Ejercicio 1:
Indicar si son correctas o no las siguientes afirmaciones, si se intenta realizar una comparación entre distintos sistemas de memoria tecnológicamente diferentes. Justificar su respuesta.

a. A menor tiempo de acceso, mayor coste por bit.
b. A menor capacidad, menor coste por bit.
c. A mayor capacidad, menor tiempo de acceso.
d. A mayor tiempo de acceso, mayor capacidad.
e. A mayor frecuencia de acceso, menor capacidad y mayor coste por bit.

a. Verdadero

b. Falso
c. Falso
d. Falso
e. Verdadero

Ejercicio 2:
Considerando un procesador que trabaja a 1.7GHz y los siguientes tiempos de acceso a memoria:

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5ns	$500–$1000
DRAM semiconductor memory	50–70ns	$10–$20
Flash semiconductor memory	5,000–50,000ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000ns	$0.05–$0.10

¿Cuantos ciclos de clock implica leer un dato de caché (SRAM) y de memoria principal (DRAM)?

t = 1 / hz
t = 1 / 1.7*10^12 = 0.588235294ns esto es lo que tarda un ciclo
F_{clk} = 1.7GHz Esto nos indica que cada 0.588235294ns se ejecuta un ciclo de reloj. Por cada ciclo podemos hacer un acceso a memoria.
Entonces

SRAM: 0.86 - 4.25 ciclos de reloj por acceso de memoria
DRAM: 86 - 120 ciclos de reloj por acceso de memoria

Ejercicio 2:
Considerando un procesador que trabaja a 1.7GHz y los siguientes tiempos de acceso a memoria:

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5ns	$500–$1000
DRAM semiconductor memory	50–70ns	$10–$20
Flash semiconductor memory	5,000–50,000ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000ns	$0.05–$0.10

¿Cuantos ciclos de clock implica leer un dato de caché (SRAM) y de memoria principal (DRAM)?

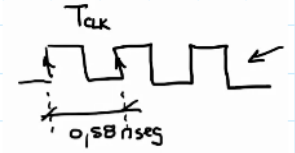
Nombre	Símbolo	Valor Exponencial	Valor decimal
Tera	T	$1x10^{12}$	1 000 000 000 000
Giga	G	$1x10^{9}$	1 000 000 000
Mega	M	$1x10^{6}$	1 000 000
Kilo	k	$1x10^{3}$	1 000
Hecto	h	$1x10^{2}$	100
Deca	da	$1x10^{1}$	10
Deci	d	$1x10^{-1}$	0,1
Centi	c	$1x10^{-2}$	0,01
Mili	m	$1x10^{-3}$	0,001
Micro	µ	$1x10^{-6}$	0,000 001
Nano	n	$1x10^{-9}$	0,000 000 001
Pico	p	$1x10^{-12}$	0,000 000 000 001

$f_{clk} = 1,7\,GHz$
t_{acc} (SRAM) $= 0,5$ nseg
t_{acc} (DRAM) $= 50$ nseg

ciclos MP → caché → 5%0,58 = 0,86 MP
mem ppal → 86 ciclos

$T_{clk\,MP} = \frac{1}{1.7 \times 10^{9}} = 5,88 \times 10^{-10} = 0,58$ n seg

Disco Duro: (HDD)
$\frac{5\ m\,seg}{0,58} = \frac{5 \times 10^{-3}}{0,58 \times 10^{-9}} = 8620689$ ciclos $\simeq 8,6$ M ciclos



En el tiempo en el que el procesador hace un clk se puede hacer un acceso a memoria

Ejercicio 3:
Calcular el tamaño total (en bits) de una caché de mapeo directo de 16KiB (16K x 8 bits) de datos y tamaño de bloque de 4 palabras de 32 bits c/u. Asuma direcciones de 64 bits.

Sabemos que tenemos 16KiB para datos en la memoria caché, ahora falta sumarle el bit de validación y el tag. Sabemos que en el área de datos guardamos 4 palabras de 32b por Línea y que tenemos 16KiB para datos, entonces podemos tener un máximo de 16*8*10^3 / 32*4 = 1000 líneas. Ahora para calcular la cantidad total de memoria el calculo seria 52*1000 +1000*4*32 + 1000 = 179000b
La longitud del tag está dada por el siguiente cálculo. Sabiendo que tenemos 64b de dirección los dividimos de la siguiente forma:

2 bits de palabra, 10 bits de index y 52 bits de tag

2 bits de palabra porque tenemos 4 palabras por bloque y 10 bits de index porque tenemos un máximo de 1000 líneas.
Por ende cada tag de la caché tendrá una longitud de 52bits.

