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

**IIC2343 — Arquitectura de Computadores (2020-2)**

# Ayudantía 2

**Lógica Digital (Circuitos)**

## 1 Circuitos Lógicos Combinacionales

## Ejercicio 1: I1 2020/1

Un laboratorio está haciendo una investigación en hámsteres a los cuales se les pusieron electrodos para determinar cuándo tienen hambre y/o sueño. Además el laboratorio cuenta con una bodega que puede o no tener comida. Al hámster se le podrá dar comida en su plato, **o** poner una cama para que duerma, de acuerdo a las siguientes condiciones:

**(I)** Si tiene hambre y queda comida en la bodega, entonces se le rellena su plato de comida.

**(II)** Si tiene hambre pero no queda comida en la bodega, se le coloca su cama, tenga o no sueño.

**(III)** Si tiene hambre y sueño y además queda comida en la bodega, sólo se le rellena su plato.

**(IV)** Si sólo tiene sueño, se le coloca su cama.

**(V)** Si no tiene hambre ni sueño, el hámster es feliz y no necesita nada.

**Debes:**

1. Crear la tabla de verdad que siga las condiciones planteadas. Utiliza las letras **H** para hambre, **S** para sueño, **Q** para "queda comida", **P** para "rellenar plato" y **B** para "colocar cama".

2. A partir de la tabla, crea la fórmula lógica para cada salida (**P** y **B**).

3. A partir de la fórmula, crea el circuito lógico cominacional.

## Solución:

Considerando las restricciones, tenemos que la tabla de verdad es:

| **H** | **S** | **Q** | **P** | **B** |
|---|---|---|---|---|
| 0 | 0 | 0 | **0** | **0** |
| 0 | 0 | 1 | **0** | **0** |
| 0 | 1 | 0 | **0** | **1** |
| 0 | 1 | 1 | **0** | **1** |
| 1 | 0 | 0 | **0** | **1** |
| 1 | 0 | 1 | **1** | **0** |
| 1 | 1 | 0 | **0** | **1** |
| 1 | 1 | 1 | **1** | **0** |

Ahora para formar las fórmulas lógicas, nos fijamos en en las combinaciones de variables (o sus negaciones) que mediante **ANDs** hacen que la salida sea 1 y si hay más de un caso las unimos mediante **ORs**.

Para **P** tenemos que:

$P \equiv (H \wedge \neg S \wedge Q) \vee (H \wedge S \wedge Q)$

Consideremos ahora el axioma de lógica proposicional que establece que:

$(A \wedge \neg B) \vee (A \wedge B) \equiv A$

Sea, entonces, $A = (H \wedge Q)$

$P \equiv (A \wedge \neg S) \vee (A \wedge S) \equiv A \equiv (H \wedge Q)$

Por otro lado, para **B** tenemos que:

$B \equiv (H \wedge \neg S \wedge \neg Q) \vee (H \wedge S \wedge \neg Q) \vee (\neg H \wedge S \wedge \neg Q) \vee (\neg H \wedge S \wedge Q)$

Dado el axioma anterior, se cumple que:

