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

IIC2343 – Arquitectura de Computadores

Ayudantía 1 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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)** Describa el valor decimal del número 0x94A6, si este se interpreta como binario con signo.

   Una forma rápida de expresar un número hexadecimal como uno binario, es transformando cada dígito de este a base binaria con 4 dígitos (recordando que $2^4 = 16$):

   - $9 = 1001_2$
   - $4 = 0100_2$
   - $A = 1010_2$
   - $6 = 0110_2$

   Luego, nuestro número en base binaria corresponde a 1001010010100110_2. Como este se interpreta como binario con signo, y el bit más significativo corresponde a un 1, utilizamos el complemento a 2 para obtener la representación correcta:

$$1001010010100110_2 = -C_2(1001010010100110_2) = -0110101101011010_2 = -27482$$

b. **(I1 - II/2011)** Dados $A = 45$ y $B = 57$, ¿cuál es el resultado, en binario, de la operación $A - B$?

Notemos que si queremos representar estos números en binario, y estamos realizando una operación que implica una resta, entonces necesariamente debemos considerar el bit de signo. Tenemos entonces que $A = 0101101_2$ y $B = 0111001_2$. Luego, restarle a un número positivo uno negativo es equivalente a sumarle su complemento a 2. Entonces:

$$-B = C_2(0111001_2) = 1000111_2$$

$$A - B = 0101101_2 + 1000111_2 = 1110100_2$$

Ahora, este número es negativo, por lo que si queremos su valor decimal correspondiente, realizamos nuevamente el complemento a 2:

$$A - B = -C_2(1110100_2) = -0001100_2 = -12$$

Finalmente, vemos que el número obtenido en binario es consistente con el resultado de la operación en base decimal.

c. Suponga que se tiene un total de 6 bits, usados para representar números positivos y negativos. Dados $A = 27$ y $B = 8$, ¿cuál es el resultado, en binario, de la operación $A + B$? ¿Por qué da este resultado?

Si se tiene un total de 6 bits, podemos representar sin problemas los A y B dados como números positivos. Tenemos entonces que $A = 011011_2$ y $B = 000100_2$. Luego, al realizar la operación:

$$A + B = 011011_2 + 001000_2 = 100011_2$$

Podemos ver que el resultado, utilizando representación binaria con signo, es negativo. El número, en base decimal, sería entonces:

$$100011_2 = -C_2(100011_2) = -011101_2 = -29$$

Esto sucede debido a que sobrepasamos nuestro poder de representación. Con 6 bits, el máximo número que podemos representar es $011111_2 = 31$, Si la suma nos da un resultado mayor a ese, al no ser capaces de representar dicho número, obtenemos un número incorrecto (pues la suma de dos números positivos no pueden dar como resultado uno negativo). Este resultado se conoce como **overflow**.

d. **(I1 - II/2014)** Si hay algún problema eléctrico, como un alza de voltaje, es muy fácil corromper datos almacenados en binario. Por ejemplo, el número 10 (1010b) puede transformarse en 14 (1110b), con tan solo modificar un bit. Describa una codificación binaria para los números 0 y 1, de manera que esta permita detectar y corregir errores de a lo sumo 1 bit, i.e., un bit de la codificación se ve alterado.

Una codificación binaria sencilla para realizar esto, es codificar cada bit como 3 bits de sí mismo, es decir:

$$f(x) = \begin{cases} 111, & x = 1 \\ 000, & x = 0 \end{cases}$$

De esta forma, si tuvieramos el número 0101, este resultaría $f(0101) = 000111000111$. Si uno de los bits se corrompe, basta ir revisando de a 3 dígitos el número, y detenernos cuando no veamos que los 3 bits coinciden para encontrar el espacio que fue corrompido. El bit que predomine corresponderá al número original.

Notar que esto no habría funcionado para la siguiente codificación:

$$g(x) = \begin{cases} 11, & x = 1 \\ 00, & x = 0 \end{cases}$$

Esto, ya que si bien podemos identificar el lugar de corrupción revisando de a 2 dígitos, no podemos saber cuál correspondía al números original (por ejemplo, 10 pudo haber sido 11 o 00, no lo sabemos con dicha codificación).

e. **(I1 - I/2017)** ¿Cuál es la cantidad máxima de pixeles que puede tener una imagen en blanco y negro de 1KB no comprimida? Asuma que cada pixel solo almacena su valor de color y que el archivo de la imagen solo almacena pixeles, por lo que no es necesario considerar el encabezado.

Como nos interesa saber si un color es blanco o negro, tenemos dos posibilidades de color, por lo que basta con un bit para representarlo. Luego:

$$1KB = 1024B = 1024 * 2^3 b = 8192b$$

Es decir, necesitamos 8192 bits para la imagen, lo que equivale a 8192 pixeles.

2. a. Implemente, utilizando solo las compuertas lógicas `AND`, `OR` y `NOT`, el conectivo binario condicional ($\rightarrow$), que está definido por la siguiente tabla de verdad:

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Se puede implementar fácilmente si recordamos que:

$$A \rightarrow B \Leftrightarrow \neg A \vee B$$

Esto se puede corroborar con la equivalencia existente en sus tablas de verdad.
Ahora, la fórmula lógica se puede implementar de la siguiente forma con conectivos lógicos:

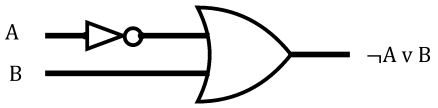


Figura 1: Circuito resultante.

b. **(Apuntes - Operaciones aritméticas y lógicas)** Implemente un circuito 2 bit Multiplier, que realice la multiplicación entre dos valores de 2 bits
Para ver esto de forma sencilla, primero vemos cómo implementar una multiplicación entre dos números de un bit:

A	B	$A * B$
0	0	0
0	1	0
1	0	0
1	1	1

Es fácil ver que la multiplicación la podemos realizar a partir de la compuerta lógica `AND`. Ahora, como nos piden una multiplicación entre dos números de dos bits, tenemos que seguir el método de multiplicación tradicional. Para cumplir este objetivo es importante recordar la composición de los *half-adders*, en la que la suma entre dos números se representa por la compuerta `XOR`, mientras que el *carry* resultante se hace con la compuerta `AND`[1].

[1] **¿Por qué?** Porque si ambos bits de entrada son iguales a 1, entonces sabemos que la suma resultará en 0, "sobrando" una unidad (como lo vemos en la suma tradicional).

Sean A y B dos números de dos bits de la forma A_1A_0 y B_1B_0. Si queremos obtener el número $C = A * B$, seguimos el siguiente procedimiento:

- Multiplicamos el bit menos significativo de B por los dos bits de A. De esta forma, tendremos A_0 `AND` $B_0 = C_0$, y A_1 `AND` $B_0 = C_1^{'}$.
- Multiplicamos ahora el bit más significativo de B por los dos bits de A. De esta forma, tendremos A_0 `AND` $B_1 = C_1^{''}$ y A_1 `AND` $B_1 = C_2^{'}$
- El bit menos significativo de nuestro resultado será C_0. Luego, el bit siguiente se obtiene de la siguiente forma: $C_1 = C_1^{'}$ `XOR` $C_1^{''}$ (tal como lo hacemos con la suma). Como esto nos puede generar un carry, tomamos $C_2^{''} = C_1^{'}$ `AND` $C_1^{''}$
- Ahora, el siguiente bit lo conseguimos como la suma entre el producto de los bits más significativos de cada número, sumado al carry anterior: $C_2 = C_2^{'}$ `XOR` $C_2^{''}$.
- Finalmente, como nuestro número puede tener máximo 4 bits (11 * 11 = 1001), tomamos el bit más significativo del resultado como el carry de la última suma: $C_3 = C_2^{'}$ `AND` $C_2^{''}$.

Finalmente, nuestro circuito queda de la siguiente forma:

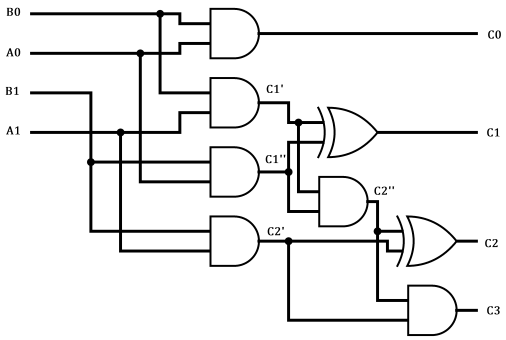


Figura 2: Resultado del circuito descrito.

Notar que este diagrama se puede simplificar haciendo uso de *half-adders* y *full-adders*.

c. **(I1 - I/2017)** Construya un circuito que permita detectar la ocurrencia de *oveflow* al sumar o restar dos números enteros de 8 bits en una ALU.

Para construir este circuito es necesario identificar los cuatro casos en los que se produce *overflow*, ya sea por la adición o sustracción de dos números A, B:

- **Caso 1:** $A \geq 0, B \geq 0, A + B = M < 0$
- **Caso 2:** $A < 0, B < 0, A + B = M \geq 0$
- **Caso 3:** $A \geq 0, B < 0, A - B = M < 0$
- **Caso 4:** $A < 0, B \geq 0, A - B = M \geq 0$

Para cada caso formaremos un circuito distinto, de forma que se puedan conectar todos al final. Para identificar el signo, utilizamos el bit más significativo de cada número (A, B, S) y lo conectamos en un puerto `AND` que reciba, además, el bit esperado. Luego, para identificar la operación, hacemos uso de otro puerto `AND` que recibe dos entradas: El número esperado del comando ejecutado en la ALU (según sea el caso) y el número de operación recibido (que denotaremos por S). Entonces, los circuitos generados son los siguientes:

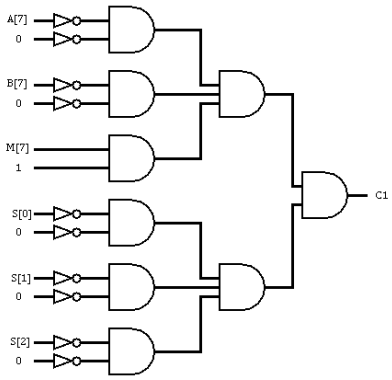


Figura 3: Caso 1.

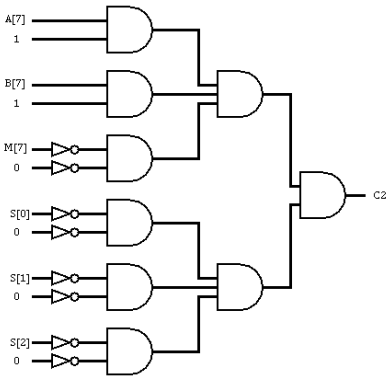


Figura 4: Caso 2.

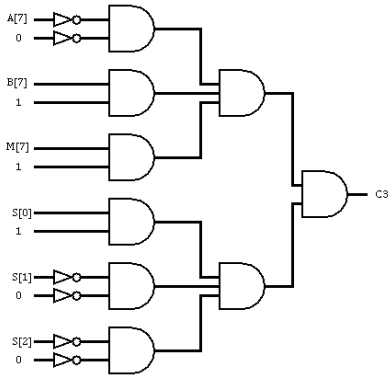


Figura 5: Caso 3.

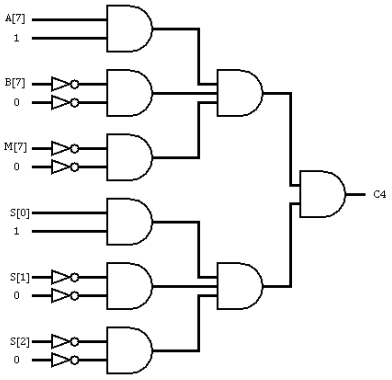


Figura 6: Caso 4.

Finalmente, conectamos los resultados a un puerto `OR` para obtener el bit que indica si se generó o no *overflow* (el que llamamos O):

Figura 7: Conexión final entre los cuatro circuitos antes descritos.

d. **(I1 - I/2012)** Diseñe un De-Multiplexor con bus de datos de 1 bit y bus de control de 2 bits.

A diferencia de los multiplexores, los de-multiplexores se encargan de transmitir la señal que reciben solo a una de sus salidas, la que se especifica mediante una señal de selección. A partir de esta descripción, el diagrama de un Demux es el siguiente:

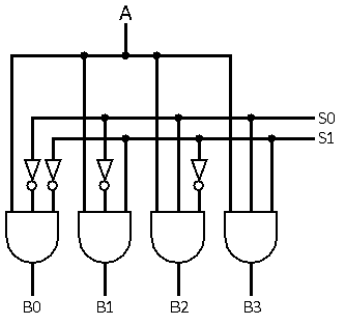


Figura 8: De-Multiplexor con entrada de 1 bit y bus de control de 2 bits.

e. **(I1 - I/2017)** ¿Qué número entero es generado al realizar cuatro operaciones **shift right** seguidas de cinco operaciones **rotate left** a un registro de 8 bits que inicialmente almacena el número entero 79?

Para interpretar bien el resultado, consideramos la representación binaria para números enteros con signo (pues, por enunciado, sabemos que debemos considerar números negativos).

Tenemos que $79 = 01001111$. Ahora, iremos listando los números resultantes:

- `shift right`: $00100111 \rightarrow 39$
- `shift right`: $00010011 \rightarrow 19$
- `shift right`: $00001001 \rightarrow 9$
- `shift right`: $00000100 \rightarrow 4$
- `rotate left`: $00001000 \rightarrow 8$
- `rotate left`: $00010000 \rightarrow 16$
- `rotate left`: $00100000 \rightarrow 32$
- `rotate left`: $01000000 \rightarrow 64$
- `rotate left`: $10000000 \rightarrow$ -128

Notemos que en esta secuencia el uso de `rotate left` no difiere de `shift left`, ya que no hubo ningún bit entre medio que pudiera ser conservado. Si se hiciera un `rotate left` más:

- `rotate left`: $00000001 \rightarrow 1$

Aquí sí se aprecia la diferencia, donde el bit más significativo ahora se vuelve el menor en vez de perderse, como es en el caso de `shift left`.

3. a. **(I1 - II/2016)** Modifique un *latch* tipo RS agregando una señal de control C, tal que los cambios en el estado del *latch* solo se realicen cuando $C = 1$.
Una posible implementación consiste en el siguiente circuito:

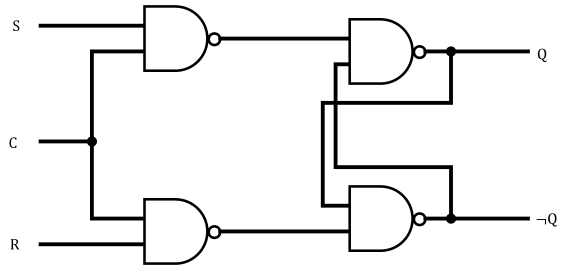


Figura 9: Primera opción.

La diferencia con el *latch* original es que se invierten las señales S y R para mantener sus funciones en el circuito (pues se intercambian con la compuerta `NAND` y queremos que la señal R y S almacenen un 0 y un 1, respectivamente). Por otra parte, la combinación de señales indefinida correspondía a R = 0 y S = 0, pero ahora corresponde a R = 1 y S = 1 (lo que no es relevante siempre que se indique). La tabla, entonces, queda así:

R	S	C	Q	$\overline{Q}$
X	X	0	Q	$\overline{Q}$
0	0	1	Q	$\overline{Q}$
0	1	1	1	0
1	0	1	0	1
1	1	1	?	?

Cuadro 1: Tabla de verdad asociada a la primera opción.

Sin embargo, el siguiente circuito también sirve:

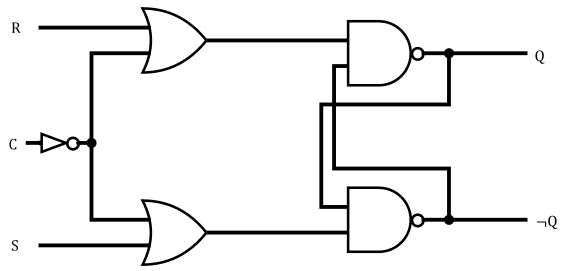


Figura 10: Segunda opción.

En este también se cumple con el objetivo de controlar los cambios de estado con la señal C, pero además no es necesario invertir el orden de las señales R y S (pues cumplen sus funciones como corresponde). Por otra parte, se mantiene la combinación que indefine los estados del circuito. Finalmente, la tabla queda de la siguiente forma:

R	S	C	Q	$\overline{Q}$
X	X	0	Q	$\overline{Q}$
0	0	1	?	?
0	1	1	1	0
1	0	1	0	1
1	1	1	Q	$\overline{Q}$

Cuadro 2: Tabla de verdad asociada a la segunda opción.

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?

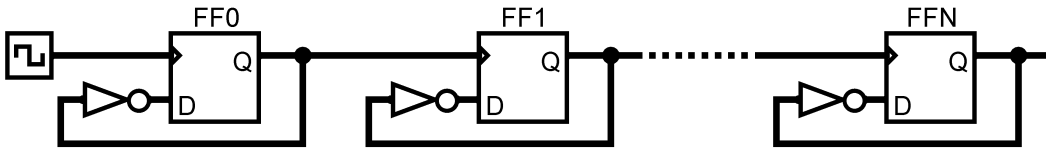


Figura 11: Secuencia de *flip-flops*, donde el *clock* de uno es la señal de estado del que lo antecede, salvo para el primero.

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 $\frac{\text{F}}{2}$.

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:

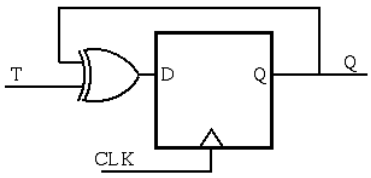
$$f_{\text{FF}}(i) = \begin{cases} \frac{f_{\text{FF}}(i-1)}{2}, & i > 0 \\ F, & i = 0 \end{cases}$$

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:



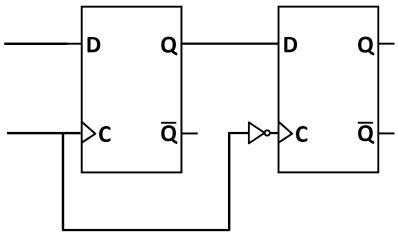
De partida, vemos que la salida del *flip-flop* D correspondiente a la señal Q se conecta a una compuerta `XOR`, la que recibe además la señal T. Esto genera la siguiente tabla de casos:

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 - II/2012)** Implemente mediante compuertas lógicas, elementos de control y latches, un *flip-flop* tipo D que funcione con flanco de bajada.
Lo que se busca, básicamente, es lo siguiente:



En el *flip-flop* tipo D tradicional, se tiene una negación antes de la entrada de la señal de control para ambos *latches*. Esto permite que solo uno de estos circuitos almacene un valor a la vez (según la señal C), habilitando cambios en el contenido guardado en la estructura completa solo en los flancos de subida. Ahora, notemos que al eliminar la primera negación, el primer *latch* (llamémoslo de entrada) solo permitirá el almacenamiento de un dato cuando $C = 1$. En cambio, el segundo *latch* (que llamaremos de salida) solo guarda el estado para $C = 0$. De esta forma, cuando existe la transición C: $1 \rightarrow 0$ (i.e., el flanco de bajada) el *latch* de entrada permite almacenar el dato y lo transfiere al *latch* de salida, siendo este último capaz de guardarlo correctamente. Esto se logra antes de que el *latch* de entrada vuelva a cambiar su valor (al habilitarse nuevamente por C), lo que permite que el *flip-flop* D pueda guardar la señal en cada flanco de bajada.

e. **(I1 - I/2012)** Implemente mediante compuertas lógicas y *flip-flops* tipo D, el registro de la figura, con señales de control (C), carga (*Load*) y *reset* (*Reset*), que funciona con flanco de subida.

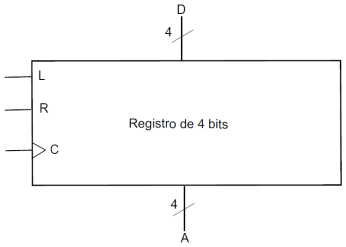
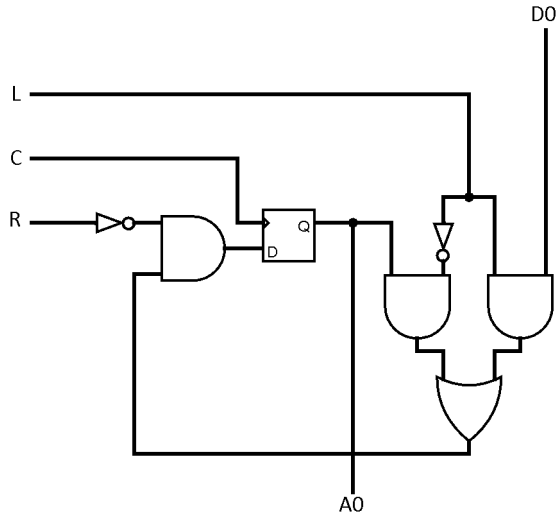


Figura 12: Registro de 4 bits que funciona con una señal de control, carga y *reset*.

Antes de ver la figura, es necesario entender bien la funcionalidad de cada señal:

- ***Load*:** Habilita el almacenamiento del valor que está recibiendo el registro.
- **C:** Permite que existan cambios en el registro solo en los flancos de subida. Corresponde al clock.
- ***Reset*:** *Resetea* el valor almacenado en 0.

El siguiente diagrama representa el circuito que permite lograr la funcionalidad antes explicada:



Es importante notar que esto solo ilustra el almacenamiento del bit menos significativo en el registro. No obstante, el almacenamiento del resto de los bits es el mismo.

Analicemos ahora las partes relevantes del circuito:

- **L:** Si la señal L está activa, se debe permitir el paso del nuevo bit para que se dirija a la entrada del *flip-flop* D. A su vez, debe evitar que este pueda almacenar el antiguo valor contenido por este. Por esta razón, se utiliza la señal L en dos compuertas `AND`: En la de la derecha, se preocupa de dejar pasar al dato que se busca ingresar, mientras que en la de la izquierda se bloquea el paso del bit original (por eso se utiliza una negación en L). Es importante notar que si L está inactiva, los papeles se invierten: Se bloquea el paso del bit de entrada y se permite el flujo del valor original del *flip-flop*. Finalmente, se escoge el resultado a partir de la compuerta `OR` (el que haya sido negado quedará descartado al ser igual a 0).
- **C:** Al corresponder a la señal del *clock*, se conecta directamente al *flip-flop* D para permitir el cambio de valores almacenados solo en los flancos de subida.
- **R:** Si esta señal está activa, se debe cambiar el valor almacenado en el *flip-flop* a 0, sin importar el resto de las señales. Esto se logra a partir de una compuerta `AND` entre la negación de la señal R y el resultado del paso de bits explicado en el primer punto. Si R está activo, su negación será igual a 0, lo que implicará que el resultado de la compuerta será 0 independiente de lo que ingrese en la otra entrada, logrando resetear el *flip-flop*. En cambio, si R está inactivo, su negación será igual a 1, lo que hace que el valor a almacenar finalmente sea equivalente al resultado obtenido por la señal L.

f. **(I1 - II/2012)** ¿Cuántas direcciones tiene una memoria RAM de 4.5 KB que utiliza palabras de 3 bytes? (1KB = 1024 bytes).

Recordemos que el espacio de memoria utilizado por una RAM está definido como el producto entre el número de direcciones y el tamaño de la palabra almacenada en cada dirección. Luego, nos basta con despejar la siguiente ecuación (hacemos el cálculo usando los bytes como unidad):

$$\text{Número de direcciones} * 3[bytes] = 4{,}5 * 1024[bytes] \rightarrow \text{Número de direcciones} = 1536$$

g. Suponga que se tiene una matriz almacenada en la dirección de memoria 0x0A. Esta posee un total de 4 filas y 5 columnas. Si se sabe que en una dirección de memoria se puede almacenar 1 byte, y la matriz almacena en cada celda un dato de 2 bytes, ¿cuál es la dirección del dato que se encuentra en la tercera columna de la segunda fila de la matriz? Asuma que se utiliza la convención de filas.

Para este ejercicio, basta con recordar la fórmula para la obtención de datos dentro de una matriz (según la convención de filas):

$$dir(matriz[i][j]) = dir(matriz) + i * sizeof(matriz[i][j]) * M + j * sizeof(matriz[i][j])$$

Donde:

- $dir(matriz[i][j])$ es la dirección que buscamos.
- $dir(matriz)$ es la dirección donde se comienza a almacenar la matriz.
- $sizeof(matriz[i][j])$ es el tamaño utilizado por cada celda en la matriz.
- M es la cantidad de columnas.
- $[i][j]$ es la fila y columna correspondiente a la dirección buscada.

Finalmente, reemplazando, tenemos que:

$$dir(matriz[1][2]) = 0x0A + 1 * 2 * 5 + 2 * 2 = 0x0A + 14 = 0x18$$

Notar que si bien parte utilizando la dirección 0x18, al ser este dato de dos bytes y la capacidad por dirección de un byte, tenemos que utiliza las direcciones 0x18 y 0x19 para almacenar el dato completo.

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

IIC2343 – Arquitectura de Computadores

Ayudantía 2 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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 - I/2018)** Construya un circuito que obtenga la siguiente tabla de verdad, pero solo usando compuertas `OR` y `NOT`:

`R`	`S`	`Q(t+1)`	$\overline{\texttt{Q(t+1)}}$
0	0	-	-
1	0	0	1
0	1	1	0
1	1	`Q(t)`	$\overline{\texttt{Q(t)}}$

Cuadro 1: Tabla de verdad del *latch* RS.

Un circuito que cumple lo pedido es el siguiente.

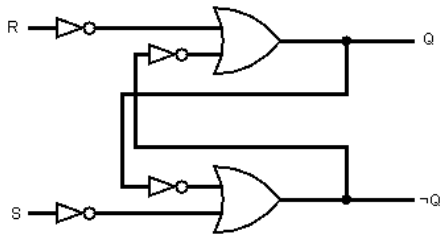


Figura 1: *Latch* RS con compuertas `OR` y `NOT`.

