DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343 - Arquitectura de Computadores (I/2023)

Tarea 1

Pauta de evaluación

Consideraciones

- Respuestas sin desarrollo o justificación no tendrán puntaje.
- Cada pregunta podría tener más de un desarrollo válido. La pauta evalúa eso y este documento solo muestra una alternativa de solución.
- Cualquier detección de infracción al código de honor será sancionada.

Código de Honor de la UC

"Como miembro de la comunidad de la Pontificia Universidad Católica de Chile me comprometo a respetar los principios y normativas que la rigen. Asimismo, prometo actuar con rectitud y honestidad en las relaciones con los demás integrantes de la comunidad y en la realización de todo trabajo, particularmente en aquellas actividades vinculadas a la docencia, el aprendizaje y la creación, difusión y transferencia del conocimiento. Además, velaré por la integridad de las personas y cuidaré los bienes de la Universidad."

Pregunta 1: Representación de números (12 ptos.)

(a) (4 ptos.) El complemento de 2 para la representación de números en base 2 estudiado en clases cuenta con la limitante de ser desequilibrado al contar con más números negativos que positivos. Comente qué tipo de representaciones posicionales de números enteros, según su base, son equilibradas al aplicar su respectivo complemento. Señale, además, una desventaja que presenten frente a la representación binaria con complemento de 2.

Solución: Si se revisa la representación de números según su base, se puede evidenciar que la representación de números de base impar cumplen con ser balanceadas. Esto ocurre porque la cantidad de números que se puede representar es impar y, por ende, se puede tener una única representación para el 0 y un mapeo directo entre números pares e impares. Además, al hacer uso del complemento de la base (por ejemplo, complemento de 3 para base 3), se sigue respetando la aritmética ($A + (-A) = 0$), por lo que efectivamente presentan representaciones equilibradas para números enteros. Una desventaja existente se encuentra en el hecho de que, en este tipo de representación, no nace un dígito o un conjunto de dígitos de signo. Por ejemplo, se puede evidenciar que para bases pares siempre existirá un conjunto fijo para dígitos de signo negativo ({ 1 } para representación binaria; { 2, 3 } para representación en base 4, { 3, 4, 5 } para representación en base 6), pero para representaciones de base impar esto no ocurre. Por ejemplo, para representaciones en base 3, existirán números con dígito más significativo igual a 1 que pueden ser positivos o negativos: $12_3 = 110 \rightarrow C_3(110) = 120 = -12$. Esto hace que la lectura del número sea menos intuitiva y se pierde la ventaja del dígito o conjunto de dígitos de signo de una representación de base par.

Distribución de puntaje

- **2 puntos** por señalar que las bases impares presentan representaciones de números enteros equilibradas. Es válido también que señalen una sola. En cualquiera de los dos casos, debe estar explicada la respuesta (ya sea con demostraciones o ejemplos).
- **2 puntos** por señalar una desventaja justificada. Es válido si señalan que no se podría traducir esto en un circuito digital al tener que representar más de dos estados, pero se debe destacar que esto aplica para cualquier base mayor a 2 y no solo para las impares.

(b) (4 ptos.) Describa una codificación para números enteros binarios distinta al complemento de 2, que utilice bit de signo y tenga una única representación para el cero. No es necesario que en esta codificación se cumpla que $A + (-A) = 0$.

Solución: Una posible solución es la codificación "zig-zag". Este esquema utiliza los números naturales para codificar de manera alternada los números negativos y positivos, *i.e.*: 0 $\rightarrow$ 0000, -1 $\rightarrow$ 0001, 1 $\rightarrow$ 0010, -2 $\rightarrow$ 0011, 2 $\rightarrow$ 0100, etc. De esta manera, los números negativos siempre tendrán un 1 como bit menos significativo y los positivos tendrán un 0, mientras que el cero será representado únicamente como 0. La diferencia con el complemento de 2 sería que el bit de signo es el menos significativo (y no el más significativo), además de no respetar que $A+(-A) = 0$ (que es lo esperado por la pregunta).

Distribución de puntaje

- **2 puntos** por indicar una codificación que posea bit de signo.
- **2 puntos** por indicar una codificación que respete una única representación para todo número.

(c) (4 ptos.) Dados números naturales K, N, T, donde $K \geq 2$ y $N > T > 0$, describa un algoritmo para transformar de manera **eficiente** números naturales codificados en base K^N a K^T. No es válido utilizar la fórmula general de transformación de bases $\sum_{k=0}^{n-1} s_k \times b^k$, dado que **no es eficiente** realizar una transformación intermedia para lograr el resultado esperado.

Solución: Un algoritmo eficiente puede realizarse en dos pasos:

1. Transformar cada cifra del número original en N cifras que representen el mismo número en base K, manteniendo el orden correlativo entre cifras.
2. Agrupar las cifras del número expandido en grupos de T cifras y transformar cada grupo en un número en base K^T, nuevamente manteniendo el orden correlativo.

Este algoritmo funciona ya que K^N y K^T son de la misma base, por lo que se puede **mostrar** a través de transformación a base decimal que efectivamente se respeta el valor del número transformado.

Distribución de puntaje

- **2 puntos** por indicar un algoritmo que funciona, independiente de su eficiencia.
- **2 puntos** por señalar un algoritmo que no requiere transformación a una base intermedia para lograr lo solicitado.

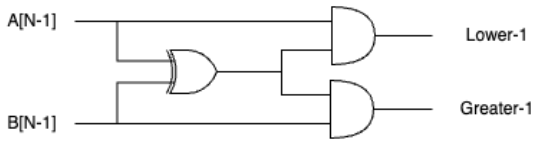
Pregunta 2: Operaciones aritméticas y lógicas (24 ptos.)

(a) (4 ptos.) Un comparador de números es un circuito que, dados dos números A y B en representación posicional, indica cuál es el mayor, o si estos son iguales. El circuito posee tres salidas, donde la primera entrega un 1 solo si A es el mayor, la segunda un 1 solo si ambos son iguales, y la tercera un 1 solo B es mayor. Haciendo uso de las compuertas lógicas vistas en clases, diseñe un comparador de números **enteros** de N bits, explicando la funcionalidad de cada uno de los circuitos que elabore.

