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

IIC2343 – Arquitectura de Computadores

Ayudantía 2 – Solución propuesta

Profesor: Yadran Francisco Eterovic Solano

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

Nota al lector

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

Representación de números enteros

1. **(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:

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

2. **(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.

3. **(I1 - II/2017)** Sea x un número binario de 8 bits; y sea $\tilde{x}$ una secuencia de 8 bits tal que cada bit de $\tilde{x}$ es la negación del correspondiente bit de x; p.ej., si $x = 01101001$, entonces $\tilde{x} = 10010110$. ¿Cuál es la relación aritmética/algebraica entre -x y $\tilde{x}$?

   La relación existente no es más que **el complemento a 2**. Este se basa en dos procedimientos principales: la negación de todos los bits del número y la posterior adición de una unidad. Usando la nomenclatura del enunciado:

   $$-x = \tilde{x} + 1$$

4. **(I1 - II/2017)** Sea x un número binario de 8 bits; ¿cómo se lo lleva a 16 bits?, tanto para números positivos como para números negativos.

   Veremos ambos casos por separado:

   - **Números positivos:** Aquí basta con añadir 8 bits iguales a 0 por la **izquierda** (para no cambiar el valor del número). Bajo la representación de complemento a 2, sigue siendo positivo y mantiene su magnitud.
   - **Números negativos:** Aquí no basta con añadir los 8 bits por la izquierda, debido a que el número dejaría de ser negativo (por la representación de complemento a 2). Entonces, lo correcto es seguir el siguiente procedimiento:
     a) Se transforma el número a positivo a través del complemento a 2.
     b) Se convierte en un número de 16 bits con el procedimiento descrito en el primer punto.
     c) Este número se transforma en negativo a través del complemento a 2.

     De esta forma, se obtiene el equivalente correcto bajo la representación de 16 bits.

5. **(I1 - II/2017)** ¿Cómo se detecta *overflow* después de una operación aritmética?, tanto para números positivos como para números negativos. Justifica.

   Tenemos cuatro casos a considerar:

   a) $A \geq 0, B \geq 0, A + B < 0$: Se revisa que la operación sea de suma, se comprueba que el bit más significativo de A y B sea igual a cero y, por último, se corrobora que el bit más significativo resultante de la operación sea igual a uno.

   b) $A < 0, B < 0, A + B \geq 0$: Se revisa que la operación sea de suma, se comprueba que el bit más significativo de A y B sea igual a uno y, por último, se corrobora que el bit más significativo resultante de la operación sea igual a cero.

   c) $A \geq 0, B < 0, A - B < 0$: Se revisa que la operación sea de resta, se comprueba que el bit más significativo de A sea cero y el de B sea uno y, por último, se corrobora que el bit más significativo resultante de la operación sea igual a uno.

   d) $A < 0, B \geq 0, A - B \geq 0$: Se revisa que la operación sea de resta, se comprueba que el bit más significativo de A sea uno y el de B sea cero y, por último, se corrobora que el bit más significativo resultante de la operación es igual a cero.

6. Suponga que tiene un total de N bits, y desea representar tanto números positivos como números negativos. ¿Cuál es la cota superior y la cota inferior de los números que se pueden representar?

   Al poseer N bits, tenemos un total de 2^N números posibles que podemos construir. Luego, si queremos dejar de esta representación una mitad para los positivos y una mitad para los negativos, entonces tenemos $\frac{2^N}{2}$ números positivos, y $\frac{2^N}{2}$ números negativos. Sin embargo, también queremos representar el 0. Considerando la representación del bit de signo utilizada en el dígito más significativo, este número utiliza una de las representaciones positivas. Finalmente, los números que podemos representar quedan acotados de la siguiente forma:

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

   Importante: Como el número correspondiente a la cota inferior no posee inverso aditivo, su complemento a 2 es redundante (nos entrega el mismo número). Sin embargo, sigue teniendo sentido aritmético. Si tenemos solo 4 bits:

$$1000_2 + C_2(1000_2) = 1000_2 + 1000_2 = (1)0000_2$$

   Es decir, si se le suma su complemento a dos, nos entrega el número nulo. Además:

$$-8 + 1 = (-8) + 1 = 1000_2 + 0001_2 = 1001_2 = -C_2(1001_2) = -0111_2 = -7$$

   Es decir, no presenta problemas al momento de operar. Por otra parte, también se puede obtener como resultado:

$$-7 - 1 = (-7) + (-1) = 1001_2 + 1111_2 = (1)1000_2$$

   Finalmente, la única diferencia es que no se puede operar con su inverso aditivo y obtener resultados consistentes.