Cantidad de bloques que puede tener la memoria principal / Cantidad de lineas = bits que necesito para los tags

Ejercicio 4:
Las memorias caché son fundamentales para elevar el rendimiento de un sistema de memoria jerárquico respecto del procesador. A continuación se da una lista de referencias de acceso a memoria (direcciones de 64 bits) las cuales deben ser consideradas como accesos secuenciales en ese mismo orden. El formato que se utiliza para cada dirección está reducido a sólo 16 bits, solo con fines prácticos:

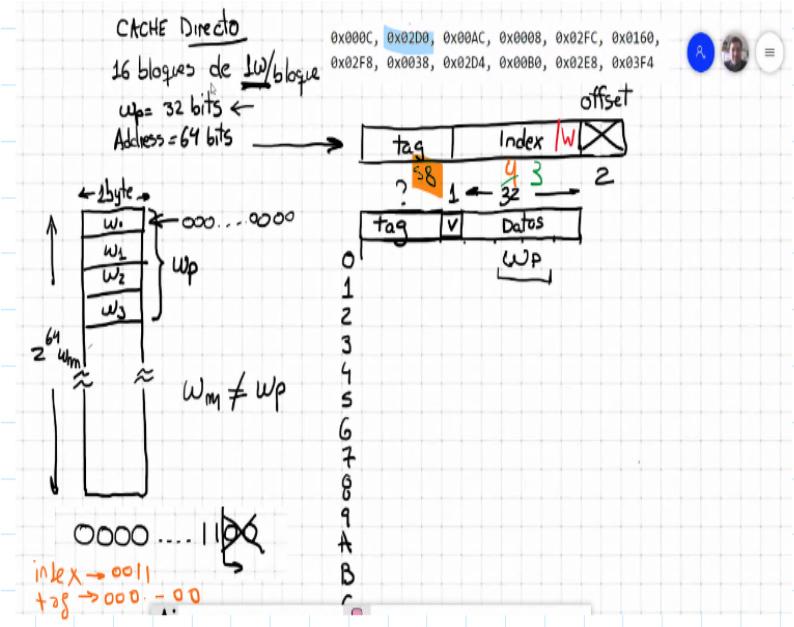
```
0x000C, 0x02D0, 0x00AC, 0x0008, 0x02FC, 0x0160,
0x02F8, 0x0038, 0x02D4, 0x00B0, 0x02E8, 0x03F4
```

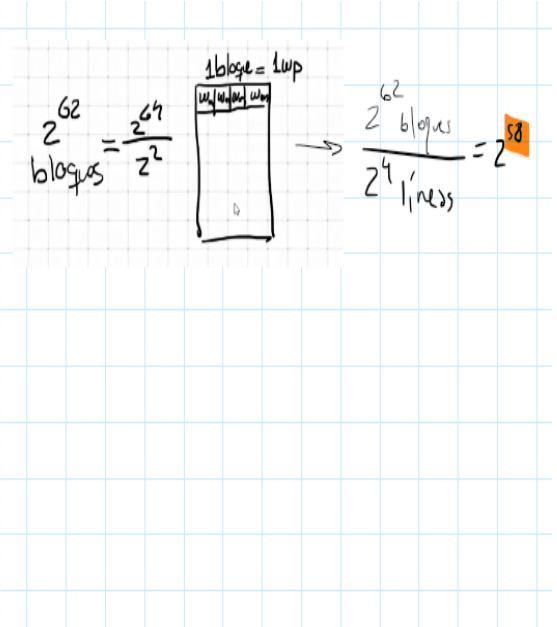
Se debe:

- **a.** Para cada una de estas referencias a memoria, determinar el binario de la dirección de cada palabra (cada palabra de 32 bits), la etiqueta (tag), el numero de linea (index) asignado en una cache de mapeo directo, con un tamaño de 16 bloques de 1 palabra c/u. Además liste qué referencias produjeron un acierto (hit) o un fallo (miss) de caché, suponiendo que la cache se inicializa vacía.
- **b.** Para cada una de estas referencias a memoria, determinar el binario de la dirección de cada palabra (cada palabra de 32 bits), la etiqueta (tag), el numero de linea (index) asignado en una cache de mapeo directo, con un tamaño de 8 bloques de 2 palabra c/u. Además liste qué referencias produjeron un acierto (hit) o un fallo (miss) de caché, suponiendo que la cache se inicializa vacía.