Solución: Las tres salidas de nuestro circuito se llamarán *Greater*, *Equal* y *Lower* según lo descrito en el enunciado. Luego, para facilitar la construcción de nuestro circuito, vemos que existen cuatro casos a evaluar:

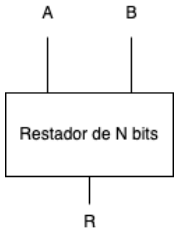
- **Caso 1:** $A \geq 0, B \geq 0$
- **Caso 2:** $A < 0, B < 0$
- **Caso 3:** $A \geq 0, B < 0$
- **Caso 4:** $A < 0, B \geq 0$

Los casos 3 y 4 se pueden resolver rápidamente haciendo uso del **bit de signo**: Verificamos que A y B posean signo distinto a través de una compuerta `XOR` (cuya salida es 1 solo si ambas señales son distintas) y este resultado lo conectamos con compuertas `AND` que nos ayudarán a determinar si se cumple que la condición *Greater-1* o *Lower-1* (primer caso para identificar si el resultado es mayor o menor):



Se observa en el circuito que si $A_{N-1} = 0, B_{N-1} = 1$, entonces la señal *Greater-1* estará activa y la señal *Lower-1* no, mientras que se dará el caso contrario para $A_{N-1} = 1, B_{N-1} = 0$. Si $A_{N-1} = B_{N-1}$ ninguna de las señales se activará, independiente del signo de estos números.

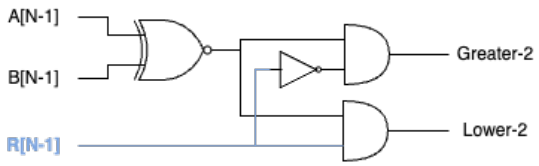
Para resolver los casos 1 y 2, haremos uso de un restador de N bits para obtener un nuevo bus de N bits R:



R nos sirve por las siguientes observaciones:

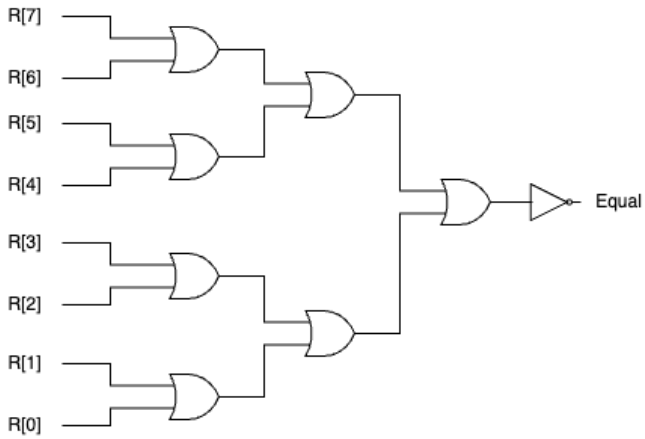
- $A - B > 0 \rightarrow A > B \rightarrow (A - B)_{N-1} = R_{N-1} = 0$
- $A - B < 0 \rightarrow A < B \rightarrow (A - B)_{N-1} = R_{N-1} = 1$

Notamos entonces que el bit más significativo de R nos indica si $A > B$ o $A < B$, lo que podemos verificar con el siguiente circuito:



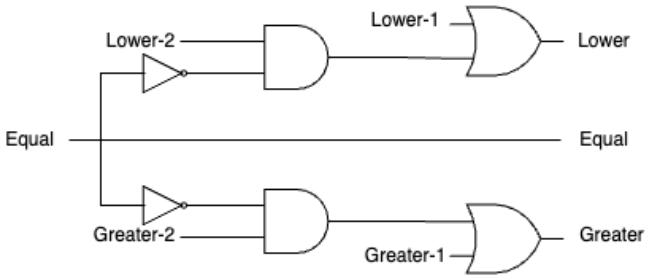
En contraste con el circuito anterior, aquí hacemos uso de una compuerta `XNOR` para verificar que los signos de A y B sean **iguales**. Observamos entonces que si $R_{N-1} = 0$, entonces se activará la señal *Greater-2* que indica que $A > B$ y, en contraste, si $R_{N-1} = 1$ entonces se activará la señal *Lower-2* que indica que $A < B$ (siempre que se cumpla A `XNOR` $B = 1$ para ambos casos).

El caso anterior no está del todo completo, dado que no estamos considerando la posibilidad donde $A = B$. Para ello, definimos un nuevo circuito que verifique que **todos los bits de R sean iguales a cero**:



En este caso se muestra para $N = 8$ pero su extensión a cualquier N es trivial. El circuito activa la señal *Equal* si, y solo si **ninguna** compuerta `OR` entrega como resultado 1 (que se da solo si alguno de los bits de R es igual a 1).

Usamos este último resultado para entregar la señal faltante *Equal* y para que los casos *Greater-2* y *Lower-2* sean válidos solo si $Equal = 0$:



De esta forma, *Greater*, *Lower* y *Equal* se activan correctamente según los casos señalados al comienzo.

Distribución de puntaje

- **0.75 puntos** por cada caso considerado correctamente.
- **1 punto** por la inclusión del caso $A = B$ en los casos $A \geq 0, B \geq 0$ y $A < 0, B < 0$.