Este se deduce a partir de la **Ley de Morgan**. En el *latch* RS tradicional, las componentes siguen las siguientes fórmulas de lógica proposicional:

- `NOT(R AND` $\overline{\texttt{Q}}$`)`
- `NOT(S AND Q)`

Al desarrollar ambos términos con la Ley de Morgan, se obtiene lo siguiente:

- $\overline{\texttt{R}}$ `OR Q`
- $\overline{\texttt{S}}$ `OR` $\overline{\texttt{Q}}$

Que es lo que finalmente representa el circuito anterior.

b. **(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 al flanco de bajada (*i.e.*, un cambio de 1 a 0) del 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:

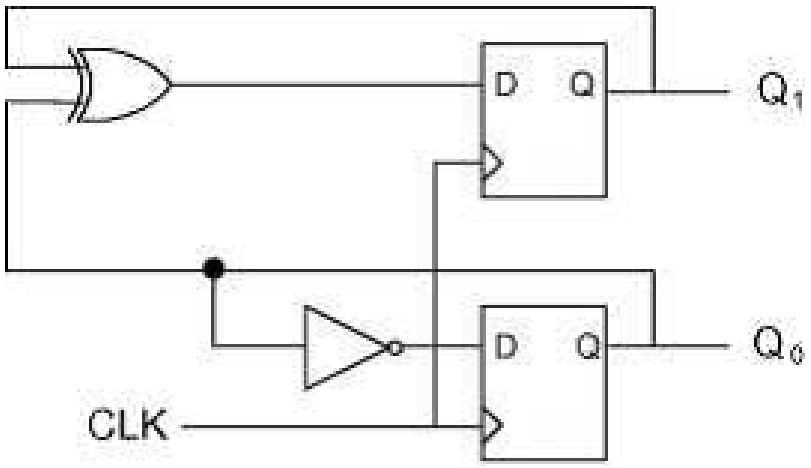


Figura 2: Contador secuencial de dos bits.

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.

   El uso de la compuerta `XOR` se hace evidente si vemos la siguiente tabla de verdad, siendo $\mathtt{A}_1, \mathtt{A}_0$ los bits más y menos significativos del contador, respectivamente:

$\mathtt{A}_1^t$	$\mathtt{A}_0^t$	$\mathtt{A}_1^{t+1}$
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 2: Tabla de verdad para determinar el siguiente valor de $\mathtt{A}_1$.

También es válido considerar para este contador 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.

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

c. **(I1 - II/2012)** Implemente mediante compuertas lógicas, elementos de control y *flip-flops*, una memoria RAM de 16 palabras de 1 byte.

Ya conociendo mejor la estructura de los registros, los Mux y Demux, se puede crear la siguiente estructura para una RAM:

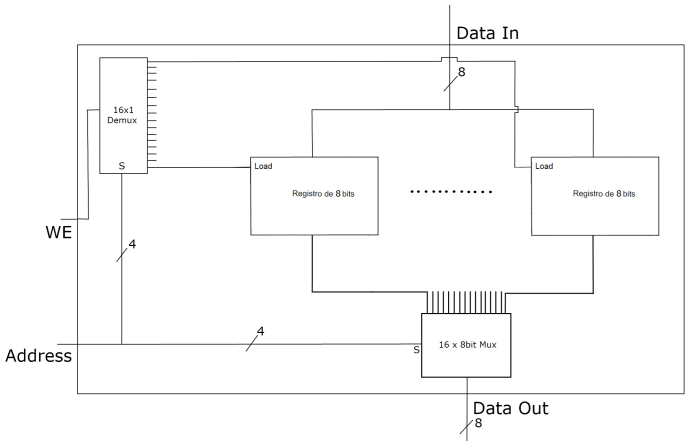


Figura 3: RAM de 16 palabras de 1 byte.

Se explica por parte cada una de sus componentes:

- **Señal *Data In*:** Corresponde al bus de un byte que busca ser almacenado en la RAM.
- **Señal *Address*:** Corresponde a la señal que determina la ubicación de un registro dentro de la RAM. Como la RAM posee 16 palabras de un byte, se necesitan 16 direcciones, *i.e.* 4 bits para poder ubicar un registro.
- **Señal WE:** Consiste en la señal que habilita la escritura en los registros de la RAM.
- **Señal Data Out:** Si `WE = 0`, corresponde a la palabra almacenada en la dirección indicada por *Address*. En cambio, si `WE = 1`, será la misma palabra que se acaba de almacenar (*i.e. Data In*).
- **Registro de 8 bits:** Corresponden a los registros de memoria de la RAM. Como se solicita en la pregunta, hay 16 en total.
- **16x1 bit Demux:** Consiste en un Demux de 16 salidas de un bit. Este se encarga de propagar la señal `WE` **solo** al registro del que se requiere su acceso (ya sea por lectura o escritura). La selección de la salida se hace a partir de la señal *Address*.
- **16x8 bit Mux:** Consiste en un Mux de 16 entradas de un byte (8 bits). A partir de la señal *Address*, retorna a través del bus *Data Out* el valor del registro seleccionado[2].

[2]Por esta razón, si `WE = 1`, el Mux selecciona el valor del registro que acaba de ser sobreescrito, explicando la coincidencia de los valores *Data In* y *Data Out*.

d. **(I1 - I/2018)** Diseñe un registro de 1 byte haciendo uso de *flip-flops* D y una señal de control `L` que habilite la sobreescritura del estado Q solo cuando `L = 1`. Luego, explique cómo puede modificar el circuito realizado para acceder individualmente (tanto para lectura como escritura) a cada uno de los bits del componente.

En primer lugar, se muestra el diagrama para un solo bit del registro, en particular, el bit menos significativo.

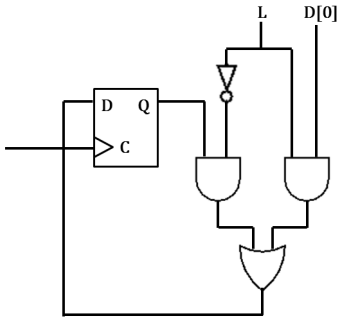


Figura 4: Diagrama del almacenamiento del bit menos significativo del registro.

En este caso, `D[0]` representa el bit menos significativo que se busca almacenar, mientras que `L` es la señal de carga. Así, se obtiene la siguiente tabla de verdad.

L	D[0]	Q
0	0	Q
0	1	Q
1	0	0
1	1	1

Cuadro 3: Tabla de verdad del almacenamiento del bit menos significativo del registro.

De esta forma, se evidencia el cumplimiento del primer objetivo: mantener el estado de un bit para `L = 0`. Para formar el byte, consiguientemente, basta por realizar este circuito para todos los bits del registro, conectados con todos los bits de `D`.

Para extender la lectura y la escritura a bits individuales del registro, existen muchas posibilidades, aquí se muestra una. En primer lugar, definimos un bus de control llamado S_{bit} que permita seleccionar el bit al que se desea acceder. Aquí, se debe cumplir que:

$$|S_{bit}| = \lceil \log_2(\text{Número de bits en el registro}) \rceil$$

Esto permitirá poder seleccionar todos los bits del registro. En este caso, se cumple que $|S_{bit}| = 3$.

Ahora, queremos que la señal `L` llegue **solo** al bit de interés. Para este propósito, podemos usar un Demux con entrada `L` y con bus de control igual a S_{bit}.

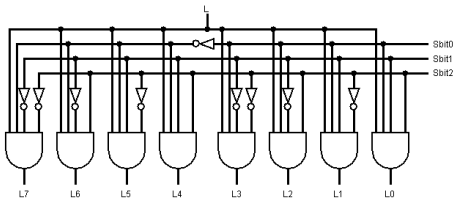


Figura 5: Demux con bus de control de 3 bits y señal de entrada `L`.

De esta forma, solo la señal $\texttt{L}_i$ va a ser igual a 1, habilitando la escritura del bit i del registro (sin importar el valor del resto de los bits que se encuentran en el bus de entrada).

Por último, necesitamos que el bus de salida solo posea el bit de interés. Para este propósito, usamos un Mux que tenga como entradas todos los bits del registro y que haga uso del mismo bus de control S_{bit}.

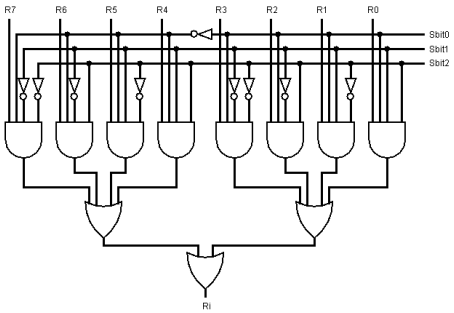


Figura 6: Mux con bus de control de 3 bits y bus de entrada `R` (valor del registro).

Finalmente, conectamos el bit resultante con un bus de 7 bits iguales a cero, de forma que el bus de salida del registro posea 8 bits (para respetar el formato del registro), siendo el bit menos significativo igual al seleccionado en un principio.

2. a. **(I1 - I/2016)** ¿Cuál es la frecuencia máxima que puede tener el *clock* del computador básico? ¿Qué pasa si un *clock* con una frecuencia mayor a la máxima es conectado al computador básico?

   La frecuencia máxima del *clock* **debe** ser el el inverso multiplicativo del tiempo que toma ejecutar la instrucción más lenta del listado. ¿Por qué? Porque en caso contrario, esta instrucción no alcanzaría a ejercutarse por completo y, por ejemplo, no se alcanzarían a almacenar los nuevos valores de los registros en el flanco de subida, causando errores de consistencia con las instrucciones posteriores.

   b. Diseñe un circuito que le permita a la unidad de control del computador básico identificar un *opcode*, retornando una señal de 1 bit que indique si corresponde a la instrucción esperada o no.

   Supongamos que `Op` es el *opcode* recibido e `Ins` la instrucción esperada. Para cada bit i se espera una tabla de verdad como sigue:

`Op[i]`	`Ins[i]`	**Resultado**
0	0	1
0	1	0
1	0	0
1	1	1

Cuadro 4: Tabla de verdad esperada para la identificación de una instrucción.

De aquí se puede deducir que la compuerta que nos sirve es la negación de `XOR`, `XNOR`. Siguiente a eso, queremos verificar que todos los bits de la secuencia sean iguales a 1. El siguiente circuito, entonces, cumple con lo pedido:

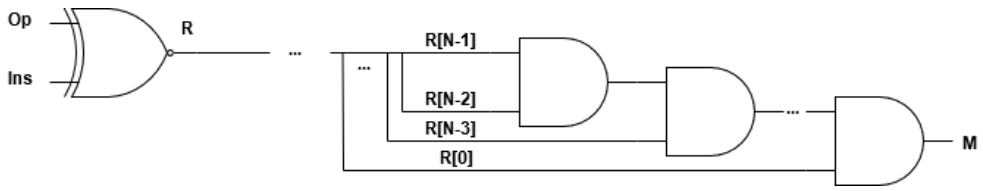


Figura 7: Circuito que retorna una señal igual a 1 para dos buses de bits equivalentes.

En este caso, `M = 1` si, y solo si todos los bits de `Op` coinciden con `Ins`, que es lo buscado.

c. Considere el siguiente programa:

```
x = 2
y = 4
z = 0 # Variable auxiliar
z = x
x = x + y
y = y - z
```

En base a este:

I. Construya un programa en **Assembly** que obtenga el mismo resultado (considere que x, y y z parten con sus valores almacenados en memoria).

```
DATA:
  x 2
  y 4
  z 0
CODE:
  MOV A,(x) ;Guardamos x en A
  MOV (z),A ;Guardamos A en z, variable auxiliar
  MOV B,(y) ;Guardamos y en B
  ADD A,B ;Guardamos en A x+y
  MOV (x),A ;Guardamos en x el resultado de la suma
  MOV A,(y) ;Guardamos y en A
  MOV B,(z) ;Guardamos z en B
  SUB A,B ;Guardamos en A y-z
  MOV (y),A ;Guardamos en y el resultado de la resta
```

II. Ahora, programe en **Assembly** un código que obtenga el mismo resultado de x e y, pero sin hacer uso de la variable z en el segmento **DATA**.

```
DATA:
  x 2
  y 4
CODE:
  MOV A,(x) ;Guardamos x en A
  MOV B,(y) ;Guardamos en B el valor de y
  ADD A,B ;Guardamos en A x+y
  MOV B,(x) ;Guardamos x en B
  MOV (x),A ;Guardamos en x el resultado de la suma
  MOV A,(y) ;Guardamos y en A
  SUB A,B ;Guardamos en A y-z
  MOV (y),A ;Guardamos en y el resultado de la resta
```

III. A partir de los programas anteriores, explique el flujo resultante de cada uno de ellos en el diagrama del computador básico que se muestra a continuación.

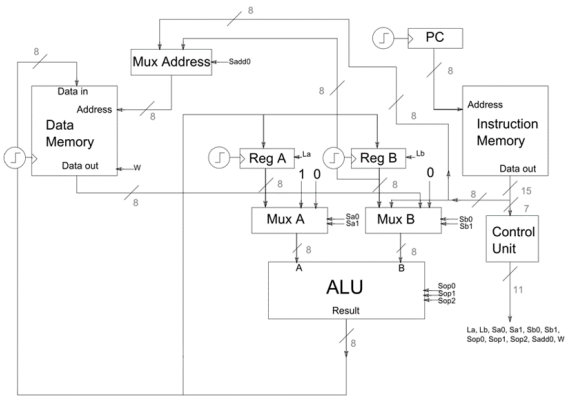


Figura 8: Computador básico visto hasta ahora.

Antes de detallar el flujo, veremos cómo se almacena cada variable en la memoria de datos:

1) `MOV A,2`: Almacena en el registro `A` el valor de la variable x.
2) `MOV (0),A`: Almacena en la primera dirección de la memoria de datos el valor del registro `A` (correspondiente a x).
3) `MOV A,4`: Almacena en el registro `A` el valor de la variable y.
4) `MOV (1),A`: Almacena en segunda dirección de la memoria de datos el valor del registro `A` (correspondiente a y).
5) `MOV A,0`: Almacena en el registro `A` el valor de la variable z.
6) `MOV (2),A`: Almacena en la primera dirección de la memoria de datos el valor del registro `A` (correspondiente a z).

Es decir, por cada variable declarada se ejecutan dos instrucciones que permiten almacenar en la memoria de datos sus valores. Por lo general, el *Assembler* parte en la primera dirección y sigue de forma sucesiva para el resto de las variables declaradas[3].

Siguiendo con el código:

1) `MOV A,(x)`: El literal de la ROM (correspondiente a la dirección de x) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `A` para que el número sea almacenado en él.
2) `MOV (z),A`: El literal de la ROM (correspondiente a la dirección de z) es escogido por el multiplexor `Address` y se habilita la señal W para poder escribir en la RAM. Luego, se configura la `ALU` de forma que el resultado sea el valor del registro `A`, escribiendo ese número en la dirección z.
3) `MOV B,(y)`: El literal de la ROM (correspondiente a la dirección de z) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `B` para que el número sea almacenado en él.

[3]Esto es muy útil al declarar arreglos.

4) `ADD A,B`: Se configuran los multiplexores de `A` y `B` para escoger los valores de sus registros. Luego, se configura la `ALU` para obtener la suma de ambos valores y, finalmente, se habilita la escritura en el registro `A` para que almacene el resultado.
5) `MOV (x),A`: El literal de la ROM (correspondiente a la dirección de x) es escogido por el multiplexor `Address` y se habilita la señal W para poder escribir en la RAM. Luego, se configura la `ALU` de forma que el resultado sea el valor del registro `A`, escribiendo ese número en la dirección x (resultado de $x + y$).
6) `MOV A,(y)`: El literal de la ROM (correspondiente a la dirección de y) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `A` para que el número sea almacenado en él.
7) `MOV B,(z)`: El literal de la ROM (correspondiente a la dirección de z) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `B` para que el número sea almacenado en él.
8) `SUB A,B`: Se configuran los multiplexores de `A` y `B` para escoger los valores de sus registros. Luego, se configura la `ALU` para obtener la suma de ambos valores y, finalmente, se habilita la escritura en el registro `A` para que almacene el resultado.
9) `MOV (y),A`: El literal de la ROM (correspondiente a la dirección de y) es escogido por el multiplexor `Address` y se habilita la señal `W` para poder escribir en la RAM. Luego, se configura la `ALU` de forma que el resultado sea el valor del registro `A`, escribiendo ese número en la dirección y (resultado de $y - z$).

Se obvia el flujo del segundo programa, dado que es una versión simplificada del primero sin mayores cambios en las instrucciones utilizadas.

d. **(I1 - I/2016)** ¿En qué casos es posible soportar la instrucciones `ADD B,Lit` en el computador básico, sin modificar su *hardware* ni sobrescribir datos? Para los casos negativos, indique qué modificaciones al *hardware* y/o `Assembly` se deberían hacer para soportarla.

Si dentro del computador básico revisamos los componentes Mux `A` y Mux `B`, podemos ver que los únicos literales que podrían ser seleccionados para ser almacenados en `A` son 0 y 1. Por lo tanto, sin modificar el computador básico, podemos soportar `ADD B,0`[4] y `ADD B,1` (que existe y llamamos `INC B`). Si quisiéramos habilitar la instrucción `ADD B,Lit` para cualquier literal, sería necesario entonces tener una conexión entre el Mux `A` y la ROM (para así poder recibir el literal a utilizar, al igual como está incluido en `B`) y definir la combinación S_{a0}, S_{a1} para escogerlo y ajustar las señales de control para esta nueva instrucción a partir de un nuevo *opcode*.

[4]Esto claramente no tiene mucho sentido.

e. **(I1 - II/2015)** Modifique el diagrama del computador básico de manera que soporte la ejecución de la instrucción `GOTO dir`, que fuerza que la siguiente instrucción en ejecutarse sea la ubicada en la dirección `dir`.

Para lograr esto, basta con agregar una conexión directa entre el literal de la memoria de instrucciones y el registro `PC`. A continuación se muestra esta idea:

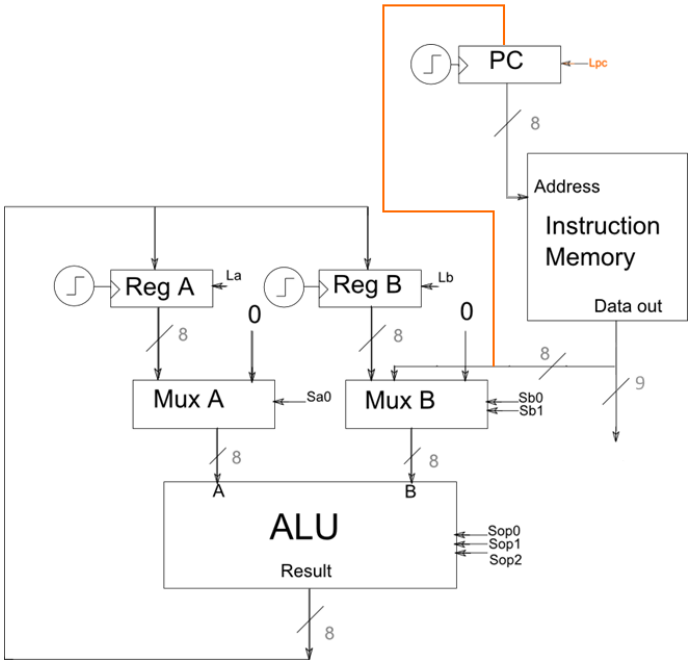


Figura 9: Computador básico con la implementación para la instrucción `GOTO dir`.

Basta con activar la señal L_{pc} una vez que esta instrucción sea llamada. Notar que esta instrucción es el equivalente al `JMP dir` que se encuentra en la versión final del computador básico, el que incluye saltos y subrutinas.

f. **(I1 - I/2018)** Modifique el computador básico para que acepte el comando `MOV A,[DIR]`, que toma el valor almacenado en la dirección `DIR` y luego considera ese valor como una dirección, va a esa dirección y almacena su valor en `A`.

Una posible solución consiste en agregar un registro interno temporal para este tipo de indirecciones, llamémoslo `T` de temporal. Este registro tendrá su entrada de datos del mismo bus de datos de `A` y `B` y su salida será conectada al `Mux A` del computador. Hay que agregar que la instrucción que se busca dar soporte **no puede** ser implementada en un ciclo de reloj, debido al funcionamiento de la RAM (necesita de dos flancos de subida para poder tener los dos accesos solicitados por la instrucción).

Lo que se hace por ciclo, entonces, es lo siguiente:

- **Ciclo 1:** Se hace uso del literal para cargar el valor almacenado en la dirección `DIR` de la RAM en el registro `T`.
- **Ciclo 2:** Se usa la salida del registro `T` como entrada para la dirección de la RAM (lo que implica su conexión con el `Mux Address`) y se obtiene el valor solicitado por la instrucción.

La siguiente figura muestra la adición del registro en conjunto con sus conexiones:

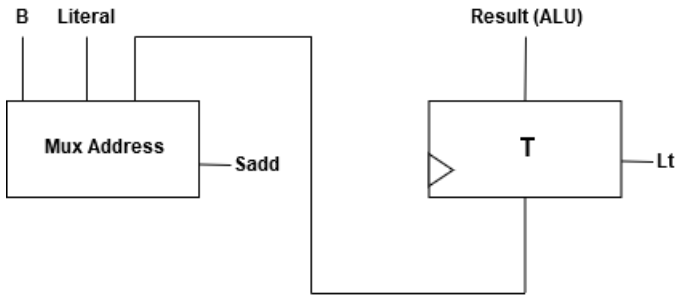


Figura 10: Registro `T` en conjunto con sus conexiones en el computador básico.

Note que es necesario el uso de la señal L_T, dado que no siempre queremos cambiar el valor del registro. Por otra parte, considerando la arquitectura del computador básico sin saltos y subrutinas, tenemos que ahora el `Mux Address` recibe tres entradas, por lo que se necesitan dos bits para seleccionar la salida de dicha componente.

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

IIC2343 – Arquitectura de Computadores

Ayudantía 3 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

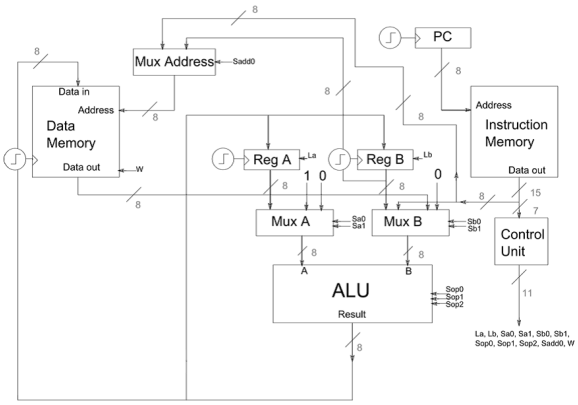
Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

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.

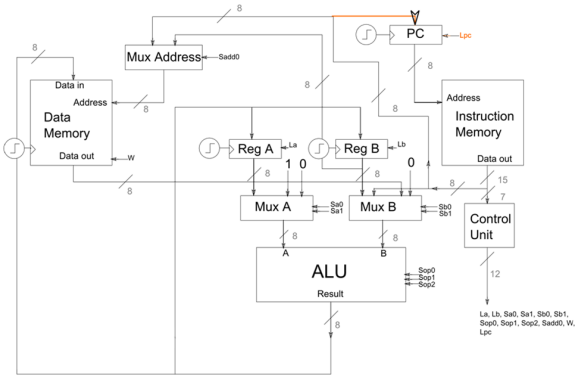
Preguntas

1. a. **(I1 - II/2017)** Considere el siguiente diagrama de bloques del computador básico, aún incompleto.



I. Explica el rol del multiplexor *Address*, da un ejemplo de su funcionamiento.
Su rol es poder seleccionar la dirección que se desea utilizar en la memoria de datos (tanto para lectura como escritura). Existen dos posibilidades, la dirección proveniente como literal de la memoria de instrucciones (direccionamiento directo) y la dirección obtenida desde el registro `B` (direccionamiento indirecto). Un ejemplo de esto es el contraste entre las instrucciones `MOV (Dir),A` y `MOV (B),A`. En el primer caso, sin pérdida de generalidad, la unidad de control le asignará el valor 0 a la señal S_{add0} para que el multiplexor `Address` seleccione el bus proveniente de la memoria de instrucciones (que será la dirección de `Dir`), mientras que en el segundo asignará el valor 1 para seleccionar el valor del registro `B`. No obstante, en ambos casos el valor escrito será el contenido en el registro `A`.

II. Explica qué es necesario agregar para permitir instrucciones de tipo salto incondicional; y explica cómo funcionaría en ese caso un salto incondicional.
Como los saltos incondicionales se realizan con la instrucción `JMP label` (siendo `label` la etiqueta de referencia de la instrucción que queremos ejecutar posteriormente), lo que se necesita es poder entregarle al registro `PC` el número de instrucción deseado y cargarlo para poder obtener el *opcode* y el literal correspondientes. Esto es posible transfiriendo la dirección de la instrucción[1] como literal desde la memoria de instrucciones[2] hacia `PC`. Como solo se transfiere una señal no es necesario un multiplexor por el momento, no obstante, sí se requiere una señal que permita decidir cuándo sobreescribir el valor del registro, que llamaremos L_{pc}. El diagrama resultante se muestra a continuación.



Entonces, su funcionamiento es el siguiente:

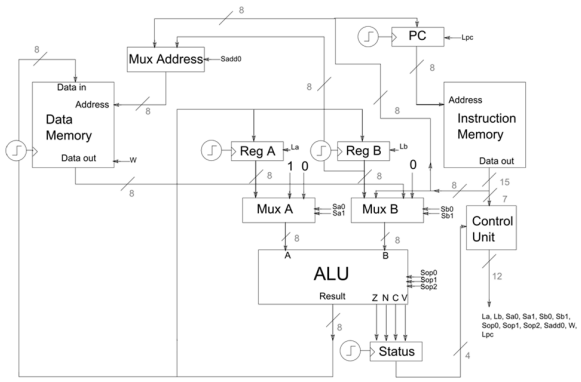
- Se obtiene el *opcode* y el literal de la instrucción `JMP label`.
- La unidad de control identifica la instrucción y habilita la escritura sobre el registro `PC`, es decir, $L_{pc} = 1$.
- El literal (correspondiente a la línea de instrucción que se desea saltar) llega al registro `PC` y es escrito en el mismo una vez que comienza el flanco de subida.

[1] En general, la dirección será equivalente al número de la línea de código del programa.
[2] En `JMP label`, el *Assembler* se encarga de convertir `label` en el literal correspondiente a la línea de la instrucción.

III. Explica qué es necesario agregar a tu respuesta en **II.** para permitir instrucciones de tipo salto condicional; y explica cómo funcionaría en ese caso un salto condicional.
La implementación de saltos condicionales implica dos tipos de instrucción esenciales: `CMP A,x` (comparación entre el valor del registro `A` y `x`) y `JEQ, JNE,` etc. (saltos según condición). En primer lugar, el objetivo de la instrucción `CMP` es poder obtener el resultado de $A - x$ sin tener que cambiar los valores de los registros, lo que en el siguiente ciclo nos ayudará a obtener información sobre la operación. Este resultado se interpreta a partir de *flags*, las que no son más que señales de un bit que nos permiten definir qué pasó con el resultado. Se suelen utilizar cuatro:

- **Z:** Indica si $A - x = 0$ o no.
- **N:** Indica si $A - x < 0$ o no.
- **C:** Indica si hubo *carry* o no.
- **O:** Indica si hubo *overflow* o no.