4.a) Dirección de memoria de 64 bits. Determinar el binario de la dirección de cada palabra de 32b, teniendo un tamaño de 16 bloques de 1 palabra c/u.
Para el campo de tag vamos a necesitar 48 bits ya que la cantidad de bloques que puede almacenar la memoria principal es 2^64 / 2^2, y el resultado de esto dividido en 16 que es la cantidad de líneas que tenemos en la caché, dejándonos 2^48. Para el campo index solamente vamos a necesitar 4 bits ya que tenemos 2^4 líneas y para el campo de offset no vamos a necesitar ningún bit ya que tenemo 1 sola palabra por linea pero vamos a ignorar los 2 primeros bits que armaron la palabra de 32 bits con 4 bytes de la memoria principal.

mente 0 de tag. Como la caché empieza vacía tenemos un miss.





a) RESUELTO POR TODOS.

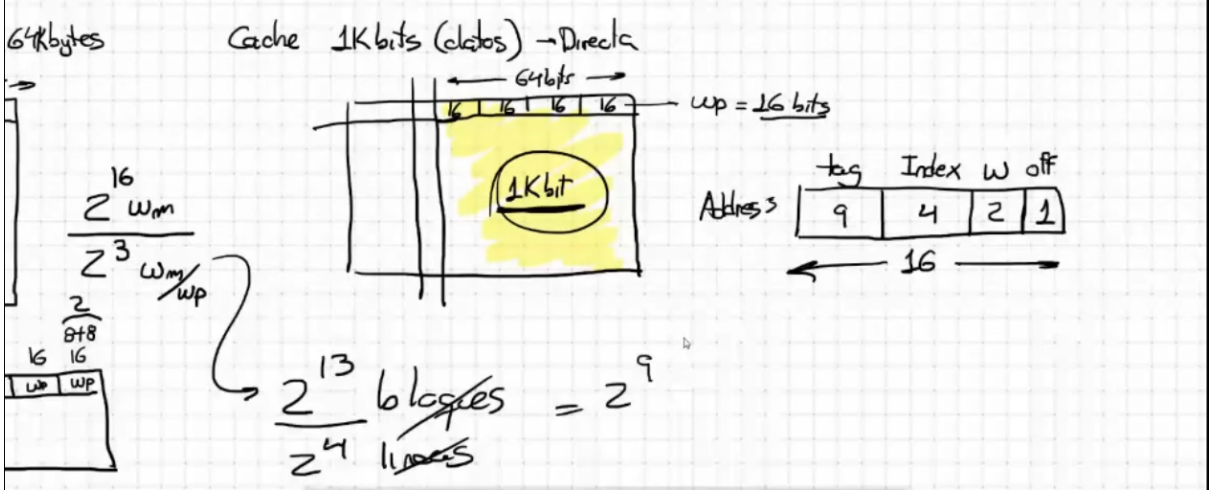
ADDRESS de 64 bits Wc = 32b DATA = 16 bloques de 1 Wc c/u es decir 16 Wc
Wp = 1 Byte = 8 bits

64 bits:
TAG →
INDEX → 4 bits.
W → Para distinguir palabras dentro de la caché.
OFFSET → Para distinguir Wp de Wc. en este caso Wc = 16b y Wp = 8bits => tenes dos Wp dentro del Wc. 1 bits para distinguirlas

donde Wp son las words de RAM o memoria principal
donde Wc son las words de cache