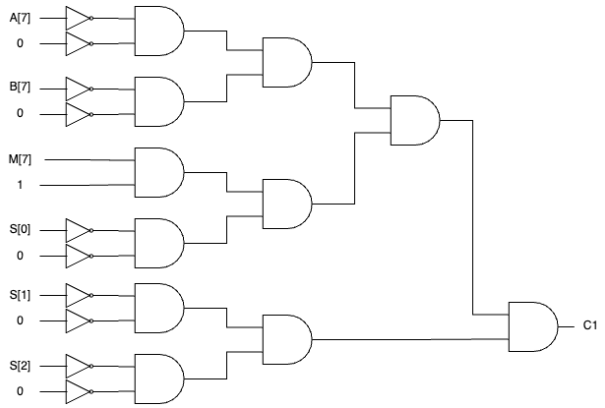
(b) (4 ptos.) Construya un circuito que permita detectar la ocurrencia de *overflow* al sumar o restar dos números enteros de 8 bits en una ALU.

Solución: 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:

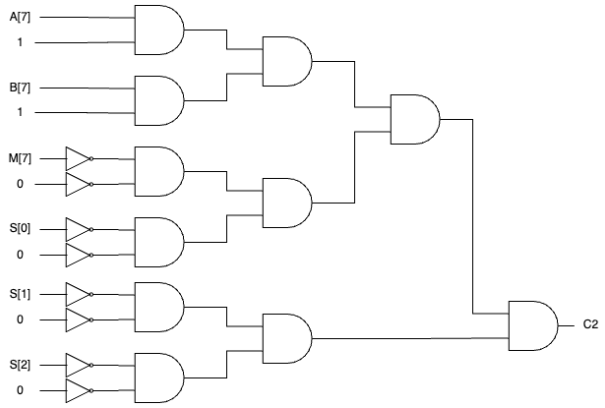
- **Caso 1:** $A \geq 0, B \geq 0, A + B = M < 0$
- **Caso 2:** $A < 0, B < 0, A + B = M \geq 0$
- **Caso 3:** $A \geq 0, B < 0, A - B = M < 0$
- **Caso 4:** $A < 0, B \geq 0, A - B = M \geq 0$

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:

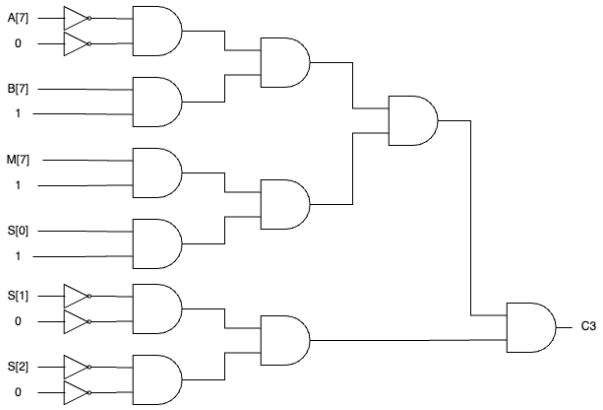
- **Caso 1:**



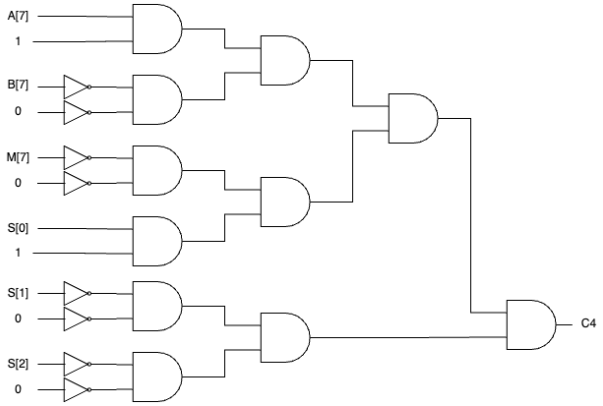
- **Caso 2:**



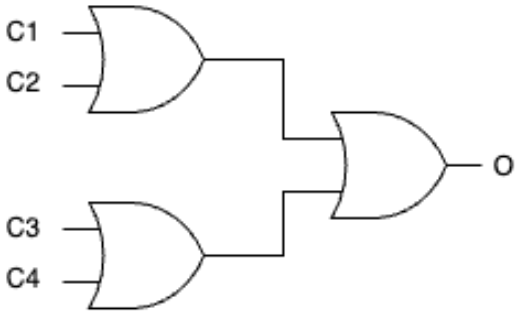
- **Caso 3:**



- **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):

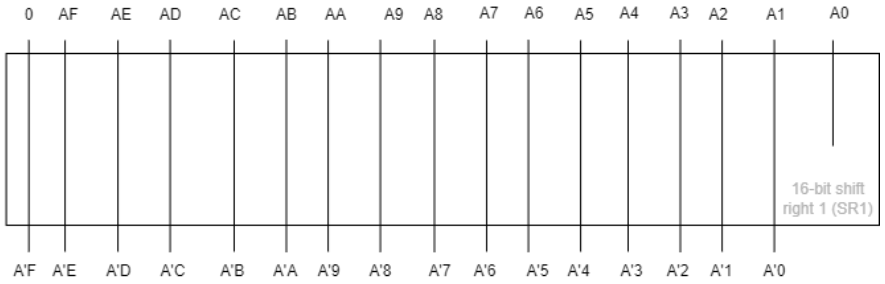


Distribución de puntaje:

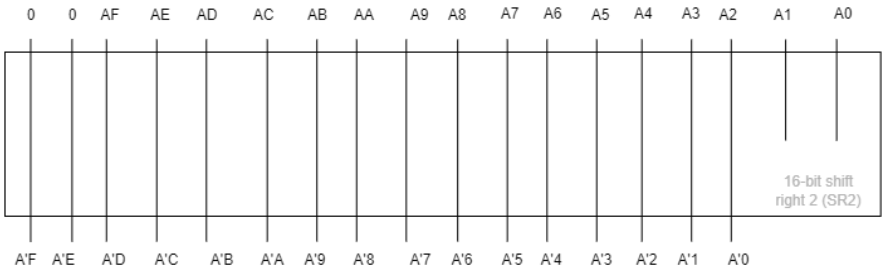
- **0.75 puntos** por cada caso abordado correctamente.
- **1 punto** si el circuito completo entrega el *output* esperado.

