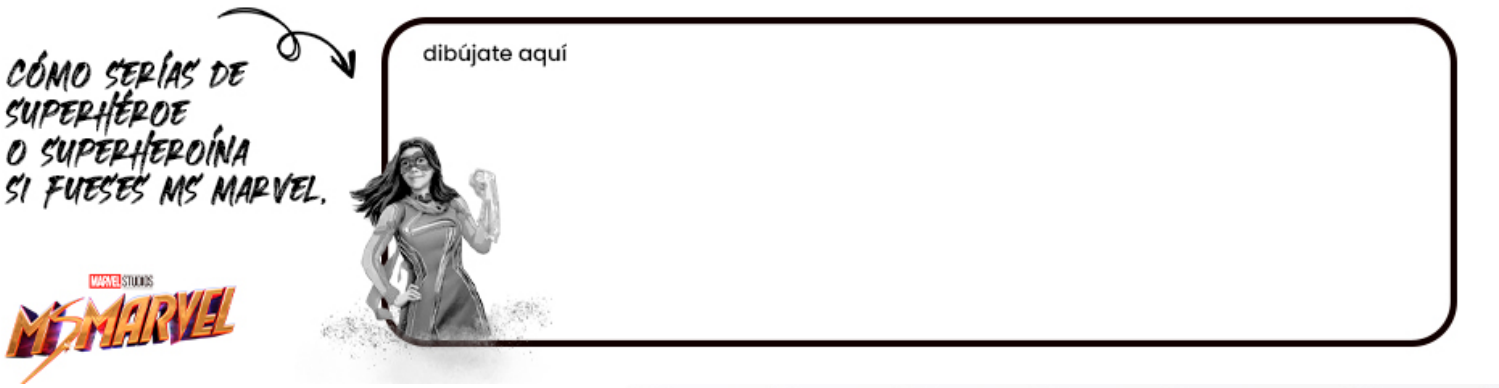
Ejercicios-examen-AC.pdf

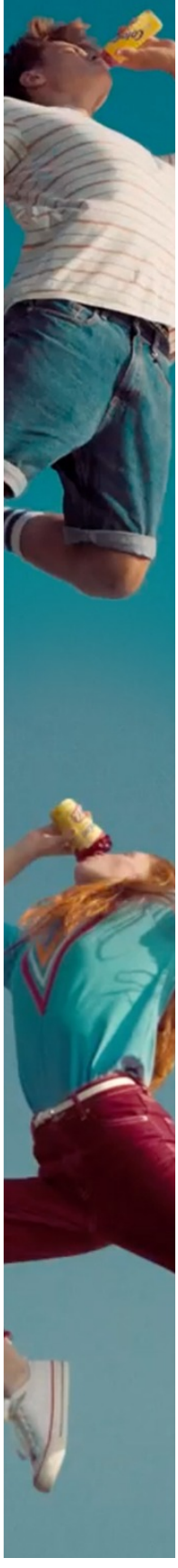
BorjaLive

Arquitectura de Computadores

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería
Universidad de Huelva



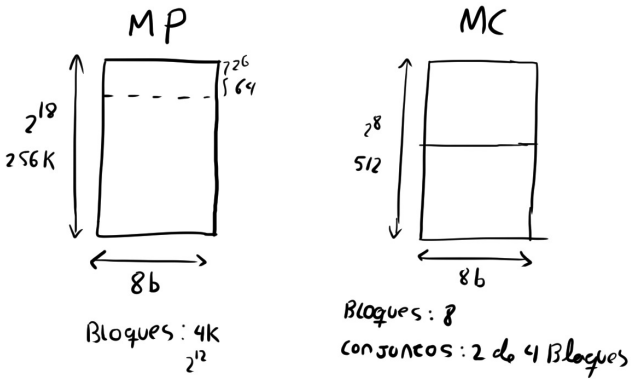


Ejercicio cache

1º. Describir la memoria principal y caché a partir de los datos proporcionados por el enunciado. Contestar a las preguntas indicando las unidades correctamente.

EJ: 19 Jun

PROBLEMA 1. (2,5 ptos.). Un sistema computador (sin memoria virtual) tiene una memoria principal de 256 Kpalabras de 8 bits cada una de ellas, dividida en bloques de 64 palabras/bloque; y una memoria caché de 512 Bytes, dividida en 2 conjuntos.



a: 8 bits por palabra * 256 Kpalabras = 2 Mb (2*10^21)
b: 256Kpalabras / 64 palabras por bloque = 4K bloques (2*10^12)
c: 8 bits en un byte * 512 bytes = 4Kb (2*10^12)
d: 8 bits en un byte * 512 bytes / 8 bits en una palabra = 512 palabras
e: 512 palabras / 64 palabras por bloque = 8 bloques
f: 8 bloques / 2 conjuntos = 4 bloques por conjunto
g: 64 palabras por bloque, igual que la MP

