Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2343 – Arquitectura de Computadores

Guía 4 – Repaso Examen II

Profesor: Yadran Francisco Eterovic Solano

Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

Nota al lector

El título dice "solución propuesta" por una razón bien sencilla: Estos ejercicios pueden tener más de un desarrollo correcto. Lo que se pretende hacer aquí es mostrar un camino a la solución, sin excluir la posibilidad de rutas alternativas igual de correctas.

Preguntas

1. a. **(I1 - II/2014)** ¿Qué relación existe entre el tamaño de los elementos de memoria y su velocidad? Explique claramente su respuesta.

   La velocidad de los elementos de memoria está directamente relacionada a su precio. Un ejemplo de esto es el contraste entre la unidad de disco duro (`HDD`, que trabaja con almacenamiento magnético) y la unidad de estado sólido (`SSD`, que utiliza circuitos eléctricos ensamblados para la memoria). Por lo mismo, en el mercado el tamaño y la velocidad de estos dispositivos están relacionados de forma inversamente proporcional: a mayor velocidad, menor es su tamaño (lo que ayuda a evitar precios muy elevados).

   b. **(I1 - I/2017)** En la siguiente figura, si la frecuencia del *clock* que entra al *flip-flop* FF0 es F Hz, ¿cuál es la frecuencia del *clock* del *flip-flop* FFN?

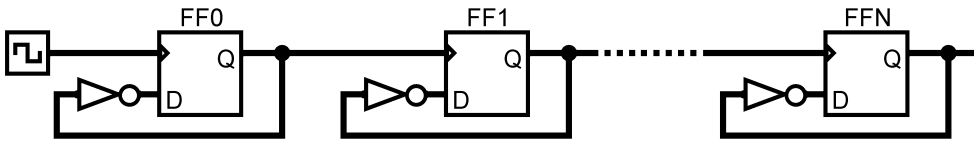


Figura 1: Secuencia de *flip-flops*, donde el *clock* de uno es la señal de estado del que lo antecede, salvo para el primero.

La parte clave de esta pregunta es notar la señal de *clock* que recibe FF1. Podemos ver que en vez de ser la misma señal del principio, es el estado Q del *flip-flop* FF0. Ahora, pensemos cómo varía la señal Q_{FF0} (*i.e.* la señal Q del *flip-flop* FF0). Si la frecuencia de dicho *flip-flop* es F, entonces a Q_{FF0} le toma un tiempo $\frac{1}{\text{F}}$ cambiar su valor (independiente si es de 0 a 1 o de 1 a 0). El hecho de que la entrada D reciba como valor la negación del estado Q_{FF0} es lo que permite simular una frecuencia (alternando sus valores). Finalmente, como Q_{FF0} se demora $\frac{1}{\text{F}}$ segundos en cambiar su valor y otros $\frac{1}{\text{F}}$ segundos en volver al inicial, su periodo es de $\frac{1}{\text{F}} + \frac{1}{\text{F}} = \frac{2}{\text{F}}$, lo que implica que su frecuencia será igual a $\frac{\text{F}}{2}$.

Si seguimos esta idea de forma análoga para la señal de *clock* que recibe FF2, veremos que a Q_{FF1} le toma $\frac{2}{\text{F}}$ segundos cambiar su valor y $\frac{2}{\text{F}}$ segundos volver al original, obteniendo un periodo de $\frac{2}{\text{F}} + \frac{2}{\text{F}} = \frac{4}{\text{F}}$ y una frecuencia de $\frac{\text{F}}{4}$.

A partir de lo anterior, obtenemos la siguiente relación de recurrencia:

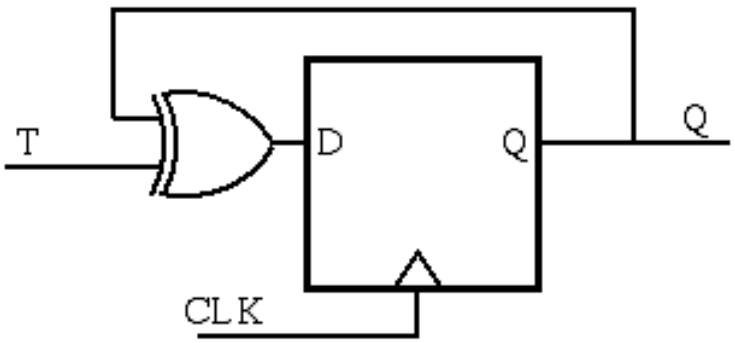
$$f_{\text{FF}}(i) = \begin{cases} \frac{f_{\text{FF}}(i-1)}{2}, & i > 0 \\ F, & i = 0 \end{cases}$$

Donde $f_{\text{FF}}(i)$ representa la frecuencia del *flip-flop* i. Finalmente, podemos ver que:

$$f_{\text{FF}}(N) = \frac{f_{\text{FF}}(N-1)}{2} = \frac{f_{\text{FF}}(N-2)}{4} = ... = \text{F} \prod_{i=0}^{N-1} \frac{1}{2} = \frac{\text{F}}{2^N}$$

c. **(I1 - I/2013)** Diseñe usando compuertas lógicas y *flip-flops* D, un *flip-flop* T. El comportamiento de este *flip-flop* consiste en invertir el valor de su salida Q si su señal de entrada T está en 1 y la señal de control C pasa de 0 a 1 (flanco de subida). En cualquier otro caso, la salida Q se mantiene igual.