Por ejemplo, si queremos ejecutar la instrucción `JEQ label`, queremos ver si se activó la señal **Z** (ya que eso indicaría que la diferencia entre ambos es 0, siendo iguales). Para poder lograr esto entonces, primero se necesita añadir un registro `STATUS` que esté conectado al *clock* del computador básico y que reciba las *flags* provenientes de la `ALU`. Esto permitirá coordinar el estado del ciclo **anterior** con la unidad de control y la instrucción que esté leyendo en su momento, de forma que pueda decidir si habilitar o no la señal L_{pc} para permitir o no la escritura sobre el registro `PC`. El diagrama, entonces, se ve como sigue:



Finalmente, su funcionamiento es el siguiente:

- En el primer ciclo, se obtiene el *opcode* de la instrucción `CMP`, realizando la operación de resta en la `ALU` con el objetivo de actualizar las *flags* en `STATUS`.
- En el segundo ciclo, se obtiene el *opcode* y el literal de la instrucción de salto condicional.
- La unidad de control identifica la instrucción y habilita la escritura sobre el registro `PC` si, y solo si se cumplen las condiciones dadas por las *flags* de `STATUS`.
- Si se cumple la condición, el literal (correspondiente a la línea de instrucción a la que se desea saltar) llega al registro `PC` y es escrito en el mismo una vez que comienza el flanco de subida. En caso contrario, L_{pc} será igual a 0 y simplemente se obtendrá el *opcode* de la instrucción siguiente en el programa.

b. ¿Cuántos ciclos toma llamar una subrutina? ¿y cuántos toma retornarla? Justifique.

Para llamar a una subrutina, se siguen los siguientes pasos:

1) Guardar `PC+1` en la posición actual de `SP`.
2) Decrementar en 1 `SP`.
3) Guardar la dirección de la subrutina en `PC`.

En cambio, para retornarla, se siguen estos:

1) Incrementar en 1 `SP`.
2) Guardar el valor de memoria por `SP` incrementado en `PC`.

Si revisamos el computador básico, podemos ver que el llamado a la subrutina toma solo un ciclo, ya que:

1) `PC+1` ya se encuentra esperando en el `Mux DataIn`, espera a que la unidad de control habilite su paso a la RAM y, en el flanco de subida, se almacena en la dirección `SP`.
2) Se habilita la señal *DecSp* en el registro `SP` y, llegado el flanco de subida, decrece correspondientemente.
3) La dirección de la subrutina ya se encuentra en el `Mux PC`. Se habilita su paso a partir de *Spc*, y llegado el flanco de subida, se almacena en el registro `PC`.

Es decir, los 3 cambios se realizan simultáneamente dentro de un solo flanco de subida, por lo que un ciclo es más que suficiente.

Ahora, veamos el retorno:

1) Se habilita la señal *IncSP* en el registro `SP` y, llegado el flanco de subida, decrece correspondientemente.
2) Se guarda en el `PC` lo almacenado en `SP+1` (que sería la siguiente línea de código desde la que se saltó). Sin embargo, `SP` lo cambiamos recientemente, por lo que el valor final llega a `PC` una vez que acaba el flanco de subida y debe esperar al siguiente flanco para habilitar su escritura en el registro.

Por ende, el retorno es imposible realizarlo en un ciclo, son necesarios dos para el funcionamiento correcto de la instrucción.

c. **(Examen - II/2012)** ¿Qué implicancia tiene en el tamaño de los programas el eliminar la conexión entre memoria de datos y `PC` (*program counter*) en el computador básico?

Eliminar esta conexión repercute en la implementación de subrutinas, ya que no sería posible cargar en el registro `PC` las líneas del código que se almacenan en el *stack* para su posterior retorno. Esto implicaría un tamaño mucho mayor en los programas, al tener que repetir código constantemente (un ejemplo es el caso de la multiplicación).

d. **(Apuntes - Saltos y subrutinas)** ¿Cómo se podría implementar en el computador básico la opción de que este avise luego de realizar una operación cuando el resultado es par o impar?

Esto se puede hacer de forma muy sencilla si nos damos cuenta del siguiente hecho: Todo número par en representación binaria termina en 0 y todo número impar termina en 1. Esto, ya que todo bit representa una potencia de 2, salvo el último que representa un 1. Por lo tanto, bastaría con añadir a la salida de la `ALU` una señal P equivalente al último bit del resultado (o su negación) y que fuera almacenada dentro del registro Status de forma directa. Solo faltaría una nueva instrucción asociada a un nuevo *opcode* (que podríamos llamar `JEN` -*Jump Even Number*- o `JON` - *Jump Odd Number*-, por ejemplo).

e. **(I1 - I/2016)** ¿Qué pasaría si se quita el registro `STATUS` del computador básico y se conectaran directamente las señales `ZNCV` a la unidad de control?

Al no estar conectadas a la señal *clock*, estas señales perderán la sincronización con el flanco de subida. Esto implica que las señales que estén ingresadas en la unidad de control no necesariamente sean correspondientes a la última instrucción ejecutada, causando posibles errores en las instrucciones de salto (y por ende, en todo el programa).

f. **(I1 - I/2017)** Si se elimina la instrucción `CMP` del computador básico, ¿cómo deben modificarse las instrucciones de salto, sin alterar el *hardware*, para que estas no dependan del resultado de la última instrucción ejecutada? Escriba detalladamente todas las modificaciones necesarias y sus implicancias. Asuma que solo es necesario resolver el caso de la comparación de los registros A y B y que no es posible sobreescribir los registros para realizar la comparación.

Como no queremos que los saltos dependan de la instrucción anterior (esto es, que no se basen en cómo haya quedado el registro `STATUS` después de ejecutarlas), lo que se necesita hacer es añadir más operaciones a las instrucciones de salto. En este caso, lo que se debe hacer es replicar la instrucción `CMP` dentro de la ejecución que realizan los saltos. De esta forma, cada instrucción de salto contendrá dos *opcodes*:

1) El primero corresponderá al mismo que solía tener `CMP`: Se realiza la resta entre los registros A y B sin guardar el resultado para poder actualizar las flags del registro `STATUS`.
2) El segundo será el que tenía cada salto originalmente.

De esta forma, el cambio sustancial que se genera es que las instrucciones de salto toman **dos ciclos** en vez de uno.

g. **(Examen - I/2017)** Modifique la arquitectura del computador básico para que el registro `STATUS` se actualice solo después de la ejecución de una instrucción `CMP`.

La modificación más simple que se puede hacer para lograr el objetivo consiste en crear una nueva señal (llamémosla L_{stat}). L_{stat} será la señal que habilita la carga de datos en el registro `STATUS`. Finalmente, basta con que la Unidad de Control se encargue de transmitir $L_{stat} = 0$ para todo *opcode* que no corresponda a la instrucción `CMP`.

h. **(I1 - I/2018)** Modifique el *hardware* del computador básico para que las instrucciones `RET` y `POP` tomen un solo ciclo.

Una opción es hacer uso de un sumador de entradas `SP` y 1. Este puede tener su salida conectada a un registro llamado "`SP+1`" conectado al *clock* del computador básico, de forma que esté sincronizado con el resto de los registros. Luego, su salida puede estar conectada al `Mux Address` y, en caso de las operaciones `RET` y `POP`, se configura *Sadd* para su selección. Luego, se puede obtener `Mem[SP+1]` de forma inmediata para su posterior escritura en el registro que corresponda (ya sea `PC`, `A` o `B`). El siguiente diagrama muestra esta idea:

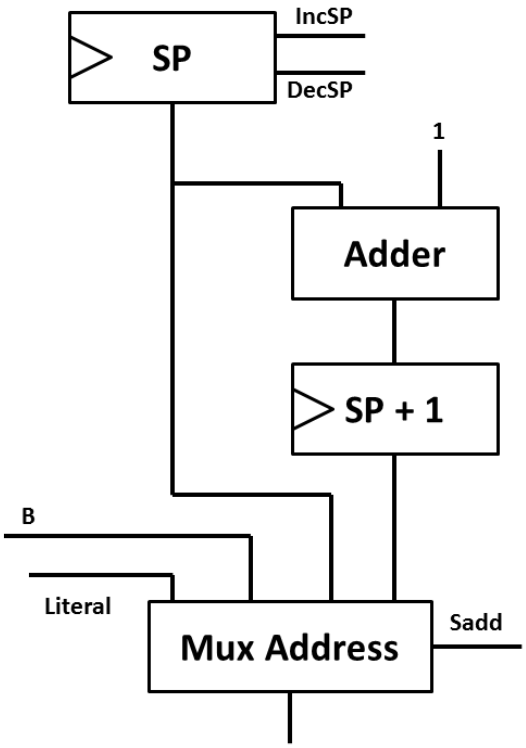


Figura 1: Diagrama del registro `SP+1`.

Esto, finalmente, permite reducir las operaciones `POP` y `RET` a un solo *opcode*, configurando correspondientemente al `Mux Address` para la selección del registro creado.

i. A partir del siguiente código `Assembly`, explique el flujo resultante dentro del computador básico, indicando el valor final de los registros `A` y `B`:

```
DATA:
  r 3 ;Resultado final
CODE:
  MOV A,(r)
  MOV B,2
  PUSH B
  CALL func
  POP B
  JMP finish
  func:
    shift:
      MOV A,B
      CMP A,0
      JEQ end
      DEC A
      MOV B,A
      MOV A,(r)
      SHL A,A
      MOV (r),A
      JMP shift
    end:
      RET
  finish: ;Termina el programa
```

1) `r 3:` Aquí se declara que el *label* `r` (una dirección de memoria) almacenará el literal 3. Al ser la primera (y única), se guarda por *default* en la dirección 0. Entonces, esto es equivalente a las instrucciones `MOV A,3` y `MOV (0),A`. Esto implica que, en primer lugar, se escribe el literal 3 en el registro `A` (habilitando su escritura previamente) y en el siguiente ciclo se habilita la escritura en la RAM y se escribe el valor del registro `A` en la dirección 0.
2) `MOV A,(r):` Se obtiene el dato de memoria en la dirección de `r` (que ya sabemos que es 0). Este dato se extrae de la RAM, lo seleccionamos para que pase por `Mux B` y permitimos la carga en el registro `A` para el resultado (que será la operación suma de la `ALU`, con `A = 0` y `B = Mem[0]`). Entonces, queda `A = Mem[0] = 3`.
3) `PUSH B:` Se almacena en la RAM el valor del registro `B`, en la posición `SP` (correspondiente a la última dirección). El valor de `SP` decrece (para almacenar el próximo valor).
4) `CALL func:` Guarda la posición `PC+1` en la dirección de memoria `SP`, esta señal decrece (para almacenar el siguiente valor), y se carga en `PC` el número correspondiente a la línea de código donde se encuentra la subrutina func (para ejecutar desde ahí).
5) `MOV A,B:` Se guarda en el registro `A` lo que está almacenado en `B` (en un comienzo, el valor igual a 2).
6) `CMP A,0:` Se realiza la operación `A-0` (en este caso, `2-0`) y se guardan las *flags* en el registro `Status` para el próximo comando a ejecutar.

7) `JEQ end:` Desde la unidad de control se revisa si la *flag* `Z` es igual a 0. Como esto no pasa, no se habilita la señal L_{pc} y el `PC` aumenta en uno para proceder con la siguiente línea de código.

8) `DEC A:` Equivalente a `SUB A,1`. El registro almacenado en `A` decrece en una unidad (en este caso entonces, queda `A = 1`).

9) `MOV B,A:` Se guarda en el registro `B` el valor almacenado en `A` (por lo que queda `B = 1`).

10) `MOV A,(r):` Se obtiene el dato de memoria en la dirección de `r = 0`. Este dato se extrae de la RAM, lo seleccionamos para que pase por `Mux B`, y permitimos la carga en el registro `A` para el resultado (que será la operación suma de la `ALU`, con `A = 0` y `B = Mem[0]`). Entonces, queda `A = Mem[0] = 3`.

11) `SHL A,A:` Almacenamos en `A` el resultado de hacerle un `shift left` al número. En este caso, al ser equivalente a una multiplicación por 2, queda `A = 6`.

12) `MOV (r),A:` Se habilita la escritura en la RAM y en la dirección `r = 0` se almacena el valor del registro `A` (entonces, `Mem[0] = 6`).

13) `JMP shift:` Se realiza un salto incondicional a la *label* `shift` (es decir, L_{pc} `= 1`, y se carga la línea de código correspondiente para la próxima ejecución).

14) `MOV A,B:` Se guarda en el registro `A` lo que está almacenado en `B` (ahora, un valor igual a 1).

15) `CMP A,0:` Se realiza la operación `A-0` (en este caso, `1-0`) y se guardan las *flags* en el registro `Status` para el próximo comando a ejecutar.

16) `JEQ end:` Desde la unidad de control se revisa si la *flag* `Z` es igual a 0. Como esto no pasa, no se habilita la señal L_{pc} y el `PC` aumenta en uno para proceder con la siguiente línea de código.

17) `DEC A:` Equivalente a `SUB A,1`. El registro almacenado en `A` decrece en una unidad (en este caso entonces, queda `A = 0`).

18) `MOV B,A:` Se guarda en el registro B el valor almacenado en `A` (por lo que queda `B = 0`).

19) `MOV A,(r):` Se obtiene el dato de memoria en la dirección de `r = 0`. Este dato se extrae de la RAM, lo seleccionamos para que pase por `Mux B` y permitimos la carga en el registro `A` para el resultado (que será la operación suma de la `ALU`, con `A = 0` y `B = Mem[0]`). Entonces, queda `A = Mem[0] = 6`.

20) `SHL A,A:` Almacenamos en `A` el resultado de hacerle un `shift left` al número. En este caso, al ser equivalente a una multiplicación por 2, queda `A = 12`.

21) `MOV (r),A:` Se habilita la escritura en la RAM y en la dirección `r = 0` se almacena el valor del registro `A` (entonces, `Mem[0] = 12`).

22) `JMP shift:` Se realiza un salto incondicional a la *label* `shift` (es decir, L_{pc} `= 1` y se carga la línea de código correspondiente).

23) `MOV A,B:` Se guarda en el registro `A` lo que está almacenado en `B` (ahora, un valor igual a 0).

24) `CMP A,0:` Se realiza la operación `A-0` (en este caso, `0-0`) y se guardan las *flags* en el registro `Status` para el próximo comando a ejecutar.

25) `JEQ end:` Desde la unidad de control se revisa si la *flag* `Z` es igual a 0. Como esto pasa, se habilita la señal L_{pc} y el registro `PC` carga la dirección de la línea de código del *label* `end`.

26) `RET: SP` aumenta en 1 y se carga en el registro `PC` lo almacenado en `Mem[SP]` (en este caso, la línea de código correspondiente a la siguiente de donde se hizo el llamado a la función).

27) `POP B: SP` aumenta en 1 y se carga en el registro `B` lo almacenado en `Mem[SP]` (en este caso, el valor original del registro `B` antes de ingresar a la función, `B = 2`).

28) `JMP finish:` Se realiza un salto incondicional a la *label* `finish` (es decir, L_{pc} `= 1`, y se carga la línea de código correspondiente). Como aquí ya no hay más código, es la última ejecución a realizar (la ROM suele tener después solo líneas de código que hacen que no se ejecute nada dentro del computador, manteniendo el estado final).

Finalmente, podemos ver que los valores finales de nuestros registros son A = 0 y B = 2. Notemos que el valor de A **no es** 12, ya que este cambió dentro de la rutina, y no terminamos por almacenarle el resultado escrito en memoria. Sí sabemos que en la memoria lo calculado se mantuvo (Mem[0] = 12).

2. a. **(Examen - II/2016)** Modifique la arquitectura del computador básico para que funcione con lógica ternaria en vez de binaria. Más específicamente, modifique los tamaños de los elementos (buses, registros, señales de control, etc.) de modo que el nuevo computador tenga una capacidad similar a la versión binaria. Asuma que existen todos los componentes vistos en clases en versión ternaria. **Nota:** No es válido utilizar los valores ternarios como si fueran binarios.

Aquí, lo relevante es ver cómo cambia la representación de datos que se tenía antes. Con un trit, podemos representar un número más que con un bit, con dos trits, podemos representar 4 números más que con dos bits, y así sucesivamente. Por lo tanto, iremos estudiando cómo cambiaría cada parte del computador básico:

- **Registros y buses de datos:** Estos almacenaban 8 bits. Para poder abarcarlos todos, resolvemos el siguiente problema para K trits y N bits:

$$\begin{array}{lcl} 3^K & = & 2^N \\ \log_3(3^K) & = & \log_3(2^N) \\ K & = & N * \log_3(2) \\ K & = & \lceil N * \log_3(2) \rceil \end{array} \tag{1}$$

Notar que como K debe ser un número entero, usamos la función techo para aproximar al entero mayor (si usáramos la función piso, podríamos perder números representables). Por lo tanto, para $N = 8$ bits, tenemos que $K = \lceil 5{,}047 \rceil = 6$.
Otra forma de obtener el resultado es simplemente ver el intervalo de representación con N bits:

$$-\frac{2^N}{2} \leq x \leq \frac{2^N}{2} - 1$$

Y tomando el intervalo para los trits:

$$-\left\lfloor \frac{3^K}{2} \right\rfloor \leq x \leq \left\lfloor \frac{3^K}{2} \right\rfloor$$

Buscamos el K tal que el intervalo de representación de bits (con $N = 8$) sea considerado **por completo**. (Notar que nos importa representar todo, da lo mismo si nuestro poder de representación aumenta, no queremos que disminuya). De esta forma, también se llega a $K = 6$.
Es importante ver que esto no lo cambiamos para el registro `STATUS`, ya que aquí nos interesa ver si cada *flag* cumple o no para generar los saltos (utilizar base trinaria dificultaría un poco el funcionamiento de este).

- **Señales de control:** Algunas de las señales de control se pueden abreviar de la siguiente forma:
  - Selector de operaciones de la `ALU`: Aquí tenemos 8 operaciones que se seleccionan a partir de 3 bits. Sin embargo, bastan 2 trits para tener 8 combinaciones diferentes, y así, escoger la operación correspondiente.
  - Incrementar/decrementar `SP`: Ahora, podemos usar un trit para las siguientes combinaciones: 0 (no incrementar ni decrementar), 1 (incrementar) y 2 (decrementar).
  - Selector del Mux `A` y Mux `Address`: Como escogemos entre tres señales para estos dos multiplexores, podemos quedarnos con un solo trit para obtener las 3 combinaciones. Notar que esto no sirve para el `Mux B`, ya que se selecciona entre 4 entradas (por lo que se siguen necesitando dos trits).

Estos corresponden a los cambios aplicables para mantener el funcionamiento de nuestro computador, utilizando trits en vez de bits.

b. **(I1 - II/2016)** En esta pregunta deberá diseñar un computador especializado en el manejo de matrices. El computador debe ser capaz de: i) copiar una matriz desde la memoria de datos a un registro y viceversa, ii) sumar 2 matrices almacenadas en registros distintos y almacenar el resultado en un registro.

   I. Haga el diagrama del computador, considerando que las matrices pueden tener como máximo $N \times N$ elementos, cada uno de 1 byte.
   En este caso, no es necesario modificar de forma significativa el diagrama del computador básico. Basta con aumentar el tamaño de los registros, buses de datos, `ALU` y las palabras de memoria a N^2 bytes para que puedan almacenar la matriz en su totalidad.

   II. Diseñe el *assembly* del computador. Cada instrucción debe estar asociada a un *opcode* y estos a sus respectivas señales de control.
   En base al diagrama anterior, basta utilizar el *assembly* del computador básico. La única diferencia se da en los literales, donde existen dos opciones: i) separar cada uno de los elementos de una matriz con coma (,) y ii) declarar la matriz completa como un único número de tamaño N^2 bytes (este último hace más sentido dada la representación anterior).

   III. Agregue tanto al *hardware* como al *assembly* soporte para una instrucción que permita modificar el valor de un elemento arbitrario de una matriz almacenada en la memoria de datos.
   Una posible solución es agregar unidades de *shifting* para aislar el elemento buscado y modificarlo y luego realizar operaciones lógicas para integrar este valor con la matriz completa. Notar que esto añade una gran complejidad en términos de *hardware*, pero logra cumplir el objetivo.

c. **(I1 - I/2018)** El computador básico solo trabaja con números enteros, sin embargo, para muchas aplicaciones es útil poder trabajar con números decimales. En consecuencia, usted deberá añadir soporte para números decimales al computador básico.

I Indique un esquema de números decimales que podría emplear en el computador básico, mencionando sus ventajas y desventajas.
Un esquema posible es el uso de precisión fija decimal. De este modo se utiliza un número fijo de bits para parte entera y parte decimal. Tiene la ventaja de ser relativamente simple y barato de implementar, pero la desventaja de estar limitado en el rango de números que puede abarcar. Otros esquemas pueden ser esquemas de punto flotante como `IEEE754`.

II Añada los componentes necesarios para trabajar con el esquema de números decimales argumentados en la parte **I.**, indicando qué hace cada componente y cómo se conecta a las partes existentes del computador básico, detallando su interacción con este. Si bien no es necesario hacer a nivel de compuertas todos los componentes, no se aceptarán componentes mágicos como "Unidad de cómputo decimal". Sí se aceptará la `ALU`, registros de n bits y otros componentes desarrollados en clases.
Es posible añadir 4 registros de 8 bits, iguales a los `A` y `B` del computador básico, que pueden ser llamados `DAi`, `DAd`, `DBi` y `DBd`, por `Decimal{A,B}(integer/decimal)`. Dichos registros tendrán sus `DataIn` conectados a la salida de la `ALU` y sus salidas conectadas a una `ALU` de partes decimales y otra de partes enteras. Además, el `CarryOut` de la `ALU` de partes decimales estará conectada al `CarryIN` de la `ALU` de partes enteras. Las salidas de dichas `ALUs` realimentarán las rutas que se alimentaban originalmente de la salida de la `ALU` normal. Este esquema hace que se requieran 4 cargas para cargar los operandos decimales y 2 ciclos para guardar sus resultados. Además, modifica las operaciones enteras para que pasen sin operar en las `ALUs` de decimales.

Otro acercamiento válido es usar los registros y `ALUs` al mismo nivel de la `ALU` normal, incluso pudiendo reutilizar esta con los nuevos registros, y usar un `Mux` de salida para seleccionar qué parte del resultado es la que se carga en registro o se guarda en memoria.

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

IIC2343 – Arquitectura de Computadores

Ayudantía 4 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. **(I2 - II/2015)** Compare las arquitecturas Harvard y Von Neumann desde el punto de vista del tiempo de ejecución de las instrucciones. Fundamente y explique claramente las diferencias.

   En la arquitectura de Harvard (utilizada para el computador básico estudiado en el curso) se tiene un tiempo promedio menor en ejecución si se compara con la arquitectura Von Neumann. Esto, debido a que en la primera (salvo por las instrucciones `POP` y `RET`) utiliza un ciclo por instrucción, mientras que Von Neumann utiliza al menos dos o tres (un ciclo donde se obtiene el *opcode* de la instrucción y se transfiere, otro donde se obtiene el literal y se transfiere -estas dos primeras se podrían realizar en uno asumiendo un *tradeoff* con respecto al rango de valores de los literales y *opcodes*-, y el último para ejecutar la instrucción en sí). Si bien hay instrucciones que se podrían seguir ejecutando en un ciclo (por ejemplo, las operaciones de la `ALU` sobre los registros `A` y `B`), todo lo que conlleve a accesos a memoria necesitaría al menos dos ciclos (uno para acceder a la instrucción y literal en memoria, y el otro para acceder a la variable almacenada). Por esto, se asume que para toda instrucción en la arquitectura Von Neumann se necesitan dos o tres ciclos en promedio (por comodidad, en general se asumirán tres).

b. **(I2 - I/2017)** ¿Cuántos ciclos como mínimo puede tomar en un computador con arquitectura Von Neumann, una instrucción que lea y luego modifique el contenido de una posición de memoria?

Antes de desarrollar la pregunta, sin pérdida de generalidad, se asume que **leer** hace referencia a una instrucción del tipo `MOV A,(Dir)`, mientras que **modificar** usa una instrucción del tipo `MOV (Dir),A`, habiendo realizado un cambio sobre el registro donde se almacenó el dato de la memoria (por ejemplo, `ADD A,3`).

Aquí nos encontramos con dos casos:

- Asumiendo que no podemos obtener el *opcode* y el literal en un solo ciclo:
  1) Se obtiene el *opcode* de `MOV A,(Dir)` y se envía a la unidad de control, la que posteriormente propaga las señales de control necesarias para la ejecución.
  2) Se obtiene el literal (dirección de memoria) y se propaga a los multiplexores correspondientes (en este caso, al `Mux Address`).
  3) Se ejecuta la instrucción completa en el flanco de subida del tercer ciclo, donde se obtiene el dato almacenado en `Dir` en el registro `A`.
  4) Se obtiene el *opcode* de la operación a realizar sobre el registro `A` y se envía a la unidad de control, la que posteriormente propaga las señales de control necesarias para la ejecución.
  5) Se obtiene el literal para la operación y se propaga a los multiplexores correspondientes (podemos asumir, sin pérdida de generalidad, un literal `Lit` a sumarse con `A`).
  6) Se ejecuta la instrucción completa en el flanco de subida del sexto ciclo, donde el resultado de la operación se almacena, nuevamente, en el registro `A`.
  7) Se obtiene el *opcode* de `MOV (Dir),A` y se envía a la unidad de control, la que posteriormente propaga las señales de control necesarias para la ejecución.
  8) Se obtiene el literal (dirección de memoria) y se propaga a los multiplexores correspondientes (en este caso, al `Mux Address`).
  9) Se ejecuta la instrucción completa en el flanco de subida del noveno ciclo, donde el dato del registro `A` se almacena en la dirección de memoria `Dir`.
- Asumiendo que sí podemos:
  1) Se obtiene el *opcode* de la instrucción `MOV A,(Dir)` y el literal de la dirección, propagándose a los componentes correspondientes.
  2) Se ejecuta la instrucción, almacenando el dato de la dirección `Dir` en el registro `A`.
  3) Se obtiene el *opcode* de la instrucción a ejecutar sobre el registro `A` y el literal para operar, propagándose a los componentes correspondientes.
  4) Se ejecuta la instrucción, almacenando el resultado de la operación en el registro `A`.
  5) Se obtiene el *opcode* de la instrucción `MOV (Dir),A` y el literal de la dirección, propagándose a los componentes correspondientes.
  6) Se ejecuta la instrucción, almacenando en la dirección `A` el resultado que se encuentra en el registro `A`.

En ambos casos, se ve que la cantidad de ciclos necesaria para la ejecución de lo pedido aumenta considerablemente si se hace el contraste con una arquitectura Harvard.

c. **(I2 - II/2014)** Modifique el computador básico, para que este utilice un esquema Von Neumann, *i.e.*, memoria de datos e instrucciones unificadas en una sola.