(c) (4 ptos.) Construya dos componentes *shift left* y *shift right* de 16 bits que permitan seleccionar la cantidad de *shifts* a realizar en un número. Considere que se admite un máximo de 16 *shifts* para cada componente y que son de tipo lógico, no aritmético.

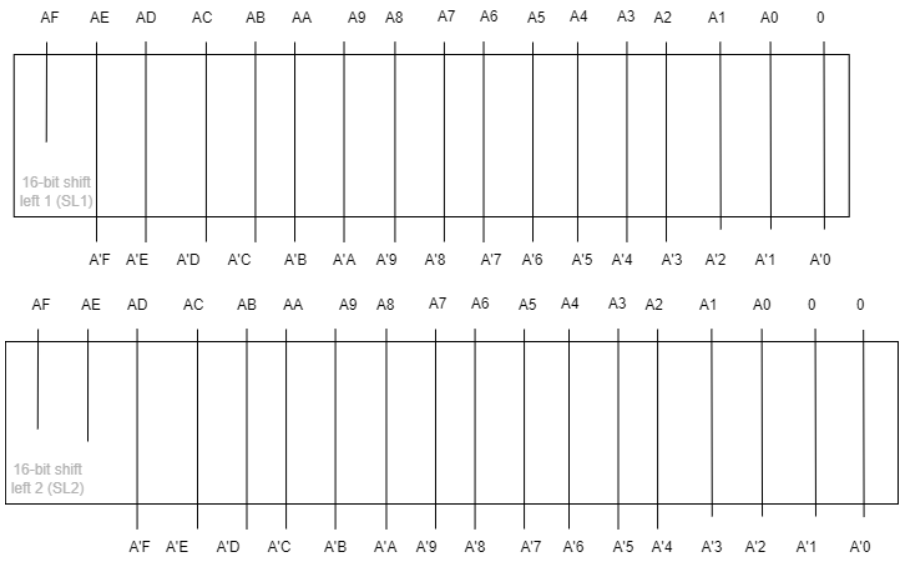
Solución: Para la solución primero definermos dos componentes, *16-bits shift right N (SRN)* y *16-bits shift left N (SLN)*, donde N indicará la cantidad de *shifts* que son dados. Para ello, el valor N representará los ceros conectados hacia los bits más significativos del *output* en caso de ser un *shift right* y los bits menos significativos en caso de ser un *shift left*, además de la cantidad de conexiones a tierra. Se muestra en primer lugar el caso de SR1 (de un solo *shift*) a continuación:



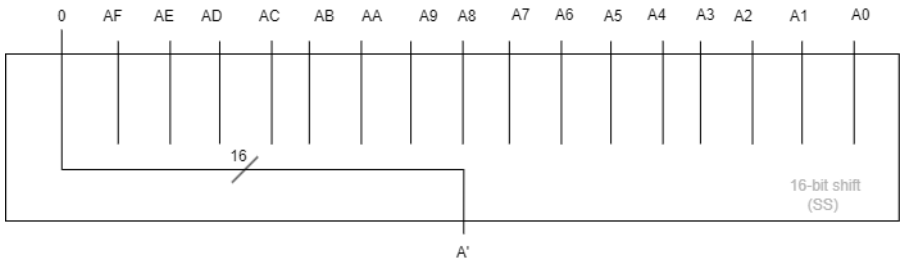
Como $N = 1$, se realiza una conexión con cero al bit más significativo y una conexión a tierra al bit menos significativo. En el caso $N = 2$, el resultado será el siguiente:



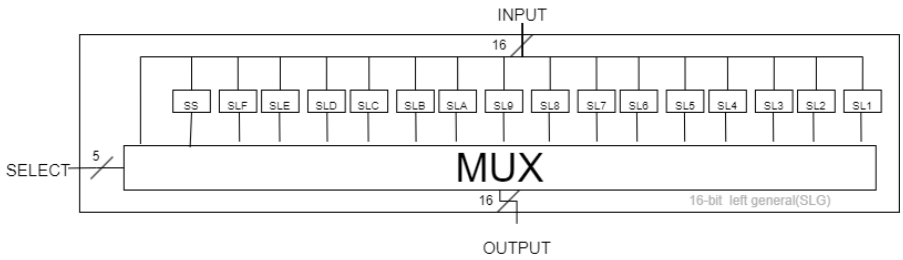
Como $N = 2$, se realiza una conexión con cero a los dos bits más significativos y una conexión a tierra a los dos bits menos significativos. Análogamente, se tiene un resultado similar con el *shift left*, pero invirtiendo los roles. Aquí se muestran los componentes para $N = 1$ y $N = 2$:



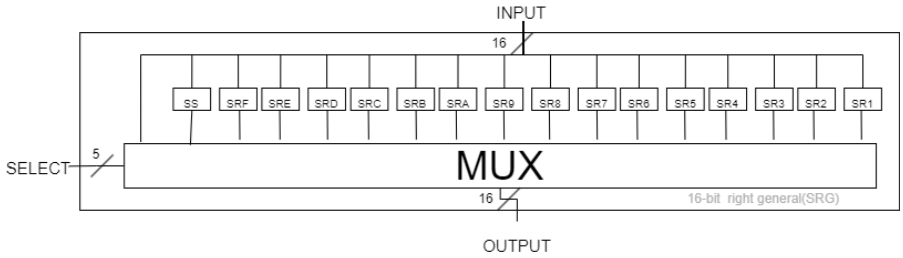
Aplicaremos la misma regla para $N = 3, 4, ..., 15$. Para $N = 16$, independiente del tipo de *shift* tendremos el siguiente resultado:



Es decir, el resultado es 0. Este componente se compartirá para ambos diagramas. Finalmente, para construir el primer componente de *shift left* de 16 bits utilizaremos un multiplexor con un bus de selección de 5 bits. En caso de dar un valor mayor a 16, este seleccionará la entrada sin aplicar ninguna operación. El diagrama se muestra a continuación:



Análogamente, tenemos el siguiente diagrama para el componente *shift right* de 16 bits:



Distribución de puntaje:

- **2 puntos** por el *shift right* bien construido. Es opcional la adición del caso donde no se realice una modificación sobre el *input* (*i.e.* 0 *shift rights*).
- **2 puntos** por el *shift left* bien construido. Es opcional la adición del caso donde no se realice una modificación sobre el *input* (*i.e.* 0 *shift lefts*).