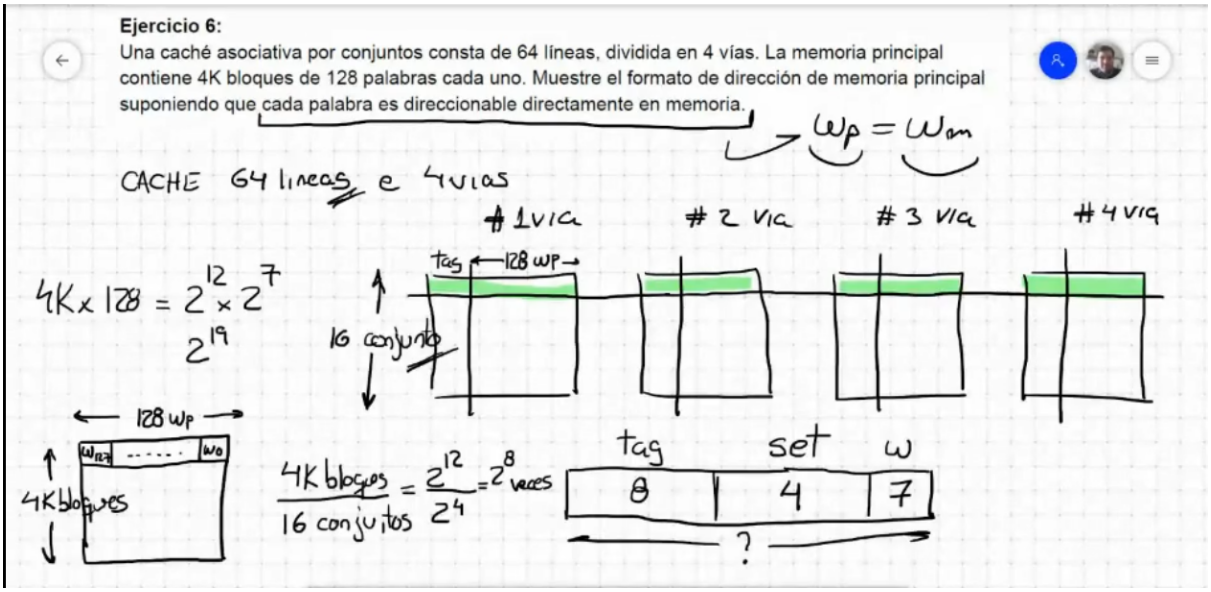
5.

¿Cuántos bits hay en cada línea de la memoria caché y cómo se dividen según su función?



6.

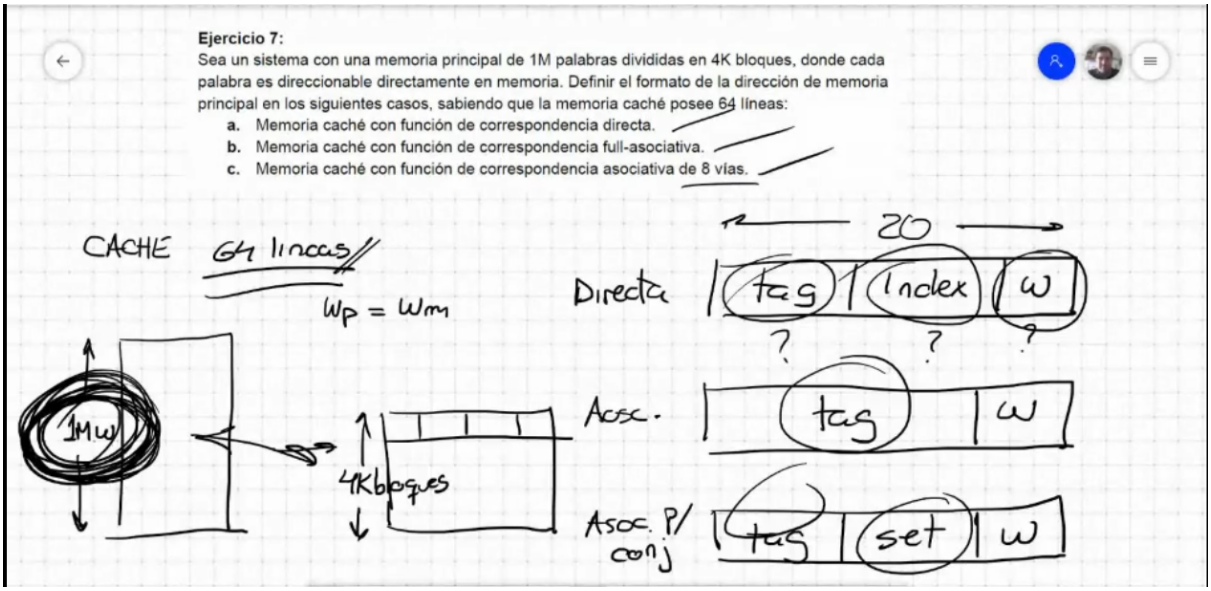
Ejercicio 6:
Una caché asociativa por conjuntos consta de 64 líneas, dividida en 4 vías. La memoria principal contiene 4K bloques de 128 palabras cada uno. Muestre el formato de dirección de memoria principal suponiendo que cada palabra es direccionable directamente en memoria.