2º. El formato de la dirección depende de la correspondencia de la caché. Hay 3 opciones, en ellas se usan las siguientes variables:
n = tamaño de una dirección de memoria = log2(nº de palabras en MP)
p = bits para identificar una palabra dentro de un bloque = log2(nº de palabras por bloque)
B = bits para identificar un bloque en MP = log2(nº de bloques de MP)
b = bits para identificar un bloque en MC = log2(nº de bloques de MC)

Correspondencia directa:
e = bits de etiqueta = B - b = n - b - p

n		
e	b	p

Correspondencia asociativa por conjuntos:
c = bits de conjunto = log2(nº de conjuntos)
e = bits de etiqueta = B - c = n - c - p

n		
e	c	p

Correspondencia totalmente asociativa:
e = bits de etiqueta = B = n - p

n	
e	p

EJ: Jun 19

2. Mostrar el **formato de la dirección** que define la unidad central de proceso en base a la función de correspondencia empleada, definiendo cada uno de los campos en los que se divide.

n = log2(256*1024) = 18
p = log2(64) = 6
c = log2(2) = 1
e = 18 - 6 - 1 = 11

18b		
Etiqueta: 11b	Conjunto: 1b	Palabra: 6b

yo elijo cerveza SIN

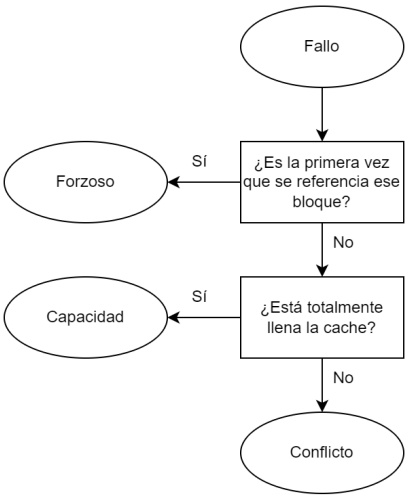
Sea cual sea el vehículo que conduces, elige cerveza SIN.

WWW.CONDUCCIONRESPONSABLECERVEZASIN.COM

UNA GRAN CERVEZA.
UNA GRAN RESPONSABILIDAD.

3º. Simular el estado de la caché. Dibujar los bloques de la caché, teniendo en cuenta la función de correspondencia y el algoritmo de reemplazo.

Identificar los tipos de fallos de caché:
Forzoso (Compulsory): Cuando es la primera vez que se referencia ese bloque.
Capacidad (Capacity): Cuando no caben todos los bloques porque la caché está llena.
Conflicto (Conflict): Cuando hay espacio en la caché, pero el mecanismo de asignación no permite guardarlo (Es asociativo y el conjunto que le toca está lleno, pero hay otros conjuntos con hueco libre)



EJ: Jun 19

Se supone que, después de haber estado la memoria caché "vacía", a continuación, en el *instante 1* se encuentran en la memoria caché los bloques de memoria principal B6, B8, B15, B17, B0, B1 y B2, leídos en ese orden y todas sus direcciones ordenadamente una vez. Suponer que el algoritmo de reemplazamiento de bloques es el LRU (Least Recently Used).

3. Mostrar el contenido de la memoria caché en el *instante 1*.

Opción 1: Pasar a binario el bloque y mirar el campo C para saber en qué conjunto se guarda.
Opción 2: Hacer el módulo: nº de bloque % nº de conjuntos = c

B6			6%2 = 0
00000000011	0	XXXXXX	

Instante 1

c0b0: B6	c0b1: B8	c0b2: B0	c0b3: B2
c1b0: B15	c1b1: B17	c1b2: B1	c1b3:

4. Identificar cuántos fallos de caché se han producido y de qué tipo hasta ese *instante 1*. Indicar, en binario y en decimal, con la lectura de qué dirección/es se produjeron el/los fallo/s.

Se han producido 7 fallos, todos de tipo forzoso ya que es la primera vez que se referencian estos bloques.
Como las direcciones de cada bloque se han leído en orden, la dirección

en la que se producen los fallos es la primera de cada bloque:
00000000011 0 000000 = 384 (6 * 2^6, no hay que convertirlo)
00000000100 0 000000 = 512
00000000111 1 000000 = 960
00000001000 1 000000 = 1088
00000000000 0 000000 = 0
00000000000 1 000000 = 64
00000000001 0 000000 = 128

A continuación, la CPU lee la secuencia de direcciones de memoria: 385, 520 y 260 (*instante 2*), según el orden marcado en la misma.

5. Mostrar el contenido de la memoria caché en el *instante 2*.

Obtener el bloque de cada dirección: binario o dividiendo entre el número de palabras por bloque y haciendo floor.

