj \neq 0 \rightarrow$ True Dependence

Ejem:

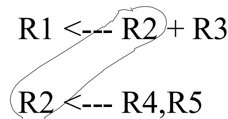


Si R1 <--- R2 + R3

Sj R4 <--- R1 - R5

- WAR (Write After Read); Di $\cap Rj \neq 0 \rightarrow$ Antidependence

Ejem:

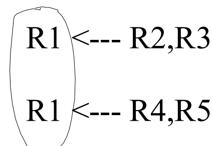


R1 <--- R2 + R3

R2 <--- R4,R5

- WAW (Write After Write); Ri $\cap Rj \neq 0 \rightarrow$ Output Dependence

Ejem:



R1 <--- R2,R3

R1 <--- R4,R5

- RAR (Read After Read); D $\cap D \rightarrow$ Las dos consultan la misma variable; no lo consideraremos una dependencia, no importa el orden.

NOTA: Las dependencias de datos WAR y WAW son culpa del compilador que intenta reutilizar los registros. Con ∞ registros no se producirían nunca.

En un procesador segmentado lineal WAR, WAW son estructuralmente imposibles.

<u>SOLUCIONES</u>

1.- SOFTWARE

- **Scheduling (planificación de instrucciones)**: El compilador busca instrucciones útiles para poner entre las que producen conflicto, para que no sea efectivo el riesgo. Si no encuentra instrucciones puede poner NOP's.

---> se reduce la eficiencia y aumenta el tamaño estático del código.

Ejem:

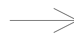
R4 <--- R5 + R6

R1 <--- R3 + R7

R2 <--- R1

R1 <--- R3 + R7

R4 <--- R5 + R6
NOP
NOP
R2 <--- R1



La dependencia todavía existe pero las instrucciones están lo suficientemente lejos.

- **Renombrar variables**: Elimina las dependencias WAR y WAW en tiempo de compilación; requiere más registros (o más memoria).

Ejem:

R1 <--- R2,R3

R2 <--- R4,R5

R6 <--- R2, R7

R1 <--- R2,R3

R2 <--- R4,R5

R6 <--- R2,R7

Esta desaparece a base de scheduling

2.- HARDWARE

- BLOQUEO del procesador hasta que desaparezca el riesgo. En cuanto a ciclos perdidos es equivalente a poner NOP's, pero genera programas más cortos (estáticamente).

¿ Que es mejor: bloqueo o NOP's ?

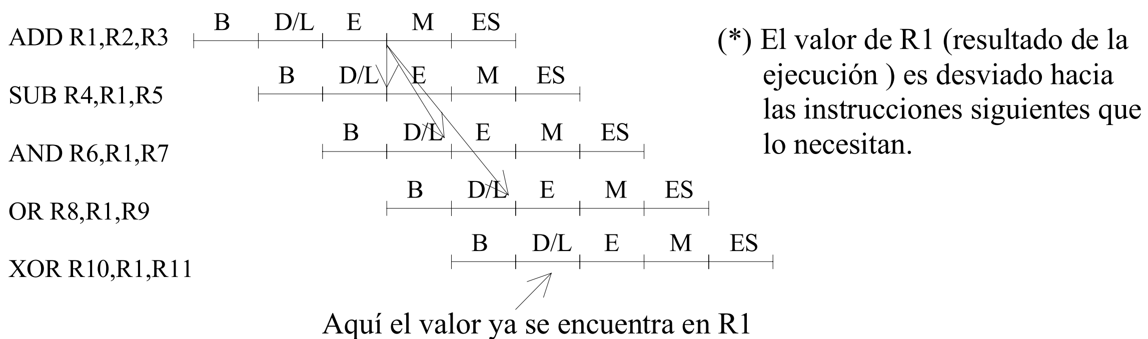
- En tiempo de ejecución son idénticos

- En longitud de código mejor bloqueo

- En simplificar el control mejor NOP's pero se complica el compilador.

- <u>CORTOCIRCUITO</u> (Forwarding): Disminuye (o elimina) el efecto RAW. Si el hardware de adelantamiento detecta que la operación previa de la ALU ha escrito en un registro, correspondiente a una fuente para la operación actual de la ALU, la lógica de control selecciona el resultado adelantando como entrada de la ALU en lugar del valor leído en el fichero de registros.

Ejem:



- <u>RENOMBRAR REGISTROS POR HARDWARE</u>: (lo veremos en procesadores avanzados).

Ejemplo: Una máquina tiene un 20 % de LOADS. De estos un 50 % produce un riesgo de datos. Se pierde un ciclo en cada riesgo. ¿Cual es el CPI real?

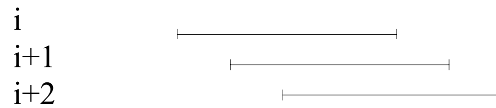
CPIideal = 1

CPIreal = 0.8 *1 + (0.2*0.5*1 + 0.2*0.5*2) = 1.1

$S = \frac{1.1}{1} = 1.1 \rightarrow$ la máquina ideal es un 10 % más rápida.

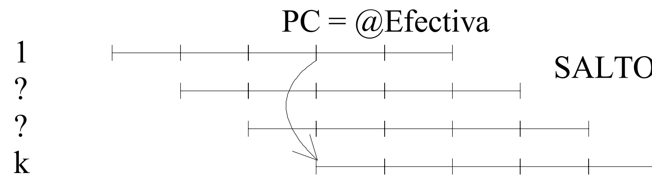
C) RIESGOS DE CONTROL (o de secuenciamiento)

- Hasta ahora hemos considerado el <u>secuenciamiento implícito</u>.



- Los saltos modifican el secuenciamiento implícito de las instrucciones; esto hace perder rendimiento ya que es necesario algunos ciclos para calcular la dirección efectiva de la siguiente instrucción (y saber si hay que saltar o no).

Ejem:



Ejem: -30 % de saltos

- 2 ciclos perdidos por riesgo de control

- CPIideal = 1

- CPIreal = 0.7*1 + 0.3* (1+2) = 1.6

- S = 1.6 / 1 = 1.6 ----> el ideal va 60 % mejor que el real.

3.2 - PROCESADORES SEGMENTADOS LINEALES

INTRODUCCION:

Son los más sencillos, su segmentación es lineal. Se caracterizan por:

- Búsqueda de la instrucción
- Inicio de la ejecución (ISSUE) { En orden }
- Final de la ejecución

Lo que pretendemos es:

CPI = 1 $\rightarrow \overline{L}=1$

nº marcas $\leq$MAL $\leq n^{o}$ 1's VCI
(= 1) (=1) $\rightarrow$ MAL=1 $\rightarrow CPI = 1$

- Para garantizar esto:

  - LINEAL (No reutiliza ninguna etapa) ---> una instrucción pasa una única vez por una etapa.

  - Todas las instrucciones pasan por las mismas etapas en el mismo orden $\Leftrightarrow$ todas las instrucciones tienen la misma TR.