Una posible solución se presenta en el siguiente diagrama:



De partida, vemos que la salida del *flip-flop* D correspondiente a la señal Q se conecta a una compuerta `XOR`, la que recibe además la señal T. Esto genera la siguiente tabla de casos:

T	Q	T `XOR` Q
0	0	0
0	1	1
1	0	1
1	1	0

Así vemos que, efectivamente, para $T = 1$ el resultado de la compuerta es el inverso de la señal Q. Se obtiene finalmente la siguiente tabla para el *flip-flop* T:

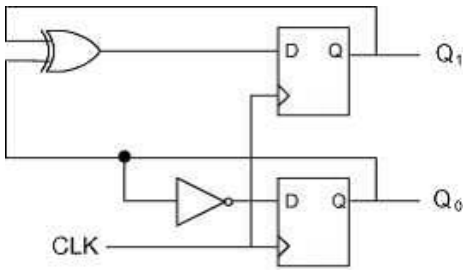
T	D	Q
0	Q	Q
0	Q	Q
1	$\overline{Q}$	$\overline{Q}$
1	$\overline{Q}$	$\overline{Q}$

d. **(I1 - I/2013)** Diseñe, utilizando todos los elementos de circuitos lógicos vistos en clases, un contador secuencial de 2 bits que se incrementa con cada flanco de subida de la señal de control.

Antes de presentar la solución, hay que ver intuitivamente lo que se necesita. Como el contador es de dos bits, lo que se busca replicar es la siguiente secuencia: 00, 01, 10, 11, 00. De aquí se aprecian las siguientes tendencias:

- El bit más significativo cambia su valor cada dos saltos y está sujeto a un flanco de bajada con respecto al bit menos significativo.
- El bit menos significativo, independiente del otro bit, va alternando su valor constantemente.

De esta forma, un contador que cumple los comportamientos antes señalados es el siguiente:



En este caso, el *flip-flop* inferior representa al bit menos significativo, mientras que el superior representa al más significativo. Se puede ver entonces que:

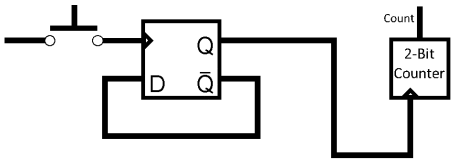
1) El *flip-flop* inferior recibe siempre como dato la negación del estado que poseía en la iteración previa, por lo que irá alternando su valor constantemente.

2) El *flip-flop* superior cambia su valor en dos casos:
   - Asumiendo que parte con un cero almacenado, se ve que de la compuerta `XOR` retorna un 1 una vez que el *flip-flop* inferior cambia su estado a 1. No obstante lo anterior, el *flip-flop* superior no cambia su valor hasta el próximo flanco de subida, lo que produce la secuencia 01-10.
   - Ya con un 1 almacenado, se ve que la compuerta `XOR` retorna un 0 cuando, nuevamente, el *flip-flop* inferior cambia su estado a 1[1]. De esta forma, en el siguiente flanco de subida el *flip-flop* superior cambia su estado a 0, generando ahora la secuencia 11,00.

   También es válido considerar la implementación de una secuencia de dos *flip-flops* para el bit más significativo, siendo el *clock* del poseedor del estado el bit la salida del primer *flip-flop*. De esta forma, este bit se actualizaría cada dos ciclos, generando el comportamiento correcto del contador de dos bits.

e. **(I1 - II/2014)** Diseñe, utilizando todos los elementos de circuitos lógicos vistos en clases que necesite, un contador secuencial circular ascendente de 2 bits, que se incrementa cada dos flancos de subida de la señal de control.

   Se utiliza un contador de dos bits simplificado, debido a que esto ya fue implementado en la pregunta anterior. Lo que se busca entonces es que el incremento se haga cada dos flancos de subida. Un circuito que logra dicho objetivo es el siguiente:



   Notar que esto es similar a lo que se vio con los *flip-flop* en serie: el *clock* que recibe el contador de dos bits provienen de una frecuencia generada por la alternancia de un *flip-flop* intermedio. De esta forma, la frecuencia de la señal que recibe el contador será igual a la mitad de la que recibe el *flip-flop* intermedio, esto implicará entonces un incremento en el contador cada dos iteraciones, o mejor dicho, cada dos flancos de subida.

[1] Recordar que `1 XOR 1 = 0`

f. **(I1 - I/2017)** Diseñe una memoria `RAM` que permita acceder (lectura y escritura) de manera individual a cada uno de los bits de una palabra. Tenga en consideración que los buses de datos de entrada y salida deben mantener su tamaño de una (1) palabra.

Nota: Puede basarse en esta como referencia.