La principal modificación que se le debe realizar al esquema del computador básico es unir la memoria de instrucciones con la memoria de datos en una sola. Esto claramente modifica los ciclos requeridos por instrucción, por lo que una forma de solucionarlo es dejar por *default* 3 ciclos: El primero se utiliza para enviar el *opcode* y que la unidad de control propague las señales correspondientes (cuidando que no se altere el contenido de los registros), el segundo para enviar el literal correspondiente para realizar las operaciones y, el tercero, para hacer finalmente la ejecución de la instrucción. Entre las consideraciones importantes se tiene el manejo del registro `PC`, donde debe suspender su incremento durante el tercer ciclo de ejecución. Si no se hiciera esto, durante el tercer ciclo saldría de la memoria el *opcode* de la siguiente instrucción, afectando la ejecución original. Por otra parte, también se debe tomar en cuenta que en la unidad de control es necesario realizar un bloqueo de la entrada durante el segundo y tercer ciclo de una instrucción. Esto, con el fin de evitar un cambio en las señales de control provenientes de esta componente. A continuación, **un posible** diagrama de esta modificación.

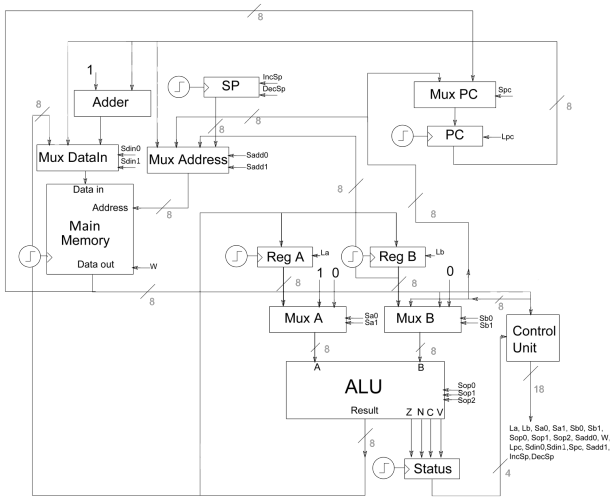


Figura 1: Diagrama del computador básico con arquitectura Von Neumann.

De aquí notamos lo siguiente:

- La salida de la memoria principal **debe** estar conectada directamente a la unidad de control, ya que puede corresponder a un *opcode*. Esta misma salida se usa como el literal de las instrucciones y los datos de memoria.
- `Mux Address` ahora también recibe el valor del registro `PC` como dirección, dado que las instrucciones se encuentran en la memoria principal.
- Al hacer uso de la memoria principal, se trabaja con una RAM de palabras de 8 bits. Por lo tanto, los *opcodes* y los literales se ajustan a estas dimensiones. Si se dejara un *opcode* y literal por palabra, ambos se verían perjudicados por un rango menor.

d. **(I2 - II/2014)** Dada la microarquitectura del computador básico, ¿es posible crear una ISA distinta la actual? Argumente su respuesta.

Sí, es posible, ya que esta se puede modificar de dos formas:

- Se puede usar un nuevo *opcode* que haga uso de una combinación de señales no utilizada antes para un nuevo comando. Por ejemplo, la combinación de señales que obtiene la suma del registro `A` y `B`, y luego guarda el resultado en ambos ($L_A = 1, L_B = 1, S_A =$ `A`, $S_B =$ `B`, $S_{op} =$ `ADD`).
- Usando una nueva instrucción que corresponda a la combinación de distintos *opcodes*. Por ejemplo, una instrucción que incremente en una unidad una variable almacenada en una dirección de memoria: `INC (dir) = PUSH B - MOV B,(dir) - INC B - MOV (dir),B - POP B`. Notar que la primera y última instrucción se usan de forma que no perdamos el valor almacenado en el registro `B`, pues la funcionalidad de la instrucción definida **no debiese** modificar el contenido de los registros si no se indica.

e. **(I2 - I/2015)** ¿Es posible agregar al `Assembly` del computador básico la instrucción `MOV A,(A+B)`, sin modificar la microarquitectura? Justifique su respuesta en cualquiera de los dos casos.

Sí, es posible. Basta con asignarle al nuevo comando los *opcodes* de las siguientes instrucciones existentes de forma consecutiva: `PUSH B - ADD B,A - MOV A,(B) - POP B`.

f. **(I2 - II/2016)** Modifique (solo) la ISA del computador básico para soportar la instrucción `CALL reg`, que permite llamar a la subrutina ubicada en la dirección de memoria almacenada en el registro `reg`.

Definimos en nuestra ISA la instrucción `CALL reg` de la siguiente forma:

- Primero se ejecuta la instrucción `MOV (dirreg),reg` para almacenar en la dirección `dir` el valor almacenado en uno de los registros (ya sea `A` o `B`). Se puede asumir que `dirreg` es una dirección fija con uso exclusivo para esta instrucción, lo que se puede definir en el *Assembler*.
- Añadimos la instrucción `CALL (dir)`, que carga en el registro `PC` el valor almacenado en la dirección `dir`. Notar que esto debe ocurrir en dos ciclos, ya que primero tenemos que guardar en el *stack* la posición `PC+1` para poder volver al retornar el llamado (*i.e.* como en las instrucciones `PUSH, CALL`, pero con $L_{pc} = 0$), y otro para obtener el valor almacenado en `dir` y cargarlo en el registro `PC`, lo que se puede hacer con la siguiente combinación de señales en el computador básico: $L_{pc} = 1$, $S_{pc} =$ `RAM Data out`, $S_{add} =$ `Lit`. En este caso, `Lit` será el literal correspondiente a la dirección `dir`, por lo que en este caso se define `dir = dirreg`.

El cuidado que tendría que tener el programador es de no alterar el contenido de la dirección `dirreg` en ningún momento.

2. a. **(I1 - I/2013)** ¿Cuál es el valor del número 11000001100000000000000000000000, representado mediante el tipo de dato `float`?

Sabemos que los `float` siguen la siguiente estructura (en el orden mencionado):

1) **Signo:** Un bit.
2) **Exponente:** 8 bits.
3) **Significante:** 23 bits.

Entonces, tenemos que:

- Signo = 1 (que representa un número negativo).
- Exponente = 10000011 = 131 (en base 10).
- Significante = 00000000000000000000000

En este caso, nuestro número es:

$$x = (-1)^1 * 1,00000000000000000000000 * 10^{(131-127)_2}$$

Entonces, tenemos finalmente que $x = -(10^{100})_2 = -2^4 = -16$.

b. **(I1 - II/2012)** Escriba en formato `float` el número 16,375 (decimal). Indique cómo se divide y qué significa cada una de las partes del *string* de bits.

Primero, necesitamos pasar el número a base binaria. Para ello, usamos el método de las restas sucesivas, que funciona de la siguiente forma:

- Obtenemos el mayor número 2^N que sea menor o igual al número que deseamos transformar.
- Al número original se le resta 2^N, ponemos un 1 en su posición (que en este caso sería la primera), y pasamos a una potencia de una unidad menor.
- Si 2^{N-1} es mayor a nuestro resultado anterior, disminuimos nuevamente la potencia en una unidad, y esta no la utilizamos (ponemos un 0 en su posición). En caso contrario, se resta (poniendo un 1 en su posición), y seguimos el mismo procedimiento con el nuevo número ya sustraido, y la potencia 2^{N-2}
- Esto se realiza hasta que el resultado sea 0. Si nunca nos da ese número, entonces estamos en presencia de un número binario infinito (que puede o no tener periodo).

En este caso, tenemos el siguiente desarrollo:

$$\begin{aligned}
16,375 - 1 * 2^4 &= 00,375 \\
00,375 - 0 * 2^3 &= 00,375 \\
00,375 - 0 * 2^2 &= 00,375 \\
00,375 - 0 * 2^1 &= 00,375 \\
00,375 - 0 * 2^0 &= 00,375 \\
00,375 - 0 * 2^{-1} &= 00,375 \\
00,375 - 1 * 2^{-2} &= 00,125 \\
00,125 - 1 * 2^{-3} &= 0
\end{aligned}$$

Por lo tanto, nuestro número en binario corresponde a 10000,011 (notar que la coma se coloca después de que comienzan a utilizarse potencias negativas). Esto es lo mismo que 1,0000011 $*10^{100}$ (notar aquí que multiplicar por 10 en binario es equivalente a hacer un `shift left`, y la potencia es 100 ya que este número corresponde a 4 en binario).

Para escribir nuestro número, analizamos las 3 partes importantes:

- **Signo:** Como el signo es positivo, tenemos que el bit de signo es 0.
- **Exponente:** El exponente es 4, pero como está desfasado: $x - 127 = 4 \rightarrow x = 131$. Utilizamos los 8 bits asignados para representar este número: 10000011
- **Significante:** Tenemos que el significante es 0000011, y para completar los 16 bits faltantes, utilizamos solo ceros.

Finalmente, el número resultante es 0 10000011 00000110000000000000000.

c. **(I1 - II/2011)** Se tienen dos números de punto flotante de precisión simple en formato IEEE754: A = 0x3E200000 y B = 0x00000000. ¿Cuál es el resultado, en formato IEEE754, de A : B?

Aquí, la gracia es ver que bajo el estándar IEEE754, la división por el 0 entrega un número infinito. Antes de anotar el resultado, necesitamos saber si el número es positivo, lo que vemos a partir de A:

0x3E200000 = 00111110001000000000000000000000

Como este estándar nos dice que el primer bit indica el signo, sabemos que A es positivo. Finalmente, la representación de un infinito positivo en IEEE754 es:

0x7F800000 = 01111111100000000000000000000000

d. **(Examen - I/2016)** Explique por qué el número $2^{50}+5$ no puede representarse de manera exacta usando el tipo de dato `float` de 32 bits.

Es necesario recordar la composición de un `float` de 32 bits, en el orden dado (de izquierda a derecha):

- **Signo:** 1 bit.
- **Exponente:** 8 bits.
- **Significante:** 23 bits.

Recordando que el exponente está desplazado en 127 unidades, tenemos que 2^{50} se expresa de la siguiente forma:

- **Signo:** 0 (positivo).
- **Exponente:** 10110001 (177).
- **Significante:** 00000000000000000000000.

Ahora, es importante notar que si se le suman 5 unidades al número, estas se deben ver reflejadas en el significante. Esto corresponde a un 1 que se encuentra 50 posiciones a la izquierda (proveniente de 2^{50}), y un 101 al final (proveniente de 5). Como la cantidad de ceros entre el 1 y el 101 supera el número de bits a disposición para el significante, $2^{50}+5$ no se puede representar.

Nota: Esto pasa para todos los números de la forma $2^{24}+x$ en adelante, $2^{23}+x$ se puede representar.

e. **(I1 - I/2017)** Sea P, el conjunto de representaciones de punto flotante de $s + e + 2$ bits, con s bits para el significante normalizado, 1 bit para el signo de este, e bits para el exponente (no desplazado) y 1 bit para el signo de este. Considere además x y $\tilde{x}$, números pertenecientes a $\mathbb{Z}$, ambos codificados usando una representación perteneciente a P, tales que:

- $sucesor(x) \neq x + 1$
- $\forall \tilde{x} < x, sucesor(\tilde{x}) = \tilde{x} + 1$

Donde $sucesor(y)$ es una función que retorna el siguiente número entero mayor que y (el sucesor de y).

En base a esto, responda las siguientes preguntas:

I. Indique cuál es el valor de x en función de los parámetros s y e y muestre que dado x, siempre existe un número $x^{'} \in \mathbb{Z}$, codificado en la misma representación que x, tal que $x^{'} + 1 = x$.
Lo primero que necesitamos para determinar el valor de x es entender sus propiedades. Supongamos que $s = 2$ y $e = 4$, que el bit 0 representa el signo positivo y por ahora obviemos los números negativos. La representación a utilizar tendrá el siguiente orden[1]:

$$se + exp + ss + sig$$

Donde:

- se: Signo exponente.
- exp: Exponente.
- ss: Signo significante.
- sig: Significante.

Entonces, la secuencia de números enteros que podemos generar partiendo de 1 es la siguiente:

$$\begin{array}{rcl} 1 & \rightarrow & 00000000 \\ 2 & \rightarrow & 00001000 \\ 3 & \rightarrow & 00001010 \\ 4 & \rightarrow & 00010000 \\ 5 & \rightarrow & 00010001 \\ 6 & \rightarrow & 00010010 \\ 7 & \rightarrow & 00010011 \\ 8 & \rightarrow & 00011000 \end{array}$$

¿Cómo podemos representar ahora el número 9? Es fácil ver que no se puede, pues el siguiente número, bajo este esquema, será:

$$00010001 \rightarrow (1010)_2 = 10$$

[1] Para no causar confusiones, llamamos a exp el valor del exponente y sig el valor del significante. En cambio, e y s representan el número de bits de estos.

Entonces, tenemos que $sucesor(8) = 10 \neq 9$, por lo que 8 sería el x buscado en esta representación. Se puede ver que este número consiste en el que ya no es posible aumentar su tamaño de forma unitaria, sino que solo a través de saltos en potencias de 2. Esto, ya que nuestro significante no es capaz de representar el bit menos significativo con un 1 para valores mayores en el exponente[2].
De la conclusión anterior se puede ver que el valor de x está sujeto al valor de s. Cuando el valor del exponente supera al número de bits para el significante es que se genera esta propiedad, es decir, $exp > s \rightarrow exp >= s+1$. Como x es el primer número con estas características:

$$x = 2^{s+1} = 0[bin(s+1)]0[zero(s)]$$

Donde:

- $bin(s+1)$: $s+1$ representado en base 2.
- $zero(s)$: s bits iguales a cero.

Ahora, para probar la existencia de $x^{'}$ simplemente usamos la definición del enunciado:

$$\forall \tilde{x} < x, sucesor(\tilde{x}) = \tilde{x} + 1 \rightarrow x - 1 < x, sucesor(x-1) = x - 1 + 1 = x$$

Finalmente: $x - 1 = x^{'} \rightarrow x^{'} + 1 = x$ ■

II. ¿Existe un número con las características de x en el estándar IEEE754 de 32 bits? En caso positivo, indique su valor, y en caso negativo indique por qué no es posible encontrar este número.
Sí, existe. Las diferencias son las siguientes:

(1) Los valores de s y e son fijos: $s = 23$, $e = 8$.
(2) No existe el bit se. Para tener exponentes positivos y negativos, el valor de exp está desfasado en 127.

De esta forma, adaptando la respuesta anterior, el número generado es el siguiente:

$$x_{IEEE754(32b)} = 2^{23+1} = 0[bin(23+1+127)][zero(23)]$$

III. Caracterice, en función de los parámetros s y e, el conjunto de representaciones de punto flotante $\tilde{P} \subset P$, tales que no contienen un número con las características de x. Para caracterizar al conjunto $\tilde{P}$ basta con ajustar el parámetro e en función de s. Esto, de forma que s sea lo suficientemente grande para poder representar siempre al bit menos significativo, sin importar el rango de valores de exp. Como el problema se da para $exp = bin(s+1)$:

$$e \leq \sum_{i=0}^{s-1} 2^i = \frac{1-2^s}{1-2} = \frac{2^s-1}{2-1} = 2^s - 1$$

Notemos que al requerir $exp < bin(s+1)$, lo que debemos hacer es que e sea menor o igual a la suma de todos los bits de s, esto es, $\sum_{i=0}^{s-1} 2^i$. El resultado se desprende de la suma de los primeros n términos de una serie geométrica[3].

[2] Asumimos que, dado el caso donde $exp > s$, los bits generados después de lo que puede representar el significante son todos iguales a 0. Por ejemplo, si $s = 2$ y $e = 3$, tomando $sig = 11$ y $exp = 011$ generamos 1110.

[3] https://es.wikipedia.org/wiki/Serie_geom%C3%A9trica

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

IIC2343 – Arquitectura de Computadores

Ayudantía 5 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. Indique la ISA del computador básico visto en el curso y la correspondiente a la arquitectura `x86`, explicando sus diferencias.

   La ISA del computador básico corresponde a un set de instrucciones RISC (*Reduced Instruction Set Computer*), la que posee funcionalidades básicas al tener una arquitectura más sencilla en el computador. Esto implica que para realizar operaciones más avanzadas sería necesario realizar un programa más complejo. En cambio, la ISA del computador con arquitectura `x86` corresponde a un set de instrucciones CISC (*Complex Instruction Set Computer*), la que posee funcionalidades más avanzadas por una arquitectura más compleja y costosa. Un ejemplo de esto son los comandos de multiplicación y división que se pueden realizar de forma directa (en vez de tener que hacer un programa completo para realizarlo, como lo es en el caso de la RISC del computador básico).

   b. **(I2 - I/2016)** ¿Por qué es necesaria la existencia de la instrucción `RET Lit` en un computador `x86`?

   Su existencia es necesaria ya que permite dejar el *stack pointer* (`SP`) en la posición que le corresponde según **la cantidad de parámetros ingresados a la llamada**. De no hacerlo, asumiendo que hubo un llamado anterior, al retornar a este no se podría acceder de forma correcta ni a los parámetros ni a las variables locales (en caso de haber), dado que el registro `SP` no estaría ubicado en el tope del *stack*, sino que más arriba.

c. **(I2 - I/2013)** Al iniciar el cuerpo de una subrutina, ¿por qué es necesario ejecutar las instrucciones `PUSH BP` y `MOV BP, SP`? ¿Qué pasa si no se ejecutan?

Veremos la importancia e implicancia de cada una por separado.

- `PUSH BP`: Respalda el valor previamente almacenado en el registro `BP`. Si no lo utilizamos, quizás no habría problema si estamos seguros de que usaremos una única subrutina con una llamada (ya que después no volveríamos a usar el registro). Sin embargo, imposibilitaría el llamado consecutivo de subrutinas (por ejemplo, con una recursión), ya que no podría volver a la dirección correspondiente, y los accesos a memoria para los parámetros y variables locales sería erróneo.
- `MOV BP, SP`: Le otorga al registro `BP` el valor almacenado en `SP`, lo que permite luego usarlo como referencia para tener la dirección de los parámetros y las variables locales. Si no lo utilizáramos, tendríamos solo la posibilidad de usar `SP` como referencia, lo que complejizaría de sobremanera el código ya que el programador debería estar siempre atento a la última posición adquirida del registro.

d. **(I2 - II/2014)** ¿Es posible emular el funcionamiento del registro `BP` en el computador básico, sin modificar la arquitectura? Si su respuesta es positiva, esboce la solución. Si es negativa, explique el motivo.

Sí, es posible emular el registro `BP`. Esto se puede hacer utilizando una dirección fija de memoria. De esta manera, en vez de hacer referencia a `BP`, se hará referecia a esta dirección (se podría configurar el *Assembler* de forma que reemplace `BP` por su dirección). El único cuidado necesario es asegurarse que el valor almacenado en esa dirección no sea sobreescrito.

e. **(I2 - I/2017)** ¿Cuántas palabras de la memoria son modificadas al ejecutar la instrucción `ADD [BH], AX`?

Se modifica un total de **cero** palabras, ya que la operación no es válida. En la arquitectura `x86` las direcciones de memoria son de 16 bits, mientras que en este caso se está tratando de acceder a una con 8 bits (recordar que el registro `BH` es uno de los registros de 8 bits de `BX`).

f. **(I2 - I/2017)** ¿Cuántas llamadas recursivas a una función es posible hacer como máximo en un computador `x86` de 16 bits? Indique claramente sus supuestos.

Para obtener el número concreto, se utilizará el código más simple posible:

```
CALL f
f: CALL f
```

Asumiendo una microarquitectura Von Neumann, se tiene el almacenamiento inicial de los *opcodes* de las instrucciones y los literales asociados (en este caso, la dirección del label `f`).

Supuesto: Tanto los literales como los *opcodes* son de 16 bits (para simplificar el análisis).

Con el supuesto anterior, vemos que se hace uso de un total de 8 direcciones de memoria para el programa escrito. Esto es:

- Dos direcciones por cada *opcode* (registro de 16 bits almacenado en dos palabras de memoria de 8 bits). Se hace uso de un total de 4 direcciones de memoria en total.
- Dos direcciones por cada literal. Se hace uso de un total de 4 direcciones de memoria en total.

Esto conlleva a un total de $2^{16} - 8$ direcciones de almacenamiento disponibles. Ahora, por cada llamado recursivo, se almacenará en el *stack* la dirección de retorno. Al ser esta de 16 bits, hace uso de dos direcciones de memoria por salto. Finalmente, la cantidad de llamadas recursivas R será igual a:

$$R = \frac{2^{16} - 8}{2} = 2^{15} - 4$$

g. **(I2 - II/2016)** Describa un mecanismo para, en tiempo de ejecución, escribir el código de una subrutina y luego llamarla, utilizando el `Assembly x86` de 16 bits. Asuma que tiene disponible la especificación completa de la ISA.

Separamos el mecanismo en distintas fases:

- **Fase 1:** Almacenamos una dirección de memoria específica donde comenzaremos a escribir el código de la subrutina.
- **Fase 2:** Desde dicha dirección, escribimos el *opcode* correspondiente a cada instrucción dentro de la subrutina y sus literales asociados.
- **Fase 3:** Una vez que termina la escritura, utilizamos el comando `CALL` para saltar directamente a la primera instrucción escrita.

h. **(I2 - II/2014)** Describa una convención de llamada para `x86`, que sea más rápida que `stdcall` al momento de leer y escribir parámetros y valores de retorno. Contrapese las posibles ventajas y desventajas.

Una posible convención podría utilizar los registros de la arquitectura `x86` para almacenar los parámetros de entrada de las subrutinas. Si la cantidad de parámetros excede la cantidad de registros, se utiliza el *stack* para los restantes parámetros. Comparada con *stdcall*, esta convención es más rapida para leer los argumentos, pero también es más compleja, ya que no todos se ubican en el mismo lugar y habría que tener cuidado con el uso de los registros dentro de las subrutinas.

2. *a*) **(I2 - I/2017)** Para el siguiente programa escrito en `Assembly x86-16`, indique los valores de los registros `SP` y `BP` y del *stack* completo, al momento de ingresar al *label* `set`.

```
        MOV BL,3
        PUSH BX
        CALL func
        RET
        func: PUSH BP
              MOV BP,SP
              MOV BL,[BP + 4]
              CMP BL,0
              JE set
              MOV CL,BL
              DEC CL
              PUSH CX
              CALL func
              MOV BL,[BP + 4]
              MUL BL
              JMP end

        set: MOV AX ,1

        end: POP BP
             RET 2
```

Para poder ver el estado final de nuestro *stack* y los registros, se describirá la ejecución del programa paso a paso hasta ingresar al label `set`. Por simplicidad, se asumirá como la primera línea de código un registro `PC = 0x0000`.

1) (`MOV BL,3`): Se almacena el número 3 en el registro `BL` (bits menos significativos de `BX`). Entonces, tenemos que `BX = 0x0003`.
2) (`PUSH BX`): Antes de ver en detalle la ejecución de esta instrucción, es importante notar que la función `func` tiene un solo parámetro, que en este caso será recibido a través del registro `BX`. El tope del *stack* (`SP`) inicialmente corresponde al final de la memoria, *i.e.* `0xFFFE`[1]. Luego, al ejecutar esta instrucción se almacena en el *stack* el valor del registro. Al ocupar dos palabras de memoria, se tiene que `SP = 0xFFFE`[2].
3) (`CALL func`): Se llama a la subrutina '`func`', por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). En este caso, `PC + 1 = 0x0003` y `SP = 0xFFFC`.
4) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFFA`. Cabe destacar que inicialmente no se conoce el valor de `BP` por lo que, sin pérdida de generalidad, se asumirá igual a `0x0000`.
5) (`MOV BP,SP`): Se tiene que `BP = 0xFFFA`.

[1] Al trabajar con elementos de 2 bytes en el *stack*, se parte de este valor con valores "basura" almacenados en un comienzo para ser reemplazados posteriormente.

[2] No cambia, solo se reemplaza el valor en memoria. Por otra parte, es importante recordar que en *Assembly* `x86` el registro `SP` apunta a la última palabra ingresada (en el *Assembly* del computador básico el registro `SP` apunta a una posición más arriba).

6) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del *stack*, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0003`
7) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
8) (`JE set`): Este salto **no será ejecutado**, puesto que la *flag* `Z` no estará activa (`BL` es distinto de cero).
9) (`MOV CL,BL`): Se almacena en el registro `CL` el valor del registro `BL`. Entonces, `CX = 0x0003`.
10) (`DEC CL`): El registro `CL` decrece en una unidad. Por lo tanto, `CX = 0x0002`.
11) (`PUSH CX`): Se almacena en el *stack* el valor del registro `CX`. Se tiene ahora `SP = 0xFFF8`. Este corresponderá al parámetro de la primera llamada recursiva.
12) (`CALL func`): Se llama a la subrutina '`func`' (primera llamada recursiva), por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). Entonces, `PC + 1 = 0x000D` y `SP = 0xFFF6`.
13) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFF4`. Cabe destacar que ahora **sí se conoce** el valor de `BP`: `BP = 0xFFFA`.
14) (`MOV BP,SP`): Se tiene ahora que `BP = 0xFFF4`.
15) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del *stack*, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0002` (Notar que ahora se almacenó el valor disminuido en una unidad anteriormente).
16) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
17) (`JE set`): Este salto **no será ejecutado**, puesto que la *flag* `Z` no estará activa (`BL` es distinto de cero).
18) (`MOV CL,BL`): Se almacena en el registro `CL` el valor del registro `BL`. Entonces, `CX = 0x0002`.
19) (`DEC CL`): El registro `CL` decrece en una unidad. Por lo tanto, `CX = 0x0001`.
20) (`PUSH CX`): Se almacena en el *stack* el valor del registro `CX`. Se tiene ahora `SP = 0xFFF2`. Este corresponderá al parámetro de la segunda llamada recursiva.
21) (`CALL func`): Se llama a la subrutina '`func`' (segunda llamada recursiva), por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). Entonces, `PC + 1 = 0x000D` y `SP = 0xFFF0`.
22) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFEE`, mientras que el valor almacenado es `BP = 0xFFF4`.
23) (`MOV BP,SP`): Se tiene que `BP = 0xFFEE`.
24) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del *stack*, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0001` (Notar que ahora se almacenó el valor disminuido en una unidad anteriormente).
25) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
26) (`JE set`): Este salto **no será ejecutado**, puesto que la flag `Z` no estará activa (`BL` es distinto de cero).
27) (`MOV CL,BL`): Se almacena en el registro `CL` el valor del registro `BL`. Entonces, `CX = 0x0001`.