<u>ARQUITECTURA EJEMPLO</u>: DLX

- Es una arquitectura que sólo opera con registros.

- Instrucciones:

  - CALCULO (ENTERA o COMA FLOTANTE)

  - ACCESO A MEMORIA (LOAD y STORE)

  - SECUENCIAMIENTO (SALTOS)

- En una instrucción no se puede especificar direccionamiento a memoria y cálculo a la vez $\rightarrow$ SON OPERACIONES DISTINTAS.

- Este sería el modelo más sencillo de procesador segmentado. Facilita mucho su diseño y estudio.

LENGUAJE MAQUINA DEL DLX

- Hay 3 tipos de instrucciones:

(i) CALCULO (operación)

- OP Rd,Rf1,Rf2 ; Rd <--- Rf1 OP Rf2

(ii) ACCESO A MEMORIA

- LOAD Rd,DIR ; Rd <--- MEM [DIR(1) + DIR(2)]

DIR = {Rf1,Rf2}
DIR = {Rf1,#X}

- STORE DIR,Rf2

DIR = {#X,Rf1}; MEM[DIR(1) + DIR(2)] <--- Rf2

[Solo tiene un modo de direccionamiento para simplificar el DATA PATH]

(iii) TRANSFERENCIA DE CONTROL (SALTOS)

- JMP COND, Rf1, #X ; IF (Rf1 COND 0) THEN
PC <--- PC + #X

Solo permite comparar con '0':
- BEQ Z
- BNE Z

- A grandes rasgos, el DATA PATH necesario (sin segmentar) para este lenguaje máquina será:

CONTROL

BANCO REGISTROS

ALU

MAR @DATOS

MDR DATOS

PC @INST.

IR INST

MEMORIA

DATA PATH

- Sería un Data Path parecido al de la máquina sencilla.

<u>SEGMENTACIÓN EN ETAPAS</u> (Depende del lenguaje máquina escogido)

- Hay que identificar instrucciones que puedan ser etapas.
- Se miran las fases de cada tipo de instrucción y generalmente se busca una división en etapas que sirva para todos los tipos.

<u>INSTR. ARITMETICA</u>	<u>INSTR. ACCESO A MEMORIA</u>
Búsqueda	Búsqueda
Decodificación	Decodificación
Lectura registros	Lectura registros
Ejecución (ALU)	Cálculo @ efectiva a memoria
Escritura registro	Acceso a memoria
	Escritura registro

B D/L ALU ES

B D/L ALU (@EFEC) M ES

Tienen tamaños diferentes ----> problemas

B D/L ALU (@EFEC) M ES

B D/L ALU ES

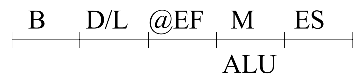
Quieren escribir en el mismo ciclo → PROBLEMAS

Utilizaremos un único formato como resultado de juntar las dos:

Que según el tipo de instrucción hará:

- Búsqueda
- Decodificación + lectura de registros
- ALU o cálculo de @ efectiva
- Nada o acceso a memoria
- Escritura registro

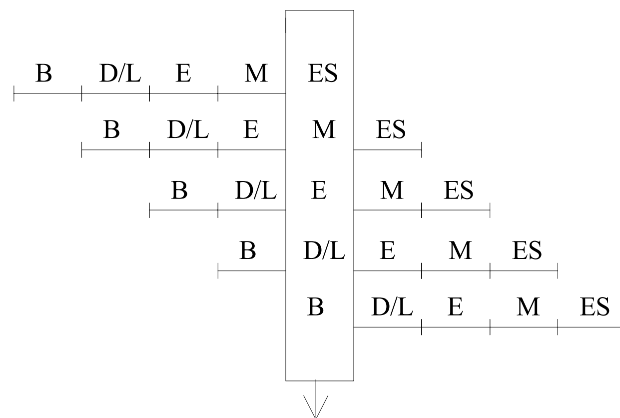
Otra posibilidad sería:



5 ETAPAS ----> máximo 5 instrucciones ejecutándose en paralelo

ESTUDIO DE LOS RIESGOS ESTRUCTURALES

- Se tiene que mirar en pleno rendimiento cual es el máximo de recursos necesarios (ahora que ya hemos escogido una división en etapas concreta).



Pleno rendimiento: Tenemos que mirar que hardware necesitamos para implementar esto.
Tenemos que mirar el peor caso.

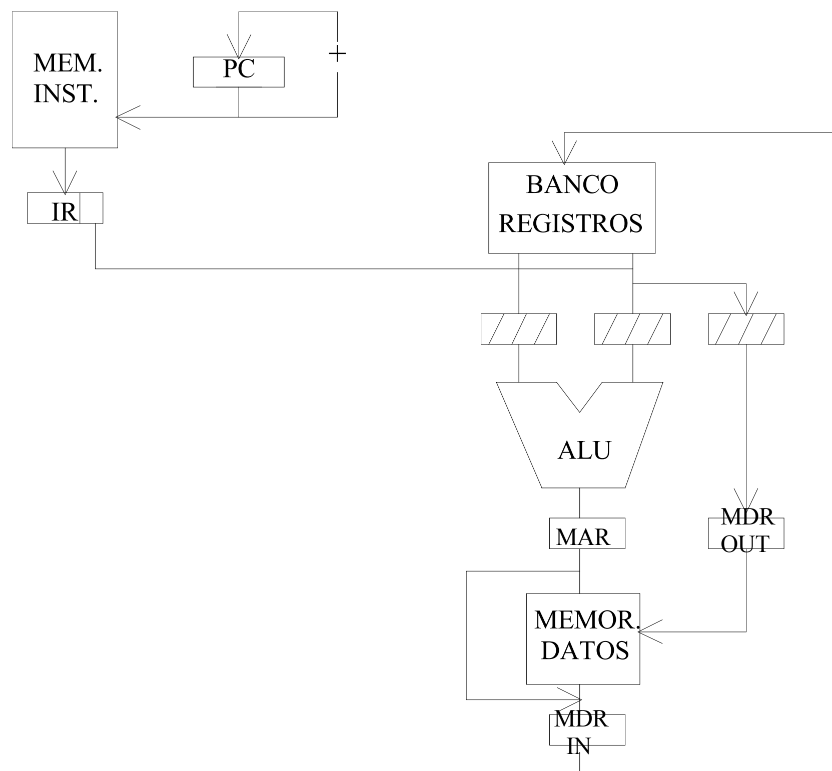
- 2 caminos de lectura a registros (las op. tiene 2 reg. fuentes) ---> D/L
- 1 camino de escritura a registro ----> E
- 1 camino de lectura de instrucciones a memoria ----> B
- 1 camino de lectura o escritura de datos a memoria ----> M
- 1 ALU ---> ALU
- 1 sumador para calcular el nuevo PC en algún punto del pipeline ---> B

- Si los registros los podemos leer en la mitad de tiempo de ciclo, habrá suficiente con un único camino de lectura.

- A partir de esto podemos diseñar un DATA PATH donde no hayan problemas por riesgos estructurales.