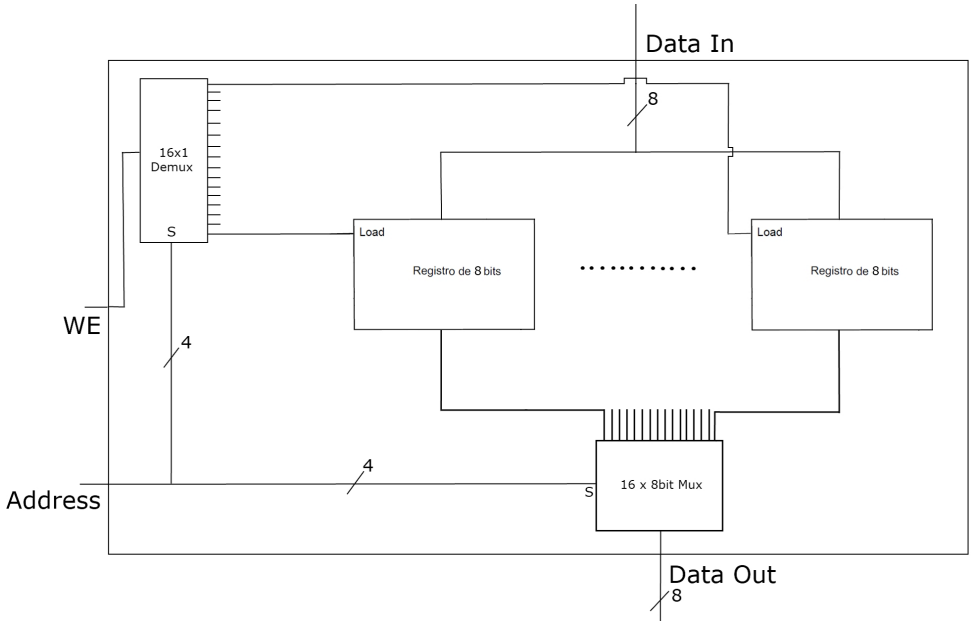


Figura 2: Diseño `RAM` de referencia.

Una metodología que permite cumplir este objetivo es la siguiente:

a) Se añade un nuevo bus de datos que llamaremos *bAddress*. Este debe ser de largo $\lceil \log_2(\text{largo palabra}) \rceil$, ya que de esa forma se tienen los bits suficientes para poder ubicar a todos los bits de las palabras almacenadas en la `RAM`.

b) Este nuevo bus de datos se añade como entrada en la `RAM`.

c) Ahora, los registros se modifican de forma que cada uno pueda actuar como una `RAM` de registros de un bit (en este caso, corresponderían a *flip-flops*). Cada uno recibe la señal `WE` de la RAM, la que será transmitida a través de un Demux. La diferencia es que este, en vez de recibir el bus *Address*, recibe *bAddress* para poder transmitir `WE` a un *flip-flop* específico. Cabe destacar que los *flip-flops* no poseen una entrada para habilitar la escritura, por lo que utilizamos un *Mux* con dos entradas: El estado actual `Q` del *flip-flop* y el bit que se busca sobreescribir. Si `WE = 0`, se escoge `Q`, en otro caso, el bit recibido. Ahora, como `WE` solo se transmite a uno de los *flip-flops*, el resto mantiene su estado bajo este esquema.

d) Lo siguiente a realizar es el manejo de los buses de entrada y salida. Esto se puede ver de dos formas:

- Se busca almacenar en el bit de un registro el i-ésimo bit de la palabra de entrada. En este caso, haciendo uso de *bAddress*, hacemos que el bus pase por un *Mux* de entradas de un bit, seleccionando solo el que es de interés. Este, posteriormente, es transmitido al resto de los registros. Para la salida, en cambio, existen dos formas: Se puede devolver el valor completo del registro, lo que se hace de la misma forma que en la `RAM`. En cambio, si se quiere solo una cadena de 0's con el bit escrito en la i-ésima posición, lo que se puede hacer es instalar un *Enabler* para cada bit, cuya señal de habilitación sea transmitida por el mismo *Demux* descrito en el comienzo. Así, solo el *Enabler* de la i-ésima posición permite pasar su bit, generando un valor de la forma 00..i..00.
- El bit a almacenar corresponde al bit menos significativo. Este caso es trivial, ya que en la entrada solo nos encargamos de transmitir el bit menos significativo obviando al resto, mientras que en la salida conectamos un bus auxiliar que contenga $[bitsporpalabra-1]$ 0's y se conecte con el bit recientemente modificado para retornar el resultado.

2. a. **(I3 - II/2012)** Dada un memoria *caché* de 1 KB con bloques de 16 palabras de 1 byte y función de correspondencia *fully associative*, que se encuentra llena, ¿cuantas comparaciones secuenciales se deben realizar para decidir qué bloque sustituir? Describa además un esquema que permita minimizar el tiempo requerido para este proceso, usando la misma cantidad de comparaciones.

   La *caché* tiene 64 líneas ($\frac{2^{10}}{2^4} = 2^6 = 64$), por lo que serán necesarias 63 comparaciones secuenciales dado que al ser *fully associative* se puede reemplazar en cualquiera de estas, surgiendo la necesidad de encontrar un mínimo o un máximo (dependiendo del algoritmo de reemplazo). Si estas se hacen se manera paralela, de 2 en 2 (en vez de ir revisando una por una), se realizan las mismas 63, pero demorarían aproximadamente $frac110$ del tiempo.

   b. **(I3 - II/2012)** Describa una posible solución al problema de consistencia entre *caché* y `RAM` del esquema *write-back*.