385			385/64 = 6
00000000011	0	000001	

385 -> B6: Acierto, el bloque 6 está en caché.
520 -> B8: Acierto, el bloque 8 está en caché.
260 -> B4: Fallo forzoso, el bloque 4 no está en caché y es la primera vez que se referencia. Se guarda en el conjunto 0, pero este está lleno, aplicando LRU se cambia por el bloque 0 ya que este es el que lleva más tiempo sin referenciarse.

Instante 2

c0b0: B6	c0b1: B8	c0b2: B4	c0b3: B2
c1b0: B15	c1b1: B17	c1b2: B1	c1b3:

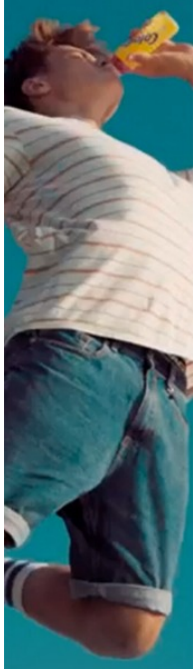
6. Hasta ese *instante 2,* ¿cuántos fallos y de qué tipo se han producido en total?. Indicar, en binario y en decimal, la/s dirección/es con la que se ha/n producido el/los nuevo/s fallo/s.

Hasta este instante se han producido 8, todos forzosos ya que era la primera vez que se diferenciaba ese bloque.

El nuevo fallo producido ha sido en la dirección:
00000000010 0 000100 = 260

7. Hasta ese *instante 2*, ¿cuántos aciertos se han producido?.

Hasta el instante 2 se han producido:
-> 7 * 63 aciertos hasta el instante 1, porque la primera dirección de cada bloque falla, pero el resto acierta.



-> 2 aciertos entre el instante 1 y el 2.
Total: 443 aciertos.

Por último, la CPU lee la dirección 3 (*instante 3*).

8. Mostrar el contenido de la memoria caché en el *instante 3*.

Instante 2

c0b0: B6	c0b1: B8	c0b2: B4	c0b3: B0
c1b0: B15	c1b1: B17	c1b2: B1	c1b3:

El bloque 0 reemplaza al bloque 2 porque este es el que llevaba más tiempo sin referenciarse.

9. Con esas última lectura, ¿se ha producido un fallo o un acierto?. Si ha sido un fallo, ¿de qué tipo?.

Se ha producido un fallo ya que el bloque 0 no estaba en caché. El fallo es de tipo conflicto porque la caché tiene espacio libre, pero la función de asignación nos fuerza a guardarlo en el conjunto 0 dónde no hay espacio.

Ejercicio de Von Neumann

1º. Escribir las instrucciones en binario.

Ej: 19 Jun

PROBLEMA 2. (2,5 ptos.). Para el sistema computador representado en la figura, y para la secuencia de instrucciones siguiente:

Dirección de Memoria (en hexadecimal)	**Instrucción (en ensamblador)**
A0000h	BNZ A0002h
A0001h	MOVE .2, 120(.3)
A0002h	CALL A0003h
A0003h	RETI

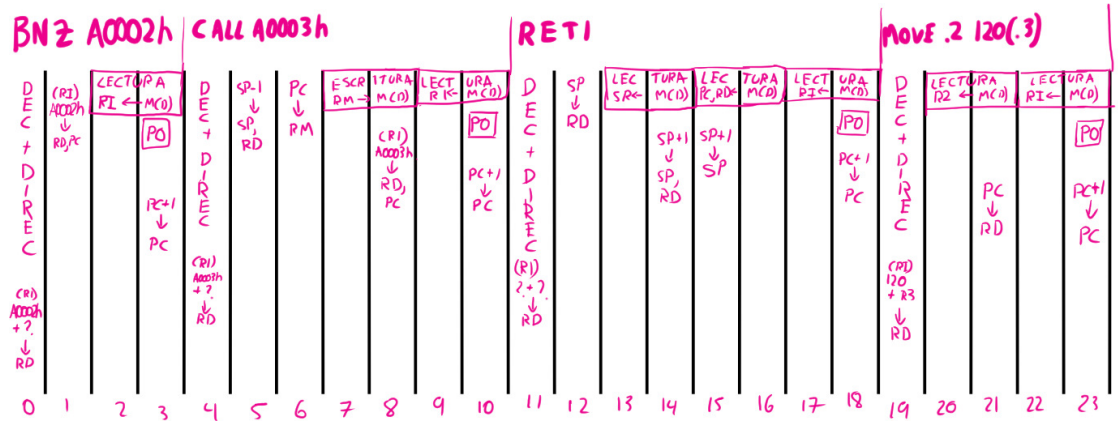
1. Mostrar los formatos de las instrucciones.