DATA PATH SEGMENTADO

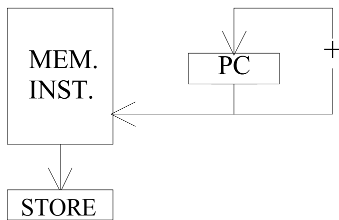
- De momento por sencillez no dibujaremos la parte de saltos.
- Básicamente miramos las necesidades de cada tipo de instrucciones y se añaden los registros necesarios al Data Path base para separar las diferentes etapas.



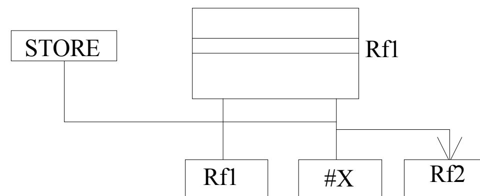
- Ejemplo: ocupación de los recursos en un STORE

STORE   #X,Rf1,Rf2   MEM[DIR(1) + DIR(2)] <--- Rf2

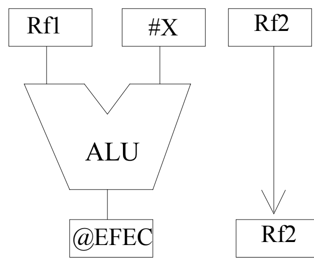
B



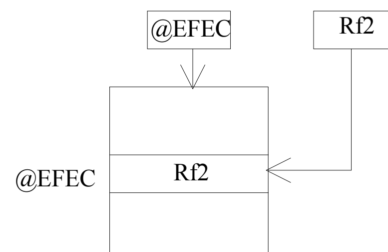
D/L



ALU



ES



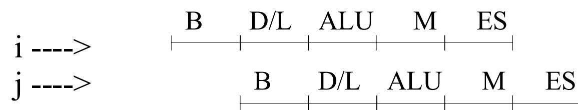
ESTUDIO DE LOS RIESGOS DE DATOS

- WAR (Antidependencias)

R1 <--- R2, R3
R2 <--- R4, R5

Por construcción (diseño del pipeline) no se pueden producir nunca

- Todas las instrucciones tardan lo mismo
- Todas escriben al final
- Cuando la instrucción 'j' escribe todas las anteriores ya han leído



- WAW (Dependencias de Salida)

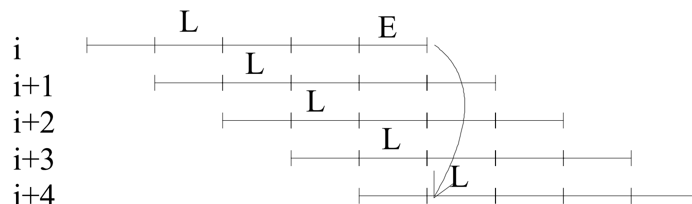
R1 <--- R2, R3
R1 <--- R5, R5

Por el mismo motivo que las WAR no se producirán nunca, son estructuralmente imposibles.
Todas las instrucciones escriben en la misma etapa (sólo una instrucción por ciclo).
Cuando la 'j' escribe, todas las demás ya lo han hecho.

- RAW (Dependencias de Flujo) ---> Provocadas por los cálculos del programa

R1 <--- R2, R3
R4 <--- R1, R5

Es necesario el valor de R1 en la etapa D/L cuando todavía no está escrito:



Las instrucciones i+1, i+2, i+3 no pueden usar aquello que la 'i' calcula o lleva a memoria.

SOLUCIONES

- BLOQUEO ---> Se pierden hasta 3 ciclos en el peor de los casos.

- SOFTWARE SCHEDULNING ---> El compilador deja instrucciones en las posiciones i+1, i+2, i+3 que dependen de la 'i' (ó NOP's).

- CORTOCIRCUITO.

CORTOCIRCUITOS PARA SOLUCIONAR LOS RIESGOS DE DATOS RAW

- En general el valor necesario ya está calculado, el problema es que no está escrito en los registros cuando la instrucción que lo necesita finaliza su etapa de D/L.

- CORTOCIRCUITO: Caminos especiales desde las UF's a la etapa de D/L ó ALU.

- Hay que determinar que etapas consumen datos (leen) y cuales escriben datos:

Teniendo en cuenta que:

Dominio ---> operandos de la instrucción (Datos de entrada).

Rango ---> resultado de la instrucción.

Tenemos:

- Etapas generadora de rango (escritura)
    - ALU (op)
    - MEMORIA (load)

- Etapas consumidoras de dominio (lectura)
    - ALU (op, @ef, load, @ef store)
    - MEMORIA (store)

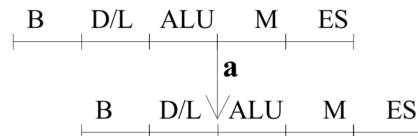
- Ahora estudiaremos cada posible combinación de generador con consumidor:

A ALU ---> ALU

Ejem:

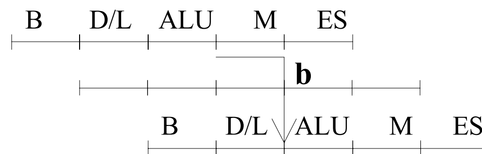
R1 <--- R2 + R3
LOAD R4 <--- M[R1 + #X]

- Distancia 1:



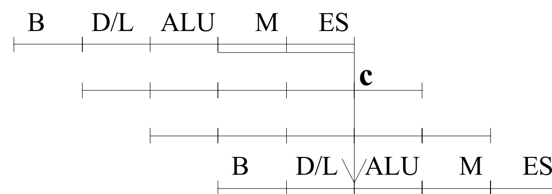
- Ponemos un cable antes de acabar la etapa ALU y antes de finalizar D/L ---> ---> podemos trabajar con el dato antes de que llegue al banco de datos.
- '0' ciclos de bloqueo. Cortocircuito a1, a2.

- Distancia 2:



- '0' ciclos de bloqueo. Cortocircuito b1, b2.

- Distancia 3:



- '0' ciclos de bloqueo. Cortocircuito c1,c2

- Distancia 4 ---> ya no hay problemas.

- Estamos suponiendo que los registros se escriben al final del ciclo. Se puede eliminar este cortocircuito haciendo que la lectura del registro se haga en el 2º 1/2 ciclo

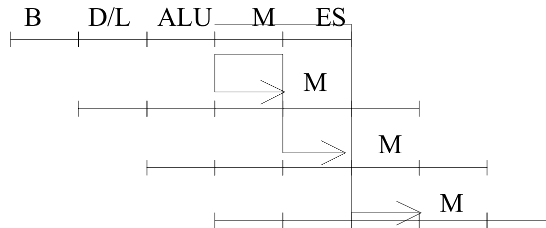
y la escritura el 1er 1/2 ciclo (Es posible ya que el acceso al banco de registros no suele estar en el camino crítico del procesador.

ATENCIÓN: Un cortocircuito mas simplifica añadir un Bus del tamaño de un registro(32 o 64 bits) ---> puede ser muy caro (Área del chip, Routing...).