7. Suponga que tiene un total de N trits (dígitos trinarios), y desea representar tanto números positivos como números negativos. ¿Cuál es la cota superior y la cota inferior de los números que se pueden representar?

   Siguiendo el mismo procedimiento, le asignamos a los negativos una mitad, y a los positivos la otra. Sin embargo, hay que notar que toda potencia de un número impar es también impar.

   Demostración: Utilizando la representación de número impar $2k+1$, tenemos que:

$$(2k+1)^N = \sum_{i=0}^{N} \binom{N-i}{i} (2k)^{N-i} 1^i = \binom{N}{0}(2k)^N 1^0 + \binom{N}{1}(2k)^{N-1}1^1 + ... + \binom{0}{N}(2k)^0 1^N$$

   Es fácil ver que todos los términos, salvo el último (que es igual a 1), son factores de 2 (esto, debido a las potencias de $2k$). Entonces, podemos abreviar esta expresión como:

$$(2k+1)^N = 2k^{'} + 1$$

   la que corresponde a un número impar. ■

   Nota: Esta demostración no es necesaria en términos de evaluación, solo la mostramos para corroborar la sentencia anterior.

   Tomando esto en consideración, tenemos que $\frac{3^N}{2}$ no es un número entero, por lo que nos vemos obligados a utilizar la función piso $\left\lfloor \frac{3^N}{2} \right\rfloor$ Al utilizar esta función para los dos tramos, sin embargo, estamos perdiendo un número para representar. En este caso, optaremos por darle ese espacio al cero, y finalmente:

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

   Notar que se pudo haber hecho, por ejemplo, que el número extra representara a un positivo más (y por consistencia, un negativo menos). Sin embargo, optamos por este caso ya que nos permite tener para todo número su valor negativo.

   Importante: En esta representación se pueden apreciar dos cosas:

   - En esta base (y por consiguiente, en bases impares) no se genera un dígito (o intervalo de estos) que represente el signo del número. Por ejemplo, si tomamos la base 4, se puede ver fácilmente que los números cuyo dígito más significativo es 0 o 1 son positivos, mientras que los que poseen un 2 o un 3 son negativos. Esto no se ve en la base ternaria. Si se toman tres trits, el máximo número positivo es 13, que corresponde a 111_3. Su 'complemento a 2' corresponderá a 112_3[1], y al sumar obtenemos el número nulo (pero su complemento a 2 tiene el mismo trit en la posición más significativa).
   - A pesar de lo anterior, se puede corroborar que las sumas son consistentes con los números positivos y negativos (siempre y cuando no se salga del intervalo que se puede representar).

[1] Es importante notar que el complemento a 2 con bases mayores a 2 es distinto. Para obtenerlo, se toma el mayor dígito de la base y se le resta la del número actual, a la que posteriormente se le suma una unidad. Por ejemplo, si tomamos 123_4, su 'complemento a 2' será $(3-1)(3-2)(3-3)_4 + 1_4 = 211_4$.

Representación de números racionales

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

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

   a) **Signo:** Un bit.
   b) **Exponente:** 8 bits.
   c) **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:

   $$x = (-1)^1 * 1,00000000000000000000000 * 2^{131-127}$$

   Entonces, tenemos finalmente que $x = -2^4 = -16$.

2. **(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.

3. **(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

4. **(C1 - II/2017)** Multiplica los números $0{,}5_{10}$ y $-0{,}4375_{10}$ en notación científica normalizada ***en binario***, siguiendo los pasos del algoritmo de multiplicación en punto flotante estudiado en clase; muestra el resultado al ejecutar cada paso.