(d) (12 ptos.) En la década de los 90, los videojuegos eran limitados tanto en aspectos gráficos como en sonido y funcionalidad. En este último aspecto, podemos tomar como ejemplo la generación de números aleatorios (RNG o *Random Number Generator*). Basándonos en videojuegos como *Super Mario World* de la consola *Super Nintendo*[1], podemos crear una función en pseudo-código que describe una forma de generar números al azar:

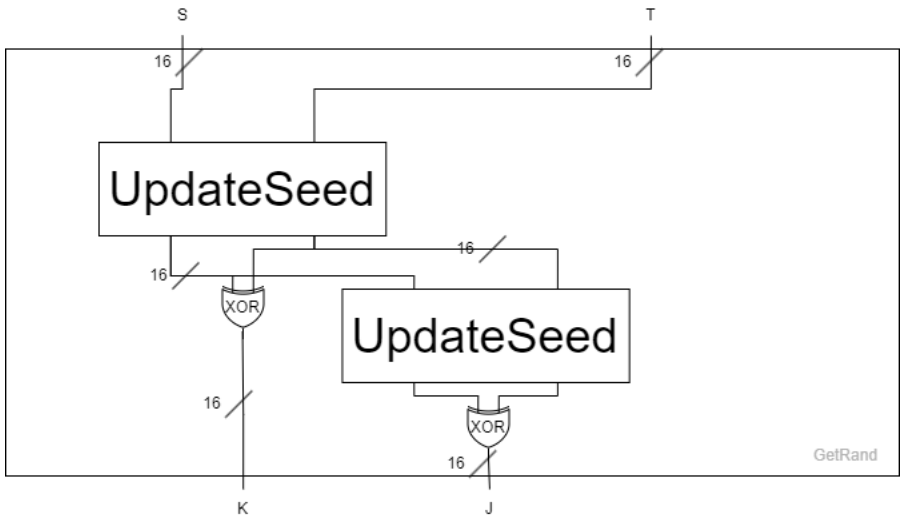
```
func get_rand(S, T):
  S, T = update_seed(S, T)
  K = S XOR T
  S, T = update_seed(S, T)
  J = S XOR T
  return J, K
end

func update_seed(S, T):
  S = 17S + 1
  if T[4] == T[7] then
    T = 2T + 1
  else
    T = 2T
  end
  return S, T
end
```

Digamos que queremos tener este mismo generador de *seeds*, pero ahora integrado al *hardware* de nuestro dispositivo. Cree el circuito lógico que permita hacer esto, teniendo como *inputs* S y T (ambos de 16 bits) y *outputs* J y K, también de 16 bits.

Restricciones: Solo se permiten circuitos combinacionales, no se permiten circuitos secuenciales ni unidades de almacenamiento de datos (latch, flip-flop, registros, contadores, memorias). Los únicos circuitos "pre-hechos" que se pueden utilizar (aparte de las compuertas lógicas) son el sumador-restador de N bits, *enablers*, multiplexores y compuertas *bitwise* vistas en clases. Puede, y se recomienda, crear componentes intermedias si así lo requiere su implementación.

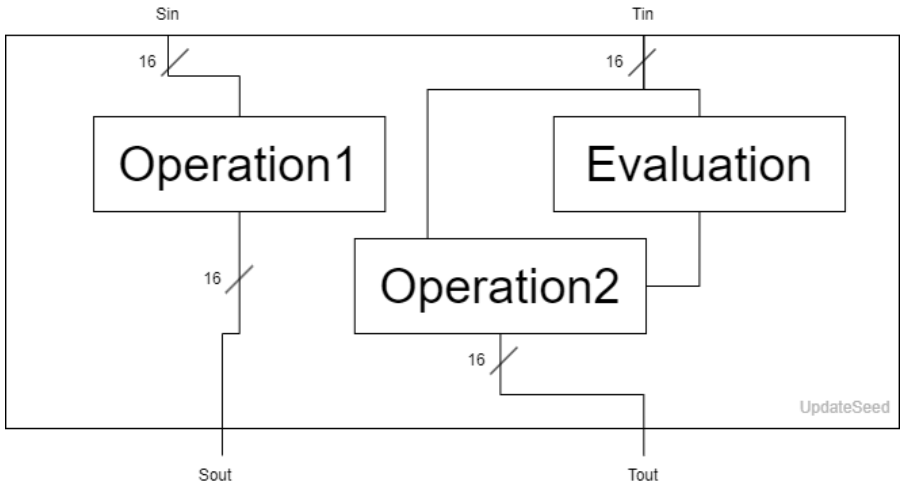
[1] Puede aprender más de esto en el siguiente video.

Solución: Para describir este circuito, elaboraremos el diagrama general con "cajas negras" y lo iremos describiendo a más bajo nivel de forma gradual. En primer lugar, la función `getRand` puede ser descrita por el siguiente circuito:

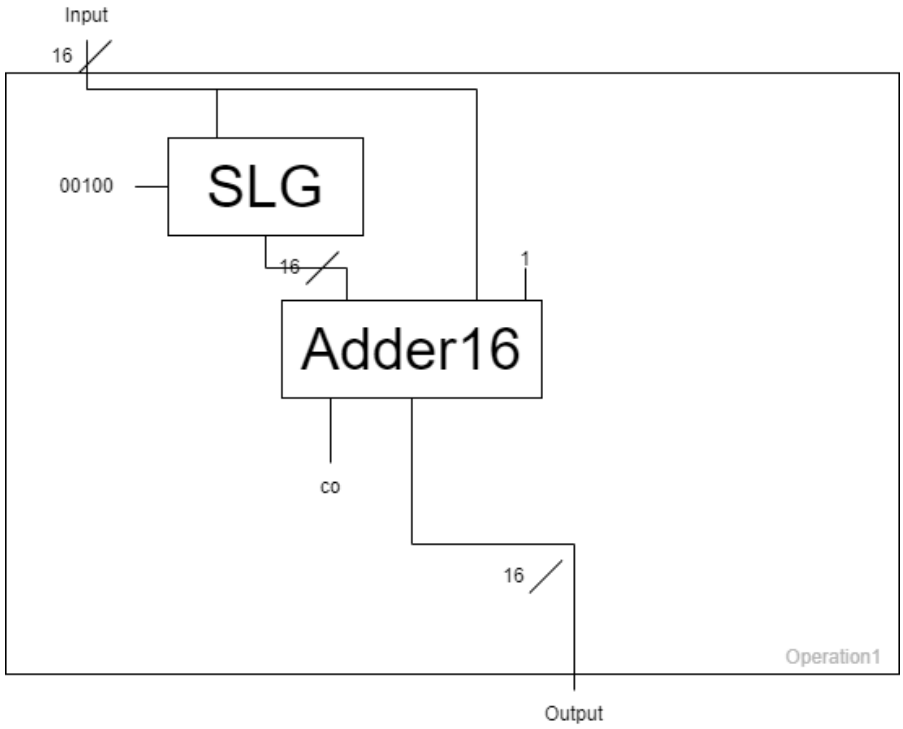


Aquí ocupamos operaciones *bitwise* para hacer uso del componente `XOR`, así como también dos instancias para manejar el caso de J.

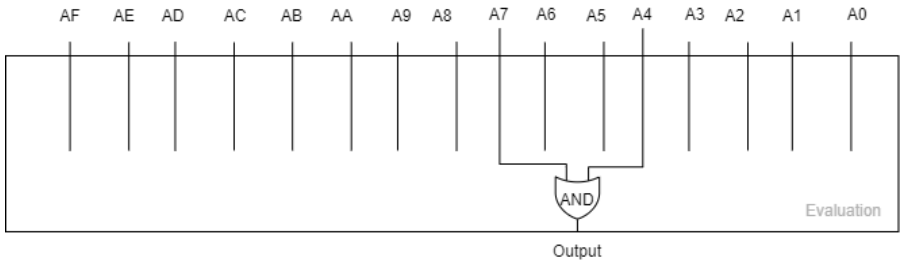
Para la implementación de *UpdateSeed* se tiene lo siguiente:



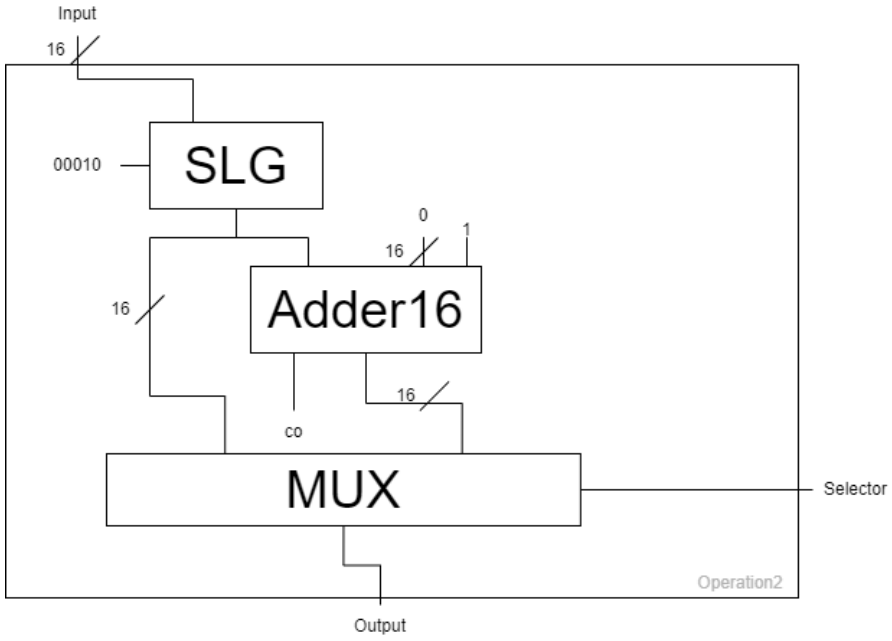
Para `Operation1` haremos uso del componente `SGL` descrito en la pregunta 2.c. para realizar cuatro *shifts lefts* (equivalente a multiplicar por 16) y luego usar el `Adder16` para completar la multiplicación de 17 ($16S + S = 17S$) (el 1 adicional lo sumamos con la señal C_{in} del sumador). El diagrama resultante es el siguiente:



Luego, `Evaluation` se encarga de hacer la comparación del bit 4 con el bit 7 por medio de la operación `AND`. Este resultado se utilizará como el bit de selección de un multiplexor dentro de `Operation2`. El circuito de esta comparación se describe en la siguiente figura:



Finalmente, `Operation2` ocupa un multiplexor y el componente `SNG` para modificar T de la forma pedida. El diagrama es el siguiente:

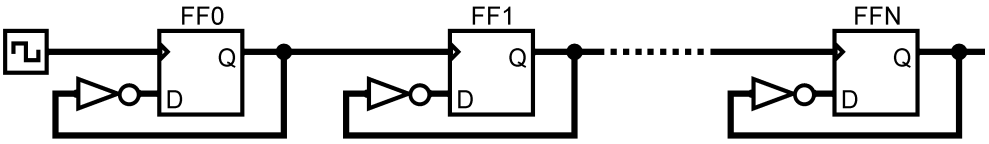


Distribución de puntaje

- **2 puntos** por correctitud de *GetRandom*.
- **2 puntos** por buen uso de compuertas *bitwise*.
- **2 puntos** por describir correctamente lo centrado en *Evaluation*.
- **3 puntos** por describir correctamente lo centrado en *Operation1*.
- **3 puntos** por describir correctamente lo centrado en *Operation2*.