B <u>ALU ---> MEMout</u>

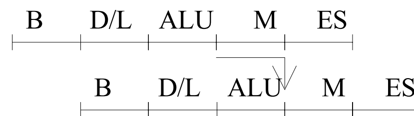
Ejem: (Todos con bloqueo)

R1 <--- R2 + R3
STORE M[DIR], R1



- Distancia 1 --> a3
- Distancia 2 --> b3
- Distancia 3 --> c3

- En muchos casos hay más de un cortocircuito posible. Se trata de escoger el mínimo número de cortocircuitos posibles.



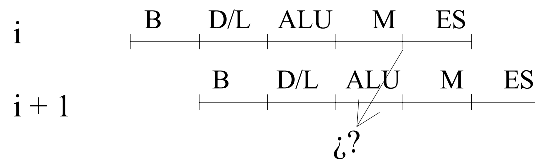
C <u>MEMin ---> ALU</u> (Load seguido de operación)

Ejem:

LOAD R1 <--- M[DIR]
R2 <--- R1 + R3

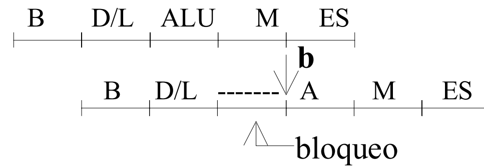
LOAD R1 <--- M[DIR]
LOAD R2 <--- M[R1 + #X]

- Distancia 1:



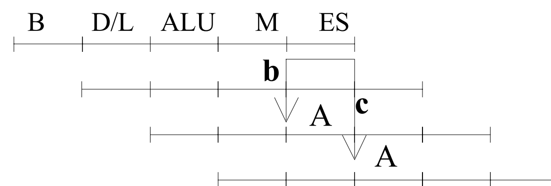
- En la i+1 no se pude usar, cuando la instrucción 'i' tiene el dato, la i+1 ya ha pasado la etapa ALU.

----> BLOQUEO + CORTOCIRCUITO



- Cortocircuito: b1, b2

- Distancia 2,3:



- Cortocircuito: b1, b2, c1, c2. Bloqueo '0'.

- En el caso problemático de que i+1 dependa de un LOAD, hay dos opciones:

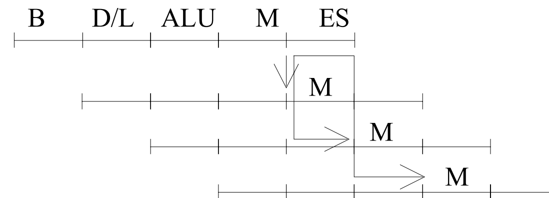
(i): detectar el riesgo y bloquear.

(ii): No comprobar el riesgo ---> El compilador tiene que garantizar que no se dará nunca.Si no encuentra operaciones independientes pondrá NOP ----> LOAD RETARDAD (DELAYED LOAD).

C <u>MEMin ---> MEMout</u> (Load seguido de un Store)

Ejem:

LOAD R1 <--- M[DIR1]
STORE M[DIR2] <--- R1



- En memoria no existen problemas porque en la misma etapa es donde se hace la lectura y la escritura. En cambio en los registros si porque se hace L y E en 2 etapas diferentes.

- Un cortocircuito está determinado por:

  - De que etapa sale

  - A que etapa entra

  - Cual la consume

y no porque etapa genera el dato

- En este caso la i+1 si que tiene tiempo de coger el dato; aquí hay dos opciones:

  - i) implementar el cortocircuito nuevo (d)

  - ii) no permitir este caso para que la solución sea simétrica a MEM ---> ALU

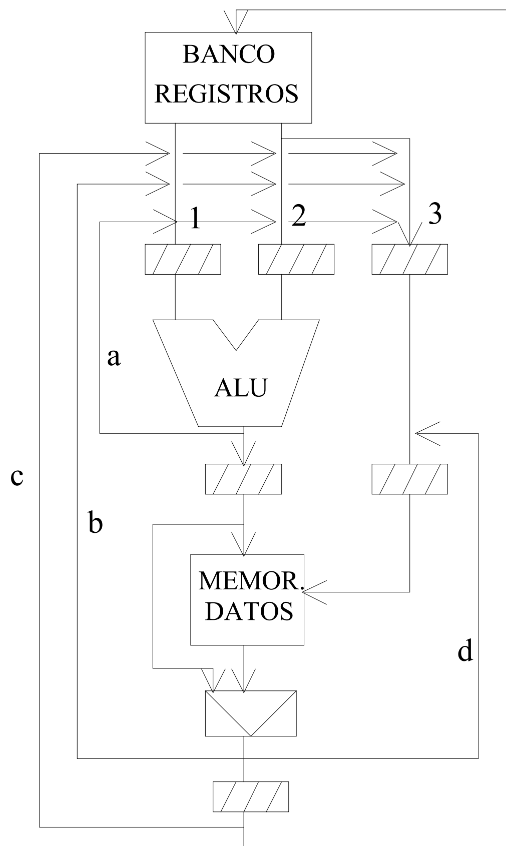
- Según quien produzca, quien consuma y la separación entre instrucciones obtenemos el siguiente recuadro de cortocircuitos:

Al final de la etapa de

Al inicio de la etapa de (Consum. rango) \ Gener. rango	ALU			MEM		
	DIST	BLO	CORT	DIST	BLO	CORT
ALU	1	0	a	1	1	b
ALU	2	0	b	2	0	b
ALU	3	0	c	3	0	c
MEM	1	0	a3	1	0	d
MEM	2	0	b3	2	0	b3
MEM	3	0	c3	3	0	c3

DATA PATH CON CORTOCIRCUITO

- Sólo nos centramos en la parte de datos que es la que se ve afectada por los cortocircuitos.



ESTUDIO DE LOS RIESGOS DE CONTROL

- Por defecto en los programas el secuenciamiento es implícito. Lo que se hace se automáticamente incrementar el contador del programa según el tamaño de la instrucción.

- Nuestro procesador sólo necesita un sumador en la etapa de búsqueda para sumar una constante.

- ¿Qué pasa cuando la instrucción buscada es una instrucción de ruptura de la secuencia? (Salto incondicional, condicional, llamada a subrutina, retorno a subrutina)

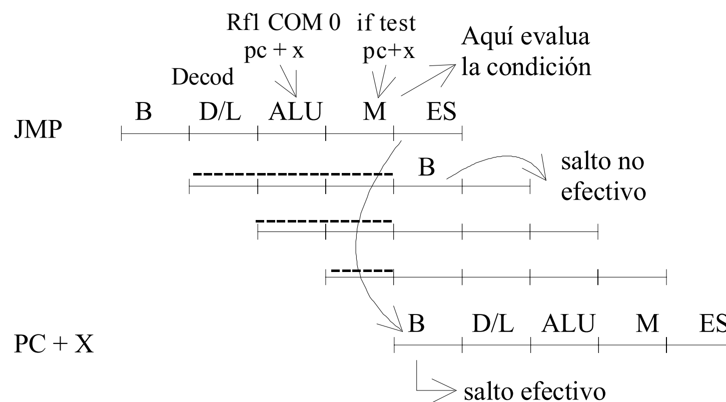
- En el DLX la instrucción de ruptura de secuencia es una instrucción de tipo COMPARE&BRANCH (es una máquina sin códigos de condición)

BEQ Z Rf1,#X ; BNEZ Rf1, #X

Semántica: **IF (Rf1 COND 0) THEN PC = PC + X**

Una única instrucción hace el test de la condición y calcula la @efectiva de salto (relativo al PC).