   Es posible solucionar esto agregando un bit de consistencia a cada bloque de la memoria `RAM`. Si el bloque ha sido modificado en la *caché* y no en la `RAM`, se *setea* este bit en 0. Luego, en caso que se requiera utilizar ese bloque por algún elemento del computador (ej. controlador `DMA`), si el bit de consistencia está en 0, deberá actualizarse el bloque en la memoria `RAM`.

   c. **(I3 - II/2012)** Considere una jerarquía de memoria dada por memoria `RAM` y *caché*, donde ambas utilizan palabras de 16 bits (16 bits = 2B = W). La memoria `RAM` tiene 1 GW (G = 1 Giga = 1024*1024*1024) de capacidad, mientras que la *caché* tiene de capacidad 1 MW (M = 1 Mega = 1024*1024) y bloques de 256W. Responda las siguientes preguntas, asumiendo que la *caché* utiliza una función de correspondencia *4-way associative*:

   - Calcule el número de bloques y conjuntos de la memoria *caché*.
     La *caché* de 1 MW tiene bloques de 256W, por lo tanto tiene $\frac{1024\times1024}{256} = 4096$ bloques. Luego, si la *caché* es *4-way associative*, tiene $\frac{4096}{4} = 1024$ conjuntos.
   - Describa la división de la direcciones de memoria, incluyendo *offset* dentro del bloque, conjunto y *tag*.
     La memoria principal es de 1 GW, por lo tanto tiene 2^{30} palabras de 16 bits, luego, las direcciones de memoria tienen 30 bits. Como los bloques son de 256W y tenemos además 1024 conjuntos, necesitamos los 8 bits menos significativos de la dirección para el *offset* y los 10 bits siguientes para el conjunto. Por lo tanto, los 12 bits más significativos se utilizan para el *tag*.

   d. **(I3 - I/2017)** La contención de bloques es un problema del esquema de mapeo directo, donde 2 o más bloques pelean por la misma línea, existiendo otras líneas no utilizadas en la *caché*. ¿Existe un problema similar en el esquema *N-way*? Si su respuesta es negativa, justifiquela, y si es positiva, indique detalladamente un caso en que esto se de.

   Sí, existe un problema similar, pero en menor grado. Se puede dar el caso dentro de un esquema *N-way* en el que los accesos sean solo de bloques pertenecientes a un mismo conjunto. Esto implicará que, eventualmente, el conjunto de la *caché* estará lleno y se deberán aplicar los algoritmos de reemplazo, existiendo la posibilidad de tener más conjuntos sin llenar. No obstante lo anterior, sigue teniendo un menor impacto que en el caso del mapeo directo, dado que primero se tiene que llenar el conjunto.

e. **(I3 - II/2011)** El siguiente programa se ejecutó en un computador con arquitectura x86 que tiene una *caché* de 4 bloques de 2 palabras cada uno:

Dirección	*Label*		
0		`MOV`	`[var2],1`
1	`loop1:`	`MOV`	`AL,[var2]`
2		`MUL`	`[var2]`
3		`CMP`	`[var1],AL`
4		`JL`	`end`
5		`INC`	`[var2]`
6		`JMP`	`loop1`
7	`end:`	`DEC`	`[var2]`
8		`RET`	
9	`var1`	`db ?`	
10	`var2`	`db 0`	

Cuadro 1: Programa con las direcciones y *labels*.

Al ejecutar el programa completo se obtuvo la siguiente secuencia de accesos a memoria:

0-10-1-10-2-10-3-9-4-5-10-6-1-10-2-10-3-9-4-5-10-6-1-10-2-10-3-9-4-7-10-8

Esta secuencia de accesos a memoria generó los siguiente estados en la *caché*:

Dir	B0	B1	B2	B3
0	0-1			
10	0-1	10-11		
1	0-1	10-11		
10	0-1	10-11		
2	0-1	10-11	2-3	
10	0-1	10-11	2-3	
3	0-1	10-11	2-3	
9	0-1	10-11	2-3	8-9
4	0-1	10-11	2-3	4-5
5	0-1	10-11	2-3	4-5
10	0-1	10-11	2-3	4-5
6	6-7	10-11	2-3	4-5
1	6-7	10-11	0-1	4-5
10	6-7	10-11	0-1	4-5
2	6-7	10-11	0-1	2-3
10	6-7	10-11	0-1	2-3

Dir	B0	B1	B2	B3
3	6-7	10-11	0-1	2-3
9	8-9	10-11	0-1	2-3
4	8-9	10-11	4-5	2-3
5	8-9	10-11	4-5	2-3
10	8-9	10-11	4-5	2-3
6	6-7	10-11	4-5	2-3
1	0-1	10-11	4-5	2-3
10	0-1	10-11	4-5	2-3
2	0-1	10-11	4-5	2-3
10	0-1	10-11	4-5	2-3
3	0-1	10-11	4-5	2-3
9	8-9	10-11	4-5	2-3
4	8-9	10-11	4-5	2-3
7	6-7	10-11	4-5	2-3
10	6-7	10-11	4-5	2-3
8	8-9	10-11	4-5	2-3

Cuadro 2: Estado de la *caché* por cada acceso.