Pregunta 3: Almacenamiento de datos (24 ptos.)

(a) (4 ptos.) En la siguiente figura, si la frecuencia del *clock* que entra al flip-flop FF0 es de F Hz, ¿cuál es la frecuencia del *clock* del flip-flop FFN? Debe incluir una explicación en su respuesta.



Solución: Si la frecuencia del *clock* del flip-flop FF0 es de F Hz, esto quiere decir que su periodo es de $\frac{1}{F}$ segundos. Luego, asumiendo que el flip-flop FF0 posee un estado inicial $Q = 0$, le toma $\frac{1}{F}$ segundos cambiar su estado a $Q = 1$ y otros $\frac{1}{F}$ segundos volver a su estado inicial $Q = 0$, lo que se traduce en un ciclo de periodo $\frac{2}{F}$ segundos, o bien una frecuencia de $\frac{F}{2}$ Hz. Se puede observar entonces que el *clock* de entrada del flip-flop FF1 será de $\frac{F}{2}$ Hz. Si se realiza este ejercicio con cada flip-flop de la secuencia, se puede observar que la frecuencia del *clock* **se va disminuyendo a la mitad con cada componente**. Entonces, de forma generalizada, se puede deducir que el *clock* de entrada del flip-flop FFN será de $\frac{F}{2^N}$ y su estado de salida Q tendrá una frecuencia de $\frac{F}{2^{N+1}}$.

Distribución de puntaje

- **2 puntos** si señala que la frecuencia del *clock* disminuye a la mitad por cada flip-flop de la secuencia.
- **2 puntos** si indica el valor final correcto, que puede ser la frecuencia del *clock* de entrada de FFN o de la salida de su estado Q (ambos se deducen de la misma forma).

(b) (4 ptos.) Una alternativa para bloquear el almacenamiento en un flip-flop D a partir de una señal de control es unir esta con la señal de flancos (*clock*) a través de una compuerta `AND`, tal como se ilustra en la figura. ¿Le parece esto una buena idea? Comente en base a las implicancias de esta implementación.

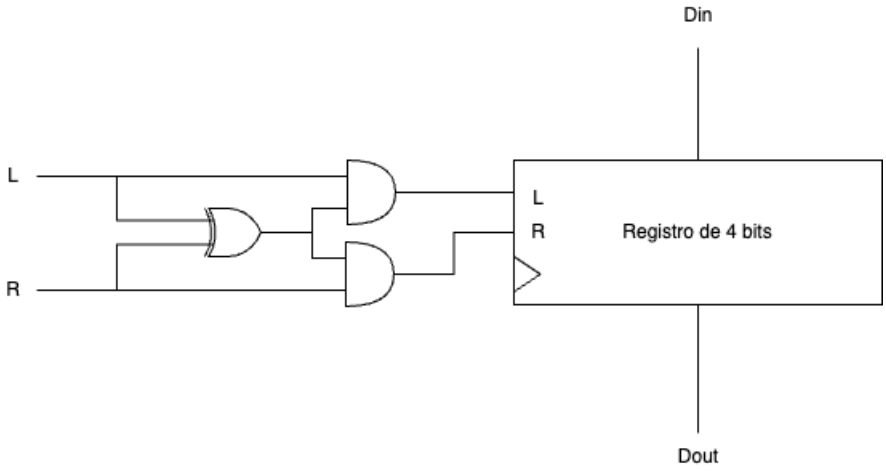
Solución: Esto **no es una buena idea**. Si bien cumple con bloquear el almacenamiento en el flip-flop, se pierde la capacidad de carga en los flancos de subida (motivo por el cual usamos este componente y no latches). Esto evita tener componentes de almacenamiento sincronizadas dentro de una microarquitectura y hace que su uso dificulte aún más la lógica de ejecución de instrucciones, por lo que es preferible controlar el almacenamiento a partir de la entrada D del flip-flop (como en los registros y contadores).

Distribución de puntaje

- **1 punto** si señala que, efectivamente, no es una buena idea.
- **3 puntos** por explicar que se pierde la capacidad de almacenamiento en flancos de subida, dificultando la sincronización dentro de la microarquitectura.

(c) (6 ptos.) A partir de la construcción de un registro de 4 bits con señales *Load* y *Reset*, realice un diagrama de esta componente con una modificación para que la combinación de señales `Load = 1, Reset = 1` **preserve el estado del registro**, es decir, que no sea sobreescrito con el bus de entrada ni reseteado.

Solución: Una opción directa para no modificar el comportamiento interno del registro es conectar las señales *Load* y *Reset* a una compuerta `XOR` (cuyo valor de verdad será 1 si, y solo si ambas señales son distintas) y hacer que la señal real que reciba el componente en sus entradas L y R esté condicionada por el resultado de la compuerta `XOR` con una compuerta `AND`. A continuación, un diagrama que ilustra lo anterior.