Una posible implementación daría:



Por culpa del salto se pierden 3 ciclos (latencia salto = 4) cuando se detecta un salto hay que bloquear el procesador hasta que conozcamos la dirección efectiva.

Un estudio estadístico nos muestra que en media del 20 al 25 % de instrucciones son de salto ----> según la ley de Amdahl hay una gran perdida de rendimiento.

Ejem:

- CPI = 1,2 (excepto salto)

- Salto ---> 4 ciclos
- 20 % de inst. salto

CPI = 0.8*1.2 + 0.2 * 4 = 1.76

S = 1.76 / 1.2 = 1.46 ---> máquina ideal 46 % mejor

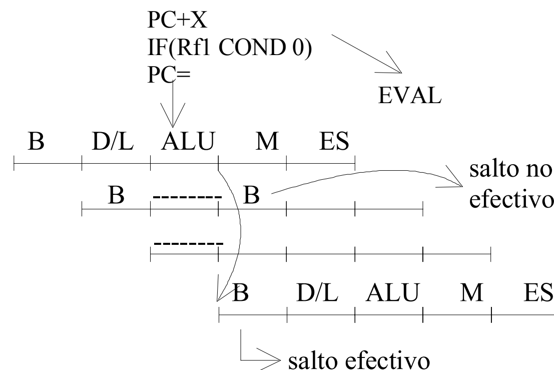
Ejercicios: 5,6,16,10

SOLUCIONES

- No podemos permitir está gran perdida de rendimiento. Hay diferentes soluciones que pueden ser hardware, software o una combinación delas dos. Cuando no se pueden arreglar tenemos que bloquear para mantener la semántica del programa.

A --> REDUCIR EL NÚMERO DE CICLOS PERDIDOS (hardware)

- Adelantar el test de la condición y si se puede también el cálculo del nuevo PC. No sirve de nada adelantar mucho uno y no el otro ya que el que tarde más determinará el número de ciclos perdidos:



(VER FOTOCOPIA 62-A)

Latencia = 3 ---> perdemos un ciclos más

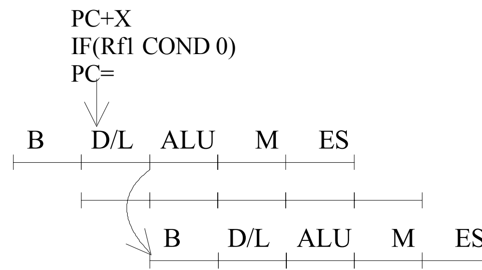
-Ejem:

CPI = 0.8 * 1.2 + 0.2 * 3 = 1.56

S = 1.76 / 1.56 = 1.12 ---> Hemos mejorado un 12% respecto al caso anterior.

- Hasta ahora hemos aprovechado que la etapa de la ALU "esta libre" durante un salto, para calcular la @EF. Si añadimos hardware para calcular la @EF en la etapa de D/L más cortocircuitos necesarios y podemos adelantar el cálculo de la condición y el test solo perdemos 1 ciclo. Esto es posible gracias a lo limitado del tipo de condición y

del simple cálculo de @. En máquinas más complejas no siempre se pueden anticipar tanto los cálculos:



(Ver fotocopía 63 - A)

- Ejem:

CPI = 1.2 * 0.8 + 2 * 0.2 = 1.36

S = 1.76 / 1.36 = 1.29 ---> 29 % mejora

B --> DELAYED BRANCH --> Salto Retardado (hardware)

- Se modifica la semántica de la instrucción de salto. El salto se hace efectivo 'n' instrucciones después de la instrucción de salto propiamente. Es decir, se hace visible al usuario la latencia de salto (por ej. 2 ciclos).

- El compilador / programador tiene que poner instrucciones después del salto (DELAY SLOTS) que no modifiquen la semántica del programa.

C --> SALTO RETARDADO CON ANULACIÓN (Soft + hard)

- Es la misma idea del salto retardado pero el hardware puede anular las instrucciones de los slots de retardo si el salto no se resuelve según esperaba el compilador.

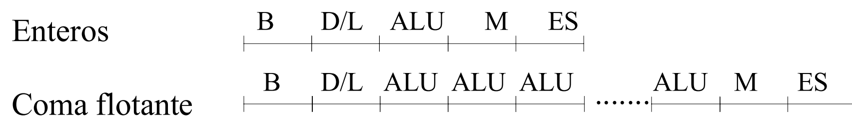
- Cuando ha movido el código ---> el compilador tiene que poder indicar de donde proviene el código para que el hardware puede decidir que hace.

3.3 - PROCESADORES SEGMENTADOS CON OPERACIONES MULTICICLO

INTRODUCCIÓN:

- Hasta ahora hemos supuesto que todas las instrucciones necesitan el mismo número de ciclos para ejecutarse. Esto no siempre es posible, especialmente en las operaciones de punto flotante.

- Vamos a introducir en nuestra máquina instrucciones de punto flotante. La segmentación es la misma que la vista para enteros pero la etapa de ejecución (ó ALU) puede durar más de un ciclo.



- Una nueva diferencia aparece cuando podemos tener más de una unidad funcional de punto flotante. Así, la nueva máquina la vemos como:

- B, D/L común a todas las UF's.

- Diferentes UF's de ejecución

  - Ejecución de enteros (la máquina anterior)
  - Punto flotante 1
  - Punto flotante 2

- M, E

- Para simplificar la máquina y reducir los problemas, la dividiremos en dos máquinas:
  - Enteros
  - Punto flotante

- Cada una tiene su banco de registros separado, así evitamos instrucciones innecesarias.

- Las instrucciones de LOAD y STORE de registros en punto flotante se ejecutarán en la parte entera.

----> No necesitamos etapa de memoria en las instrucciones de punto flotante.