7.

Ejercicio 7:
Sea un sistema con una memoria principal de 1M palabras divididas en 4K bloques, donde cada palabra es direccionable directamente en memoria. Definir el formato de la dirección de memoria principal en los siguientes casos, sabiendo que la memoria caché posee 64 líneas:

a. Memoria caché con función de correspondencia directa.
b. Memoria caché con función de correspondencia full-asociativa.
c. Memoria caché con función de correspondencia asociativa de 8 vías.



PRÁCTICO 6:

Ejercicio 1: Deep Pipelines

Considere construir un procesador con pipeline dividiendo el procesador de un solo ciclo en N etapas. El procesador de ciclo único tiene un retardo de propagación a través de la lógica combinacional de 740ps. La penalidad por agregar un registro de pipeline es de 90ps. Suponga que el retardo de la lógica combinacional se puede dividir arbitrariamente en cualquier número de etapas y que la lógica de hazard del pipeline no aumenta el retardo.

Asumiendo que un pipeline de cinco etapas tiene un CPI de 1.23 y que cada etapa adicional aumenta el CPI en 0.1 debido a las predicciones de salto erróneas y otros hazard. ¿Cuántas etapas de pipeline deberían usarse para hacer que el procesador ejecute los programas lo más rápido posible?

Retardo de propagación a través de la lógica combinacional de 740ps.
La penalidad por agregar un registro de pipeline es de 90ps.
Siendo N la cantidad de etapas para ejecutar una instrucción.
Latencia de una etapa= Tiempo de un ciclo = (740/N) + 90 ps
CPI = 1.23 + 0.1(N-5)
Tiempo por instrucción = Tc * CPI

N	CPI	T_c	T_i
5	1.23	238 ps	1.23 x 238 ps = 292,74 ps
6	1.33	213,33 ps	1.33 x 213,33 ps = 283,7289 ps
7	1.43	195,71 ps	279,8653 ps
8	1.23 + 0.3 = 1.53	92.5 + 90 = 180.5	1.53 * 180.5 = 276,165 ps

Ejercicio 2: Predictores de saltos

Asuma un microprocesador con 20 etapas de pipeline, con un fetch que levanta 5 instrucciones por ciclo. Este procesador ejecuta un código donde 1 de cada 5 instrucciones es un salto y esta conformado por bloques de 5 instrucciones donde la última es un salto.

¿Cuántos ciclos de instrucción toma hacer fetch de todas las instrucciones? Considerando predictores con las siguientes precisiones: 100%, 99%, 98%, 95%.

Precisión del 100%:
100 ciclos. No se hace ningún fetch incorrecto
Precisión del 99%:
100 (camino de ejecución correcto) + 20 (fetches incorrectos) = 120 ciclos
20% instrucciones extras levantadas
Precisión del 98%:

100 (camino de ejecución correcto) + 20 * 2 (fetches incorrectos) = 140 ciclos
40% instrucciones extras levantadas
Precisión del 95%:
100 (camino de ejecución correcto) + 20 * 5 (fetches incorrectos) = 200 ciclos
100% instrucciones extras levantadas

Ejercicio 3: Predictores de saltos

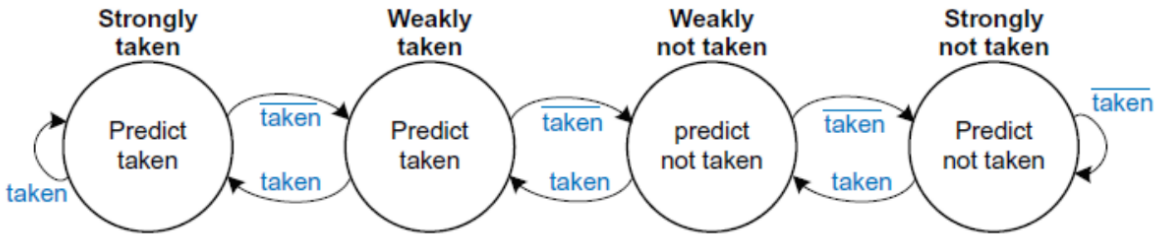


Figure 7.62 Two-bit branch predictor state transition diagram

Este ejercicio analiza la precisión de varios predictores de saltos para el siguiente patrón repetitivo (ej, en un loop) donde los saltos resultaron: **Taken - Not Taken - Taken - Taken - Not Taken.**

a) ¿Cuál es la precisión de los predictores *always-taken* y *always-not taken* para el patrón dado?
b) ¿Cuál es la precisión del predictor de *2-bits* para los primeros 4 saltos de este patrón? Asumir que el predictor arranca en *Strongly not taken*.
c) ¿Cuál es la precisión de este predictor de *2-bits* si el patrón completo se repite infinitamente?