C.OP BNZ			A0002h
C.OP MOVE	R2	R3	120
C.OP CALL			A0003h
C.OP RETI			

2. Definir la secuencia de operaciones elementales y el solapamiento posible de las mismas. Especificar el cronograma según el flujo marcado por el programa propuesto, y **considerando una sola vez la ejecución de cada instrucción**.

Importante: tener el diagrama de Von Neumann siempre presente y el manual de cronos para consultar las instrucciones.
Variantes en el ejercicio:

- Tiempo que tarda la memoria en responder
- Si el cálculo del direccionamiento relativo ocurre en el mismo ciclo que la decodificación.
- Dirección en la que crece la pila
- Cual es el dato al que apunta el SP
- Si se guarda primero SR o PC
- El direccionamiento de cada instrucción, aunque se puede deducir por la propia instrucción



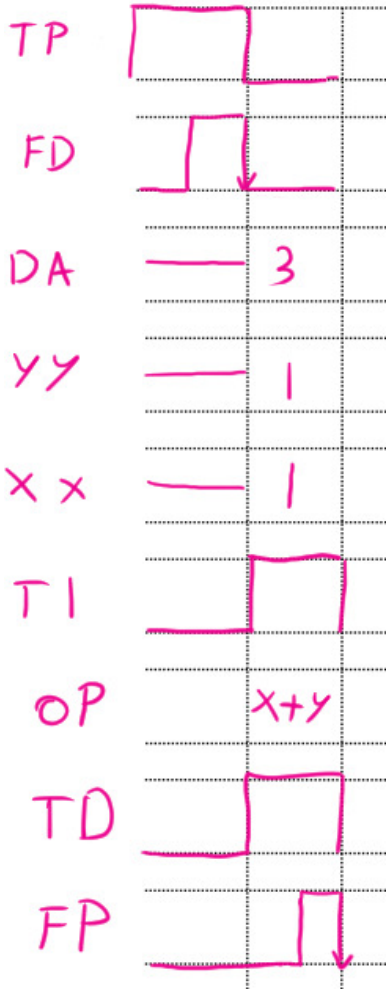
BNZ bifurca, porque Z vale 0.
Call tiene que guardar primero PC y luego SP. SP apunta al último dato introducido, hay que incrementarlo antes de escribir. La única forma de que PC llegue a RM es a través de la ALU, que está ocupada por el cálculo de SP, debe esperar. El ciclo 5 y 6 se pueden intercambiar. SP decrece porque la pila crece hacia direcciones decrecientes.
(Esto no ocurre porque es un CALL y no una interrupción, pero nótese que si quisiéramos guardar SR también, llevar SR a RM es compatible con el cálculo de SP ya que la ALU tiene una salida que la conecta a SP y RD mediante el buffer alto, mientras que SR se conecta a RM por el buffer bajo). Acto seguido ocurre la bifurcación.
RETI carga PC y SR, aunque sólo hemos guardado PC. Esto sería un fallo en la programación; pero el ejercicio parece pensado para que ocurra esto ya que nos dan el dato necesario para saber cómo continuaría el programa. RETI Comienza restaurando SR porque el PC se guarda primero, luego SR está arriba en la pila.
En la pila ya había escrito A0001h en la cabecera. Nosotros agregamos A0003h, que era la dirección de la instrucción siguiente al momento de ejecutar el CALL. En el RETI tomamos A0003h como si fuera el SR y PC se restaura con lo que había debajo, que era A0001h. Así que el programa continúa por el MOVE.
Aquí es la única vez que el cálculo prematuro del desplazamiento es útil.

Notas:
Hay que respetar la causalidad. Si una instrucción va a lanzar una excepción porque una operación en la ALU falla, no puede preparase para el tratamiento de la excepción antes de que esta se produzca; y tampoco deben emitirse las acciones previas a la excepción que resulten inútiles debido a la excepción.
La regla más importante para el paralelismo es respetar los buffers. Si dos operaciones elementales requieren hacer uso del mismo buffer, no se pueden simultanear

RD y RM deben mantener su valor durante todos los ciclos de la operación en memoria, excepto en el último ciclo.
P0 Ocurre siempre en el último ciclo de cada instrucción. (Pone RF a 0)
FEST ocurre siempre que la ALU hace un cálculo que fue pedido explícitamente por el programador. Es decir, no se tiene en cuenta el incremento del PC, las operaciones con la pila…
La instrucción CALL sólo guarda el PC. Una excepción guarda PC y SR.
La instrucción RET restaura sólo el PC, RETI restaura PC y SR.