   Se ejecuta paso a paso el algoritmo visto en clases:

   I. Se convierten los números a notación científica normalizada en binario, con cuatro bits de precisión:

   $$1{,}000_2 \times 2^{-1}, -1{,}110_2 \times 2^{-2}$$

   II. Se suman los exponentes defasados y se resta el desfase (127_{10}) a la suma para obtener el nuevo exponente desfasado:

   $$-1 + (-2) = -3 \rightarrow 124$$

   III. Se multiplican los significantes:

   $$1{,}000_2 \times 1{,}110_2 = 1110000_2 \rightarrow 1{,}110_2 \times 2^{-3}$$

IV. Se normaliza el producto si es necesario, realizando *shift right* e incrementado el exponente. No se hace nada al ya estar normalizado.

V. Se redondea el significante al número apropiado de bits. En este caso, no cambia nada. Como el resultado ya está normalizado, se avanza al siguiente paso.

VI. Se pone el signo del producto en positivo si los signos de los operandos son iguales; en negativo en caso contrario. Por lo tanto, el resultado es:

$$-1{,}110_2 \times 2^{-3}$$

Operaciones aritméticas y lógicas

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

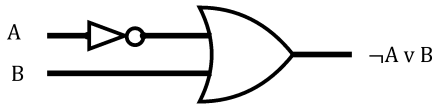
A	B	A $\rightarrow$ B
0	0	1
0	1	1
1	0	0
1	1	1

   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:



2. **(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

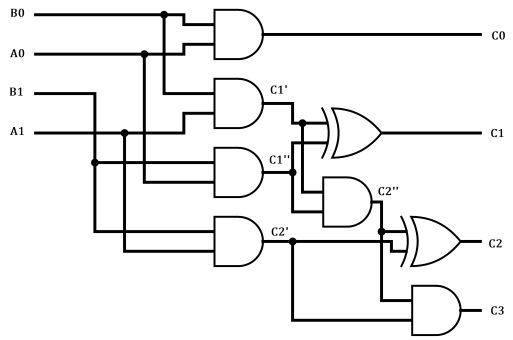
   Es fácil ver que la multiplicación la podemos realizar a partir de la compuerta lógica `AND`. Ahora, como nos piden una multiplicación entre dos números de dos bits, tenemos que seguir el método de multiplicación tradicional.

   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:



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

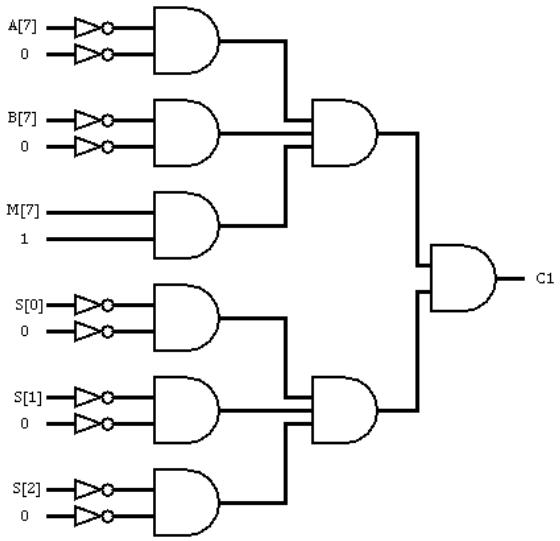
3. **(I1 - I/2017)** Construya un circuito que permita detectar la ocurrencia de *oveflow* al sumar o restar dos números enteros de 8 bits en una ALU.