3.a)

Patrón	Taken	Not taken	Taken	Taken	Not Taken
Predicción Not taken	NT	NT	NT	NT	NT
Predicción Taken	T	T	T	T	T

Para el branch predictor Not Taken la certeza será de un %40.
Para el branch predictor Taken la certeza será de un %60.

3.b)

Patrón	T	NT	T	T
Predicción	NT	NT	NT	NT
Acierto	x	OK	x	x

El branch predictor de 2 bits comenzando en strongly not taken da una certeza de %25.

3.c)
SNT(T) -> WNT(NT) -> SNT(T) -> WNT(T) -> WT(NT) ->
WNT(T) -> WT(NT) -> WNT(T) -> WT(T) -> ST(NT) ->
WT (T) 1 -> ST(NT) -> WT(T) 2 -> ST(T) 3 -> ST(NT) -> }
WT (T) 4 -> ST(NT) -> WT(T) 5 -> ST(T) 6 -> ST(NT) -> } Patrón principal.
WT (T) -> ST(NT) -> WT(T) -> ST(T) -> ST(NT) -> }
WT (T) -> ST(NT) -> WT(T) -> ST(T) -> ST(NT) -> }
AD INFINITUM

Dentro del ciclo de predicciones que se genera hay un 60% de certeza.

Ejercicio 4: Predictores de saltos
El siguiente código en C puede escribirse en ARMv8 de la siguiente forma:

```
for (i = 0; i <= 100; i++) {
   for (j = 0; j < 3; j++) {
      …
   }
}
```

```
0x00: L2:    add x0, xzr, xzr
0x04: L1:    add x1, xzr, xzr
0x08:          …
0x0C:          addi x1, x1, 1
0x10:          cmpi x1, 3
0x14:          b.lt L1
0x18:          addi x0, x0, 1
0x1C:          cmpi x0, 99
0x20:          b.lt L2
```

---

a) Mostrar como queda la tabla de historial de patrones (PHT) considerando que el procesador que ejecuta este código, cuenta con un predictor de saltos local de dos niveles.
b) Comparar la precisión de este predictor con uno de 2-bits (despreciando los primeros ciclos de iniciación).

Predictor de saltos local de dos niveles.

Evaluacion	**Valor**	**GR**	**Resultado**
j < 3	j = 0	1101	Taken
j < 3	j = 1	1011	Taken
j < 3	j = 2	1011	Taken

j < 3	j = 3	0111	Not Taken
i < 100	i = 10	1110	Taken

Ejercicio 5: Predictores de saltos

Asuma que el siguiente código itera en un array largo y lleno de números enteros positivos aleatorios. El código cuenta con 4 saltos, etiquetados B1, B2, B3 y B4. Cuando decimos que un salto es Taken, nos referimos a que el código dentro de las llaves es ejecutado.

```
for (int i=0; i<N; i++) {        /* B1 */
   val = array[i];               /* TAKEN PATH for B1 */
   if (val % 2 == 0) {           /* B2 */
      sum += val;                /* TAKEN PATH for B2 */
   }
   if (val % 3 == 0) {           /* B3 */
      sum += val;                /* TAKEN PATH for B3 */
   }
   if (val % 6 == 0) {           /* B4 */
      sum += val;                 /* TAKEN PATH for B4 */
   }
}
```

a) Determinar cuál de los cuatro saltos muestra una correlación local.
b) ¿Existe correlación global entre algunos de los saltos? Explicar.

a) El primer salto muestra una correlacion local ya que los demas saltos dependen de valores aleatorios.
b) Existe una corellacion global entre el salto B2, B3 y B4. Si se toma B2 y B3 entonces tambien se toma B4.

quien sos

Ejercicio 7: Static Multiple Issue Processor

En este ejercicio se compara el rendimiento de los procesadores de 1-issue y 2-issue, teniendo en cuenta las transformaciones que se pueden realizar en un programa para optimizar la ejecución de 2-issue.