Ej: 19 Jun

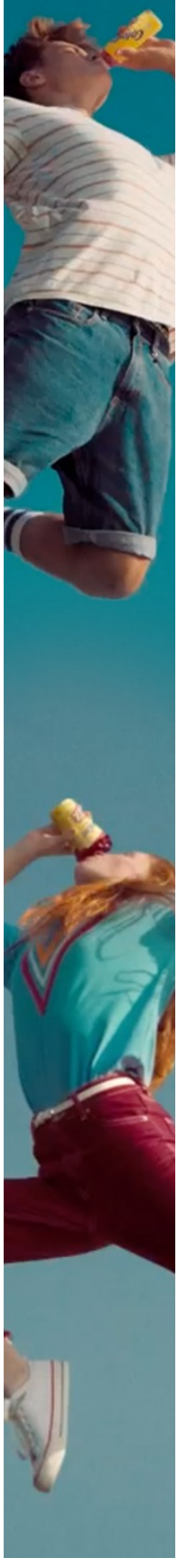
3. Definir el valor de las señales de control correspondiente a la secuencia ordenada de operaciones elementales **D ← PC** y **PC ← Desplazamiento** (en RI) + **R3**, realizadas en el menor tiempo posible. Mostrar únicamente el valor de las señales que intervienen directamente en dicha operación.



La primera operación consiste en abrir TP para que el dato de PC inunde el buffer alto, y activar FD para que RD fije este valor.

La segunda operación selecciona en el primer canal del banco de registros el R3 e YY pasa a ser 1 para que entre a la ALU. El otro valor viene de RI así que hay que abrir TI, inundando el buffer bajo, y seleccionar 1 en XX. La OP es una suma. El resultado puede ir directamente al resultado de destino ya que sale por el buffer alto activando TD y se guarda en PC con la señal FP.

Nótese que las señales que tienen los registros son distintas ya que el chip necesita un flanco de bajada.



Nota: Aunque en este ejercicio diga que las operaciones dadas son elementales, en realidad no suelen serlo. Por ejemplo, si piden que el resultado de sumar un registro con un dato en RM y que se guarde en otro registro, eso serían dos operaciones elementales ya que el resultado tendría que guardarse en RA antes de poder llevarlo al banco de registros. O cuando mencionan la memoria.

Notas:
El dato en las operaciones de bifurcación (BZ, BNZ, JUMP, CALL) es la instrucción, no la dirección de la instrucción. Lo que imposibilita direccionamiento inmediato, porque una instrucción no puede contener otra instrucción.

Modos de direccionamiento

- **Inmediato**: El operando está en la instrucción.
  No se requieren accesos a memoria. Ej: ADD #24
- **Directo**: Apunta a la dirección del elemento
  - **Absoluto**: No se requieren operaciones
    - **A memoria**: Ej: ADD 24
    - **A registro**: Ej: ADD .4
    - **A página 0**: Desplazamiento con respecto a una dirección de MP. Ej: ADD !24
  - **Relativo**: Desplazamiento con respecto a un puntero
    - **Contador de programa**: Ej: ADD $24
    - **Registro base**: Ej: ADD 12(.4)
    - **Registro índice**: el valor del registro se ve alterado
      - **AutoPostIncremento**: Ej: ADD 12(.4++)
      - **AutoPostDecremento**: Ej: ADD 12(.4--)
      - **AutoPreIncremento**: Ej: ADD 12(++.4)
      - **AutoPreDecremento**: Ej: ADD 12(--.4)
    - **Pila**:
- **Indirecto**: Apunta a un puntero
  - **Absoluto**:
    - A memoria: Ej: ADD [24]
    - A registro: Ej: ADD [.4]
    - A página 0: Ej: ADD [!24]
  - **Relativo**:
    - **Contador de programa**: Ej: ADD [$24]
    - **Registro base**: Ej: ADD [12(.4)]
    - **Registro índice**: el valor del registro se ve alterado
      - **AutoPostIncremento**: Ej: ADD [12(.4++)]
      - **AutoPostDecremento**: Ej: ADD [12(.4--)]
      - **AutoPreIncremento**: Ej: ADD [12(++.4)]
      - **AutoPreDecremento**: Ej: ADD [12(--.4)]
- **Implícito**: El código de la instrucción define la dirección del dato.
  Ej: CLC pone a 0 el flag C.

Ejercicio de DLX

Adelantamientos en DLX

Son optimizaciones que suelen evitar que se produzca una detención o acortan los ciclos de la detención.
Sin adelantamiento, las fases ID tienen que esperar a que los datos estén disponibles. Es decir, si una operación requiere un dato que va a ser actualizado por otra operación, la fase ID se tendrá que ejecutar en el ciclo siguiente en el que la otra operación haga WB. Los ciclos entre IF e ID serán una detención.
Mientras que, si tenemos adelantamiento, podemos llevar los datos de una fase, directamente a la fase de otra operación. Los adelantamientos se clasifican en función de qué fase es la fuente y cuál es el destino. En estos casos ID se puede realizar justo después de IF y sólo serán necesarias las detenciones mínimas para que la fase destino suceda en el instante siguiente al de la fase de origen.