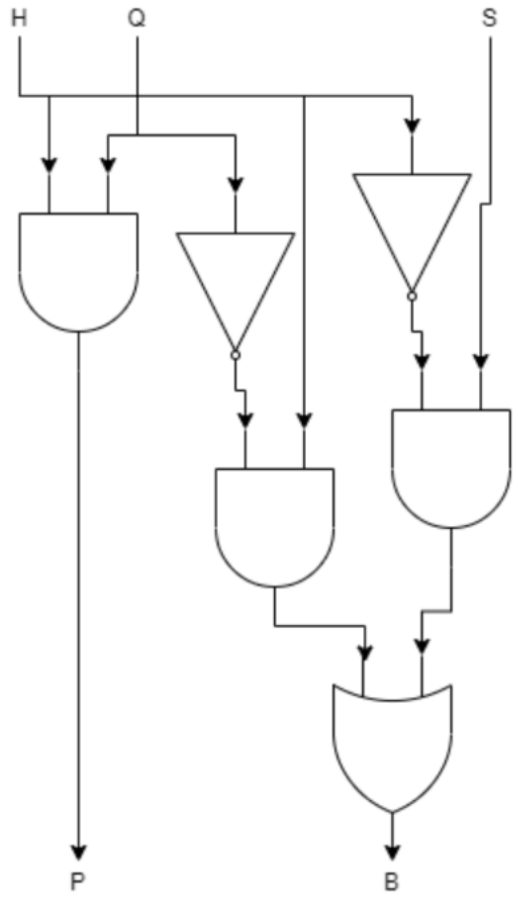
$(H \wedge \neg S \wedge \neg Q) \vee (H \wedge S \wedge \neg Q) \equiv (H \wedge \neg Q)$

$(\neg H \wedge S \wedge \neg Q) \vee (\neg H \wedge S \wedge Q) \equiv (\neg H \wedge S)$

Si juntamos ambas expresiones, tenemos que:

$B \equiv (H \wedge \neg Q) \vee (\neg H \wedge S)$

De esta forma ahora podemos representar el circuito tal como sigue:




# Ejercicio 2: I1 2013/1

1. Considere la siguiente fórmula en lógica booleana:

$$Y = (A \wedge B) \vee \neg(A \wedge C) \wedge \neg B$$

**(I)** Escriba la tabla de verdad correspondiente a la función anterior y diseñe un circuito lógico que la implemente.

**(II)** A partir de la tabla de verdad, re-escriba la función como conjunto de **ANDs** de las variables (o sus negaciones), combinados mediante **ORs.**

## Solución:

NOTA: Esta es una de varias soluciones posibles, ya que depende de como se simplifica y/o factoriza la expresión.

Primero simplifiquemos un poco la expresión:

$Y = (A \wedge B) \vee \neg(A \wedge C) \wedge \neg B$

Ocupando la ley de Morgan tenemos que:

$\neg(A \wedge C) \equiv (\neg A \vee \neg C)$

Ahora aplicando la ley distributiva, se cumple que:

$(\neg A \vee \neg C) \wedge \neg B \equiv (\neg B \wedge \neg A) \vee (\neg B \wedge \neg C)$

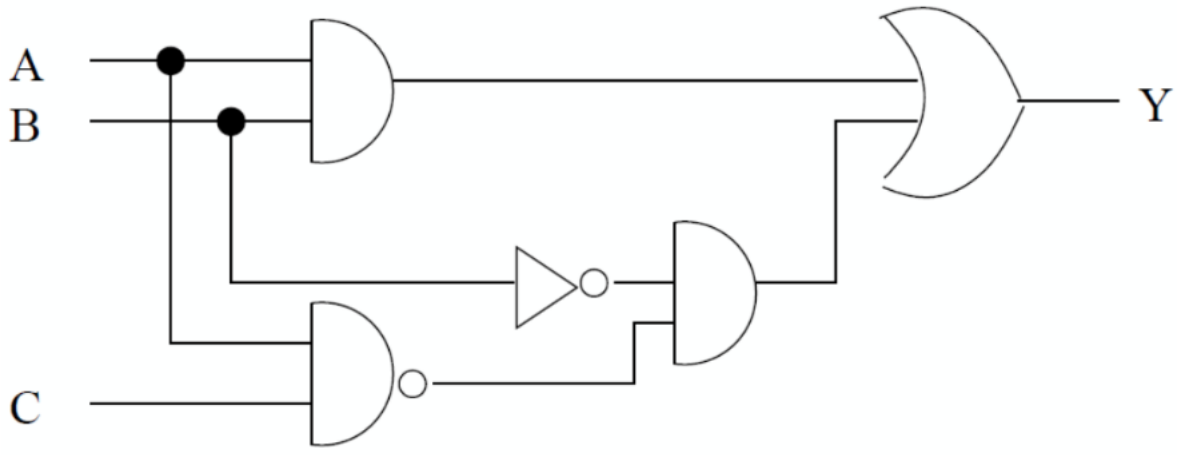
De esta forma, volvemos a aplicar la Ley de Morgan para "sacar el negativo para afuera del paréntesis", resultando:

$Y = (A \wedge B) \vee \neg(A \vee B) \vee \neg(B \vee C)$

Así, es más fácil reemplazar los posibles valores. Ahora construimos la tabla de verdad dada esta composición, obteniendo:

| $A$ | $B$ | $C$ | $A \wedge B$ | $\neg(A \vee B)$ | $\neg(B \vee C)$ | $Y$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | **1** |
| 0 | 0 | 1 | 0 | 1 | 0 | **1** |
| 0 | 1 | 0 | 0 | 0 | 0 | **0** |
| 0 | 1 | 1 | 0 | 0 | 0 | **0** |
| 1 | 0 | 0 | 0 | 0 | 1 | **1** |
| 1 | 0 | 1 | 0 | 0 | 0 | **0** |
| 1 | 1 | 0 | 1 | 0 | 0 | **1** |
| 1 | 1 | 1 | 1 | 0 | 0 | **1** |

Ahora solo falta agregar el diagrama que representa el circuito. En este caso, se agrega el circuito que representa la expresión inicial, es decir, $Y = (A \wedge B) \vee \neg(A \wedge C) \wedge \neg B$.
Para esto, se incluyen dos compuertas AND, una compuerta NAND, un NOT y un OR.



2. Si dos números enteros (con signo), A y B, son sumados (restados) mediante un **adder** (**subtractor**) de n bits, es posible que el resultado sea incorrecto: la suma de dos números positivos puede ser un número negativo, la suma de dos números negativos puede ser un positivo, etc.
Estos errores se conocen como *overflow*.
Para detectar automáticamente estas situaciones, se define una variable binaria **overflow** que toma el valor 1 cuando ocurre el error y 0 cuando no ocurre. Esta variable depende de otras cuatro variables binarias: $A_{n-1}$ (bit más significativo de A), $B_{n-1}$ (bit más significativo de B), $S_{n-1}$ (bit más significativo del resultado de la operación, $Op$ (operación: suma o resta).

   Escriba la tabla de verdad para la variable **overflow**, en base a los valores de las otras cuatro variables ya mencionadas.

## Solución:

Consideremos los casos de suma y de resta de forma separada.

Primero veamos la suma. Vamos a tener 3 casos:

**(I)** Si los dos números son de signo opuesto, NO puede existir overflow.

**(II)** Si los dos números son positivos, es decir, se cumple que $(A_{n-1} = 0, B_{n-1} = 0)$, sólo tendremos overflow si el resultado es negativo $(S_{n-1} = 1)$.

**(III)** Si los dos dos números son negativos, es decir, se cumple que $(A_{n-1} = 1, B_{n-1} = 1)$, sólo tendremos overflow si el resultado no es negativo $(S_{n-1} = 0)$.

Ahora hacemos el análisis del caso de la resta:

**(I)** Si los dos números son del mismo signo, NO puede existir overflow.

**(II)** Si $A$ es positivo $(A_{n-1} = 0)$ y $B$ es negativo $(B_{n-1} = 1)$, sólo tendremos overflow si el resultado es negativo $(S_{n-1} = 1)$.

**(III)** Si $A$ es negativo $(A_{n-1} = 1)$ y $B$ es positivo $(B_{n-1} = 0)$, sólo tendremos overflow si el resultado es positivo $(S_{n-1} = 0)$.