Los problemas en este ejercicio se refieren al siguiente bucle (escrito en C):

```
for(i=0;i!=j;i+=2)
    b[i]=a[i]-a[i+1];
```

Un compilador con poca o ninguna optimización podría generar el siguiente código de assembler LEGv8:

```
        ADD X5, XZR, XZR
        B ENT
TOP:    LSL X10, X5, #3
        ADD X11, X1, X10
        LDUR X12, [X11, #0]
        LDUR X13, [X11, #8]
        SUB X14, X12, X13
        ADD X15, X2, X10
        STUR X14, [X15, #0]
        ADDI X5, X5, #2
ENT:    CMP X5, X6
        B.NE TOP
```

El código utiliza los siguientes registros:

i	j	a	b	Temporary values
X5	X6	X1	X2	X10–X15

Asumiendo que el procesador de 2-issue tiene las siguientes propiedades:

1. En cada *issue packet* una instrucción debe ser una operación de memoria y la otra una de tipo aritmética/lógica o un salto.
2. El procesador tiene todos los caminos de forwarding posibles entre las etapas (incluyendo los caminos a la etapa ID para la resolución de saltos).
3. El procesador predice los saltos perfectamente.
4. Dos instrucciones no pueden procesarse juntas en un paquete si una depende de la otra.
5. Si se requiere un stall, ambas instrucciones en el paquete deben volverse stall.

```
1.      ADD X5, XZR, XZR
2.      B ENT
3.TOP:  LSL X10, X5, #3
4.      ADD X11, X1, X10
5.      LDUR X12, [X11, #0]
6.      LDUR X13, [X11, #8]
7.      SUB X14, X12, X13
8.      ADD X15, X2, X10
9.      STUR X14, [X15, #0]
10.     ADDI X5, X5, #2
11. ENT:CMP X5, X6
12.     B.NE TOP
```

a) Analice en el código las dependencias de datos y determine cuales generan data hazards en nuestro procesador one-issue, **sin** forwarding-stall. En cada caso indique: el tipo de hazard, el operando en conflicto y los números de las instrucciones involucradas. Suponga que los saltos están perfectamente predichos, de modo que los *control hazard* son manejados por hardware.

Tipo	Lineas	Registro
Data	1 y 11	X5

Ejercicio tipo parcial:
Un procesador 2-issue de arquitectura LEGv8 posee las siguientes propiedades:
1. En cada issue packet una instrucción debe ser una operación de acceso a memoria y la otra de tipo aritmética/lógica o un salto.
2. El procesador tiene todos los caminos de forwarding posibles entre las etapas (incluyendo caminos a la etapa ID para la resolución de saltos).
3. El procesador predice los saltos perfectamente.
4. Dos instrucciones no pueden procesarse juntas en un paquete si una requiere el resultado de la otra.
5. El compilador asume toda la responsabilidad de eliminar los hazard, organizar el código e insertar instrucciones “nop” para que el código se ejecute sin necesidad de generación de stalls.
Para el siguiente fragmento de código LEGv8 (donde X2 = 0):

1. ADDI X0, XZR, #0x100
2. ADDI X10, XZR, #50
3. loop: LDUR X1, [X0,#0]
4. ADD X2, X2, X1
5. LDUR X1, [X0,#8]
6. SUBI X10, X10, #1
7. ADD X2, X2, X1
8. STUR X2, [X0,#8]
9. ADDI X0, X0, #16
10. CBNZ X10, loop

a. Dibuje un diagrama de pipeline que muestre cómo se ejecuta el código LEGv8 dado en el procesador de 2-issue (sólo hasta completar una iteración del bucle). Sin modificar el orden de ejecución, organice el código para evitar la mayor cantidad posible de stalls. Indique los caminos de forwarding utilizados. (Completar en la tabla dada al final del ejercicio).