- MEM-MEM
- ALU-ALU
- MEM-ALU
- ALU-MEM

Por último, el adelantamiento no resuelve las detenciones estructurales.

Detenciones en DLX

En DLX las instrucciones se liberan una por cada ciclo.
Las detecciones ocurren cuando este ritmo natural provocaría que el resultado no fuera equivalente con una ejecución en serie, cuando dos instrucciones llegan a la misma fase a la misma vez, o cuando se da una combinación de fases que la arquitectura no soporta. Dependiendo de la causa de la detención se clasifican en:

- **Estructural**: Ocurre si dos instrucciones llegan a la misma fase en el mismo ciclo. La instrucción más nueva es detenida. También ocurre por una limitación de la arquitectura, IF y MEM no se pueden realizar a la vez porque las dos acceden a memoria. (Sólo si la instrucción que hace MEM tiene que acceder a la memoria, por ejemplo, el MEM de un FADD que sólo usa registros es compatible con el IF de otra instrucción)
- **RAW** (read after write): una instrucción posterior intenta leer un dato que la instrucción anterior modifica, y esta aún no la ha

escrito. La fase ID se retrasa ya que, si no hay adelantamiento, ID debe disponer de los datos para el cálculo.

- **WAW** (write after write): dos instrucciones escriben el mismo dato pero la anterior lo hace después de la posterior, sobrescribiendo. El problema no es que se pierda, sino que en una ejecución en serie el que debería perderse es el otro dato. En DLX es posible que una instrucción posterior acabe antes que una previa, si se da el caso hay que retrasar MEM si es guardada en memoria o WB si es en registro.
- **WAR** (write after read): una instrucción anterior lee un dato que será modificado por una instrucción futura y está lo escribe antes de que la primera lo lea. En DLX es imposible que ocurra.

Importante, las detenciones se propagan hacia abajo. Es decir, en una columna donde hay una detención, no puede haber ninguna fase en una fila debajo de la detención.

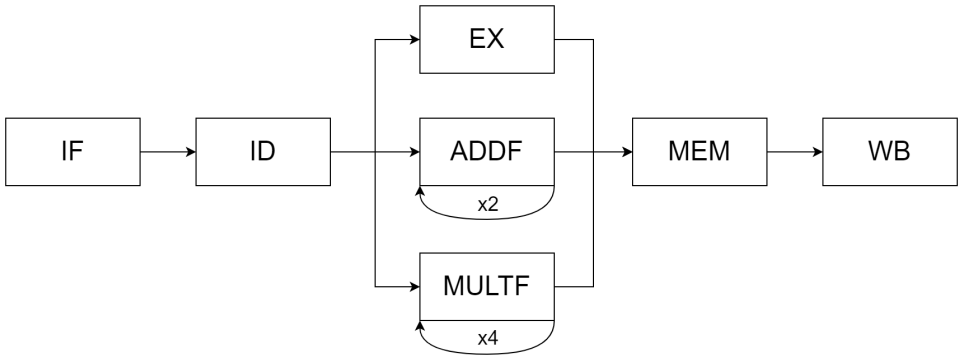
1º. Dibujar el esquema de la arquitectura en base al enunciado. Especialmente los ciclos que requieren las unidades de punto flotante y si están segmentadas o no…

Ej: 19 JUN

PROBLEMA 3. (2 ptos.). El siguiente fragmento de código se ejecuta en un procesador con arquitectura DLX. Las latencias (en ciclos de reloj) de las unidades funcionales son las siguientes: Sumador/restador entero: 1; Sumador/restador flotante: 2; y Multiplicador/Divisor: 4 (las unidades funcionales para operaciones en coma flotante **no** están **segmentadas**).

Para realizar el ejercicio necesitamos saber qué adelantamientos se pueden realizar. La caché siempre está unificada, ese dato no es importante.
Sin adelantamientos, unidades de coma flotante no segmentadas

a) Suponiendo que la **memoria caché está unificada** y **no existe adelantamiento**, indicar el estado de cada instrucción durante los ciclos de ejecución del código; calcular el número de ciclos necesarios para ejecutar este código; indicar si existen bloqueos en la cadena y a qué se deben.