28) (`DEC CL`): El registro `CL` decrece en una unidad. Por lo tanto, `CX = 0x0000`.
29) (`PUSH CX`): Se almacena en el *stack* el valor del registro `CX`. Se tiene ahora `SP = 0xFFEC`.
30) (`CALL func`): Se llama a la subrutina '`func`' (tercera llamada recursiva), por lo que se almacena el valor `PC + 1` en el *stack* (dirección de retorno). Entonces, `PC + 1 = 0x000D` y `SP = 0xFFEA`.
31) (`PUSH BP`): Se guarda el registro `BP` en el *stack*. Entonces, `SP = 0xFFE8`, mientras que el valor almacenado es `BP = 0xFFEE`.
32) (`MOV BP,SP`): Se tiene que `BP = 0xFFE8`.
33) (`MOV BL,[BP + 4]`): Se almacena en el registro `BL` el valor de lo que está almacenado en la dirección a la que apunta `BP` desplazado en 4 unidades hacia abajo del *stack*, es decir, el parámetro recibido por la función. Entonces, `BX = 0x0000` (Notar que ahora se almacenó el valor disminuido en una unidad anteriormente).
34) (`CMP BL,0`): En el registro `Status` se almacenarán las *flags* de la operación `BL - 0`.
35) (`JE set`): Este salto **será ejecutado**, puesto que la `flag Z` estará activa (`BL` es igual a cero).

Finalmente, se tiene que `SP = 0xFFE8` y `BP = 0xFFE8`. A continuación, el *stack* final:

Variable	Dirección	Palabra
BP	**0xFFE8**	**0xEE**
BP	**0xFFE9**	**0xFF**
PC + 1	**0xFFEA**	**0x0D**
PC + 1	**0xFFEB**	**0x00**
CX	**0xFFEC**	**0x00**
CX	**0xFFED**	**0x00**
BP	**0xFFEE**	**0xF4**
BP	**0xFFEF**	**0xFF**
PC + 1	**0xFFF0**	**0x0D**
PC + 1	**0xFFF1**	**0x00**
CX	**0xFFF2**	**0x01**
CX	**0xFFF3**	**0x00**
BP	**0xFFF4**	**0xFA**
BP	**0xFFF5**	**0xFF**
PC + 1	**0xFFF6**	**0x0D**
PC + 1	**0xFFF7**	**0x00**
CX	**0xFFF8**	**0x02**
CX	**0xFFF9**	**0x00**
BP	**0xFFFA**	**0x00**
BP	**0xFFFB**	**0x00**
PC + 1	**0xFFFC**	**0x03**
PC + 1	**0xFFFD**	**0x00**
BX	**0xFFFE**	**0x03**
BX	**0xFFFF**	**0x00**

Cuadro 1: *Stack* resultante del programa.

Es importante mencionar qué es lo que hace este programa finalmente. Al cumplirse `BL = 0`, se guarda en `AX` el valor igual a 1 (*i.e.* `AX = 0x0001`). Luego, al retornar a la llamada anterior, con las instrucciones `MOV BL,[BP +4]` y `MUL BL` se tendrá que `AX = AL * BL`. Dado el almacenamiento del *stack*, en primer lugar se tendrá `AX = 1 * 1 = 1`. No obstante, al volver a retornar se ejecutará `AX = 1 * 2 = 2` y, finalmente, `AX = 2 * 3 = 6`, retornando por último a la instrucción `RET` para finalizar el programa. De aquí se desprende que `func` almacena en `AX` el factorial de `BX`.

b) **(Apuntes - Arquitectura x86)** El siguiente código en `Assembly x86` obtiene una potencia mediante subrutinas.

```
MOV BL,exp
MOV CL,base
PUSH BX ; Paso de parametros (de derecha a izquierda)
PUSH CX
CALL potencia ; potencia (base,exp)
MOV pow,AL ; Retorno viene en AX
RET
potencia: ; Subrutina para el calculo de la potencia
    PUSH BP
    MOV BP,SP ; Actualizamos BP con valor del SP
    MOV CL,[BP + 4] ; Recuperamos los dos parametros
    MOV BL,[BP + 6]
    MOV AX,1 ;AX = 1
start:
    CMP BL,0 ; if exp <= 0 goto endpotencia
    JLE endpotencia
    MUL CL ; AX = AL * base
    DEC BL ; exp --
    JMP start
endpotencia:
    POP BP
    RET 4 ; Retornar , desplazando el SP en 4 bytes
base db 2
exp db 2
pow db 0
```

I. Describa cómo se ejecuta este programa.
El paso a paso se realiza de la siguiente forma:

1) Se almacena el exponente en el registro `BL` (bits menos significativos de `BX`) y la base en el registro `CL` (bits menos significativos de `CX`). Entonces, tenemos que `BX = 0x0002` y `CX = 0x0002`.
2) Si definimos la función como *potencia*(*base*, *exponente*), el paso de parámetros al *stack* es como sigue:
   - El tope del *stack* (`SP`) corresponde al final de la memoria, *i.e.* `0xFFFE`.
   - Se guarda el exponente (`PUSH BX`). El tope se mantiene en `0xFFFE`.
   - Se guarda la base (`PUSH CX`). Ahora, el tope es `0xFFFC`.
   - Tenemos entonces que `SP = 0xFFFC`.
3) Se llama a la subrutina "potencia", por lo que se almacena `PC+1` en el *stack*: `SP = 0xFFFA`.
4) Al ejecutar la subrutina, se guarda el registro `BP` en el *stack*: `SP = 0xFFF8`.
5) Ahora, con `MOV BP,SP` se tiene que `BP = 0xFFF8`.
6) Recuperamos los parámetros en los registros correspondientes: `CL = Mem[0xFFFC] = 0x02, BL = Mem[0xFFFE] = 0x02`. Notemos que esto funciona debido a que

el orden en que se almacenan las palabras en memoria es *little endian* (si fuera *big endian*, lo correcto habría sido trabajar con los registros `BH,CH` desde el principio).

7) Usamos el registro `AX` (hasta ahora vacío) para almacenar el resultado final de la potencia. `AX = 0x0001`.
8) Comienza el código de "start". Al ver que `BL` es distinto de 0, se ejecuta `MUL CL`, *i.e.* `AX = AL * CL = 0x01 * 0x02 = 0x0002`. Entonces, tenemos ahora que `AL = 0x02`.
9) Decrece `BL`, `BL = 0x01` y por ende `BX = 0x0001`.
10) Volvemos al comienzo de "start". Al ver que `BL` es distinto de 0, se ejecuta `MUL CL`, *i.e.* `AX = AL*CL = 0x02 * 0x02 = 0x0004`. Entonces, tenemos ahora que `AL = 0x04`.
11) Decrece `BL`, `BL = 0x00` y por ende `BX = 0x0000`.
12) Volvemos al comienzo de "start". Al ver que `BL` es igual a cero, ejecutamos el código de "endpotencia", que realiza lo siguiente:
    - `BP = Mem[0xFFF8,0xFFF9] = 0x0000`. Ahora, `SP = 0xFFFA`.
    - `POP PC+1` (para el retorno). Entonces, `SP = 0xFFFC`.
    - Con `RET 4`, se tiene `ADD SP,4` $\rightarrow$ `SP = 0xFFFF`. El *stack pointer* recupera su posición inicial.
13) Terminada la subrutina, se almacena en "pow" lo almacenado en el registro `Al`. Finalmente, `pow = 0x04`.

Si bien aquí no pareciera haber mucha utilidad en almacenar `BP` en el *stack* al comienzo de la subrutina, se vuelve vital al llamar a otra subrutina dentro de la primera (que sucede, por ejemplo, en subrutinas recursivas).

II. ¿Cómo se modificaría este programa para hacer uso de variables locales? ¿Cómo cambia la dinámica desde el punto de vista de los registros `BP` y `SP`?
En los mismos apuntes se presenta una forma de utilizar variables locales:

```
MOV BL,exp
MOV CL,base
PUSH BX ; Paso de parmetros (de derecha a izquierda)
PUSH CX
CALL potencia ; potencia (base,exp)
MOV pow,AL ; Retorno viene en AX
RET
potencia: ; Subrutina para el calculo de la potencia
    PUSH BP
    MOV BP,SP ; Actualizamos BP con valor del SP
    SUB SP,2 ; Reservamos espacio para variable i
    MOV CL,[BP + 4] ; Recuperamos los dos parametros
    MOV BL,[BP + 6]
    MOV AX,1 ;AX = 1
    MOV [BP - 2],0 ; Variable i = 0
start:
    CMP [BP - 2],BL ; if i >= n goto endpotencia
    JLE endpotencia
    MUL CL ; AX = AL * base
    INC [BP - 2] ; i++
    JMP start
endpotencia:
    ADD SP,2 ; Eliminamos el espacio para la variable i
    POP BP
    RET 4 ; Retornar , desplazando el SP en 4 bytes
base db 2
exp db 2
pow db 0
```

Como ahora el *stack* crece con mis variables locales, puedo decrementar `BP` para acceder a ellas (que es equivalente a "subir" por el *stack*). Es importante notar que se modifica `SP` para que considere este nuevo tope (`SUB SP,2`), por lo que es válida la notación `BP-2`. Al final de la subrutina, antes de recuperar el valor de `BP`, es importante incrementar `SP` para que "olvide" las variables locales y que `POP BP` retorne efectivamente el valor del registro, no el valor de una variable local (además de posicionar correctamente al *stack pointer* finalizada la subrutina).

c) **(I2 - II/2014)** Implemente, usando el `Assembly x86` y la convención `stdcall`, un programa que calcule el máximo común divisor de dos enteros no negativos, donde ambos no pueden ser 0 simultáneamente, utilizando el algoritmo de Euclides, descrito a continuación:

$$\text{Para } a, b \geq 0,\ mcd(a,b) = \begin{cases} a & \text{, si } b = 0 \\ mcd(b, a \bmod b) & \text{, en cualquier otro caso.} \end{cases}$$

El siguiente programa cumple con el cálculo pedido:

```
        JMP    main
        a db    ?
        b db    ?
        res db 0
    main:
        MOV    AX, 0
        MOV    AL, b
        PUSH   AX
        MOV    AL, a
        PUSH   AX
        CALL   euclides
        MOV    res, AL
    euclides:
        PUSH   BP
        MOV    BP, SP
        MOV    BX, [BP-6]
        CMP    BL, 0
        JMP    setA
        MOV    AX, [BP-4]
        DIV    BL
        MOV    CX, 0
        MOV    CL, AH
        PUSH   CX
        PUSH   BX
        CALL   euclides
        JMP    return
    setA:
        MOV    AX, [BP-4]
    return:
        POP BP
        RET 4
```

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

IIC2343 – Arquitectura de Computadores

Ayudantía 6 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. **(I2 - II/2015)** Explique cómo funciona la transferencia de direcciones y datos desde/hacia los dispositivos *mapeados* a memoria.

   La pieza clave para el mecanismo de *mapeo* a memoria corresponde al ***address decoder***. La idea es que tanto la memoria como los dispositivos I/O ocupan el mismo bus de direcciones y el *address decoder* determina si la dirección corresponde a la memoria o a alguno de los dispositivos I/O. Si el caso es el último, entonces se accede al dispositivo correspondiente, desde el que se recibe un dato (o es enviado) desde el bus de direcciones.

   b. Describa las instrucciones de la `ISA` de un computador `x86` que permiten acceder a dispositivos mediante *port* I/O.

   Las instrucciones principales son dos:

   - `IN Reg,Port:` Se copia en el registro `Reg` el dato enviado por el dispositivo I/O asociado a la dirección `Port`.
   - `OUT Port,Reg:` Se escribe en el dispositivo I/O asociado a la dirección `Port` el contenido del registro `Reg`.

   En este caso, las direcciones `Port` son independientes de las direcciones de la memoria y su valor se encuentra entre 0 y 65535 (`0xFFFF`) para direcciones de 16 bits.

c. **(I2 - I/2017)** ¿Con qué tipo de dispositivo de I/O es preferible utilizar *mapeo* de memoria por sobre puertos?

Es preferible utilizarlo con dispositivos que utilicen pocas direcciones de memoria. Si utilizan menos, el número de direcciones a ocupar será menor dentro del espacio disponible en el bus de direccionamiento (evitando llegar a una eventual *memory barrier*). Por ejemplo, no convendría en absoluto mapear a memoria una tarjeta de video (¡imagina la cantidad de direcciones que habrían tomado los pixeles!).

d. ¿Por qué es mejor hacer uso de interrupciones en vez de *polling*? Mencione un ejemplo.

Las interrupciones permiten que la interacción entre la `CPU` y los dispositivos I/O se realice solo cuando uno de estos dos lo requiera. En *polling*, se revisa constantemente si algún dispositivo I/O requiere enviar datos (o que le envíen), hasta que alguno lo solicita y se realiza la solicitud. La revisión constante limita la capacidad de la `CPU` de realizar otras tareas, haciendo que el funcionamiento general sea ineficiente.

Un ejemplo sería el uso del teclado y el *mouse*. Si el computador estuviera revisando constantemente si el usuario escribió algo o no, o si movió el cursor, la eficiencia del computador sería deplorable (más aún, si revisara más dispositivos, como el lector de discos, ¡sería prácticamente imposible que corriera!). En cambio, si espera la interrupción por teclado o por cursor, no deja de realizar su conjunto de tareas mientras no se escriba ni se mueva el *mouse*.

e. **(I2 - I/2016)** ¿Cuál es la función del vector de interrupciones? ¿Cuál es su contenido?

El vector de interrupciones alberga las direcciones de las `ISR` de los dispositivos I/O que se encuentran registrados en el sistema. Entonces, al procesar una `IRQ`, el vector otorga la dirección de la `ISR` del dispositivo que hizo la interrupción, lo que permite finalmente ejecutarla.

f. **(I2 - II/2016)** Luego de recibir la señal `INTA`, ¿qué tarea(s) debe realizar un controlador de interrupciones?

El controlador de interrupciones, al recibir la `INTA`, busca en sus registros internos el dispositivo que produjo la `IRQ`. Luego, el ID de esta es enviada a la `CPU` a través del bus de datos. Finalmente, la `CPU` utiliza este ID para buscar la dirección de la `ISR` a ejecutar en el vector de interrupciones.

g. Detalle, paso a paso, cómo se manejaría una interrupción realizada por un dispositivo (llamémoslo IO_i) que se encuentra conectado a otro dispositivo (llamémoslo IO_j), donde la `ISR` de este último se encuentra almacenada en el vector de interrupciones del computador.

Enumeramos el paso a paso lo más detallado posible.

1) Asumimos que IO_j genera una `IRQ` a partir de la que realiza IO_i.
2) La `PIC` revisa su registro `IMR` para ver si la interrupción no está enmascarada. En este caso, marca un 1 en el bit correspondiente del *Interrupt Request Register*.
3) `PIC` escoge la interrupción de mayor prioridad (menor `IRQ`). En este caso, asumimos que es la enviada por IO_i. Se marca un 1 en el bit correspondiente del *In-Service Register*.
4) `PIC` envía interrupción (`INT`) a la `CPU`.
5) `CPU` termina de ejecutar la instrucción actual y guarda en el *stack* los *condition codes* (*flags*).

6) `CPU` revisa si el *flag* de las interrupciones está activo (`IF = 1`), en cuyo caso la atiende (asumimos que lo hace).
7) `CPU` deshabilita la atención de más interrupciones (`IF = 0`).
8) `CPU` envía `INTA` para saber quién interrumpió.
9) `PIC` revisa *In-Service Register* para saber el ID del `IRQ` que está siendo atendido (en este caso, el de IO_j) y lo envía mediante el bus de datos.
10) `CPU` usa el ID para buscar la dirección de la `ISR` en el vector de interrupciones.
11) `CPU` llama a la `ISR` asociada al dispositivo (`CALL Mem[id]`).
12) `ISR` respalda el estado actual de la `CPU`.
13) `ISR` ejecuta su código. Esta es la parte clave: Definimos la `ISR` como la búsqueda en el registro de estado del controlador de IO_j de la dirección de la `ISR` asociada a IO_i (que puede ser una subrutina, por ejemplo) y la llama.
14) Terminada la subrutina, el `ISR` original envía un comando `EOI` a la `PIC`, con la que se cambia a 0 el bit asociado en el *In-Service Register*, lo que indica que se terminó de atender la interrupción.
15) `ISR` devuelve el estado previo a la `CPU`.
16) `ISR` retorna.
17) `CPU` rehabilita la atención de interrupciones (`IF = 1`).
18) `CPU` recupera los *conditions codes* (*flags*) desde el *stack*.

h. Explique la diferencia entre las interrupciones realizadas por *hardware* y *software*, dando un ejemplo de cada una.

Las interrupciones por *hardware* pueden ser de dos tipos:

- Gatilladas por dispositivos I/O: Son las que realizan estos dispositivos para actualizar su estado o el de la `CPU`, generando una interrupción que ejecuta una `ISR` específica. Un ejemplo sería la actualización del cursor en el monitor de un computador a partir de la posición del *mouse*.
- Excepciones: Son las que se gatillan al encontrarse errores en la programación, generando una interrupción que ejecuta una `ISR` especializada (*exception handlers*). Un ejemplo sería dividir por 0 en una instrucción.

A diferencia de estas, las interrupciones por *software* son las que generan los mismos programas ejecutados, generalmente para ejecutar una `ISR` específica. La principal diferencia es que este no deshabilita la atención a otras interrupciones en *hardware*, por lo que hay que modificar la `IF` directamente mediante instrucciones: `CLI` (deshabilitar las interrupciones) y `STI` (habilitar las interrupciones). Un ejemplo concreto de esto es, por ejemplo, el manejo gráfico en una consola (tomaremos la *Super Nintendo*). El programa del juego *Super Mario World* genera una interrupción por *software* para modificar la tarjeta de video y desplegar una nueva imagen (por ejemplo, el movimiento de *Mario* y de un par de *Goombas*). Entonces, se ejecuta `INT dir` (siendo `dir` la dirección de memoria correspondiente a la tarjeta en el vector de interrupciones) y la `ISR` ejecutada se encarga de actualizar el contenido gráfico. Notar que aquí se trabaja directamente con la memoria de la `CPU`, por lo que convendría, en este caso, utilizar un controlador `DMA` para actualizar el estado (y así la `CPU` se encarga de ejecutar otras tareas).

i. **(I2 - II/2016)** Para los siguientes ejercicios, considere la siguiente tabla, que presenta el vector de interrupciones completo de un computador con `ISA x86` de 16 bits. El vector de interrupciones se encuentra almacenado a partir de la dirección de memoria `0x0000`:

IRQ	Dispositivo	Pos. en vector
IRQ0	*Timer* del sistema	00
IRQ1	Disco Duro	01
IRQ2	Interfaz `USB`	02
IRQ3	Interrupción *software*	03

Cuadro 1: Vector de interrupciones del computador.

I. ¿Cuantos dispositivos que generen solicitudes de interrupción pueden conectarse?
Es importante notar que uno de los dispositivos disponibles es una interfaz `USB`. Como se puede conectar un número arbitrario de dispositivos en ella, la respuesta es una cantidad infinita.

II. Dos dispositivos, teclado y *mouse*, están conectados a la interfaz USB. Describa un mecanismo para ejecutar la `ISR` correspondiente al *mouse*, cuando este genera una interrupción.
   - Se llama a la `ISR` de la controladora `USB`.
   - Esta `ISR` en particular puede tener como función leer un registro de estado específico de la interfaz para determinar la `ISR` real que se busca ejecutar (en este caso, la del *mouse*).
   - Se invoca dicha `ISR`. Un método puede ser, por ejemplo, implementarla como subrutina. También podría generar otra interrupción por *software* (*trap*).

III. ¿Que ocurriría en este computador si se ejecuta la instrucción `MOV [0],AX`?
Esto generaría un cambio en el puntero que apunta a la `ISR` del *timer* del sistema. Esto es **fatal**, ya que es utilizado para generar interrupciones que controlan la ejecución de los procesos dentro del computador y permiten realizar cambios de contexto.

IV. Proponga un esquema para permitir el acceso (lectura y escritura) controlado y centralizado al vector de interrupciones por parte de los programas, *i.e.*, el acceso solo puede realizarse a través de una interfaz entregada por el sistema operativo (o la `BIOS`).
Hint: El esquema puede incluir cambios a la arquitectura del computador.
Una opción, sería traspasar completamente el vector de interrupciones a un registro especializado (llamémoslo `I`), que solo puede ser escrito en modo supervisor/*kernel* (*i.e.* por el sistema operativo), y que es accedido/modificado por una instrucción especializada: `SINT id,ISR`, siendo `id` el ID del vector e `ISR` la dirección nueva de un `ISR` a almacenar. Se seguiría usando `INT` en la `ISA`, pero su ejecución debe ser modificada para ir acorde a lo especificado recientemente.

2. a. **(I3 - I/2013)** Un robot simple, conectado a un computador, es accesible mediante *mapeo* a memoria. Este robot se mueve en un espacio cuadrado infinito, en el cual cada grilla puede estar vacía o contener una muralla. El robot tiene comandos para ser prendido, apagado, avanzar 1 espacio hacia adelante, girar a la izquierda en 90^{o} y examinar lo que hay adelante. Cada vez que el robot se encuentra desocupado, *i.e.* ha sido recién iniciado o ha terminado una acción, genera una interrupción para informar que es posible darle un nuevo comando.

   I. Describa el mapa de memoria necesario para manejar el robot.

Dirección	Contenido/Función asociada
0	Dirección `ISR` de manejo del robot.
1	Registro de comandos del robot.
2	Registro de estado del robot.
3-...	Memoria de uso libre

Cuadro 2: Mapa de memoria definido para el robot.

Aquí es importante notar que las direcciones no siguen un orden preestablecido, se pudieron haber usado de otra forma siempre que estén las funciones asociadas correspondientes.

   II. Defina el formato de los datos que recibirá el robot como comandos y que entregará este para informar su estado.

Ubicación	Comando/Estado	Valor
Reg. Comandos	Encender	255
Reg. Comandos	Apagar	0
Reg. Comandos	Avanzar	1
Reg. Comandos	Girar Izq.	2
Reg. Comandos	Examinar	4
Reg. Estado	Recién encendido	0
Reg. Estado	Nada que informar	255
Reg. Estado	Espacio libre adelante	1
Reg. Estado	Muralla adelante	2

Cuadro 3: Valores para definir los comandos y estados del robot.

Nuevamente, aquí lo más importante es que se contengan los comandos y estados necesarios para poder cumplir con las funcionalidades del robot. Los valores utilizados no necesariamente deben coincidir con los de su respuesta.

III. Escriba en *Assembly* `x86` la `ISR` asociada al control del robot, siguiendo el siguiente comportamiento: el robot avanza hasta encontrar una muralla, en cuyo caso girará a la izquierda hasta encontrar un espacio vacío para avanzar, teniendo la precaución de que el robot no retroceda. Asuma que el espacio ha sido diseñado para que el robot no se quede pegado girando eternamente.

```
ISR_robot:
MOV BX, 0x0002       ;Revisamos el estado del robot.
CMP [BX], 0x00       ;Si es 0, inicializamos.
JE inicializar
CMP [BX], 0xFF       ;Si es 255, el robot examina.
JE examinar
CMP [BX], 0x01       ;Si es 1, antes de avanzar verificamos
JE check_retroceso   ;que no retroceda el robot.
JMP girar_izq        ;E.O.C., hay muralla. El robot gira.

inicializar:
MOV BX, 0x0003   ;Inicializamos variable en direccion de
MOV [BX], 0x00   ;memoria 3 para que el robot no retroceda.

examinar:
MOV BX, 0x0001   ;Se le da el comando al robot para
MOV [BX], 0x04   ;examinar y termina la subrutina.
JMP end_isr

check_retroceso:
MOV BX, 0x0003
CMP [BX], 0x02   ;Verificamos direccion de retroceso. Si es
JE girar_izq     ;2, el robot retrocede. Debe girar.

avanzar:
MOV [BX], 0x00   ;El robot puede avanzar. Se resetea la
MOV BX, 0x0001   ;variable de retroceso, se da el comando
MOV [BX], 0x01   ;de avanzar y termina la subrutina.
JMP end_isr

girar_izq:
MOV BX, 0x0003   ;Se actualiza la variable auxiliar de
ADD [BX], 0x01   ;retroceso para verificar en la siguiente
MOV BX, 0x0001   ;llamada que no retroceda, se da el
MOV [BX], 0x02   ;comando de girar y termina la subrutina.

end_isr:         ;Instruccion que retorna de la
IRET             ;interrupcion.
```

Lo importante de este código, además de su funcionamiento, es que se condiga con las tablas antes definidas.

b. **(I3 - II/2012)** Suponga que se tiene un dispositivo de adquisición de imágenes térmicas conectado a un computador que tiene una microarquitectura especializada para la adquisición de imágenes, pero con `ISA` compatible con `x86` de 16 bits. El computador tiene una memoria principal de 64 kilobytes, con el siguiente mapa de memoria para los primeros 4096 bytes:

Dirección	**Función asociada**
0-5	*Exception handlers.*
6	Registro de comandos de la cámara.
7	Registro de estado de la cámara.
8-14	Vectores de interrupciones de *hardware.*
15	Vector de interrupción de escritura en disco.
16	Vector de interrupción de adquisición de imagen.
17-31	Vectores de interrupciones de *software* de uso libre.
32-123	Memoria de uso libre.
124-1023	*Buffer* de adquisición de la cámara.
1024-4096	Espacio de memoria del disco.

Cuadro 4: Tabla que muestra el mapa de memoria del dispositivo.

Se desea escribir un programa que permita adquirir imágenes mediante la cámara y luego almacenarlas en disco. Las imágenes generadas por la cámara se encuentran en escala de grises de 8 bits, ordenadas por filas en una matriz cuadrada de 30x30.

i. Escriba una `ISR` para alguna interrupción de *software* disponible, que permita adquirir una imagen y luego escribirla en disco.

   La `ISR` de la cámara no recibe parámetros y retorna en su registro de estado información sobre la adquisición. Si la adquisición fue exitosa, el registro contendrá `0xFF` y la imagen se encontrará en el `buffer` de la cámara. En caso contrario, si la adquisición falló, el registro contendrá `0x00`. Durante la adquisición, el registro contendrá el valor `0xF0`.
   La `ISR` del disco utiliza internamente el controlador de `DMA`, por lo que necesita los siguientes parámetros en los siguientes registros:

   - La dirección de inicio del origen en el registro `AX`.
   - La dirección de inicio del destino en disco en el registro `BX`.
   - La cantidad de palabras a copiar en el registro `CX`.

   Puede utilizar la cantidad de parámetros que estime conveniente para su `ISR`, pero debe dejar explícitamente escrito qué significan y dónde se almacenan.