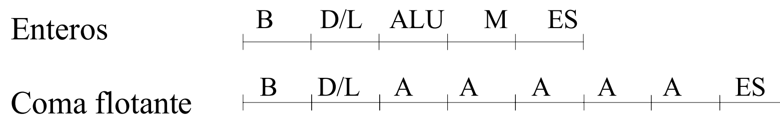
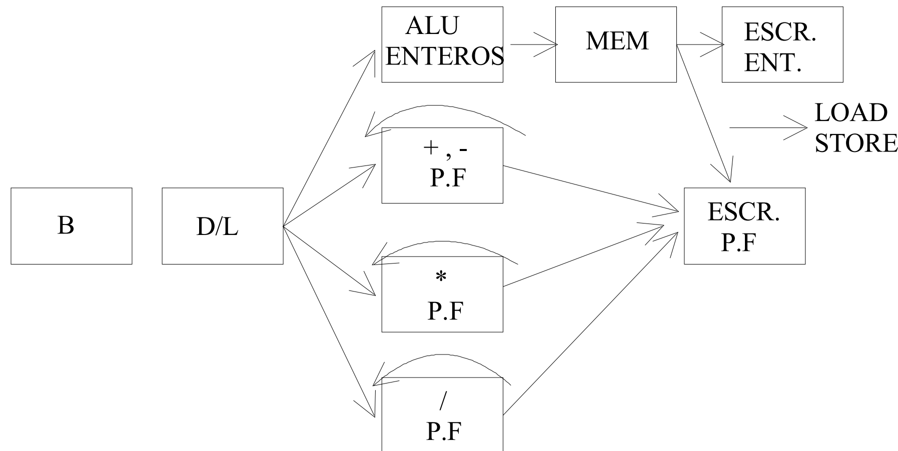


DIAGRAMA DE BLOQUES



- Obtenemos dos máquinas lo más desacopladas posible y sencillas.

- Podemos diferenciar 2 tipos de máquinas:

A --> MULTICICLO SIMPLE

- Todas las instrucciones de una misma familia (enteros o coma flotante) tardan lo mismo (mismas etapas y mismo número). ---> las instrucciones de una misma familia acaban en orden (entre familias puede haber desorden por el diferente número de etapas).

B --> MULTICICLO GENERAL

- Las instrucciones multiciclo pueden tener diferente número de etapas de ejecución.

- Nos dedicaremos a estudiar estas ya que incluyen a las anteriores (en el caso 'A' algunos de los problemas no aparecen).

- Se caracterizan por:
    - Búsqueda en orden
    - Inicio en orden (la ejecución)
    - Final en orden

EXTENSIÓN DEL LENGUAJE MÁQUINA

- Instrucciones de cálculo

FPOP FRd , FRf1 , FRf2 ; FRd = FRf1 FPOP FRf2

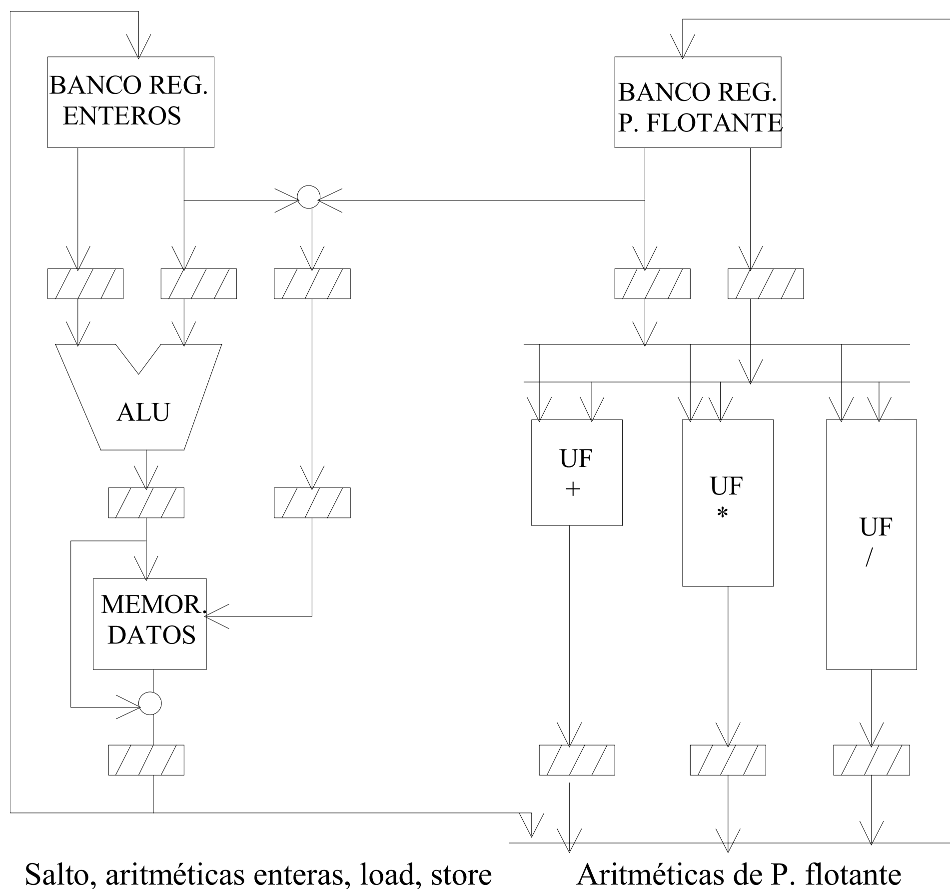
- Instrucciones de acceso a memoria

LOAD FRd, DIR ; FRd = M[DIR(1) + DIR(2)]

DIR = {#X,Rf1} Las @ se calculan siempre en
DIR = {Rf2,Rf1} la parte entera

STORE #X,Rf1,FR ; M[#X + Rf1] = FR

DATA PATH



SINCRONIZACIÓN ENTRE DATA PATHS (Enteros y P.F)

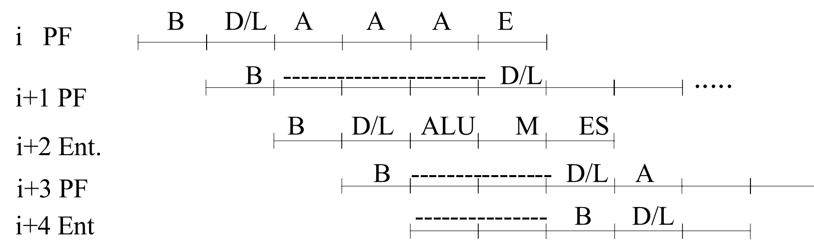
- Data path separados para enteros y p. flotante.

- Las instrucciones de LOAD y STORE comunican los dos DATA PATH y permiten / fuerzan un sincronismo.

- Si se detecta un riesgo es sólo para el data path afectado.

- Un data path parado provoca la detención del otro si la instrucción que se ha buscado tiene que usar el data path parado (ya que no sabemos cuantas instrucciones pueden llegar por el data path parado ---> colas ∞).

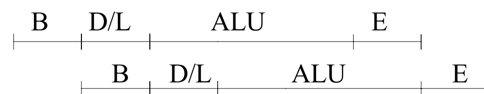
- Ejem:



i+1 para el data path de PF y deja continuar el data path entero en i+2. Al llegar una i+3 de PF que necesita utilizar el data path parado provoca que se pare el data path entero en i+4.