En base a esta información, responda lo siguiente:

I. ¿Qué valores puede tener la variable `var1` para que efectivamente se genere la secuencia de accesos detallada previamente?
Para identificarlo, es importante ver la secuencia de accesos a memoria y el código en ejecución. Vemos que el valor de `var1` impacta en la cantidad de ejecuciones de `loop1`, puesto que es el valor con el que se compara el resultado de $\texttt{var2}^2$. En la secuencia, el código de `loop1` (dirección 1) se ejecuta tres veces, por lo que el crecimiento de `var2` debe seguir la siguiente secuencia:
   i) $1 \rightarrow 1^2 = 1 < \texttt{var1} \rightarrow 1++ = 2$
   ii) $2 \rightarrow 2^2 = 4 < \texttt{var1} \rightarrow 2++ = 3$
   iii) $3 \rightarrow 3^2 = 9 > \texttt{var1} \rightarrow \texttt{end}$

   El rango de valores, dada esta secuencia, es $[4, 8]$, ya que para cualquier valor de dicho intervalo $\texttt{var2}^2$ es menor o igual hasta que $\texttt{var2} = 3$.

II. ¿Cuál es el *hit rate*?
El *hit rate* es $\frac{17}{32}$, basta con contar la cantidad de veces donde la tabla de estados de la *caché* no es modificada entre dos accesos.

III. ¿Qué tipo de *caché* es: *unified* o *split*?
La *caché* es unified, lo que se observa dado que los bloques de la memoria no hacen una distinción entre las direcciones correspondientes a datos y las correspondientes a direcciones. Incluso se copian bloques que mezclan ambas (como 8-9).

IV. ¿Qué función de correspondencia y algoritmo de reemplazo (si corresponde) utiliza esta *caché*?
La función de correspondencia es *fully associative*, lo que se evidencia del hecho de que se llenan las líneas de la *caché* de forma secuencial y el reemplazo por dirección no se basa en un conjunto de bloques, sino que se basa en cualquiera que posea disponibilidad. Con respecto al algoritmo de reemplazo, este es *random*. Si bien en un principio parecía adoptar la lógica de `LFU` con desempate mediante `LRU`, el reemplazo generado en el acceso número 13 de la secuencia deja de seguir ese comportamiento.

V. ¿Es posible mejorar el desempeño de esta *caché* durante la ejecución de este programa, sin modificar la cantidad y tamaño de los bloques? Si es posible, explique una posible mejora que se podría realizar para lograr un mejor *hit rate* que el actual y demuestre que efecticamente su modificación logra mejorarlo. Si no es posible, justifique por qué.
Lo más sencillo es definir un algoritmo de reemplazo. Por simplicidad, se usará el que parecía existir al comienzo de la secuencia: `LFU` con desempate mediante `LRU`. A continuación, se presenta el estado de la *caché* adoptando este algoritmo.

Dir	B0	B1	B2	B3
0	0-1			
10	0-1	10-11		
1	0-1	10-11		
10	0-1	10-11		
2	0-1	10-11	2-3	
10	0-1	10-11	2-3	
3	0-1	10-11	2-3	
9	0-1	10-11	2-3	8-9
4	0-1	10-11	2-3	4-5
5	0-1	10-11	2-3	4-5
10	0-1	10-11	2-3	4-5
6	6-7	10-11	2-3	4-5
1	0-1	10-11	2-3	4-5
10	0-1	10-11	2-3	4-5
2	0-1	10-11	2-3	4-5
10	0-1	10-11	2-3	4-5

Dir	B0	B1	B2	B3
3	0-1	10-11	2-3	4-5
9	8-9	10-11	2-3	4-5
4	8-9	10-11	2-3	4-5
5	8-9	10-11	2-3	4-5
10	8-9	10-11	2-3	4-5
6	6-7	10-11	2-3	4-5
1	0-1	10-11	2-3	4-5
10	0-1	10-11	2-3	4-5
2	0-1	10-11	2-3	4-5
10	0-1	10-11	2-3	4-5
3	0-1	10-11	2-3	4-5
9	8-9	10-11	2-3	4-5
4	8-9	10-11	2-3	4-5
7	6-7	10-11	2-3	4-5
10	6-7	10-11	2-3	4-5
8	8-9	10-11	2-3	4-5

Cuadro 3: Estado de la *caché* por cada acceso con el nuevo algoritmo.

A primera vista ya presenta una mejora, dado que solo el bloque `B0` es el que se termina reemplazando. Dado que se tiene un total de 13 *misses*, el *hit rate* es igual a $\frac{32-13}{32} = \frac{19}{32}$, cumpliendo el objetivo.

3. a. **(Examen - II/2014)** Un computador utiliza palabras de 8 bits, tiene un espacio direccionable de 64GB y una memoria principal de 2GB dividida en marcos de 2KB. ¿Cuántos bits se necesitan para describir el número de página virtual y el número de marco físico? Si todos los *flags* de la tabla de páginas toman 12 bits por entrada, ¿cuantos bits de espacio utiliza una tabla de páginas?

   En primer lugar, al ser los marcos de 2KB sabemos que el número de palabras por marco (y por ende, por página) es igual a $2^1 \times 2^{10} = 2^{11}$, lo que implica un total de 11 bits de *offset*. Por otra parte, al ser el espacio direccionable de 64GB y las palabras de 8 bits (o un byte), se tiene un total de $2^6 \times 2^{30} = 2^{36}$ direcciones virtuales, lo que implica un total de 36 bits para una dirección virtual. De aquí se desprende que el número de bits necesario para describir una página es igual a $36 - 11 = 25$, generando un total de 2^{25} páginas posibles. Ahora, para obtener el número de bits para el marco, es necesario hacer el mismo análisis pero con la memoria principal. Al ser de 2GB, se tiene un total de $2^1 \times 2^{30} = 2^{31}$ direcciones físicas, lo que implica un total de 31 bits para una dirección física. Por lo tanto, el número de bits necesario para describir un marco es igual a $31 - 11 = 20$, lo que implica un total de 2^{20} marcos posibles (esto también pudo haber sido determinado dividiendo el tamaño de la memoria principal por el tamaño de un marco).