La solución utiliza la `IRQ 17` y asume que recibe como parámetro la dirección de inicio de escritura en disco en el registro `BX`.

```
ISR17:
    INT 16       ; Se hace la obtencion de la imagen.
while:
    MOV AX, [7] ; Se revisa el estado de adquisicion.
    CMP AX, F0h ; Si es 0xF0, se sigue revisando.
    JE while    ;
    ; Se llega a este punto completado el proceso.
    MOV AX, 124 ; Se guarda la direccion de inicio del buffer.
    MOV CX, 900 ; Se guarda la cantidad de palabras del
                ;buffer.
    INT 15       ; Se inicia la escritura en disco.
```

II. Escriba un programa que llame a la subrutina del ítem anterior para adquirir tres imágenes y almacenarlas de manera consecutiva en disco. Considere que la adquisición puede fallar y que se intentará esta un máximo de tres veces por imagen.
La solución utiliza la `IRQ 18` y asume que recibe como parámetro la dirección de inicio de escritura en disco en el registro `BX`.

```
ISR18:
    MOV DX, 0    ; Numero de intentos de adquisicion.
    MOV BX, 1024 ; Se guarda la direccion de inicio del
                 ; destino.
    MOV SI, 0    ; Numero de imagenes adquiridas.
while:
    CMP DX, 3    ; Tres intentos fallidos = se deja de
                 ; intentar.
    JE end
    ADD DX, 1    ; Aumenta el numero de intentos.
    INT 17       ; Se llama la subrutina para la
                 ; transferencia.
    MOV AX, [7]  ; Se revisa el estado de adquisicion.
    CMP AX, 00h  ; Si es 0x00, fallo y se debe intentar de
                 ; nuevo.
    JE while
end:             ; Se llega aqui por exito o 3 fallos
                 ; seguidos.
    MOV DX, 0    ; Reiniciamos el numero de intentos.
    ADD BX, 900  ; Cambia direccion de destino para otra
                 ; imagen.
    ADD SI, 1    ; Aumenta el numero de imagenes adquiridas.
    CMP SI, 3    ; Si obtenemos tres imagenes, terminamos.
    JLT while
```

Importante: Notar que se pudo haber inicializado el valor del registro `BX` en la primera subrutina. No obstante, hacerlo en la segunda permite entender de mejor forma cómo se va cambiando su valor para almacenar las tres imágenes en el disco.

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

IIC2343 – Arquitectura de Computadores

Ayudantía 7 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. Explique los principios de localidad espacial y localidad temporal con sus propias palabras, dando un ejemplo para cada caso.

   El principio de localidad espacial nos dice que si un dato es accedido, es probable que se soliciten datos cercanos al mismo. Un ejemplo de esto sería un elemento de un arreglo, con todos sus componentes contiguos en memoria.

   Por otra parte, el principio de localidad temporal nos dice que si un dato es accedido, es probable que se vuelva a solicitar el mismo en el corto plazo. Un ejemplo de esto sería una variable utilizada dentro de un ciclo.

   b. **(I3 - I/2017)** Sin considerar el precio, ¿por qué no tiene sentido usar una *caché* infinita?

   El objetivo principal de la memoria *caché* es disminuir el tiempo de acceso, aprovechando los principios de localidad espacial y temporal. Si se tuviera una *caché* infinita, a la larga se terminarían por copiar todos los bloques de la memoria principal a las líneas de la *caché*. Esto implicará un tiempo de búsqueda similar de una dirección dentro de la memoria, por lo que no se sacaría provecho de los principios antes mencionados y se tendría una ganancia en la eficiencia prácticamente nula.

   c. Suponga que tiene una memoria principal de 16 bytes y una memoria *caché* de 8 bytes y 4 líneas. Además, asuma que tiene un programa que accede, en este orden, a las direcciones de la memoria principal: $0, 1, 5, 7, 10, 13, 4, 6$.

   Obtenga el estado final de la memoria *caché* (en una tabla) y el *hit-rate* para cada una de las siguientes funciones de correspondencia:

a. *Directly mapped.*
b. *Fully associative.*
c. *2-way associative.*

Puede asumir una política de reemplazo `LRU`, en caso de necesitarla.

Antes de comenzar con el desarrollo, notaremos ciertos elementos que nos permitirán avanzar más rápido en cada ejercicio:

- Al ser la memoria de 16 bytes, tendremos 2^4 direcciones de datos posibles. Es decir, las direcciones son de 4 bits.
- Al ser la memoria *caché* de 8 bytes, y poseer 4 líneas, tendremos $\frac{8}{4} = 2^1$ palabras (bytes) por línea. Es decir, posicionamos cada palabra (*offset*) con un solo bit, y para obtener la línea correspondiente, al ser 2^2, necesitamos de 2 bits.

Para un mejor entendimiento de esta parte, usaremos colores para identificar cada elemento en una dirección:

- Posición
- Línea
- *Tag*
- Conjunto

a. *Directly mapped*: Cada bloque de la memoria principal tiene una línea asociada en la *caché*. Para obtener la posición, usaremos el bit menos significativo (pues con uno nos basta para posicionar). Por otra parte, para obtener la línea correspondiente, usamos los dos bits siguientes. Los bits restantes (en este caso, solo el bit más significativo) serán el *tag*. Entonces, veamos cada acceso:
   - 0 = 0000. Obtenemos un *miss* al tener un bit de validez = 0. Guardamos el *tag* 0 en la línea 00, almacenando los datos Mem[0] en la posición 0 y Mem[1] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   - 1 = 0001. Obtenemos un *hit* al encontrar el dato en la posición correspondiente.
   - 5 = 0101. Obtenemos un *miss* al tener un bit de validez = 0. Guardamos el *tag* 0 en la línea 10, almacenando los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   - 7 = 0111. Obtenemos un miss al tener un bit de validez = 0. Guardamos el *tag* 0 en la línea 11, almacenando los datos Mem[6] en la posición 0 y Mem[7] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   - 10 = 1010. Obtenemos un *miss* al tener un bit de validez = 0. Guardamos el *tag* 1 en la línea 01, almacenando los datos Mem[10] en la posición 0 y Mem[11] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1.
   - 13 = 1101. Obtenemos un *miss* al ver que en la línea 10 y en la posición 1 de ella, el *tag* difiere del nuestro (¡tenemos datos distintos!). Reescribimos el *tag* 1 en la línea 10 y los datos Mem[12] en la posición 0 y Mem[13] en la posición 1 de la línea.
   - 4 = 0100. Obtenemos un *miss* al ver que en la línea 10 y en la posición 1 de ella, el *tag* difiere del nuestro. Reescribimos el *tag* 0 en la línea 10 y los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea.
   - 6 = 0110. Obtenemos un *hit* al encontrar el dato en la posición correspondiente.

Finalmente, tenemos un *hit-rate* de $\frac{2}{8}$, y el estado de la *caché* de la siguiente forma:

Índice línea	Ubicación palabra	Bit validez	*Tag*	Dato
00	0	1	0	Mem[0]
	1			Mem[1]
01	0	1	1	Mem[10]
	1			Mem[11]
10	0	1	0	Mem[4]
	1			Mem[5]
11	0	1	0	Mem[6]
	1			Mem[7]

b. **Fully associative:** Cada bloque de la memoria principal puede ser asociado a cualquier línea de la *caché*. Para obtener la posición, usaremos el bit menos significativo (pues con uno nos basta para posicionar). Los bits restantes (en este caso, los tres que quedaron) serán el *tag*. Acá, además será necesario tener constancia del tiempo en que cada línea fue accedida por última vez, para poder utilizar la política de reemplazo `LRU`. Entonces, veamos cada acceso:

- 0 = 0000. Obtenemos un *miss* al no encontrar el dato en ninguna línea. Guardamos el *tag* 000 en la línea 00 (primer espacio disponible), almacenando los datos Mem[0] en la posición 0 y Mem[1] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 1.
- 1 = 0001. Obtenemos un *hit* al encontrar el dato en la línea 00. Actualizamos el tiempo de acceso a la línea a 2.
- 5 = 0101. Obtenemos un *miss* al no encontrar el dato en ninguna línea. Guardamos el *tag* 010 en la línea 01 (primer espacio disponible), almacenando los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 3.
- 7 = 0111. Obtenemos un *miss* al no encontrar el dato en ninguna línea. Guardamos el *tag* 011 en la línea 10 (primer espacio disponible), almacenando los datos Mem[6] en la posición 0 y Mem[7] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 4.
- 10 = 1010. Obtenemos un *miss* al no encontrar el dato en ninguna línea. Guardamos el *tag* 101 en la línea 11 (primer espacio disponible), almacenando los datos Mem[10] en la posición 0 y Mem[11] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 5.
- 13 = 1101. Obtenemos un *miss* al no encontrar el dato en ninguna línea. Al quedarnos sin espacio, por `LRU` reescribimos la línea accedida hace más tiempo. Reescribimos el *tag* 110 en la línea 00, almacenando los datos Mem[12] en la posición 0 y Mem[13] en la posición 1 de la línea. Finalmente, actualizamos el tiempo de acceso a la línea a 6.
- 4 = 0100. Obtenemos un *hit* al encontrar el dato en la línea 01. Actualizamos el tiempo de acceso a la línea a 7.
- 6 = 0110. Obtenemos un *hit* al encontrar el dato en la línea 10. Actualizamos el tiempo de acceso a la línea a 8.

Finalmente, tenemos un *hit-rate* de $\frac{3}{8}$, y el estado de la caché de la siguiente forma:

Índice línea	Ubicación palabra	Bit validez	*Tag*	Dato	Tiempo de acceso
00	0 1	1	110	Mem[12] Mem[13]	6
01	0 1	1	010	Mem[4] Mem[5]	7
10	0 1	1	011	Mem[6] Mem[7]	8
11	0 1	1	101	Mem[10] Mem[11]	5

c. **2-way associative:** Cada bloque de la memoria principal está asociado a conjuntos de dos líneas cada uno, pudiendo ser almacenado en cualquiera de estas. Para obtener la posición, usaremos el bit menos significativo (pues con uno nos basta para posicionar). Para obtener el conjunto, al poseer 4 líneas y 2 líneas por conjuntos, tendremos solo 2, por lo que nos bastará un bit para identificarlo (el que se encuentra después del de posición). Los bits restantes (en este caso, los dos que quedaron) serán el *tag*. Acá, además será necesario tener constancia del tiempo en que cada línea fue accedida por última vez, para poder utilizar la política de reemplazo `LRU`. Entonces, veamos cada acceso:

- $0 = 0000$. Obtenemos un *miss* al no encontrar el dato en ninguna línea del conjunto 0. Guardamos el *tag* 00 en la línea 00 del primer conjunto (primer espacio disponible), almacenando los datos Mem[0] en la posición 0 y Mem[1] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 1.
- $1 = 0001$. Obtenemos un *hit* al encontrar el dato en la línea 00 del conjunto 0. Actualizamos el tiempo de acceso a la línea a 2.
- $5 = 0101$. Obtenemos un *miss* al no encontrar el dato en ninguna línea del conjunto 0. Guardamos el *tag* 01 en la línea 01 del conjunto 0 (primer espacio disponible), almacenando los datos Mem[4] en la posición 0 y Mem[5] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 3.
- $7 = 0111$. Obtenemos un *miss* al no encontrar el dato en ninguna línea del conjunto 1. Guardamos el *tag* 01 en la línea 10 del conjunto 1 (primer espacio disponible), almacenando los datos Mem[6] en la posición 0 y Mem[7] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 4.
- $10 = 1010$. Obtenemos un *miss* al no encontrar el dato en ninguna línea del conjunto 1. Guardamos el *tag* 10 en la línea 11 del conjunto 1 (primer espacio disponible), almacenando los datos Mem[10] en la posición 0 y Mem[11] en la posición 1 de la línea. Finalmente, cambiamos el valor del bit de validez a 1, y el tiempo de acceso a la línea a 5.

- 13 = 1101. Obtenemos un *miss* al no encontrar el dato en ninguna línea del conjunto 0. Al quedarnos sin espacio, por `LRU` reescribimos la línea accedida hace más tiempo dentro del conjunto (ya que debemos mantener consistencia con el mapeo a conjuntos). Reescribimos el *tag* 11 en la línea 00, almacenando los datos Mem[12] en la posición 0 y Mem[13] en la posición 1 de la línea. Finalmente, actualizamos el tiempo de acceso a la línea a 6.
- 4 = 0100. Obtenemos un *hit* al encontrar el dato en la línea 01 del conjunto 0. Actualizamos el tiempo de acceso a la línea a 7.
- 6 = 0110. Obtenemos un *hit* al encontrar el dato en la línea 10 del conjunto 1. Actualizamos el tiempo de acceso a la línea a 8.

Finalmente, tenemos un *hit-rate* de $\frac{3}{8}$, y el estado de la *caché* de la siguiente forma:

Conjunto	Índice línea	Ubicación palabra	Bit validez	*Tag*	Dato	Tiempo de acceso
0	00	0 1	1	11	Mem[12] Mem[13]	6
	01	0 1	1	01	Mem[4] Mem[5]	7
1	10	0 1	1	01	Mem[6] Mem[7]	8
	11	0 1	1	10	Mem[10] Mem[11]	5

d. **(I3 - I/2016)** Comente sobre las ventajas de tener una memoria *caché split* en vez de una *unified* conectada a la memoria de datos del computador básico.

No existe ninguna ventaja si está conectada a la **memoria de datos**. La gracia de la *caché split* es poder tener separada la memoria de datos con la de instrucciones. Conectarla directamente a la memoria de datos no tiene una utilidad práctica, ya que utilizaríamos la mitad del espacio en instrucciones inexistentes.

e. **(I2 - I/2018)** Suponga que tiene una *caché* con *mapeo* directo. ¿Se demora más esta en sustituir una línea que una *caché fully associative*? ¿Por qué?

No, se demora menos porque no tiene que aplicar sustitución. El *mapeo* directo indica de inmediato cuál línea sustituir, mientras que *fully associative* requiere de un protocolo de sustitución, lo cual exige revisar cada entrada para buscar el primero en entrar (`FIFO`), el menos usado (`LFU`) o el usado hace más tiempo (`LRU`).

2. a. **(I3 - I/2016)** Un computador tiene una memoria *caché* de 16KB, con líneas de 32 bytes que almacenan 8 palabras, y un tiempo de acceso de 10ns. La memoria *caché* está conectada a la memoria principal mediante un bus capaz de transferir 8 bytes en 120ns. ¿Cuál es el *hit-rate* que debe tener la memoria *caché* para tener un tiempo de acceso promedio de 20ns?

   Nos guiamos por la siguiente fórmula:

$$TP = HR * HT + (1 - HR) * (HT + MP)$$

   Donde:

   - TP : Tiempo promedio.
   - HR : *Hit-rate.*
   - HT : *Hit-time.*
   - MP : *Miss penalty.*

   Ahora, agrupando términos, tenemos que:

$$TP = HT + (1 - HR) * MP$$

   Como el tiempo de acceso es 10ns, $HT = 10ns$, y como la transferencia del bus es de 120ns por 8 bytes, $MP = \frac{32}{8} * 120ns = 480ns$. Finalmente, como queremos que $TP = 20ns$:

$$20ns = 10ns + (1 - HR) * 480$$
$$HR = \frac{470}{480} \approx 0{,}98$$

   Nota: Recordar que el tiempo de acceso al tener un *miss* incluye el tiempo de acceso en la *caché*, ya que se busca el dato de todas formas, pero no se encuentra.

   b. **(I3 - I/2017)** Considere una memoria *caché fully-associative*, con *hit-time* igual a $16L^3 - 100L$ ns, donde L es la cantidad de líneas de la *caché*. Si el *hit-rate* de esta memoria es de 0.95, ¿cuál es la cantidad de líneas que genera el tiempo de acceso promedio mínimo?

   Se hará uso de la misma fórmula antes descrita:

$$TP(L) = HR * HT(L) + (1 - HR) * (HT(L) + MP)$$

   Hay que notar dos detalles importantes para la resolución:

   1) Al estar el *hit-time* en función de la cantidad de líneas en la *caché*, el tiempo promedio también lo está.
   2) El *miss-penalty* es constante, debido a que corresponde simplemente al tiempo que toma copiar un bloque de la memoria principal en una línea de la *caché*.

   Agrupando términos:

$$TP(L) = HT(L) + (1 - HR) * (MP)$$

   De esta forma, para encontrar el mínimo, derivamos e igualamos a cero:

$$TP'(L) = HT'(L) = 0$$

Los términos $(1 - HR)$ y MP desaparecen al ser constantes.
Reemplazando:

$$TP'(L) = 48 * L^2 - 100 = 0$$

Despejando:

$$L = \sqrt{\frac{100}{48}} = 1{,}443$$

Como necesitamos un número entero para la cantidad de líneas, vemos el valor de $HT(1)$ y $HT(2)$:

$$HT(1) = 16 * 1^3 - 100 * 1 = -84$$

$$HT(2) = 16 * 2^3 - 100 * 2 = -72$$

Aquí notamos algo importante: Para los valores de L más cercanos al mínimo, el *hit-time* es **negativo**, lo que no puede suceder en la práctica[1]. Como el tiempo va aumentando con el incremento en el número de líneas (según lo observado con los cálculos anteriores), nos quedamos con el mínimo valor L para el que $HT(L) \geq 0$:

$$HT(3) = 16 * 3^3 - 100 * 3 = 132$$

Por lo tanto, el número de líneas que minimiza el tiempo promedio es $L = 3$, con un *hit-time* promedio de 132 ns.

[1] Salvo que conozcan un mecanismo para que su computador haga cómputos en el pasado.

c. **(I3 - I/2015)** Un computador de 64 bits tiene una memoria *caché* de 32KB, con 1024 líneas de 32 palabras. ¿Cuánto espacio de esta *caché* es usado por información distinta de los datos?

Primero, notemos que esto depende del tipo de función de asociación que tengamos. Para poder ejemplificar bien, asumamos que tenemos una función *4-way associative*.

- Al tener 2^5 palabras por línea, necesitamos 5 bits para identificar una específica dentro de una línea en especial (*offset*).
- Al ser *4-way associative*, tendremos conjuntos de 4 líneas. Si tenemos 2^{10} líneas, entonces tendremos 2^8 conjuntos (es decir, necesitamos 8 bits para identificar el conjunto).
- Finalmente, usamos 13 bits para posicionar (5 bits *offset* + 8 bits conjunto), por lo que el resto lo utilizamos para el *tag*: 51 bits.

Lo que almacenaremos en la tabla serán finalmente los 51 bits del *tag* para cada línea (pues todas las palabras en la línea lo comparten), además de un bit de validez (para ver si el dato fue escrito efectivamente desde la memoria principal). Finalmente, tendremos un espacio total de 52 * 1024 bits = 6.5 KB.

Veamos qué habría pasado si hubiera sido *fully associative*: solo utilizaríamos los 5 bits de posicionamiento dentro de una línea (ya que todo bloque de memoria puede ser asignado a cualquier línea de la *caché*). Entonces, ahora nuestro *tag* tendrá 59 bits y, por ende, incluyendo el bit de validez ocupará un espacio de 60 * 1024 bits = 7.5 KB.

Puede realizar el mismo análisis para otros tipos de asociación (*directly mapped*, *2-way associative*, etc.).

d. **(I3 - I/2015)** Considere un computador con microarquitectura Von Neumann, donde la tasa de ciclos de *clock* por instrucción es igual a N, cuando todos los accesos a memoria producen *hits* en la *caché*. La memoria *caché* tiene *miss-rate* de 4 % y *miss-penalty* de $25 \times N$ ciclos de *clock*. Si en un programa de K instrucciones, el 50 % de estas realizan lectura de un dato en memoria, ¿cuántos ciclos de *clock* menos tomaría la ejecución del programa, si todas las instrucciones produjeran *hits* en la memoria *caché*?

Primero, veamos el mejor caso: Solo tenemos *hits*, por lo que la cantidad de ciclos de *clock* total que se tendrá será igual al número de instrucciones por el número de ciclos por instrucción, es decir:

$$Ciclos_{mejor\_caso} = N \times K$$

Ahora, tomemos el caso real: Tenemos un *miss-rate* de un 4 %. Esto significa que en un 4 % de nuestras instrucciones tendremos un *miss*. Ahora, cuando tenemos un *miss*, **también** se ejecuta la cantidad de ciclos de un *hit* (ya que se buscó, en primer lugar, el dato en la *caché*).

Entonces, hasta ahora nuestro número de ciclos se ve de la siguiente forma:

$$Ciclos_{real} = N * K + Ciclos_{miss}$$

Notemos que la cantidad de ciclos será de $25 \times N$ por cada *miss*. Como estamos trabajando con la arquitectura Von Neumann, tendremos **al menos** un acceso a memoria por instrucción (necesitamos obtener el *opcode* de la memoria). Entonces, tenemos en primera instancia $0{,}04 \times K \times 25 \times N$ ciclos añadidos. Luego, nos dicen que en la mitad de las instrucciones, además, se tienen lecturas de memoria, por lo que en la mitad de las instrucciones tendremos **dos accesos a memoria**. Por ende, debemos incluir una cantidad de ciclos equivalente a $\frac{1}{2} \times 0{,}04 \times K \times 25 \times N$. Entonces:

$$Ciclos_{miss} = 0,04 \times K \times 25 \times N + 0,02 \times K \times 25 \times N = 0,06 \times K \times 25 \times N = 1{,}5 \times N \times K$$

Finalmente, reemplazamos:

$$Ciclos_{real} = N \times K + 1{,}5 \times N \times K = 2{,}5 \times N \times K$$

Concluimos entonces que se tendrán $1{,}5 \times N \times K$ ciclos menos en el caso ideal.

3. a. **(I3 - I/2013)** El principio de localidad espacial explica en parte el buen funcionamiento de la memoria *caché*. Sin embargo, es posible no cumplir este principio, disminuyendo el rendimiento de la memoria. Describa un ejemplo específico de esto y explique por qué se produce.

Un ejemplo particular de esto es el caso de almacenar una matriz dentro de la memoria *caché*, guardando en cada línea una fila de esta, pero sin que quepan todas en el espacio disponible. Asumiendo lo anterior, si recorriéramos la matriz por columna, tendríamos el caso de que por cada acceso a un elemento en particular, copiaríamos la fila completa. Esto, ya que en el acceso anterior se agregó la fila según lo estipulado al principio **y no la columna que contiene al siguiente dato que busca ser accedido**. Entonces, tendremos un rendimiento de memoria mucho peor en comparación con el que se tendría al recorrer la matriz por filas.

b. **(I3 - I/2015)** ¿Cómo es el rendimiento de una memoria *caché*, si el patrón de accesos a memoria distribuye de manera uniforme sobre todas las posibles direcciones? Ejemplifique el o los posibles casos.

Antes de ejemplificar, notemos que una distribución uniforme implica que se accede a cada posición en memoria en la misma proporción, sin importar el orden en el que se haga. Sabiendo esto, podemos ver dos casos generales. Para ejemplificar mejor, asumiremos una memoria *caché* con líneas de 8 palabras:

- **Caso 1:** Se accede a los datos de memoria en forma secuencial: $0, 1, 2, 3, ..., N$. En este caso, al partir en el acceso 0, tendremos un *miss*, pero guardaremos en la *caché* los datos desde la posición 0 a la 7 en la primera línea, por lo que en los accesos $1, 2, ..., 7$ solo tendremos *hits*. Luego, tendremos lo mismo: Tratamos de acceder a la posición 8 con un *miss*, guardamos en la siguiente línea los datos desde la posición 8 a la 15, y en los accesos tendremos $9, 10, ..., 15$ solo tendremos *hits*. Esto se repetirá constantemente hasta el último dato, por lo que tendremos un *hit-rate* muy alto.
- **Caso 2:** Se accede a los datos de memoria de la siguiente forma: $0, 8, 16, ..., N-7, 1, 9, 17, ..., N-6, ..., N$. Al partir en el acceso 0, tendremos un *miss*, y guardamos en la *caché* los datos desde la posición 0 a la 7 en la primera línea. Luego, en el acceso 8, tendremos otro *miss*, almacenando los datos desde la posición 8 a la 15, y así sucesivamente. Si no contáramos con el espacio suficiente en la *caché* para toda la memoria accedida, y siguiéramos una política de reemplazo `LRU`, las primeras líneas utilizadas serían sobreescritas, y al llegar al acceso 1, tendríamos un *miss* nuevamente por la sobreescritura. Esto, finalmente, conlleva a un *hit-rate* prácticamente nulo.

Bajo estos ejemplos, llegamos a la conclusión de que no es posible catalogar el rendimiento dada la información del enunciado, ya que puede ser muy alto o muy bajo, dependiendo del caso.

c. **(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, justifíquela y, si es positiva, indique detalladamente un caso en que esto se de.

Sí, existe un problema similar, pero en menor grado. Al ser un bloque asociado a un **conjunto** de líneas, podría darse el caso en el que una secuencia de accesos genere que uno solo de estos conjuntos se llene rápidamente y sea necesario efectuar las políticas de reemplazo, a pesar de existir otros conjuntos con espacio disponible.

Un ejemplo de esto sería la secuencia $0, 4, 8, 0, 4, 8, 0, 4, 8, ...$ para una memoria principal de 16 bytes, una memoria *caché* de 8 bytes y 4 líneas, una política de reemplazo `LRU` y, por último, un esquema *2-way associative*. En este caso, los accesos a 0 y 4 harán que las dos líneas del conjunto **0** se ocupen y, luego, por `LRU` se irá reemplazando la primera línea ocupada del mismo conjunto ya que el acceso a 8 será *mapeado* al mismo conjunto de los dos anteriores. Una secuencia como la descrita generará reemplazos constantes en el conjunto **0**, dejando al conjunto **1** prácticamente vacío.

d. **(I3 - II/2014)** El algoritmo de reemplazo `MRU` (*Most Recently Used*), a diferencia de `LRU`, descarta primero los elementos que han sido ocupados más recientemente. ¿En qué casos podría ser útil el uso de este esquema?

   Un caso específico en el que puede ser útil, es en la iteración constante de un arreglo. Supongamos que el arreglo completo no cabe en la *caché*, así que una vez que se nos agote el espacio, reemplazamos la última posición iterada por la siguiente a acceder. Hacemos esto hasta llegar al final, y luego, al reiniciar, tendremos en memoria una porción considerable del mismo, dándonos un buen *hit-rate* (que concluirá al llegar a la posición donde se realizó la primera reescritura).

   Notar que este caso convendría siempre y cuando la memoria *caché* pueda contener más de la mitad del arreglo (en otro caso, el *miss rate* sería mayor).

e. **(I3 - II/2014)** Describa al menos dos posibles soluciones para el problema de consistencia de memoria que se genera al tener un esquema de escritura de *caché write-back*.

   - **Solución 1:** Tener un bit dentro de la memoria principal que indique si el dato fue modificado dentro de la *caché* o no (*dirty bit*). Entonces, al querer revisar dicha posición, si se tiene un *dirty bit* igual a 1, se puede realizar una interrupción que realice una sobreescritura en la memoria principal desde la *caché*, devolviéndole al bit su estado original igual a 0. De esta forma, la inconsistencia se arreglaría solo cuando fuera necesario.
   - **Solución 2:** En general, tenemos problemas cuando otros dispositivos tratan de acceder de forma directa a la memoria (`DMA`). Entonces, podemos hacer que la `DMA` interactúe de forma directa con la *caché*, por lo que siempre tendría datos consistentes (sin embargo, en caso de necesitar datos almacenados en la memoria principal, sería necesario aplicar políticas de reemplazo).