Apartado a)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
LF F2, 20(R1)	IF	ID	Ex	MEM	WB																		
SF 30(R1), F2		IF	dat1	dat1	ID	Ex	MEM	WB															
ADDI R1, R1, #4					IF	ID	Ex	MEM	WB														
FMULT F6, F7, F8						IF	ID	MULTF 0	MULTF 1	MULTF 2	MULTF 3	MEM	WB										
FADD F6, F7, F9							dat2	IF	ID	ADDF 0	ADDF 1	dat3	MEM	WB									
FDIV F1, F2, F3									IF	ID	dat4		MULTF	MULTF	MULTF 2	MULTF 3	MEM	WB					
FADD F3, F1, F6										IF			dat5	dat6	dat6	dat6	dat6	ID	ADDF 0	ADDF 1	MEM	WB	
SF 30(R0), F1																		IF	ID	Ex	dat7	MEM	WB

Detenciones
1: RAW(F2) LF - SF
2: ESTRUCTURAL SF(MEM) - FADD(IF)
3: ESTRUCTURAL (MEM) FMULT - FADD
4: ESTRUCTURAL (MULTF) FMULT - FDIV

Detenciones
5: RAW(F6) FADD - FADD
6: RAW(F1) FDIV - FADD
7: ESTRUCTURAL (MEM) FADD - SF

Adelantamientos

Lo mejor es no pasar nunca a la siguiente instrucción sin terminar de escribir todas las fases de la anterior. Cuando no hay ningún tipo de adelantamiento la única fase que se puede escribir sin comprobar dependendencia de datos es IF, ya que ID siempre debe realizarse cuando ya estén todos los datos disponibles. Aunque es mejor comprobar las variables antes de empezar con una instrucción.

La primera instrucción siempre se puede realizar sin ninguna consideración.

Analizando la dependencia de la segunda instrucción, encontramos que se lee F2 y una instrucción anterior (LF) escribe en F2. Esto no tiene porqué ser un problema, pero en este caso hay que introducir detenciones para retrasar el ID hasta la columna 5, debajo del WB de la instrucción que escribe F2.

La tercera instrucción no puede empezar en el instante 3 ni 4 por la propagación de detenciones. Importante, el registro R1 aparece en la instrucción anterior en el campo destino, pero esa instrucción tiene un modo de direccionamiento directo relativo a registro base, lo que significa que no se escribe en R1, sino que se lee para obtener la dirección de memoria en la que se escribe. Por eso SF y ADDI no tienen

dependencia.

La cuarta instrucción no tiene ninguna dependencia. Aún así hay que seguir comprobando que no haya ninguna detención estructural. Para eso, siempre que escribimos una fase, miramos hacia arriba en la columna para ver si algo es incompatible. No es el caso.

La quinta instrucción no puede empezar en la columna 7, porque se juntaría el IF con un MEM de la segunda instrucción. Como la segunda instrucción hace uso de la memoria, porque escribe en ella, su MEM no es compatible con otros IF. Sin embargo, la tercera instrucción usa direccionamiento inmediato y directo absoluto a registro, por lo que no necesita acceder a memoria; luego el MEM del ADDI es compatible con el IF del FADD.No tiene dependencia de datos con ninguna instrucción, recordemos que sólo nos interesa las dependencias que tenga con instrucciones anteriores. Hay que introducir otra detención para que no coincidan dos fases MEM. Si nos fijamos, FMULT y FADD escriben en el mismo registro, lo que es un riesgo de WAW, pero acaban en orden natural así que no hay problema. La detección de WAWs se puede hacer únicamente cuando veamos que una instrucción posterior acaba antes que una anterior, porque suele darse poco y es la única forma de que se produzca un WAW.

La sexta instrucción no tiene dependencia de datos. La fase de ejecución se hace en la unidad MULTF, que no está segmentada, así que no puede realizarse en el instante 11 porque coincidiría con la cuarta instrucción. Tampoco puede empezar en el instante 12 por la propagación de detención.

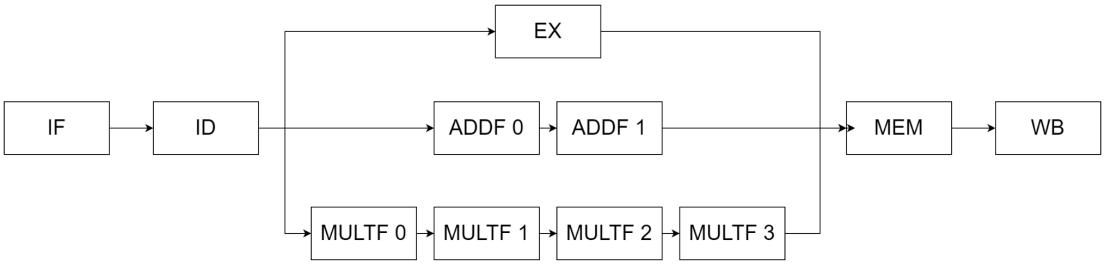
La séptima instrucción tiene dependencia con el primer FADD y el FDIV por F6 y F1, por esto hay que retrasar ID. Lo tendríamos que retrasar hasta el instante 14 para disponer de F6 y hasta el 18 para F1. En estos casos se tratan como dos detenciones diferentes. Nótese que en una columna sólo puede haber una detención, por eso det5 no aparece en la columna 11 ni 12.