RIESGOS ESTRUCTURALES

A --> UF. NO SEGMENTADA (Tuf > Tciclo , Tuf = k Tciclo)



SOLUCIONES:

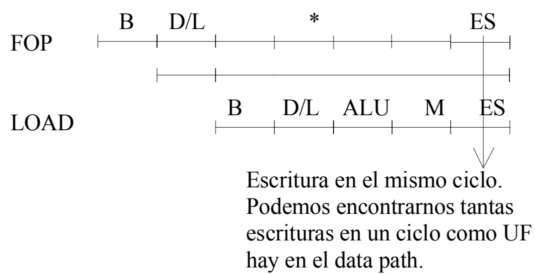
- Bloqueo:

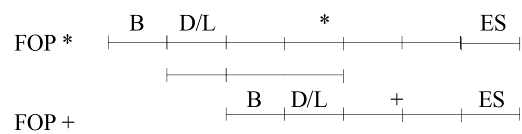
- Reordenar instrucciones

  - Tienen que estar a una distancia >= LANTENCIA UF -1

B --> CONFLICTO DE ACCESO AL BANCO DE REGISTROS

- Como las instrucciones tiene tamaños diferentes nos podemos encontrar con:





- SOLUCIONES:

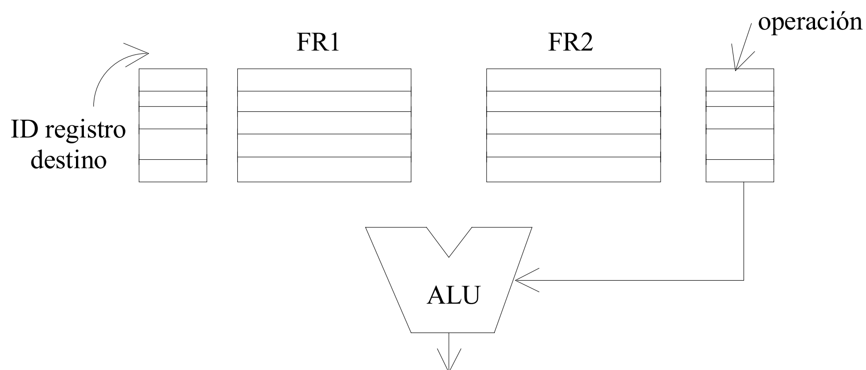
  - Bloqueo
  - Poner más caminos de escritura al banco de registros de P.F

- En ciertos casos puede interesar no solucionar los riesgos estructurales:

  - Reducir coste

  - Reducir latencia UF

  - Probabilidad de que se produzca muy baja. Si esto ocurre se puede solucionar fácilmente añadiendo colas FIFO en la entrada de la UF --> UF's virtuales.

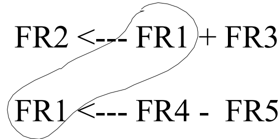


RIESGOS DE DATOS

- La parte de aritmética entera tiene los mismos de siempre.

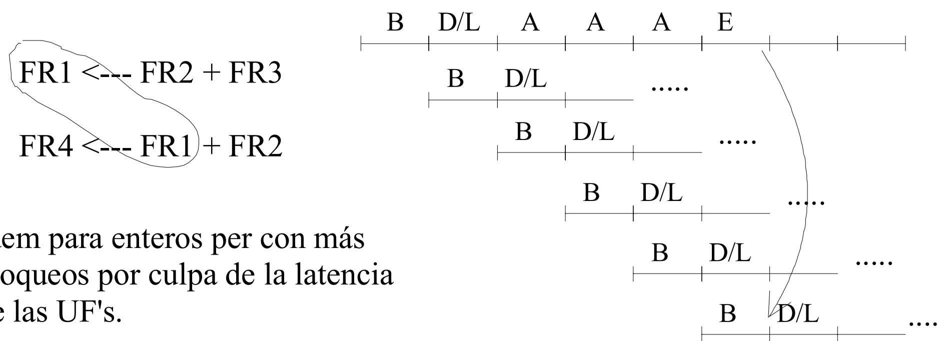
- Falta estudiar la parte de aritmética en P.F y su relación con LOAD y STORE.

1.- <u>WAR</u>



Todas las instrucciones son idénticas hasta la etapa de lectura. En todas se escribe después de leer ---> ESTRUCTURALMENTE IMPOSIBLE.

2.- <u>RAW</u>



Idem para enteros per con más bloqueos por culpa de la latencia de las UF's.

SOLUCIONES:

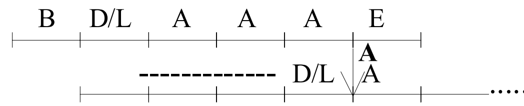
(i) - BLOQUEO ---> ciclos perdidos = LATuf - DIST + 2

(ii) - REORDENAR CÓDIGOS

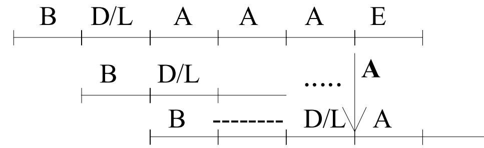
(iii)- CORTOCIRCUITOS: Aquí los c.c no resuelven mucho el problema solo ganamos un ciclo. Vamos a ver los casos que se producen.

- ALU_{FP} ----> ALU_{FP}

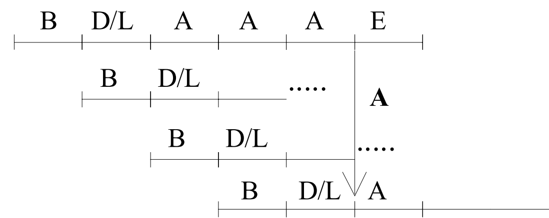
- Dist1:



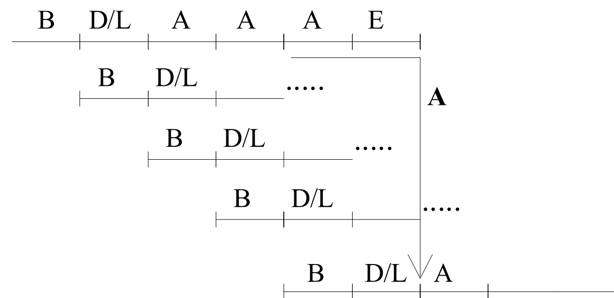
- Dist2:



- Dist3:



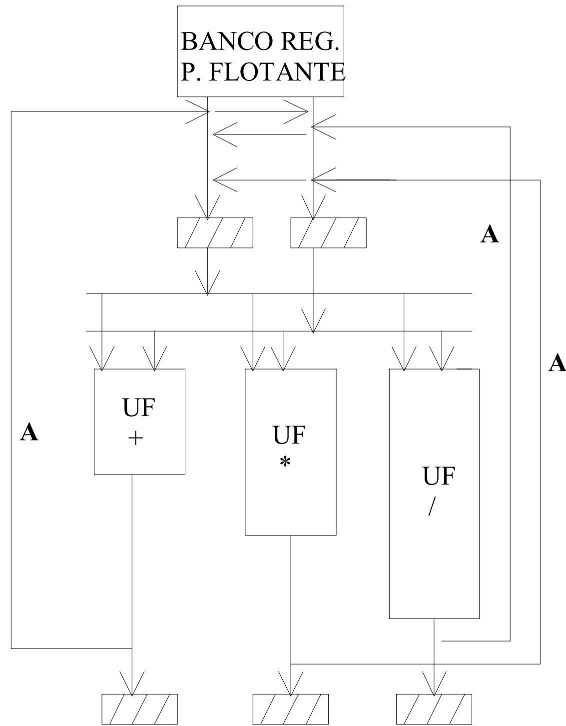
- Dist4:



Si la escritura se realiza al principio del ciclo y la lectura al final no es necesario este cortocircuito.