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

IIC2343 – Arquitectura de Computadores

Ayudantía 8 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. ¿Cuáles son las desventajas de implementar multiprogramación sin memoria virtual?

   Son, principalmente, tres:

   I. Al tener cada proceso un espacio de memoria asignado, el programador tendría que conocer de antemano las direcciones físicas asociadas a cada uno de estos, de forma que no se generen inconsistencias.

   II. No existiría ningún tipo de protección: Un proceso podría acceder a espacios de memoria física que no se le asignaron.

   III. Al repartir el espacio de memoria, cada proceso tendría un tamaño de memoria fijo. Esto haría que algunos procesos tuvieran mucho espacio de sobra, mientras que a otros les hiciera falta más para poder ejecutarse.

   b. **(I3 - I/2013)** ¿Cómo debería modificarse la arquitectura del computador básico para que tenga soporte para multiprogramación?

   Se debería añadir lo siguiente:

   I. **Bit de modo:** Este bit se añade al registro `Status`, y se encarga de indicar si se está en modo usuario o en modo supervisor (*kernel*). En este último es donde se le entrega el mando al sistema operativo, el que se encarga de realizar el cambio de contexto entre procesos.

   II. **Page Table Base Register (`PTBR`):** Registro especial que se encarga de almacenar la dirección inicial de la tabla de páginas asociada a un proceso específico. Solo puede ser modificado en modo supervisor.

III. **Memory Management Unit (`MMU`):** Esta pieza de *hardware* se encarga de realizar la traducción de una dirección virtual a una dirección física. Se hace indispensable para poder obtener los datos que requiere cada proceso.

IV. **Process Control Block (`PCB`):** Corresponde a un registro especial en el que se almacena el valor de los registros del proceso en ejecución, antes de que este le otorgue el control al sistema operativo.

c. **(I3 - I/2017)** ¿Tiene sentido usar memoria virtual, si la cantidad de memoria física es igual a la cantidad de memoria direccionable? Justifique su respuesta.

Sí, sigue teniendo sentido. La memoria virtual hace que se facilite la multiprogramación al evitar las desventajas mencionadas en **1.**, ya que:

1) Cada programa tiene la impresión de estar solo en el computador, por lo que el programador no se debe preocupar de las direcciones físicas asociadas a este.
2) Como cada programa solo puede acceder a su espacio de memoria virtual y no al de otro, se añade protección.
3) Como pueden direccionar a **todo** el espacio de memoria, no hay un límite de tamaño.

Por lo tanto, sigue cumpliendo su objetivo. No hay que olvidar que la razón de implementar memoria virtual es precisamente la ejecución de diversos procesos secuencialmente, más que el poder direccionar en un espacio mayor al de la memoria física disponible.

d. **(I3 - I/2016)** Considere un computador con un espacio direccionable virtual de 32 bits, espacio de direccionamiento físico de 30 bits y páginas de 8KB.

I. Describa la composición interna de una dirección virtual.
Al tener un tamaño de página de 8KB, se tiene un total de 2^3KB = 2^{13}B, es decir, se almacenan 2^{13} palabras. Para poder ubicar una palabra en la página, entonces, se necesitan 13 bits. Luego, como la dirección virtual consta de 32 bits, se tienen 32 - 13 = 19 bits para indicar el número de página (es decir, se puede tener un total de 2^{19} páginas).
Entonces, la composición interna de una dirección virtual es como sigue[1]:

10110100101101011010110101101110

Donde el segmento rojo[2] (porción más significativa) corresponde al número de página, y el naranjo (porción menos significativa) al *offset*.

II. ¿Cuál es el máximo número de entradas válidas que pueden existir en una tabla de páginas?
El máximo número de entradas válidas será el número máximo de marcos físicos posibles. Primero, al tener 13 bits de *offset*, se tiene un total de 30 - 13 = 17 bits para direccionar marcos, es decir, se puede tener un total de 2^{17} marcos físicos. Ahora, uno de estos marcos **debe** almacenar la tabla de páginas. Por lo que si se tiene un solo proceso, se puede tener un total de $2^{17} - 1$ entradas en tabla.

[1] Número totalmente arbitrario.
[2] Si usted es daltónico, ruego me disculpe.

2. a. **(I3 - II/2015)** ¿Cuál es la cantidad mínima de memoria física que debe tener un computador de 32 bits con un esquema de paginación simple?

   Debe tener suficiente espacio para:

   1) **Tabla de páginas:** Para poder tener la asociación entre una página y un marco físico.
   2) **Una página/marco físico:** Para poder hacer uso de la memoria física y *swapping* con el disco. Si no se tuviera este espacio disponible, pierde sentido el uso de la tabla de páginas ya que no habrían secciones de memoria por asignar.

   Notar que esta respuesta es independiente del número de bits.

   b. **(I3 - II/2016)** Indique bajo qué condiciones el uso de memoria virtual podría hacer más lenta la ejecución de los procesos en un computador.

   La ejecución se hace más lenta cuando se produce *swapping*. Estos se dan dos escenarios:

   I. Si no quedan marcos físicos disponibles para asignar, será necesario hacer un *swap out* para almacenar una página en el disco y un posterior *swap in* para guardar la nueva. De esta forma, queda un marco disponible para la página que lo necesita.
   II. Como consecuencia del punto anterior, si la página accedida se encuentra almacenada en el disco, será necesario hacer un *swap out* de una que se encuentre en la memoria física. Luego, se usa ese nuevo espacio disponible para realizar un *swap in* de la página que se buscaba acceder en un comienzo.

   En los dos casos se termina por acceder al disco, lo que implica un mayor tiempo de ejecución dado que implican una transferencia de datos entre ambas estructuras de memoria.

   c. **(I3 - II/2016)** Durante la mayoría de sus tiempo de ejecución, un proceso tiene el 51 % de sus páginas en el *swap file*. Otro proceso que utiliza la misma cantidad de memoria total, tiene el 49 % de sus páginas en el *swap file*, también durante la mayor parte de su tiempo de ejecución. Asumiendo que ambos se ejecutan durante la misma cantidad de tiempo, ¿cuál de los dos procesos generó más *page faults*?

   No es posible determinarlo. Por ejemplo, los accesos del primer proceso podrían ser solo a las páginas que se encuentran almacenadas en la memoria física y no en el disco, lo que no generaría ningún *page fault*. Por el contrario, el segundo proceso podría tener una cadena de accesos secuencial en lo que respecta al número de página, lo que haría que se produjeran constantemente *page faults* (se ocuparían todos los marcos físicos en un comienzo, y luego se harían *swap outs* de forma constante para los siguiente accesos). Por lo tanto, que un proceso posea más páginas en el *swap file* no determinará necesariamente que tenga una tasa mayor de *page faults*.

   d. **(I3 - I/2016)** En un computador con soporte para memoria virtual, ¿cómo se pueden implementar regiones de memoria protegidas y regiones de memoria compartidas?

   Para implementar regiones de memoria compartida, basta con permitir la asignación de un mismo marco a dos páginas diferentes. De esta forma, un proceso podría acceder a los datos de otro (pues comparten el espacio). Por otra parte, para implementar regiones de memoria protegidas, basta con prohibir el uso de un marco físico para la asignación de una página (por ejemplo, que exista un *stack* que contenga las direcciones de los marcos físicos que pueden ser asignados). Así, las regiones de memoria protegidas podrían ser accedidas solo por el sistema operativo (modo *kernel*).

e. **(I3 - I/2013)** La siguiente figura presenta el estado de la memoria principal de un computador con memoria virtual en un instante dado:

Dirección	Contenido
0	6
1	15
2	3
3	−2
4	−45
5	8
6	28
7	−2
8	9
9	−3
10	8
11	12

Tabla de págs. P1

12	0
13	−12
14	24
15	−
16	0
17	−12
18	−16
19	−
20	0
21	0
22	0
23	0

Tabla de págs. P2

24	5
25	−3
26	−3
27	2
28	0
29	1
30	2
31	3
⋮	⋮

En base a esto, asuma la siguiente situación:

- Tamaño de cada página y de cada tabla de páginas es de 4 palabras.
- Existen dos procesos en ejecución, P1 y P2.
- Las páginas no existentes (no asociadas a marcos) se denotan con -.
- En una tabla de páginas, el bit más significativo de cada palabra indica si la página está en memoria (0) o disco (1).
- Ambos procesos solicitan las siguientes direcciones virtuales: 0, 1, 4, 5, 8, 10, 12, 15.

Para cada proceso, transforme las direcciones virtuales en físicas. Si la transformación fue exitosa, indique el dato obtenido. En caso contrario, indique el tipo de *page fault* generado.

Antes de hacer el análisis, algunas cosas a notar:

I. Al tener 4 palabras por página, se necesitan 2 bits (2^2) para el *offset* dentro de una dirección.
II. Por simplicidad, se tomarán 5 bits para las direcciones virtuales (podrían ser más).
III. Notar que el segundo bloque corresponde a la tabla de páginas del proceso 1, y el séptimo bloque a la tabla de páginas del proceso 2.
IV. Desde la dirección física 28 en adelante, los datos almacenados van en orden creciente: 0,1,2,3,4,5,6,...
V. En el análisis, al hacer la traducción a dirección física, se tendrá que el segmento en negrita indicará el marco físico, y el segmento normal el *offset* dentro del mismo.

- **Proceso 1**
  - *Acceso 0:* Dirección 00000. Corresponde a la página 0 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a -45. Por complemento a 2, sabemos que el bit más significativo de esta palabra es igual a 1, por lo que se genera un ***page fault* por tener la página en el disco**.
  - *Acceso 1:* Dirección 00001. Corresponde a la página 0 con *offset* 1. Si revisamos la tabla de páginas, la entrada asociada es igual a -45. Por complemento a 2, sabemos que el bit más significativo de esta palabra es igual a 1, por lo que se genera un ***page fault* por tener la página en el disco**.
  - *Acceso 4:* Dirección 00100. Corresponde a la página 1 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a 8. Haciendo la traducción: **1000**00 = 32. Revisando la figura, vemos que el dato almacenado es 4.
  - *Acceso 5:* Dirección 00101. Corresponde a la página 1 con *offset* 1. Si revisamos la tabla de páginas, la entrada asociada es igual a 8. Haciendo la traducción: **1000**01 = 33. Revisando la figura, vemos que el dato almacenado es 5.
  - *Acceso 8:* Dirección 01000. Corresponde a la página 2 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a 28. Haciendo la traducción: **11100**00 = 112. Si revisamos la tabla de páginas, podemos deducir el valor: Valor = Dirección - 28 = 84.
  - *Acceso 10:* Dirección 01010. Corresponde a la página 2 con *offset* 2. Si revisamos la tabla de páginas, la entrada asociada es igual a 28. Haciendo la traducción: **11100**10 = 114. Si revisamos la tabla de páginas, podemos deducir el valor: Valor = Dirección - 28 = 86.
  - *Acceso 12:* Dirección 01100. Corresponde a la página 3 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a -2. Por complemento a 2, sabemos que el bit más significativo de esta palabra es igual a 1, por lo que se genera un ***page fault* por tener la página en el disco**.
  - *Acceso 15:* Dirección 01111. Corresponde a la página 3 con *offset* 3. Si revisamos la tabla de páginas, la entrada asociada es igual a -2. Por complemento a 2, sabemos que el bit más significativo de esta palabra es igual a 1, por lo que se genera un ***page fault* por tener la página en el disco**.

- **Proceso 2**
  - *Acceso 0:* Dirección 00000. Corresponde a la página 0 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a 0. Haciendo la traducción: **0**00 = 0. Revisando la figura, vemos que el dato almacenado es 6.
  - *Acceso 1:* Dirección 00001. Corresponde a la página 0 con *offset* 1. Si revisamos la tabla de páginas, la entrada asociada es igual a 0. Haciendo la traducción: **0**01 = 1. Revisando la figura, vemos que el dato almacenado es 15.
  - *Acceso 4:* Dirección 00100. Corresponde a la página 1 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a -12. Por complemento a 2, sabemos que el bit más significativo de esta palabra es igual a 1, por lo que se genera un ***page fault* por tener la página en el disco**.
  - *Acceso 5:* Dirección 00101. Corresponde a la página 1 con *offset* 1. Si revisamos la tabla de páginas, la entrada asociada es igual a -12. Por complemento a 2, sabemos que el bit más significativo de esta palabra es igual a 1, por lo que se genera un ***page fault* por tener la página en el disco**.
  - *Acceso 8:* Dirección 01000. Corresponde a la página 2 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a 24. Haciendo la traducción: **11000**00 = 96. Si revisamos la tabla de páginas, podemos deducir el valor: Valor = Dirección - 28 = 68.
  - *Acceso 10:* Dirección 01010. Corresponde a la página 2 con *offset* 2. Si revisamos la tabla de páginas, la entrada asociada es igual a 24. Haciendo la traducción: **11000**10 = 98. Si revisamos la tabla de páginas, podemos deducir el valor: Valor = Dirección - 28 = 70.
  - *Acceso 12:* Dirección 01100. Corresponde a la página 3 con *offset* 0. Si revisamos la tabla de páginas, la entrada asociada es igual a -. Por enunciado, sabemos que se genera un ***page fault* por marco físico no asignado**.
  - *Acceso 15:* Dirección 01111. Corresponde a la página 3 con *offset* 3. Si revisamos la tabla de páginas, la entrada asociada es igual a -. Por enunciado, sabemos que se genera un ***page fault* por marco físico no asignado**.

f. **(I3 - II/2011)** En un computador de 32 bits con 1 GB de RAM, se ejecuta un proceso que en cierto momento utiliza 1.5 GB de RAM. ¿Cómo es posible que ocurra esto sin que el proceso se caiga? Comente acerca del tiempo de ejecución de este proceso.

Esto se puede debido al uso de la memoria virtual. Al ser de 32 bits, el espacio direccionable del proceso corresponde a:

$$2^{32}\text{B} = 2^{2}\text{GB} = 4\text{GB}$$

Es decir, el proceso puede hacer uso de **hasta** 4 GB. Ahora, como en la RAM (nuestra memoria física) solo se dispone de 1 GB, lo que hace el proceso es realizar *swapping* constantemente para el espacio restante que utiliza. Esto evidentemente impacta la ejecución de forma negativa, debido al gasto proveniente de la transferencia de datos entre el disco y la memoria física. Por este motivo, el proceso tendrá un tiempo de ejecución mucho mayor al deseable.

g. **(I3 - II/2011)** Describa el peor caso posible, desde la óptica de la cantidad de accesos, que puede ocurrir cuando un programa intenta obtener un dato almacenado en memoria en un sistema con memoria virtual y caché.

El peor caso (*worst path*) sería la siguiente secuencia:

I. Se accede a la `TLB`, pero ninguna de sus entradas coincide con la página solicitada (`TLB` ***miss exception***).

II. El sistema operativo va a buscar la entrada a la tabla de páginas del proceso, pero la página posee la marca que indica que está en disco. Se genera un **page fault**, y será necesario realizar un *swap out* de una página que se encuentre en la memoria física, y un posterior *swap in* para almacenar la página deseada en el nuevo marco disponible. Luego, se deberá actualizar la tabla de páginas y la `TLB` con la nueva entrada.

III. Finalmente, se vuelve a repetir el acceso a la `TLB`, esta vez con un *hit*.

3. a. **(Examen - II/2014)** ¿Qué es un cambio de contexto y un `PCB`? ¿Qué relación existe entre ellos?

   El cambio de contexto corresponde al cambio del proceso que se encuentra en ejecución en la CPU, mientras que el `PCB` Corresponde a un registro especial en el que se almacena el valor de los registros del proceso en ejecución, además de otra información relevante (límites de memoria, estado, *flags*, etc.). La relación existente es que el cambio de contexto se realiza satisfactoriamente gracias al `PCB`, debido a que este último contiene toda la información necesaria que le permite al proceso retomar su ejecución desde su último estado. Es decir, permite respaldar los procesos para su posterior restauración.

   b. **(I3 - I/2012)** ¿Es posible reutilizar el contenido de la `TLB` cuando ocurre un cambio de contexto?

   No, ya que la `TLB` contendrá en sus entradas los valores de la tabla de páginas correspondientes al proceso que acaba de salir del control de la CPU. Por ello, es necesario borrar sus entradas para almacenar valores correspondientes a la tabla de páginas del nuevo proceso.

   c. **(I3 - I/2013)** Describa qué elementos de un sistema con soporte para multiprogramación (SO, CPU y Memoria) deben actualizarse al realizar un cambio de contexto.

   Los que deben actualizarse con los siguientes:

   1) `TLB`**:** Debe limpiarse por completo para almacenar la asociación entre páginas y *frames* del proceso que ejecutará posteriormente.
   2) `PTBR`**:** Debido a que se trabajará con la tabla de páginas de otro proceso (que, evidentemente, se encuentra almacenada en una dirección de memoria distinta).
   3) **Bit de modo:** Debido a que el sistema operativo es el que realiza el cambio de contexto, por lo que es necesario encontrarse en modo *kernel* para tener los permisos correspondientes.
   4) `PCB`**:** Para el proceso que sale de la ejecución, debido a que se quiere poder restaurar desde su último estado ejecutado en la CPU.

   d. **(I3 - I/2016)** Indique qué eventos generan un cambio de contexto en el esquema de *cooperative scheduling*.

   En este tipo de *scheduling*, solo hay dos formas de generar un cambio de contexto:

   I. Que un proceso entregue voluntariamente el control de la CPU al sistema operativo (***yield***).
   II. Una petición de interacción con un dispositivo I/O.

   e. **(I3 - II/2016)** ¿Por qué no es enmascarable la `IRQ0` en un computador `x86`?

   La `IRQ0` corresponde al *timer* del sistema. A través de este es que se pueden llevar a cabo los cambios de contextos entre procesos, dado que en el contexto de *pre-emptive scheduling* se hace uso del *timer* para determinar el momento en el que se debe **interrumpir** una ejecución para el cambio de contexto. Por lo tanto, no debe ser posible enmascararla ya que jugaría en contra de la dinámica del computador.

f. **(I3 - II/2015)** Explique cómo un proceso puede sufrir de inanición si se utiliza el esquema *fixed priority pre-emptive scheduling*. ¿Es posible que ocurra esto en el esquema *round-robin*?

El *fixed priority pre-emptive scheduling* se preocupa de ejecutar los procesos de mayor prioridad. De este modo, pueden haber procesos de baja prioridad que nunca son ejecutados, por el ingreso constante de otros de mayor importancia. De esta forma, los procesos menos importantes sufren de inanición. Esto no ocurre con *round robin*, ya que este tipo de *scheduling* se encarga de darle una cantidad de ciclos de ejecución constante a cada proceso en la cola, por lo que eventualmente todos tendrán su turno para ejecutar.

g. **(I3 - I/2016)** En un computador con soporte para memoria virtual, ¿el registro `PC` almacena direcciones virtuales o físicas? Justifique su respuesta considerando el efecto de los cambios de contexto y del *swapping*.

Si el `PC` solo almacenara direcciones físicas, se podrían generar problemas de consistencia. Por ejemplo, veamos el caso en el que se genere un cambio de contexto. Será necesario almacenar el registro del `PC` para poder continuar la ejecución del proceso una vez que vuelva a acceder control. Sin embargo, en el entre tanto, se puede generar un *swap out* de sus datos. De esta forma, una vez que vuelva a ser ejecutado, al hacer *swap in* para reingresar sus páginas, estas podrían quedar asociadas a un marco físico diferente al inicial. Entonces, al trabajar con direcciones físicas, el `PC` tendría una dirección que estaría asociada al mismo marco inicial, pero que contendría datos diferentes. Es por ello que debe almacenar direcciones virtuales, ya que estas siempre serán traducidas a la región correspondiente gracias a la `MMU`.

h. **(Examen - I/2018)** En un computador Von Neumann donde las instrucciones y los datos de un programa están en el mismo espacio de memoria, suponiendo que se tiene memoria virtual, ¿qué se podría hacer para que las páginas de datos nunca sean interpretadas como instrucciones por el computador?

Se puede agregar un bit `NX` a cada página, por ejemplo en la tabla de páginas, que indique si las direcciones contenidas en ella se pueden ejecutar o no. Si el bit `NX` está en 1, cualquier intento de ejecutar su código debiera ignorarse y/o activar un *trap*.

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

IIC2343 – Arquitectura de Computadores

Ayudantía 9 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. **(I3 - I/2018)** Al hacer un computador con *pipeline*, por ejemplo en el caso del computador básico, ¿qué componentes se agregan?

   Los elementos que se agregan son esencialmente cuatro:

   - Registros de separación entre cada fase.
   - Unidades de control de *hazard*.
   - Unidades de *forwarding*.
   - Unidad de salto.

   b. **(Examen - I/2013)** Indique qué capacidades/intrucciones gana y pierde el computador básico al agregar soporte para paralelismo a nivel de instrucción.

   Lo más importante que gana el computador básico, que se deduce del tema en sí, es el poder de ejecutar múltiples instrucciones en paralelo. Sin embargo:

   - Se pierden los llamados a subrutinas. Como el registro `SP` se elimina y no existe una conexión del `PC` a la memoria, las subrutinas no se pueden implementar.
   - No se pueden hacer instrucciones de salto que impliquen el uso de los bit de estado N, C, O (pues se eliminan). Notar que además, como se realiza la comparación $A - B$ y el salto en un mismo ciclo, la `ALU` se conecta directamente a un nuevo elemento: la unidad de salto (separando esa funcionalidad de la unidad de control). Por esto, se elimina además el registro *Status*, pues ya no es necesario.

- Ya no se pueden usar datos de la memoria directamente como parámetros en la `ALU`, ya que se elimina la conexión entre el Mux `B` y la salida de memoria. Además, tampoco se puede enviar directamente el resultado de la `ALU` a la memoria. Lo único permitido es la transferencia de memoria desde y hacia los registros `A` y `B`.

c. **(I3 - I/2018)** ¿Por qué se permite hacer *forwarding* en algunos casos en un *pipeline* en lugar de hacer *stalling* de la `CPU`?

   Debido a que es posible determinar que, en esos casos, se requiere el dato de forma rápida y es mejor permitir la ejecución en lugar de parar la `CPU` por uno o más ciclos, influyendo positivamente en la rapidez de ejecución de los programas dentro de esta arquitectura.

d. **(I3 - II/2016)** En caso que la única solución para solucionar un *hazard* de datos sea introducir una burbuja, ¿cuál es el momento más adecuado para hacerlo?

   El momento más adecuado para hacerlo es en la etapa `ID` (*Instruction Decode*), ya que se desea poder identificar el *hazard* lo más pronto posible para perder menos ciclos. La etapa `ID` es la más temprana en la que se puede detectar, por lo que al identificar el *hazard* se querrá pausar un ciclo de forma inmediata.

e. **(Examen - I/2013)** Un computador con microarquitectura avanzada posee un *pipeline* de 30 etapas. ¿Cuál es el elemento de *hardware*, sin contar los registros entre etapas, que tiene el mayor impacto (positivo y negativo) en el rendimiento? Justifique su respuesta.

   Al poseer una cantidad considerable de etapas, el elemento más crítico, a la larga, será la unidad predictora de saltos. Si se implementa mal (*i.e.* posee un bajo porcentaje de acierto en las predicciones), se podrían perder muchos ciclos a medida que se avanza en las etapas.

2. a. **(I3 - II/2016)** En promedio, ¿cuántos ciclos por salto pierde un computador con un *pipeline* de 12 etapas, si su unidad predictora acierta el 75 % de las veces? Asuma que los saltos se realizan en la penúltima etapa.

   Si los saltos se realizan en la etapa 11 (correspondiente a la penúltima etapa), entonces son 10 las instrucciones que se podrían perder por una mala predicción. Asumiendo un ciclo por instrucción, se tiene entonces que se pierden 10 ciclos por un salto mal realizado. Finalmente, si se falla en el 25 % de las veces, se pierde un total de $10 * 0{,}25 = 2{,}5$ ciclos por salto.

   b. **(I3 - I/2016)** Estime el tiempo de ejecución de un programa que toma N instrucciones en el computador básico con *pipeline*, en base a las siguientes condiciones:

   - El 50 % de las instrucciones realizan lecturas o escrituras en memoria.
   - El 10 % de las instrucciones son de salto y en el 25 % de estas, el salto finalmente se realiza.
   - En el 50 % de las ocasiones, el dato obtenido desde la memoria debe ser utilizado en la instrucción siguiente.

   Cualquier supuesto sobre la solución debe quedar claramente explicado.

   De partida, con N instrucciones, se tendrá un total de $N + 4$ ciclos si no se consideran las condiciones anteriores. Luego, se van añadiendo más iteraciones según los siguientes criterios:

   - Considerando el *pipeline* visto en clases, tenemos que se agregan 3 ciclos por cada salto realizado (correspondientes a los que se les hizo *flush*) si se asume una unidad predictora que siempre opta por no saltar. Como se tiene un 10 % de instrucciones de salto y un 25 % de estos se realiza, entonces se añade un total de $0{,}1 * 0{,}25 * 3 * N = 0{,}075N$ ciclos al total.
   - Como el 50 % de las instrucciones acceden a memoria y en el 50 % de estos casos el dato obtenido de memoria se usa en la siguiente instrucción, en $0{,}5 * 0{,}5 * N = 0{,}25N$ instrucciones se necesita hacer *stalling* (por lo que se pierde una cantidad de ciclos igual a la cantidad de veces que se realiza esta espera, pues solo se pierde un ciclo por burbuja insertada).

   Finalmente, tenemos que el tiempo de ejecución final es:

$$t \times ((N + 4) + (0{,}075N) + (0{,}25N)) = 4 \times t + 1{,}325N \times t$$

3. a. **(Examen - I/2018)** Considere la arquitectura del computador básico con *pipeline*. Una de las grandes desventajas que posee es el hecho de que si la unidad predictora de saltos se equivoca, se pierden tres ciclos, lo que impacta considerablemente el tiempo de ejecución de los programas. ¿Cómo modificaría esta arquitectura para poder perder un ciclo menos por predicción de salto? Puede hacer un diagrama o detallar su implementación.