   Para calcular el tamaño de la tabla de páginas, es necesario recordar que esta debe poseer una entrada por cada página y cada una de estas posee los bits del marco físico asociado en conjunto con las *flags* correspondientes. Por lo tanto:

$$\text{Tamaño tabla} = 2^{25} \times (20 + 12)\text{b} = 2^{25} \times 32\text{b} = 2^{25} \times 2^5\text{b} = 2^{27} \times 2^3\text{b} = 128\text{MB}$$

   b. **(Examen - II/2014)** Complete la siguiente tabla asumiendo que por cada entrada de la tabla de páginas, se utilizan 4 bits para *flags*:

Bits Dir. Virt.	Bits Dir. Fís.	Tam. Pág.	Bits Pág.	Bits Marco	Bits por entrada
32	32	16KB	18	18	22
32	26	8KB	19	13	17
36	32	32KB	21	17	21
40	36	32KB	25	21	25
64	40	64KB	48	24	28

   En azul se encuentran las respuestas esperadas. Para completarla, se hacen los cálculos de la siguiente forma:

   - **Fila 1:** Con el tamaño de página obtenemos el *offset*. Lo usamos para restar a los bits de direcciones virtuales y físicas para obtener los bits de página y marco, respectivamente. Los bits por entrada se obtienen sumándole a los bits de marco las *flags*.
   - **Fila 2:** Con los bits de marco y de dirección física obtenemos los bits de *offset*, de los que se desprende el tamaño de página. Los bits de página se obtienen restando a los bits de dirección virtual el *offset*, mientras que los bits por entrada se obtienen sumando directamente las *flags* a los bits de marco.
   - **Fila 3:** Con los bits por entrada se obtienen los bits de marco simplemente restando las *flags*. Si restamos este nuevo valor a los bits de dirección física podemos obtener el *offset*, del que se desprende el tamaño de página. Finalmente, sumamos el *offset* a los bits de página para la obtención de los bits de dirección virtual.

- **Fila 4:** Con el tamaño de página se obtienen los bits de *offset*, lo que sumado a los bits de página nos da los bits de dirección virtual. Luego, de los bits por entrada se desprenden directamente los bits de marco restando las *flags*, lo que en conjunto con el *offset* entrega los bits de dirección física.
- **Fila 5:** Restando los bits de dirección virtual con los bits de página se obtienen los bits de *offset*, de lo que se desprende el tamaño de página. Luego, restando las *flags* de los bits por entrada se obtienen los bits de marco, lo que en conjunto con el *offset* resulta en los bits de dirección virtual.

c. **(I3 - II/2011)** Asuma que una `CPU` tiene un espacio de direccionamiento virtual de 13 bits cuyas páginas son de 1KB. Esta máquina, sin embargo, cuenta tan solo con 4KB de memoria `RAM` disponibles para marcos. Asuma que los marcos están inicialmente vacíos. En un momento comienza a ejecutarse un proceso (P1) el cual, durante su ejecución, utiliza las direcciones de memoria desde la 0 hasta la 1500. Luego de esto el sistema operativo hace un cambio de contexto con lo que empieza a ejecutarse un segundo proceso (P2) el cual, durante su ejecución, utiliza las direcciones de memoria desde la 0 hasta la 500, y desde la 4500 a la 5000. Los datos que el proceso P2 almacena en estas últimas direcciones (de la 4500 hasta la 5000) son compartidos por el proceso P3 (tanto para lectura como para escritura), el que accede a ellos a través de las direcciones virtuales 2452 a la 2952. Se genera otro cambio de contexto y empieza a ejecutarse este tercer proceso (P3), el cual hace uso de las direcciones de memoria desde la 0 hasta la 1000, utilizando además datos desde la dirección 2500 a la 2600. Suponga que la política de reemplazo de páginas en los marcos es `FIFO`.

   i. Determine, para cada marco, de qué proceso o procesos es la información y/o datos que contiene. También indique qué paginas, de haber, se encuentran en disco.
   Al tener páginas de 1KB, sabemos que se tienen marcos del mismo tamaño, lo que implica que con una memoria `RAM` de 4KB se tienen solo 4 marcos. Ahora, como las direcciones virtuales son de 13 bits, se tienen 3 bits para el número de página (pues $1\text{KB} = 2^{10}\text{B}$, 10 bits de *offset*), lo que implica un total de 8 páginas por proceso. A partir de estos hechos, veamos el intervalo de accesos por proceso:
   - **P1:** Accede desde 0 hasta 1500. Con 13 bits, esto es desde 0000000000000 hasta 0010111011100. Es decir, ocupa los números de página 000 = 0 y 001 = 1. Asumiendo en un principio los marcos vacíos y una asignación *fully associative*, le asignamos los marcos físicos 00 = 0 y 01 = 1.
   - **P2:** Accede desde 0 hasta 500 y luego desde 4500 hasta 5000. En el primer intevalo, con 13 bits, accede desde 0000000000000 hasta 0000111110100, por lo que ocupa solo la página 000 = 0, la que es asignada al marco físico 10 = 2. En cambio, en el segundo intervalo con 13 bits accede desde 1000110010100 hasta 1001110001000, por lo que ocupa solo la página 100 = 4, la que es asignada al marco físico 11 = 3.
   - **P3:** Accede desde 0 hasta 1000 y luego desde 2500 hasta 2600. En el primer intervalo, con 13 bits, accede desde 0000000000000 hasta 0001111101000, ocupando solo la página 000 = 0. Como todos los marcos físicos han sido ocupados, se usa la política de reemplazo `FIFO` y se reemplaza el primer marco asignado, enviando su contenido al disco (o bien al *swap file*). En el segundo intervalo, con 13 bits, accede desde 0100111000100 hasta 0101000101000, ocupando solo la página 010 = 3.