   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:

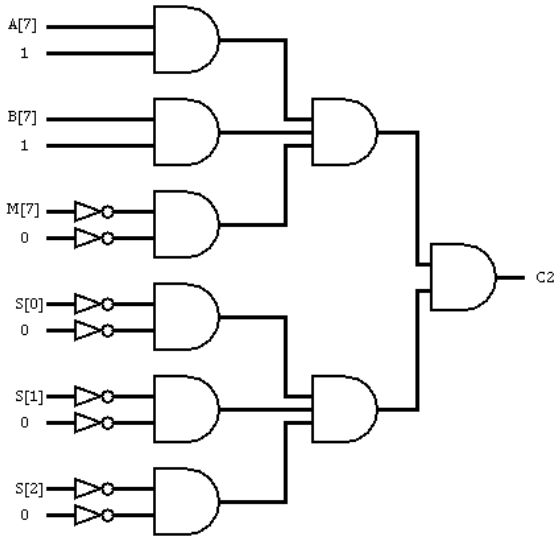
   - **Caso 1:** $A \geq 0, B \geq 0, A + B = M < 0$
   - **Caso 2:** $A < 0, B < 0, A + B = M \geq 0$
   - **Caso 3:** $A \geq 0, B < 0, A - B = M < 0$
   - **Caso 4:** $A < 0, B \geq 0, A - B = M \geq 0$

   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:

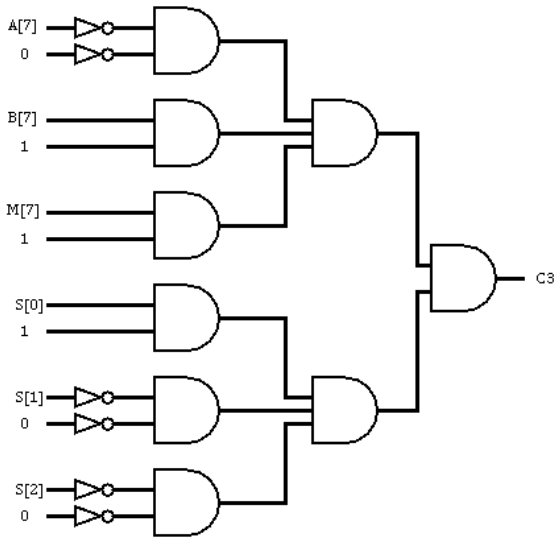
- **Caso 1**



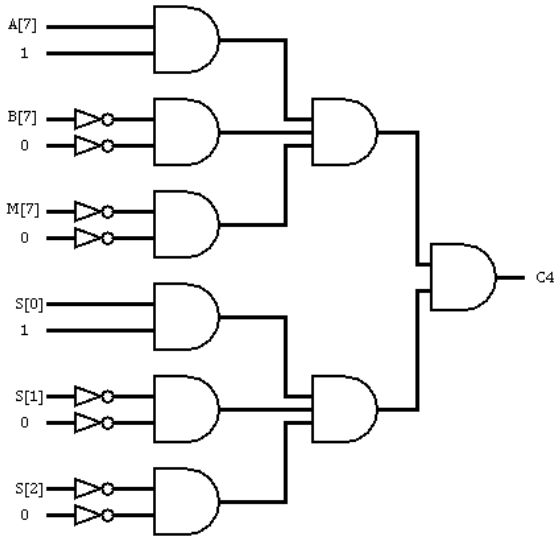
- **Caso 2**



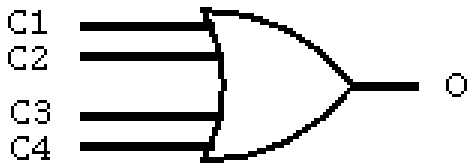
- **Caso 3**



- **Caso 4**

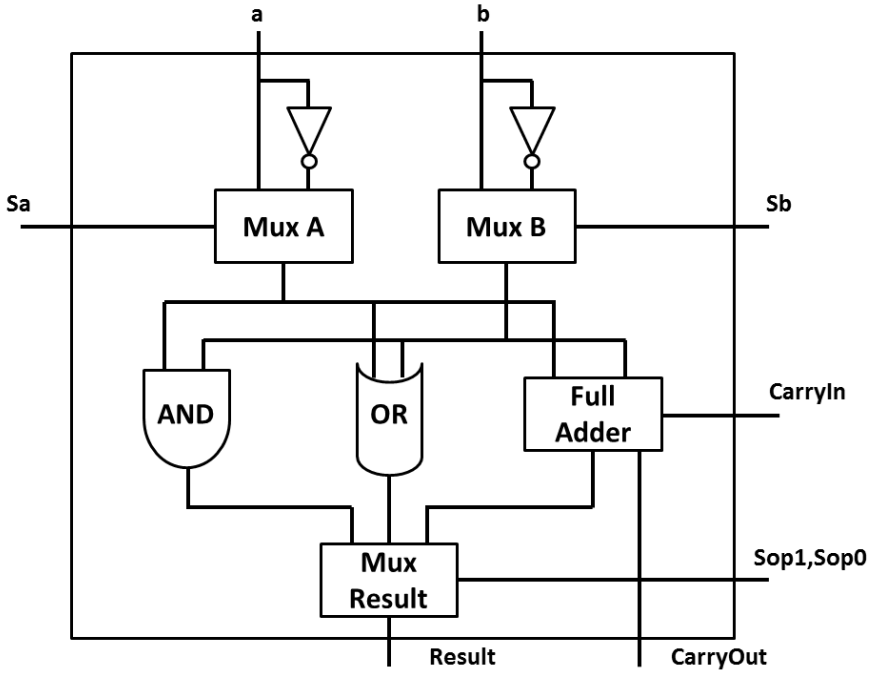


Finalmente, conectamos los resultados a un puerto `OR` para obtener el bit que indica si se generó o no *overflow* (el que llamamos O):



4. **(I1 - II/2017)** Dibuja el circuito correspondiente a una `ALU` de 1 bit, con tres entradas, a, b y *CarryIn*, y dos resultados, *Result* y *CarryOut*. La `ALU` debe ser capaz de ejecutar las operaciones `AND`, `OR` y suma sobre a y b o $\tilde{a}$ y $\tilde{b}$, de acuerdo con tres señales de control que determinan qué valores se usan y qué operación se ejecuta. Tu circuito puede contener compuertas `AND`, `OR` y `NOT` y multiplexores (selectores) de 2 y 3 bits.

   El circuito resultante es el siguiente:



   Donde:

   - **Sa:** Permite seleccionar en el Mux `A` si la señal a utilizar será a o $\tilde{a}$ dentro de la operación.
   - **Sb:** Permite seleccionar en el Mux `B` si la señal a utilizar será b o $\tilde{b}$ dentro de la operación.
   - **Sop1,Sop0:** Permiten seleccionar la operación que se utilizará como resultado, entre la suma, `AND` y `OR`.

5. **(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 → 39
   - `shift right`: 00010011 → 19
   - `shift right`: 00001001 → 9
   - `shift right`: 00000100 → 4
   - `rotate left`: 00001000 → 8
   - `rotate left`: 00010000 → 16
   - `rotate left`: 00100000 → 32
   - `rotate left`: 01000000 → 64
   - `rotate left`: 10000000 → -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 → 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`.

6. **(I1 - I/2017)** Dado un número entero x de 32 bits almacenado en una memoria con palabras de 1 byte, determine el valor de x tal que el error en valor absoluto es máximo si se confunde el endianness del número al ser leído e interpretado.

   Al almacenar el número entero en palabras de 1 byte, tenemos un total de 4 palabras (32 bits → 4 bytes). Ahora, la pregunta puede ser interpretada de dos formas, lo que entrega dos resultados diferentes:

   a) Máximo error interpretado como $|a - b|$
   El máximo error se da para a = 0x80000000. En este caso, a representa el número más negativo representable, por lo que al restarle otro valor nos quedará un número de mayor módulo. De esta forma, b = 0x00000080, por lo que:

   $$|a - b| = |-2^{31} - 2^7| = 2^{31} + 2^7 = 2147483776$$

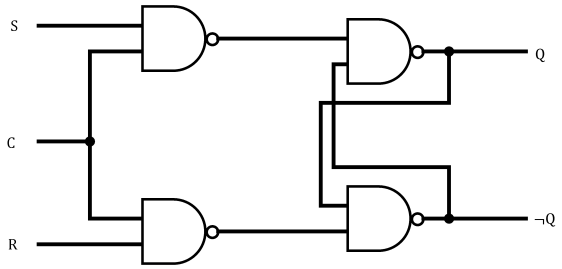
   b) Máximo error interpretado como $|a| - |b|$
   El máximo error también se da para a = 0x80000000. En este caso, el valor absoluto de a será incluso mayor al más positivo bajo esta representación. Entonces:

   $$|a| - |b| = 2^{31} - 2^7 = 2147483520$$

Almacenamiento de datos

1. **(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 $C = 1$.