   Bastaría con la adición de un sustractor que tenga de entradas los valores de las salidas de `Mux FwA` y `Mux FwB` dentro de la etapa `EX`, además de trasladar a esta sección la unidad de salto y la unidad de *hazard* de control. De esta forma, en la etapa `EX` la unidad de *hazard* de control puede determinar si es necesario hacer un *flush* o no, desechando dos ciclos en vez de tres. Notar que esto hace innecesario el uso de la *flag* `Z` de la `ALU`, ya que pierde su propósito.

b. **(Examen - I/2018)** Suponga que tiene una empresa que se encuentra desarrollando un dispositivo relativamente simple, que no ejecuta programas de más de 10 instrucciones. Uno de los desarrolladores sugiere que, en pos de la eficiencia, lo diseñen con un *pipeline* de 10 etapas para poder paralelizar las ejecuciones. ¿Es esta una decisión conveniente tomando en cuenta el tiempo de ejecución de los programas? Justifique su respuesta.

   No. Al existir el proceso de *filling* (llenado de las etapas del *pipeline*), recién en el décimo ciclo todas las instrucciones estarían ejecutándose en una etapa del *pipeline*, tomando un total de 19 ciclos de ejecución. Si se tuviera una arquitectura como la del computador simple, el programa tendría una ejecución de 10 ciclos, además de tener una menor limitante en lo que respecta a las instrucciones. En resumen, la decisión perjudicaría la *performance* del dispositivo. **Importante:** Se asume que la diferencia del tiempo de ejecución **de un ciclo** en el dispositivo con y sin *pipeline* no es significativa, validando el análisis anterior.

c. **(Examen - II/2015)** Un computador x86 monoprocesador posee un *pipeline* de 5 etapas que se ejecutan en el siguiente orden:

   a) *Fetch*: Obtiene desde la memoria el *opcode* de la instrucción a ejecutar.
   b) *Decode*: Decodifica el *opcode*, enviando las señales correspondientes a cada componente.
   c) *Read*: Lee desde registros y memoria los datos requeridos para ejecutar la operación.
   d) *Execute*: Ejecuta la operación aritmética/lógica de la instrucción usando la `ALU`.
   e) *Write*: Almacena en registros o memoria el resultado de la operación aritmética/lógica.

   Además de los mecanismos tradicionales para combatir *hazards* de datos y de control, el computador evita *hazards* estructurales de acceso a memoria entre las etapas *Fetch*, *Read* y *Write*, al utilizar una memoria RAM que permite realizar de manera simultánea tres solicitudes distintas. A pesar de esto, es posible generar un *hazard* estructural de ejecución entre las etapas *Read* y *Execute*, cuando se procesa una instrucción del tipo `MOV Reg1,[Reg2+offset]`. ¿Cómo es posible evitar este *hazard*?

   Se puede ver que el *hazard* se produce al necesitar utilizar la `ALU` dos veces dentro de la misma instrucción (una para poder almacenar en `Reg1` un dato, y otra para obtener `Reg2+offset`). Una solución **para este caso en particular** sería añadir un sumador en la etapa *Read*. De esta forma, al buscar datos en memoria, `Reg2` y `offset` podrían pasar por el sumador y el resultado ser utilizado para obtener el dato correcto a almacenar en `Reg1`.

4. **(I3 - I/2016)** Determine el número de ciclos que se demora el siguiente código, detallando en un diagrama los estados del *pipeline* por instrucción. El *pipeline* tiene *forwarding* entre todas sus etapas, el manejo de *stalling* es por software (instrucción `NOP`) y predicción de salto asumiendo que no ocurre. Indique en el diagrama cuando ocurre *forwarding*, *stalling* y *flushing*.

```
DATA:
    n 1
    index 1
    prev1 0
    prev2 1
    res 0
CODE:
    main:
        MOV A,(n)
        MOV B,(index)
        JEQ end
        ADD B,1
        MOV (index),B
        JMP main
    end:
        MOV A,(prev1)
        MOV B,(prev2)
        ADD A,B
        MOV (res),A
```

Si el *forwarding* se marca con flechas azules, el *stalling* con instrucciones `NOP` y el *flushing* con líneas rojas que marcan el código que no se ejecutará, se tiene el siguiente diagrama:

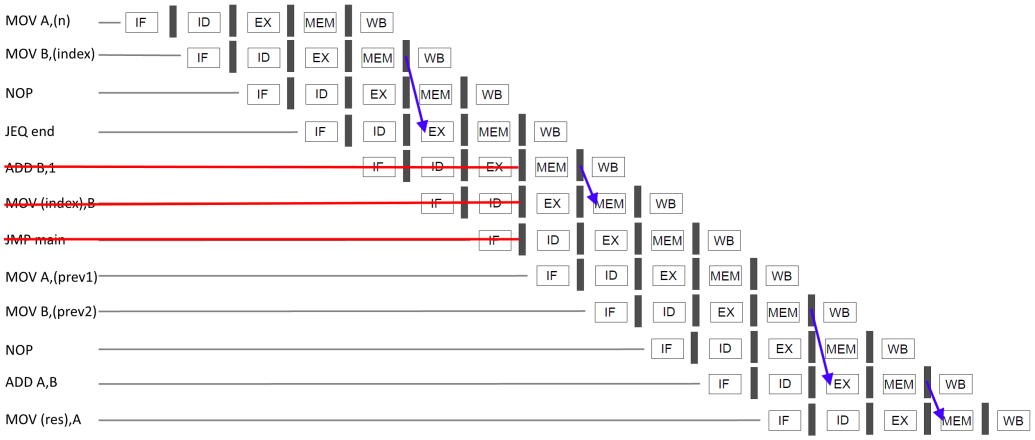


Figura 1: Diagrama final de la ejecución del programa, con un total de 16 ciclos.

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

IIC2343 – Arquitectura de Computadores

Ayudantía 10 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. ¿Qué es la taxonomía de Flynn? ¿Existe alguna categoría de esta taxonomía que sea difícil de ejemplificar en la práctica?

   La taxonomía de Flynn consiste en la categorización de tipos de arquitectura según su manejo de datos (único dato o múltiples datos) e instrucciones (única instrucción o múltiples instrucciones).

   Una categoría que es difícil de ejemplificar dado su poco uso práctico es la de las arquitecturas `MISD` (*Multiple Instruction, Single Data*. Es común aplicar un conjunto de instrucciones a un conjunto de datos, o bien aplicar una misma instrucción a muchos datos distintos, pero no es normal aplicar un conjunto de instrucciones a un único dato. En el siguiente enlace hay más información al respecto.

   b. **(Examen - II/2014)** Describa los tipos de paralelismo `SIMD` y `SISD`. ¿Qué tipo de operaciones son las que más beneficio sacan del paralelismo `SIMD`? ¿Es posible tener simultáneamente paralelismo del tipo `SISD` y `SIMD`?

   El paralelismo `SIMD` (*Single Instruction, Multiple Data*) ocurre cuando la misma instrucción/programa debe ser aplicado a datos distintos. Las operaciones vectoriales y matriciales son las que mayor provecho sacan de esto. `SISD` (*Single Instruction, Single Data*) ocurre cuando se realiza la ejecución paralela de un programa sobre un mismo dato. Generalmente este se logra mediante *pipelining*. Operaciones largas que pueden ser divididas en sub-operaciones son las que mayor ventaja obtienen de `SISD`. Para que ambos esquemas, `SISD` y `SIMD`, puedan ejecutarse simultáneamente, basta que las unidades de ejecución del *pipeline* sean capaces de aplicar la misma operación a múltiples datos distintos.

c. **(Examen - II/2015)** ¿En qué se diferencian los tipos de paralelismo `SIMD` y `SIMT`? Complemente las diferencias con ejemplos para cada uno de los casos.

`SIMD` (*Single Instruction Multiple Data*) consiste en utilizar un mismo programa (o función) para una serie de de datos. Un ejemplo de esto son las instrucciones `SSE` de la ISA `x86`. `SIMT` (*Single Instruction Multiple Threads*) consiste en utilizar un mismo programa sobre múltiples datos, pero utilizando múltiples *threads*. Un ejemplo de esto son las GPUs.

d. ¿Qué es una GPU? ¿Es siempre más conveniente que una CPU?

Una GPU es un dispositivo I/O con una arquitectura del tipo `SIMT`, utilizada para ejecutar la misma operación sobre múltiples datos con dependencia mínima. Por ejemplo, el procesamiento aplicado a cada pixel en una imagen sería una tarea en la que este dispositivo serviría bastante. Ahora, **no siempre** es más conveniente su uso con respecto a una CPU. Esta última es más útil para el paralelismo de tareas y usos de propósito general en comparación a la GPU.

2. a. Dentro del contexto de múltiples procesadores, existen dos formas de que estos tengan acceso a una misma fuente de memoria: `UMA` (*Uniform Memory Access*) y `NUMA` (`Non-Uniform Memory Access`). ¿En qué consiste cada una de estas? Mencione, además, una ventaja y desventaja para cada una.

   - `UMA:` Consiste en la arquitectura de memoria compartida en la que todos los procesadores acceden a la memoria de forma uniforme. Existen varios tipos, pero la estudiada en el curso es la `SMP` (*Symmetric Multiprocessor System*), donde todos los procesadores interactúan con la memoria a partir de un único bus de datos. Su ventaja es que el tiempo de acceso a una sección de memoria es independiente del procesador que trate de acceder a ella, haciéndolo útil en aplicaciones de propósito general y tiempo compartido para múltiples usuarios. Su desventaja, no obstante, es que es costoso de implementar (tamaño del bus de datos considerando la cantidad de procesadores), además de tener una escalabilidad restringida debido a la saturación del bus compartido en el caso de muchos procesadores.
   - `NUMA:` Consiste en la arquitectura de memoria compartida en la que esta se distribuye a partir de unidades de memoria locales a las que pueden acceder los distintos procesadores. Su ventaja radica en que es menos costoso de implementar que `UMA`, además de tener una escalabilidad mucho menos restringida al poder extenderse fácilmente (no dependen de un bus de acceso particular para todas las memorias locales disponibles). La desventaja es que los procesadores que busquen acceder a unidades de memoria local remotas tardarán más en poder concretar el acceso, debido a la red interconectada que se debe atravesar para la transferencia de datos.

b. **(I3 - I/2018)** Mencione una ventaja y una desventaja de actualizar las líneas de *caché* desactualizadas en vez de invalidarlas.

Ventaja: Permite que el computador aproveche al máximo la ventaja de tener datos en la *caché*.
Desventaja: Es muchísimo más complejo implementar un protocolo de coherencia que cubra este caso en lugar de simplemente invalidar.

c. Mencione y explique cada estado del protocolo *write-back* `MESI`.

El protocolo *write-back* `MESI` busca mejorar el desempeño de la consistencia de *caché* evitando actualizar en cada solicitud de escritura la memoria principal. Para ello, maneja un total de cuatro estados para cada línea de la caché:

- ***M**odified*: Indica que la línea de la *caché* corresponde a un bloque de la memoria principal que ha sido modificado (siendo el procesador el que lo modificó desde su *caché*).
- ***E**xclusive*: Indica que dicha línea de la *caché* corresponde a un bloque de la memoria principal que solo ha sido accedido por el procesador correspondiente, haciéndola **exclusiva**.
- ***S**hared*: Indica que dicha línea de la *caché* corresponde a un bloque de la memoria principal que ha sido accedido por más de un procesador, haciéndola **compartida**.
- ***I**nvalid*: Indica que dicha línea de la *caché* es **inválida** (es decir, está disponible para su uso pues lo contenido en ella no tiene validez alguna, ya sea por ser recientemente inicializada o por no estar actualizada).

Puntos importantes a considerar a partir de los estados anteriores:

- La primera vez que se accede a un bloque de la memoria principal, se lleva a la *caché* del procesador que solicita el acceso y se marca con estado `E`.
- Si el mismo procesador hace lecturas del mismo bloque, lo hace desde la línea de la *caché* y no actualiza su estado, no se hace uso del bus.
- Si otro procesador lleva dicho bloque a su *caché*, se da aviso de la existencia de una copia y ambos marcan la línea correspondiente con estado `S`.
- Si alguno de estos procesadores hace una lectura del mismo bloque, nuevamente se hace desde la línea de la *caché* y el estado no se modifica, no se hace uso del bus.
- Si un procesador quiere modificar un bloque de memoria que está contenido en una línea de su *caché* y tiene estado `E`, lo hace de forma directa en esta y pasa su estado a `M`, no hace uso del bus.
- Si un procesador quiere modificar un bloque de memoria que está contenido en una línea de su *caché* y tiene estado `S`, lo hace de forma directa en esta, pasa su estado a `M`, y hace uso del bus para enviar una señal de invalidación al resto de los procesadores. Los que contengan dicha línea, la pasarán a estado `I`.
- Si ahora otro procesador hace una lectura del bloque de memoria que fue actualizado y que se encuentra en una línea en estado `M` en el procesador que lo modificó, entonces este último hará una solicitud de espera, en la que el procesador que desea leer lo hace una vez que el bloque de memoria principal ha sido actualizado.
- Si este mismo procesador hubiera hecho una escritura sobre dicho bloque, entonces el procesador que posee la línea en estado `M` hará una solicitud de espera para modificar el bloque en la memoria principal y, posteriormente, marca su línea en estado `I` ya que ya no será consistente. El procesador que hace la escritura, por otra parte, no necesariamente guardará la línea en su *caché*. Al igual que en el protocolo *write through*, dependerá de la variante. En la versión estudiada en el curso no se hace, pero sí en caso de emplear una polícita *write-allocate*.

d. **(Examen - I/2018)** El protocolo `MOESI` corresponde a una variante del protocolo `MESI` estudiado en clases. Este posee el mismo comportamiento, salvo por la existencia de un nuevo estado `O` ("*Owned*"). Este es utilizado para indicar que uno y solo uno de los controladores de *caché* tiene el permiso para modificar el valor de una línea de la *caché* asociada a un bloque de la memoria y compartirlo con el resto de las *cachés* que la posean. Esto permite que, en caso de que se hagan lecturas de un recurso compartido, la transferencia de datos se haga entre *cachés* y, además, no es necesario que se escriban los cambios directamente en la memoria principal, solo cuando es estrictamente necesario.

   I. ¿Cómo se podría asegurar que sea una sola *caché* la que posea el estado `O` para un recurso compartido? Comente considerando el procedimiento del protocolo `MESI` para recursos compartidos (en particular, para los estados `M`, `S` e `I`).
   Cuando un recurso se comparte por primera vez se asigna el estado `E` para indicar que es exclusivo. Luego, cuando otro procesador solicita su lectura, tanto para dicho procesador como para el primero el estado se convierte en `S`, pues está compartido. Posteriormente, si una de las dos *caché* realiza una modificación, la *caché* que realiza la modificación cambia la línea a estado `M` y la otra invalida su copia. En el protocolo `MOESI` se puede establecer, en cambio, que la *caché* que escribe cambie a estado `O` en vez de `M` para indicar que será el encargado predeterminado para compartir el recurso entre las *cachés*, siempre que no existan solicitudes de escritura que hagan obsoleta su copia, mientras que la segunda mantiene su estado `S`, con la salvedad de que se le transfiere el nuevo valor de la línea. Naturalmente, el resto de los procesadores que soliciten la lectura del recurso para almacenarlo en sus *cachés* mantendrán un estado `S`, efectuando la transferencia desde la *caché* con la línea en estado `O`.

   II. Suponga que un par de *cachés* comparten una línea con estados `O` y `S`, respectivamente. Si la segunda *caché*, en estado `S`, solicita realizar cambios en la línea compartida, ¿qué protocolo se puede seguir para mantener la consistencia?
   Se puede manejar de dos formas, pero en ambas la *caché* que solicita la escritura pasa a estado `O`. En la primera forma, la *caché* que estaba encargada de la línea compartida puede simplemente invalidar su copia para no causar problemas de inconsistencia. En la segunda, en cambio, simplemente puede cambiar su estado a `S` y recibir a través de una transferencia el nuevo valor.

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

IIC2343 – Arquitectura de Computadores

Ayudantía 11 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. **(Examen - I/2018)**

   a. A continuación, se presenta un diagrama de una memoria `RAM` para el computador básico:

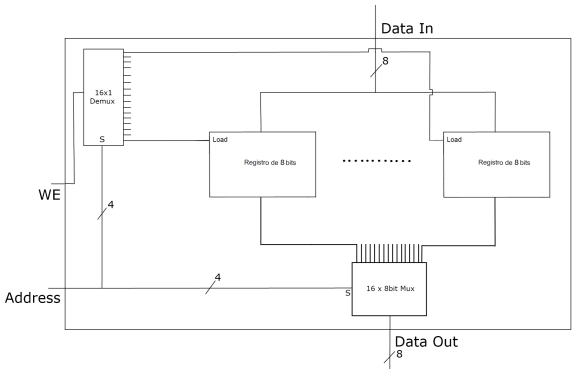


Figura 1: Representación de la memoria `RAM`.

A partir de este diagrama, ¿cómo modificaría esta componente para poder habilitar dos accesos (tanto para lectura como escritura) de forma simultánea? Puede hacer un diagrama o explicar en detalle su implementación.

Básicamente, doblando los siguientes buses de datos: *Data In*, `WE`, `Address` y `Data Out`. Esto puede ser haciendo que estos buses sean del doble de tamaño (dividiéndolos dentro del componente del diagrama) o bien que sean dos buses distintos. También se necesitaría una unidad adicional de *Demux* y una de *Mux*. En estos **no pueden** aumentar sus dimensiones para lograr lo pedido, dado que de todas formas solo pueden seleccionar un registro con cada componente. Por último, el bus de salida de cada registro debe tener dos cables, uno a cada *Mux* de salida.

b. Haciendo uso del componente anterior, modifique el computador básico para que pueda implementar las siguientes instrucciones:

   - `DREAD (dir1),(dir2)`: Se almacena la palabra ubicada en `dir1` dentro del registro `A` y la palabra ubicada en `dir2` dentro del registro `B`.
   - `DWRITE (dir1),(dir2)`: Se almacena en la dirección `dir1` el valor contenido en el registro `A` y en la dirección `dir2` el valor contenido en el registro `B`.
   - `RW (dir1),(dir2)`: Se almacena la palabra ubicada en `dir1` dentro del registro `A` y en la dirección `dir2` el valor contenido en el registro `B`.
   - `WR (dir1),(dir2)`: Se almacena en la dirección `dir1` el valor contenido en el registro `A` y se almacena la palabra ubicada en `dir2` dentro del registro `B`.

   Si su implementación genera algún cambio con respecto al funcionamiento del computador básico, debe indicarlo y explicar qué se debe hacer para que no cambie la ejecución del resto de las instrucciones. Además, para cada instrucción, debe indicar las señales a activar (sean del diagrama original o las añadidas por usted) para su ejecución correcta. Puede realizar un diagrama o explicar en detalle su implementación.
   Nota: Puede asumir que no se ejecutarán programas donde `dir1 = dir2`.

   Lo más importante para que puedan responder esta pregunta es que den cuenta de la necesidad de modificar la `ROM` para poder obtener más de un literal. Existen muchas formas válidas de hacer esto:

   - Pueden ampliar la memoria `ROM` y que cada instrucción tenga 2 literales.
   - Pueden segmentar en 2 los bits utilizados originalmente para el literal y separarlos en 2 buses (aunque esto conlleva a un menor poder de representación, algo que deberían comentar).
   - Pueden hacer lo mismo del punto 1 pero sin ampliar la `ROM`, indicando que el máximo de instrucciones por programa disminuye.

   En cualquiera de estos casos, no obstante, se espera que mencionen el impacto de que ahora en todas las instrucciones habrán dos literales. Basta con que digan que el segundo es igual a cero para el resto de las instrucciones donde no se ocupa. A partir de estos dos literales pueden comenzar a trabajar en el diagrama. Por ejemplo, pueden incluir ahora dos *Mux Address*, uno para cada dirección de ingreso de la `RAM` (lo que podría ampliar, además, las funcionalidades del computador). No obstante, también pueden mantener el mismo *Mux Address* y hacer que el literal `dir2` vaya directamente a la segunda entrada de dirección de la `RAM`. En ambos casos, deben mencionar cómo cambian los buses de selección.
   En lo que respecta a las señales de control de la `RAM`, va a depender de cómo interpre-

ten la pregunta anterior (la hayan respondido o no). Si usan dos buses, deben indicar el impacto que genera esto para las instrucciones de lectura normales (`MOV (dir),Reg`; `MOV Reg,(dir)`), donde ahora se debe deshabilitar la segunda señal `WE` añadida. Si usan un único bus del doble de tamaño, deben indicar un estándar de forma que se siga modificando un solo registro o una sola palabra en el ejemplo antes mencionado.
A partir de las modificaciones realizadas, se espera que comenten además que el bus de señales proveniente de la unidad de control aumenta (salvo que su implementación funcione y no incluya la adición de nuevas señales de selección o habilitación).
Lo último que cabe mencionar como modificación es cómo ingresan los valores de los registros a la memoria de datos. Originalmente, esto se hacía como el resultado de la `ALU` a través del *Mux DataIn*, no obstante, no se puede hacer de esa forma para las instrucciones pedida. ¿Por qué? Por el hecho de que de la `ALU` solo se podría obtener el valor del registro `A` o del registro `B`, no ambos. Una solución rápida para este problema sería conectar la salida de estos registros al *Mux DataIn* (incrementando los bits de la señal de selección, si corresponde) y, además, añadir un *Mux* de entrada para cada registro, dado que de esa forma se pueden almacenar directamente en estos los dos valores de memoria, algo que no era posible antes solo con la salida de la `ALU`. Finalmente, se presenta la tabla a partir de la cual se evidencia el funcionamiento correcto de las instrucciones (se espera la particularidad de que en `RW` y `WR` no estén habilitadas las dos señales `WE` implementadas). No es necesario indicar un *opcode*.
Importante: El análisis anterior supone la ejecución de las instrucciones en un solo ciclo. Esto **no está explicitado** en el enunciado, por lo que es válido implementarlas con dos ciclos (lo que implicaría no modificar la `ROM`) tomando en cuenta lo siguiente:

- Se debe ocupar la nueva memoria `RAM`, ya que está explicitado en el enunciado.
- Si implementan las instrucciones en dos ciclos, se espera que añadan un registro adicional que mantenga el primer literal de la operación intacto para la siguiente ejecución.
- A raíz de lo anterior, deben indicar la combinación de señales activadas para cada uno de los dos ciclos de las instrucciones.

c. Asumiendo que ya posee el computador básico de la implementación anterior, ¿qué otra instrucción podría incluir? No es necesario que mencione las señales involucradas para implementarla o el *opcode* asociado, pero sí que justifique por qué se podría implementar.

Aquí existen varias respuestas posibles, pero se espera que mencionen solo una que tenga sentido en el contexto del problema. Si bien no son las únicas, a continuación se presentan algunas opciones válidas:

- `ADD B,Lit` (conectando el segundo literal al *Mux A*, si implementan todo con un ciclo).
- `Sop B,(A) | Sop A,(A) | Sop (A),A` (si se considera la implementación de un segundo *Mux Address*, donde en vez de recibir el bus del registro `B` recibe el del registro `A`).
- `Sop (Dir1),(Dir2)` (si se conecta la salida de la segunda dirección de la `RAM` al *Mux A* de forma que se pueda realizar esta operación).

Donde `Sop` $\in$ {`ADD, SUB, AND, OR, XOR`}.

2. **(Examen - I/2018)**

a. Sea un dispositivo I/O cuya controladora tiene el siguiente diagrama:

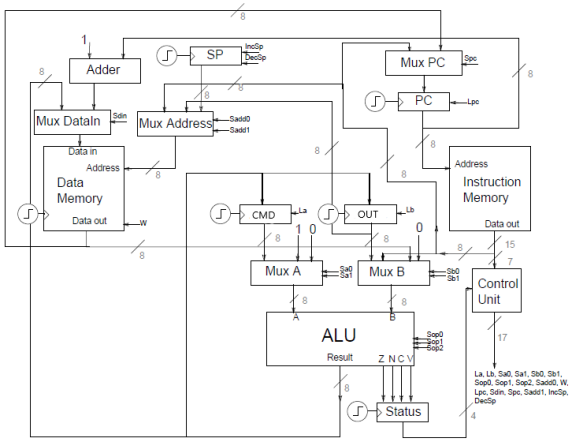


Figura 2: Diagrama de la controladora del dispositivo I/O.

Donde el registro `CMD` puede ser leído y escrito por el computador para entregar un comando al I/O y, además, el registro `OUT` es leído por el computador. En la dirección de memoria 0 siempre se encuentra el estado actual del botón que tiene este I/O. Este dispositivo se conecta vía *ports* a un computador compatible con `x86-286`. Además, se tienen las siguientes tablas de *ports*, comandos y estado del dispositivo:

Port	**Dispositivo**
0	`DMA` *ports* a memoria
1	Comandos `HDD`
2	Estado `HDD`
3-259	*Buffer* `HDD`
260	`CMD` botón
261	*Buffer* botón

Cuadro 1: *Mapeo* de puertos.

Comando	**Acción**
0	No hacer nada
1	Copiar estado del botón en `OUT`
3	Reiniciar controlador

Cuadro 2: Tabla de comandos del I/O.

Estado	**Explicación**
0	Botón no presionado
1	Botón presionado
2	Copiando información

Cuadro 3: Tabla de estados del I/O.

Suponiendo que la controladora de este dispositivo es compatible con la `ISA` del computador básico, escriba el programa que controla este dispositivo.

Lo importante es notar lo que se preguntó: el programa que ejecuta la controladora del dispositivo IO, que es básicamente el computador básico. No es necesario hacer una `ISR`, dado que no es lo que se pide. A continuación, el programa solicitado.

```
DATA:
CODE:
        JMP reset  ; Al prender el IO, se resetea
    entry:
        CMP CMD,1
        JEQ copy
        CMP CMD,2
        JEQ reset
        JMP entry
    copy:
        MOV OUT,2
        MOV OUT,(0)
        MOV CMD,0 ; Esto es opcional
        JMP entry
    reset:
        MOV CMD,0
        MOV OUT,0
        JMP entry
```

b. Qué diferencia el esquema de *ports* del esquema de *memory-mapping*?

Básicamente *ports* crea un espacio de direccionamiento separado y exclusivo para IO, mientras que en *memory-mapping* potencialmente se quita espacio direccionable de la `RAM` al reservar direcciones exclusivas para los dispositivos.

c. ¿Por qué se incluye una unidad `DMA` en los computadores?

Porque de esa forma la `CPU` puede realizar otras tareas importantes mientras se completa una copia de memoria.

d. ¿Qué diferencia los esquemas de *polling* e interrupciones?

En *polling* se pregunta continuamente por el estado del dispositivo, usando ciclos de `CPU` para ello. En interrupciones, el dispositivo IO avisa a la `CPU` cuando tiene algo que necesita atención y en el intertanto la `CPU` puede ejecutar programas sin preocupaciones.

3. a. **(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 4: 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 5: 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 6: 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.

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 1KB = 2^{10}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 7: 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:

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.

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 8: Tablas de página asociadas a cada proceso.

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	P1 en disco
P2	
P2/P3	

Cuadro 9: 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 10: 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.

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.