Lo importante es notar que los accesos son parte de la memoria compartida con el proceso P2 que ya se encuentra ubicada en el último marco, por lo que se asigna directamente al último marco y solo se generan *hits*, sin ningún reemplazo de por medio.

El análisis anterior se resume en las siguientes tabla:

Antes de ejecución de P3

P1
P1
P2
P2

Después de ejecución de P3

P3	P1 en disco
P1	
P2	
P2/P3	

Cuadro 4: Estado de la memoria física antes y después de la ejecución del proceso P3.

II. Escriba las tablas de página asociadas a estos tres procesos.
A partir del análisis planteado en la pregunta anterior, las tablas de página se pueden escribir de forma directa. Lo importante es que el bit de disco solo esté activado para la primer página del proceso P1 (pues fue lo reemplazado), mientras que el resto de las páginas utilizadas tengan su bit de validez igual a 1. Para el resto de las páginas, ambos bits son 0 y el contenido del marco se puede considerar "basura". A continuación, el resultado esperado:

Proceso 1			
Pág.	Marco	Val.	Disco
0	0	0	1
1	1	1	0
2	x	0	0
3	x	0	0
4	x	0	0
5	x	0	0
6	x	0	0
7	x	0	0

Proceso 2			
Pág.	Marco	Val.	Disco
0	2	1	0
1	x	0	0
2	x	0	0
3	x	0	0
4	3	1	0
5	x	0	0
6	x	0	0
7	x	0	0

Proceso 3			
Pág.	Marco	Val.	Disco
0	0	1	0
1	x	0	0
2	3	1	0
3	x	0	0
4	x	0	0
5	x	0	0
6	x	0	0
7	x	0	0

Cuadro 5: Tablas de página asociadas a cada proceso.

Luego de un cambio de contexto el proceso P1 lee la dirección de memoria 500. Posterior a esto, el mismo proceso requiere escribir en la dirección de memoria 600.

I. Determine en qué dirección real se escribe en la memoria principal al escribir este proceso en la dirección 600.
Al leer la dirección virtual 500 = 0000111110100 que corresponde a la página 1, se genera *swapping* al estar esta en el disco (o *swap file*) y por `FIFO` se reemplaza el segundo marco. Luego, la dirección virtual 600 = 0001001011000 corresponde a la página 0 del proceso P1. Como ahora está asociada al marco 01 de la memoria física, la dirección real correspondiente a la dirección virtual 600 será 011001011000 = 1624 (**1024 + 600 = 1624**).

II. Determine, para cada marco, de qué proceso o procesos es la información y/o datos que contiene. También indique qué paginas, de haber, se encuentran en disco.
Simplemente se actualiza la tabla de marcos en base al análisis antes planteado.

P3
P1
P2
P2/P3

P1 en disco

Cuadro 6: Asignación de marcos posterior al *swapping*.

III. Escriba las tablas de página asociadas a estos tres procesos.
Simplemente actualizamos los valores basándonos en el análisis anterior.

Proceso 1			
Pág.	Marco	Val.	Disco
0	1	1	0
1	1	0	1
2	x	0	0
3	x	0	0
4	x	0	0
5	x	0	0
6	x	0	0
7	x	0	0

Proceso 2			
Pág.	Marco	Val.	Disco
0	2	1	0
1	x	0	0
2	x	0	0
3	x	0	0
4	3	1	0
5	x	0	0
6	x	0	0
7	x	0	0

Proceso 3			
Pág.	Marco	Val.	Disco
0	0	1	0
1	x	0	0
2	3	1	0
3	x	0	0
4	x	0	0
5	x	0	0
6	x	0	0
7	x	0	0

Cuadro 7: Tablas de páginas por proceso después del *swapping*.

IV. Indique en qué direcciones físicas, de estar, se encuentran las direcciones virtuales:
- 2500 del proceso P3 → 110111000100 = 3524.
- 2000 del proceso P1 → Como 2000 es la dirección 0011111010000, su página 001 = 1 fue recientemente transferida al disco (o *swap file*), lo que implica que no existe una dirección física.
- 4548 del proceso P2 → 110111000100 = 3524.

d. **(I3 - I/2017)** ¿Tiene sentido utilizar una `TLB` *split*?
Sí, lo tiene. Sobre todo en la arquitectura Von Neumann uno seguirá trabajando con direcciones de memoria que pueden corresponder tanto a datos como instrucciones, por lo que las direcciones virtuales no se desligan de esa particularidad. Separar la `TLB` según tipo de acceso seguirá siendo enriquecedor en términos de *hits*. Más aún, es posible encontrar arquitecturas donde se haga uso de dos `TLB` en el primer nivel de jerarquía: *instruction*-`TLB` (`ITLB`) y *data*-`TLB` (`DTLB`).