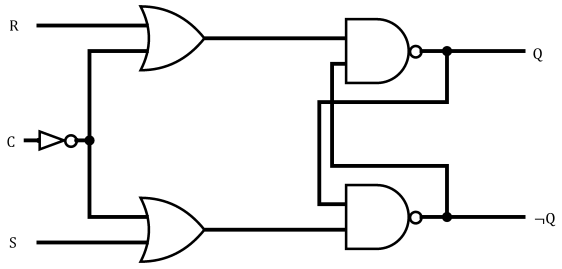
   Una posible implementación consiste en el siguiente circuito:



   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	?	?

   Sin embargo, el siguiente circuito también sirve:

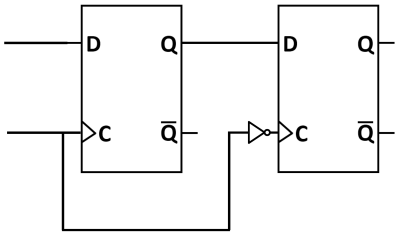


   En este también se cumple con el objetivo de controlar los cambios de estado con la señal C, pero además no es necesario invertir el orden de las señales R y S (pues cumplen sus funciones como corresponde). Por otra parte, se mantiene la combinación que indefine los estados del circuito. Finalmente,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

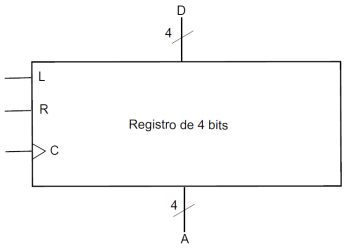
2. **(I1 - II/2012)** Implemente mediante compuertas lógicas, elementos de control y latches, un *flip-flop* tipo D que funcione con flanco de bajada.

   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: $1 \to 0$ (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.

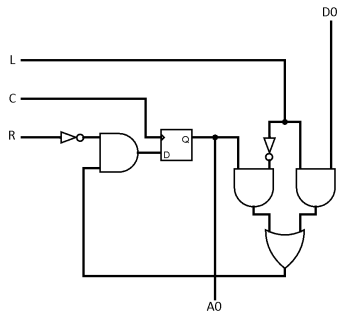
3. **(I1 - I/2012)** Implemente mediante compuertas lógicas y *flip-flops* tipo D, el registro de la figura, con señales de control (C), carga (Load) y reset (Reset), que funciona con flanco de subida.



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:



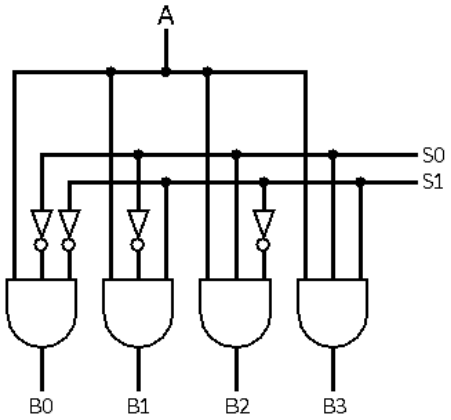
Es importante notar que esto solo ilustra el almacenamiento del bit menos significativo en el registro. No obstante, el almacenamiento del resto de los bits es el mismo.

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.