1. ADDI X0, XZR, #0x100 fow ex1
2. NOP

3. ADDI X10, XZR, #50
4. loop: LDUR X1, [X0,#0] fow mem1

STALL

5. ADD X2, X2, X1
6. LDUR X1, [X0,#8] fow mem 2

7. SUBI X10, X10, #1
8. NOP

9. ADD X2, X2, X1 fow ex 2 , fow ex 3
10. NOP

11. STUR X2, [X0,#8] fow mem 3
12. ADDI X0, X0, #16
    - Va en el mismo issue porque Xo se pisa recién en la etapa de wb y lo utilizamos en ex

13. ADDI X0, X0, #16
14. NOP

instr	clk 1	2	3	4	5	6	7	8	9	10	11	12		
1	1	F	D	E	M	W								
-	1	-	-	-	-	-								
2	2		F	D	E	M	W							
3	2		F	D	E	M	W							
-	3			-	-	-	-	-						
-	3			-	-	-	-	-						
4	4				F	D	E	M	W					
5	4				F	D	E	M	W					
6	5					F	D	E	M	W				
-	5					-	-	-	-	-				
7	6						F	D	E	M	W			
-	6						-	-	-	-	-			
8	7							F	D	E	M	W		
9	7							F	D	E	M	W		
10	8								F	D	E	M	W	
-	8								-	-	-	-	-	

b. Suponiendo que no es económicamente viable integrar los multiplexores de tres entradas que son necesarios para implementar todos los caminos de forwarding, analice el código dado y determine si es mejor reenviar solo desde el registro de pipeline EX / MEM (EX → EX) o solo desde el registro MEM / WB (MEM → EX).

- Los caminos de FW son dos de Ex → M y uno solo de M → Ex. Por lo tanto nos conviene Ex → M

c. Indique el aumento de velocidad en la ejecución del código dado al pasar de un procesador de 1-issue a un procesador de 2-issue, ambos con pipeline y forwarding-stall. Considere la totalidad de las iteraciones realizadas por el código.

1. ADDI X0, XZR, #0x100
2. ADDI X10, XZR, #50
3. loop: LDUR X1, [X0,#0] → 1 y 3 No tiene hazard por fw
   - stall
4. ADD X2, X2, X1 → 3 y 4 necesitan un 1 clk para usar el FW
5. LDUR X1, [X0,#8]
6. SUBI X10, X10, #1
7. ADD X2, X2, X1 → 7 Y 5 solucionado por fw
8. STUR X2, [X0,#8] → 7 y 8 solucionado por fw
9. ADDI X0, X0, #16
10. CBNZ X10, loop

- En total tenemos 11 clk para el procesado One-issue y por a) 8 para el procesado Two-Issue
- El aumento de la velocidad =%28 más rápido

d. Cuántas instrucciones LDUR se ejecutarán por lazo si se aplica la técnica del loop-unrolling al código dado, con el fin de minimizar la cantidad de iteraciones del lazo? Asuma que X10 se inicializa en la instrucción <2> con un valor múltiplo de 4.

1. ADDI X0, XZR, #0x100 fow ex1
2. NOP
3. ADDI X10, XZR, #50

----..----

4. loop: LDUR X1, [X0,#0] fow mem1
5.NOP
6'.NOP
7. ADD X2, X2, X1 →
8. LDUR X1, [X0,#8] fow mem 2
9. SUBI X10, X10, #2 → nueva
10. NOP
11. ADD X2, X2, X1 fow ex 2 , fow ex 3
12. NOP
13. STUR X2, [X0,#8] fow mem 3
14. ADDI X0, X0, #32

----..----

15. LDUR X1, [X0,#-16] fow mem1
16. NOP
17.STALL
18. STALL
19. ADD X2, X2, X1
20. LDUR X1, [X0,#-8] fow mem 2
21. NOP

22. ADD X2, X2, X1 fow ex 2 , fow ex 3
23. NOP
24. STUR X2, [X0,#-8] fow mem 3
25. CBNZ X10, loop
26. NOP

instr	clk 1	2	3	4	5	6	7	8	9	10	11	12		
1	1	F	D	E	M	W								
-	1	-	-	-	-	-								
2	2		F	D	E	M	W							
3	2		F	D	E	M	W							
-	3			-	-	-	-	-						
-	3			-	-	-	-	-						
4	4				F	D	E	M	W					
5	4				F	D	E	M	W					
6	5					F	D	E	M	W				
-	5					-	-	-	-	-				
7	6						F	D	E	M	W			
-	6						-	-	-	-	-			
8	7							F	D	E	M	W		
9	7							F	D	E	M	W		
10	8								F	D	E	M	W	
-	8								-	-	-	-	-	

LDURD D0, [X1, #0]
 FMULD D4, D0, D2

F	D	E	M	W

F	D	E	M	W