4. a. **(Examen - I/2015)** ¿Cómo podría solucionarse un *hazard* estructural que involucra la colisión de dos etapas que requieren la lectura de datos desde un registro?

   Existen varias formas, a continuación se listan algunas:

   - Se puede hacer uso de dos memorias de datos, pero suma un gran valor al costo y habría que implementar protocolos que aseguren la consistencia entre ambas.
   - Se puede modificar la memoria de datos de forma que se permitan dos accesos concurrentes. No obstante, habría que tener cuidado si es que se hace una escritura simultánea en ambas etapas sobre una misma dirección.
   - Se puede hacer uso de una *caché*, de forma que cada etapa acceda un tipo de memoria distinto (aunque los *caché-miss* aumentarían el tiempo de ejecución y habría que aumentar el tiempo de ejecución máximo por etapa).
   - Se pueden insertar burbujas mediante *stalling* en la etapa más temprana hasta que ninguna de las etapas intermedias deba hacer uso de ella, para así asegurar que no se genere el *hazard* (aunque se pierden más ciclos).

   b. **(I3 - I/2017)** Indique por qué agregaría una complejidad adicional el tener soporte para llamado a subrutinas en el computador básico con *pipeline*.

   El soporte de subrutinas implica el uso del *stack* para almacenar la dirección de retorno. Esto supone un problema si se considera, por ejemplo, que para escribir la dirección de retorno sería necesario que el registro `SP` no se actualizara hasta almacenar la dirección en el tope del *stack*. Si bien esto no pareciera tener dificultad, empezaría a cambiar la idea de tener etapas con distintas funcionalidades (en la etapa `MEM` se cambiaría su valor, lo que uno esperaría en la etapa `WB`). Por otra parte, sería imperante la adición de un *Mux* `PC` para seleccionar entre el literal (*Param*) y la salida de la `ALU`, lo que supone otro aumento en la complejidad de la arquitectura y en las señales de control involucradas.

   c. **(I3 - I/2017)** Considere un computador `RISC`-Harvard con un *pipeline* de 12 etapas, donde la unidad de salto se activa en la etapa 6 (40 % de las veces) o en la etapa 11 (60 % de las veces), dependiendo del origen de los parámetros necesarios para resolver el salto. Dado que la unidad de predicción de saltos de este computador acierta en el 75 % de las oportunidades, en promedio, ¿cuántos ciclos por salto pierde este computador?

   Si la predicción de la unidad de salto se activa en la etapa 6 de forma errónea, se pierde un total de 5 ciclos, mientras que si se activa de forma errónea en la etapa 11, se pierden 10. Considerando los porcentajes de activación, en promedio por salto erróneo se pierden:

$$(0{,}4 \times 5) + (0{,}6 \times 10) \text{ ciclos} = 2 + 6 \text{ ciclos} = 8 \text{ ciclos}$$

   Ahora, como solo un 25 % de los saltos son erróneos, en promedio por salto se pierden:

$$0{,}25 \times 8 \text{ ciclos} = 2 \text{ ciclos}$$

d. **(I3 - I/2017)** Indique cuándo y cómo podría implementarse la aceleración de un *pipeline*, si se sabe de antemano que una instrucción no utiliza una etapa.

Aquí hay que tener cuidado, dado que no se podrá acelerar el *pipeline* si solo una instrucción entre varias es la que no utiliza una etapa. En un principio se podría pensar que basta con propagar sus valores a la etapa siguiente, saltándose la que le correspondía. No obstante, esto podría afectar a la instrucción que estuvo ejecutando previamente, puesto que podría utilizar la etapa a la que se busca propagar los valores de la instrucción actual. Por eso es importate el "cuándo", ya que esto será solo cuando la instrucción que no utiliza la etapa que viene no tiene instrucciones "por delante" (es decir, que no haya una instrucción previa que ejecute en la etapa siguiente). Para implementar la aceleración, en ese caso, sería necesaria la inclusión de una unidad encargada de identificar que se cumplan las condiciones antes establecidas y, en dicho caso, de propagar las señales del registro intermedio al que se encuentra dos etapas adelante.

e. **(I3 - I/2016)** Diseñe un computador con *pipeline* de al menos 5 etapas, donde debido a restricciones del *hardware*, la etapa `MEM` debe ejecutarse antes que la etapa `EX`. El computador debe soportar las mismas funcionalidades que el computador básico con *pipeline*.

La clave para solucionar este ejercicio consiste en evitar que los saltos mal predecidos afecten el contenido de la memoria, que complica la aplicación de una operación de *flushing*. En el computador básico con *pipeline*, esto se evita ubicando la etapa `MEM` después de `EX`, dado que la condición de salto se genera en la etapa `EX` y se evalúa en la etapa `MEM`. Teniendo esto en consideración, una posible solución consiste en un computador con un *pipeline* de 6 etapas: `IF`, `ID`, `JMP`, `MEM`, `EX`, `WB`, donde la etapa `JMP` contiene un restador que genera la condición de salto. Luego, tal como en el computador básico con *pipeline*, en la etapa `MEM` se evalúa la condición, evitando la escritura en memoria de las instrucciones siguientes.