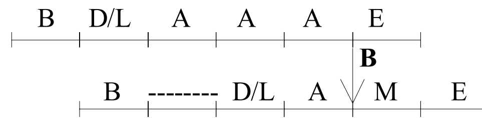
- En general $D \leq 1$

ciclos perdidos ⇔ BLOQUE: L - D
CORTOC.: A

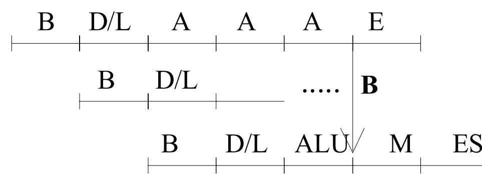


- ALU$_{FP}$ - MEM$_{STORE}$

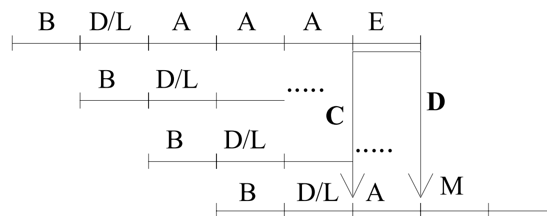
- Dist1:



- Dist2:

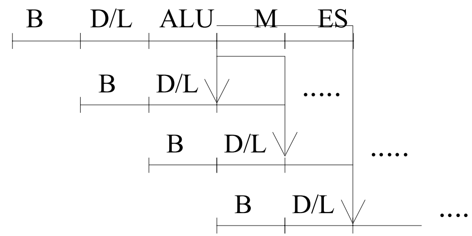


- Dist3:



Dos posibles C.C, los 2 son válidos.

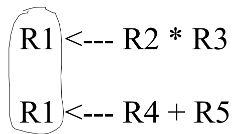
- MEM ---> ALU
  (load) FP



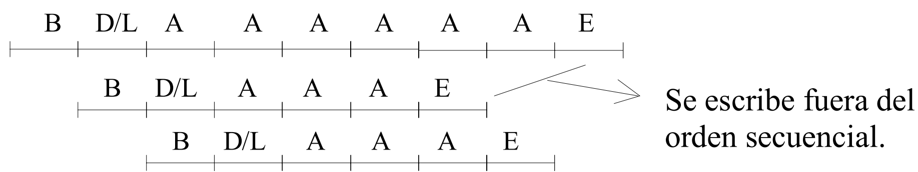
- MEM ---> MEM ----> Estos cortocircuitos ya los tenemos en la parte de
  (load) (store) enteros.

- MEM ---> MEM -----> No trae problemas.
  (store) (load)

3.- <u>WAW</u>



- Las instrucciones pueden acabar en desorden, tanto una FOP seguida de un LOAD como una FOP seguida de una FOP más corta.



- Tenemos que garantizar que se mantiene el orden en las escrituras del programa secuencial .

  SOLUCIONES:

  (i) BLOQUEO

  (ii) Anular la escritura más antigua (eliminar la instrucción)

- Mediante software podríamos hacer:

```
R1 <---
    <--- R1      ---> guarda el contenido de R1 en otro registro pero se
R1 <---               genera un RAW --> ya sabemos solucionarlo.
```

- Este riesgo también se suele producir en un salto cuando una de las ramas quiere volver a utilizar un registro.

Ejem:

```
R1 <--- R2 / R3
R4 <---
BEQ @
  .
  .
@R1 <--- R3 + R2
```

- Ejercicios: 26,27,15,30

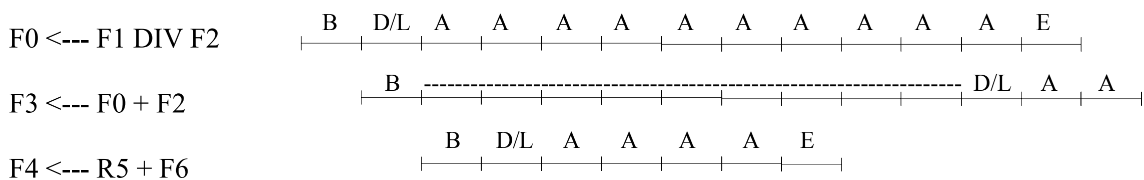
3.4 - PROCESADORES CON OPERACIÓN DINÁMICA DE INSTRUCCIONES

	Segm. lineal	Segm. multiciclo
Búsqueda Inicio Final	EN ORDEN	EN ORDEN FUERA DE ORDEN
RAW WAW WAR	Si (fácil solución) No No	Si Si --> Debido al fuera de orden No
	↓ Máquinas no reales	↓ Reordenación estática de código

- Lo que haremos es dejar más libertad al procesador. Hasta ahora el software era el responsable de la planificación de las instrucciones para minimizar las detenciones ---> ORDENACIÓN ESTÁTICA.

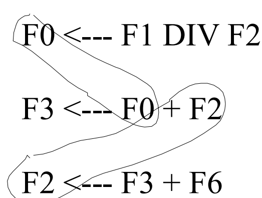
- Ahora dejaremos que el hardware reorganice la ejecución de la instrucción ---> ORDENACIÓN DINÁMICA.

- En las máquinas segmentadas es muy importante el trabajo del compilador, tiene que existir una reordenación estática de código (sino la máquina no funcionará tan bien).



- Se puede evitar por parte del compilador haciendo una reordenación de código (pasando la 3ª en lugar de la 2ª).

Ejem:



No podemos hacer reordenación de código porque existe autodependencia. Hacerlo por software es dificil porque hay pocos registros.

Solución: Renombrarlo a nivel interno.

Ejem:

F0 <--- F7 / F8

STORE <--- d1,R1,F0

LOAD <--- F5,d2,R2

En tiempo de compilación estas instrucciones no se pueden girar ya que tendríamos que estar seguros que las @ de STORE y LOAD son diferentes y esto no se sabe hasta el tiempo de ejecución.

----> Reordenación estática no siempre se puede hacer:
- pocos registros
- compiladores malos

Solución: Reordenación Dinámica.

PROCESADORES CON ANTICIPACIÓN

- Se trata de reordenar código en tiempo de ejecución (dinámicamente) (mientras tiene recursos no para de buscar instrucciones, si no se pueden ejecutar las guarda en la cola).

IMPLEMENTACIONES

A --> SCOREBOARD ---> CDC G600.