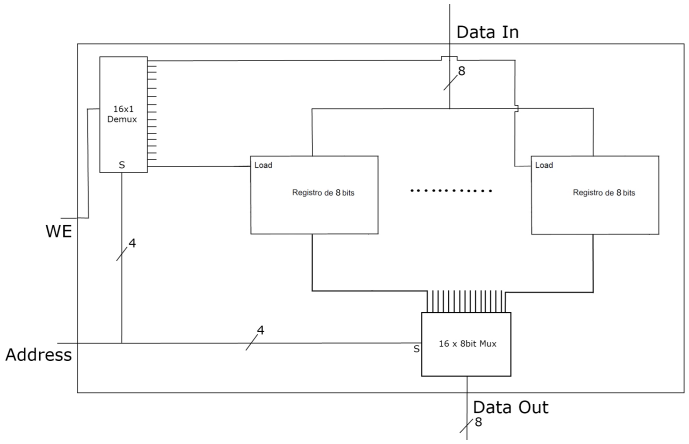
4. **(I1 - I/2012)** Diseñe un De-Multiplexor con bus de datos de 1 bit y bus de control de 2 bits.

   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:



5. **(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:



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].

6. **(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):

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

[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.

7. 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.

Arquitecturas de computadores

1. **(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-, 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 en 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 esta arquitectura se necesitan dos ciclos por lo bajo.

2. **(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. Otra consideración importante sería el manejo del PC, donde una opción es que estos apunten, dentro de la memoria, a registros de 16 bits (con los primeros 8 bits correspondientes al opcode, y los siguientes al literal), enviando primero los 8 más significativos (instrucción) y luego los 8 menos significativos (literal). Luego, al tercer ciclo se ejecuta la instrucción completa. Otro detalle a cuidar del PC sería que solo fuera alimentado al final del tercer ciclo de toda instrucción, ya que si esto sucediera en el primer o segundo ciclo, habrían inconsistencias en los resultados esperados. Finalmente, faltaría tomar en cuenta que en la ISA se tenga la precaución de que el literal enviado en el segundo ciclo no sea ejecutado en la unidad de control (lo que podría conllevar a problemas en la ejecución de la instrucción final).

   Notar que esto es posible hacerlo también en 2 ciclos, asumiendo que de la memoria se extrae el bloque completo que contiene el opcode y el literal, y se alcanzan a enviar a los componentes correspondientes antes del flanco de subida (en este caso, PC no podría ser alimentado en el primero flanco de subida, solo podría pasar en el segundo).

3. **(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.

   Aquí nos encontramos con dos casos:

   - Asumiendo que no podemos obtener el opcode y el literal en un solo ciclo:
     a) 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.
     b) Se obtiene el literal (dirección de memoria) y se propaga a los multiplexores correspondientes (en este caso, al Mux Address).
     c) 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`.
     d) 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.
     e) 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`).
     f) 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`.
     g) 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.
     h) Se obtiene el literal (dirección de memoria) y se propaga a los multiplexores correspondientes (en este caso, al Mux Address).
     i) 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:
     a) Se obtiene el opcode de la instrucción `MOV A,(Dir)` y el literal de la dirección, propagándose a los componentes correspondientes.
     b) Se ejecuta la instrucción, almacenando el dato de la dirección `Dir` en el registro `A`.
     c) Se obtiene el opcode de la instrucción a ejecutar sobre el registro `A` y el literal para operar, propagándose a los componentes correspondientes.
     d) Se ejecuta la instrucción, almacenando el resultado de la operación en el registro `A`.
     e) Se obtiene el opcode de la instrucción `MOV (Dir),A` y el literal de la dirección, propagándose a los componentes correspondientes.
     f) 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.

4. **(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 utilice 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.
   - 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) = MOV A,(dir) - INC A - MOV (dir),A`.

5. **(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`. Notar que la primera y última instrucción se usan de forma que no perdamos el valor almacenado en el registro `B`, pues no es el objetivo del comando inicial.

6. **(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 (dir),reg` para almacenar en la dirección `dir` el valor almacenado en el registro (ya sea `A` o `B`). Se puede asumir que `dir` es una dirección fija con uso exclusivo para esta instrucción.
   - Añadimos la instrucción `CALL (dir)`, que carga en el registro PC el valor almacenado en la dirección `dir`. Notar que esto debe ocurrir en 2 ciclos, ya que primero tenemos que guardar en el stack la posición PC+1 para poder volver al retornar el llamado, y otro para obtener el valor almacenado en `dir` para realizar el salto.

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