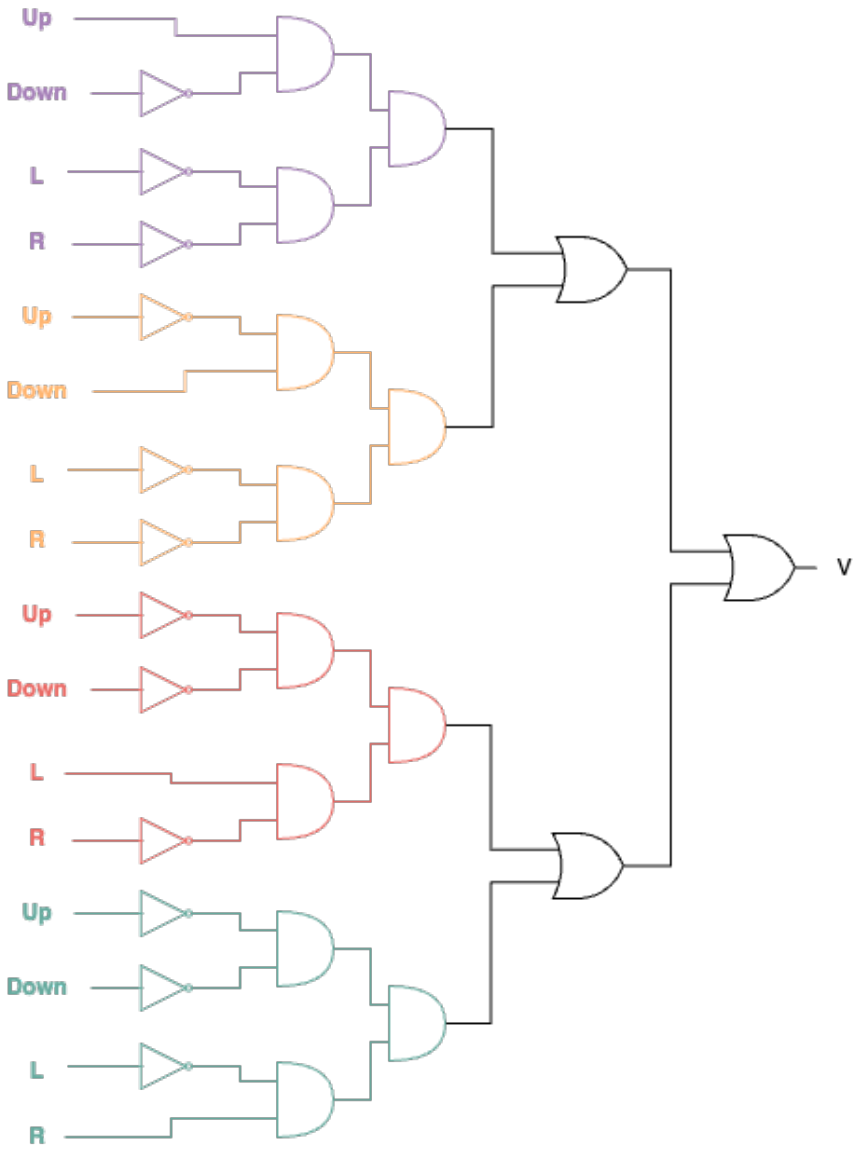
De esta forma, si $Load = Reset = 1$, entonces los *input* del registro serán $L = 0$ y $R = 0$, preservando el estado del registro. Si $Load = Reset = 0$, entonces también se tendrá $L = 0$ y $R = 0$, que es el comportamiento normal. En otro caso, los valores de las señales de entrada se mantendrán iguales y, por ende, harán lo esperado (cargar el dato de entrada o resetear el registro a 0).

Distribución de puntaje

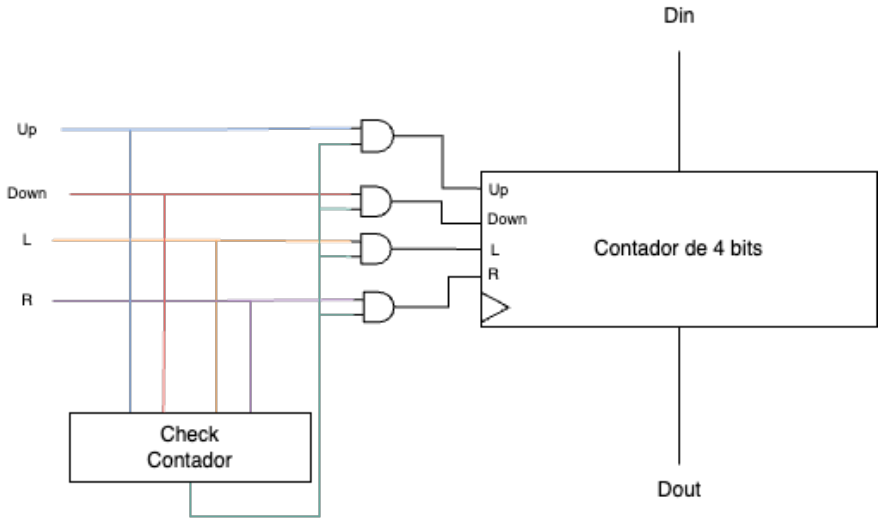
- **3 puntos** si la modificación funciona para el caso $Load = Reset = 1$.
- **3 puntos** si el resto de combinaciones *Load-Reset* no se ven alteradas.

(d) (10 ptos.) Extienda la idea de la pregunta anterior y elabore un diagrama del contador de 4 bits (con señales *Up* y *Down*) donde cualquier *input* de señales inválido preserve el estado del componente. Se consideran como *input* inválido cualquier combinación de señales que implique más de una modificación sobre el valor registrado. Por ejemplo, `Up = 1, Down = 1` es una combinación inválida ya que el registro no puede incrementar y decrementar su valor al mismo tiempo. Debe entregar una tabla de verdad para las combinaciones de señales *Up*, *Down*, *Load* y *Reset*.

Solución: Se puede implementar un circuito que verifique que cada caso se cumpla, esto es: $Up = 1$ y el resto 0; $Down = 1$ y el resto 0; $Load = 1$ y el resto 0; $Reset = 1$ y el resto 0. Un ejemplo a continuación.



Luego, al igual que con la protección de registro, podemos conectar el resultado de este circuito (que llamaremos "*Check* Contador" con cada señal del contador y una compuerta `AND`, habilitando las señales si, y solo si se cumple una de las cuatro condiciones.



Cabe destacar que si $Up = Down = Load = Reset = 0$, se seguirá cumpliendo lo pedido ya que todas las compuertas `AND` tendrán señal de salida 0 y, por ende, se mantendrá el resultado original de esta combinación de señales.

Distribución de puntaje

- **4 puntos** si se detectan los casos válidos dentro del circuito completo.
- **4 puntos** si se restringen los *input* del contador según los casos válidos.
- **2 puntos** si el caso $Up = Down = Load = Reset = 0$ sigue resguardando el estado del contador.