La última instrucción tiene dependencia con el FDIV por F1, pero no es limitante porque el ID ya se ejecuta después del WB del FDIV sin tener que intervenir. Hay una detención estructural porque coinciden dos MEM.

Ej: 19 JUN

b) Repetir del apartado anterior considerando que **adelantamiento generalizado** y las unidades funcionales para operaciones en coma flotante **están segmentadas**.

Sin adelantamientos, unidades de coma flotante no segmentadas



Apartado a)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
LF F2, 20(R1)	IF	ID	Ex	MEM	WB																		
SF 30(R1), F2		IF	ID	Ex	MEM	WB																	
ADDI R1, R1, #4			IF	ID	Ex	MEM	WB																
FMULT F6, F7, F8				JdT1	JdT2	IF	ID	MULTF 0	MULTF 1	MULTF 2	MULTF 3	MEM	WB										
FADD F6, F7, F9							IF	ID	ADDF 0	ADDF 1	MEM	JdT3	JdT5	WB		4							
FDIV F1, F2, F3								IF	ID	MULTF 0	MULTF 1		2	MULTF 2	MULTF 3	MEM	WB						
FADD F3, F1, F6									IF	ID	JdT4			JdT4	JdT9	ADDF 0	ADDF 1	MEM	WB				
SF 30(R0), F1										IF						LD	EX	JdT5	MEM	WB			

Detenciones
1: ESTRUCTURAL: LF(MEM) - FMULT(IF)
2: ESTRUCTURAL: SF(MEM) - FMULT(IF)
3: WAW: (F6) FMULT - FADD
4: RAF: (F1) FDIV - FADD

Detenciones
5: ESTRUCTURAL(MEM) FADD - SF

Adelantamientos
1: (F2) MEM-MEM
2: (F6) ALU-ALU
3: (F1) ALU-ALU
4: (F1) ALU-MEM

Cuando tenemos adelantamiento generalizado podemos escribir IF e ID sin tener en cuenta la dependencia de datos, porque, ocurra lo que ocurra, ID nunca tendrá que esperar.

La primera instrucción nunca espera.

La segunda instrucción tiene dependencia con la primera por F2, pero ID no espera, en su lugar debemos identificar el tipo de adelantamiento que se necesita. En este caso es memoria a memoria, porque el LF carga un dato de memoria en un registro y el SF lo guarda de un registro a memoria, lo que tenemos que comunicar es la operación de memoria que lee con la que escribe. Luego, MEM de la primera instrucción tiene que suceder al menos un instante antes del MEM de la segunda instrucción para que pueda darse el adelantamiento, esto ya ocurre así que no se necesitan detenciones. Señalizamos el adelantamiento MEM a MEM.

La tercera instrucción no tiene dependencia de datos, ni se dan incompatibilidades.

La cuarta instrucción no puede empezar el el instante 4, ni en el 5 porque se pisaría con la fase MEM de las operaciones LF o SF y las dos

acceden a memoria.

La quinta instrucción no tiene dependencia de datos; pero si sigue la secuencia normal terminará antes que la instrucción anterior. Esto nos pone las orejas de punta y revisamos si hay WAW, lo hay porque FMULT y FADD escriben las dos en F6. Por esto hay que procurar que la fase del FADD que escriba F6 ocurra después de la del FMULT. Cómo F6 es un registro, debe ser WB el que se retrase. Si en lugar de F6 fuera una dirección de memoria, la fase a retrasar sería MEM.

La sexta instrucción no tiene dependencia de datos, sólo debe respetar la propagación de detenciones. En este caso ocurren a la vez varias fases MULTF, lo que no es problema porque la unidad está segmentada y nunca coinciden exactamente las mismas fases de MULTF.

La séptima instrucción tiene dependencia de F6 y F1. ID ocurre antes del WB del primer FADD y del FDIV, así que se necesitan dos adelantamientos. Los dos son de tipo ALU a ALU. Aunque tengamos adelantamiento, no se permite el viaje en el tiempo. La primera parte de la fase de ejecución del segundo ADDF debe esperar a que termine la última parte de la ejecución del FDIV, para eso ponemos detenciones. Ocurre un adelantamiento del ADDF del primer FADD al ADDF del segundo FADD, podríamos pensar que no es necesario ya que ha ocurrido el WB de esa instrucción, pero el adelantamiento siempre es necesario cuando el ID de la instrucción se ejecuta antes del WB de la instrucción de la que depende. Da igual lo que ocurra entre medias. El segundo adelantamiento ocurre entre MULTF del FDIV y el ADDF del segundo FADD.

La última instrucción depende de F1 y como su ID se produce antes del WB del FDIV, se necesita un adelantamiento de tipo ALU a memoria. También ocurre una detención estructural para evitar dos fases MEM a la vez.