De esta forma hemos establecido las condiciones con las que rellenamos la tabla de verdad asociada a la variable **overflow**, y si consideramos la suma como $Op = 1$ y la resta como $Op = 0$, tenemos que:

| $Op$ | $A_{n-1}$ | $B_{n-1}$ | $S_{n-1}$ | **overflow** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 0 | 1 | **1** |
| 0 | 0 | 1 | 0 | **0** |
| 0 | 0 | 1 | 1 | **0** |
| 0 | 1 | 0 | 0 | **0** |
| 0 | 1 | 0 | 1 | **0** |
| 0 | 1 | 1 | 0 | **1** |
| 0 | 1 | 1 | 1 | **0** |
| 1 | 0 | 0 | 0 | **0** |
| 1 | 0 | 0 | 1 | **0** |
| 1 | 0 | 1 | 0 | **0** |
| 1 | 0 | 1 | 1 | **1** |
| 1 | 1 | 0 | 0 | **1** |
| 1 | 1 | 0 | 1 | **0** |
| 1 | 1 | 1 | 0 | **0** |
| 1 | 1 | 1 | 1 | **0** |

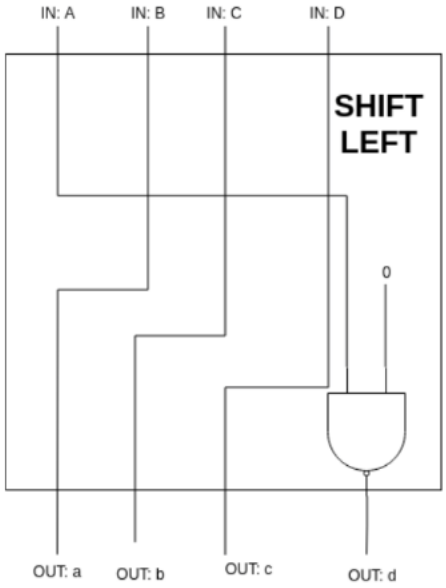
# Ejercicio 3: C1 2019/1

Diseñe un componente de 4 bits que según una señal de control multiplique o divida el valor de 4 bits de entrada por 2. (Sí, solo por 2).
**HINT:** Los multiplexores existen.

## Solución:

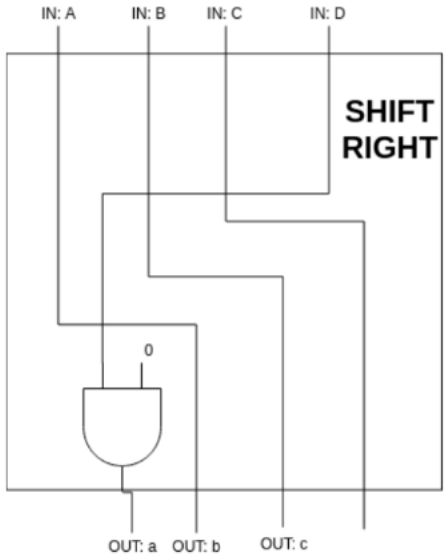
En primer lugar, tenemos que construir un componente que sea capaz de multiplicar por dos, lo que se puede conseguir con un shift-left (relleno con 0's a la derecha). (**OJO:** para la construcción de este componente se asumirá la misma cantidad de bits de input de que oputput, y no se hará uso del carry de la multiplicación ni de la división).

El componente shift-left se muestra a continuación:

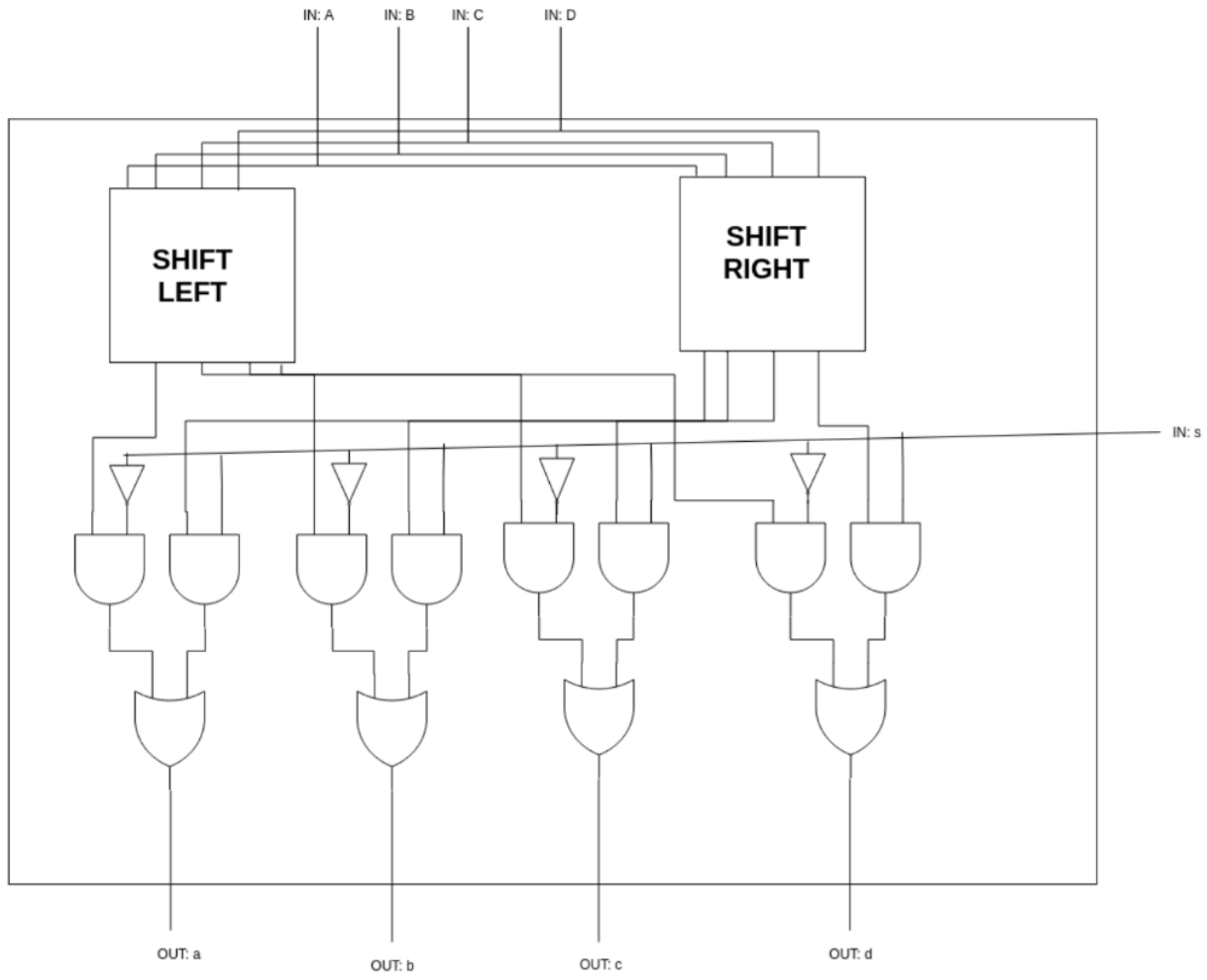



Tal como se puede ver, la lógica del circuito del shift left es que cada bit se mueve una posición a la izquierda y luego se rellena la posición menos significativa con un 0.

Ahora, necesitamos un componente que pueda dividir por 2. Para esto es fácil notar que necesitamos un shift right, ya que queremos que los bits se muevan una posición a la derecha y que ahora el bit más significativo se rellene con un 0. El shift right se muestra a continuación:




Por último, como queremos que el componente multiplique o divida por 2 dada una señal de control necesitamos incluir un multiplexor (MUX), de esta forma el diagrama queda de esta manera:




Aquí se puede notar que la lógica que sigue es que dado los 4 bits de input, se calcula tanto la posible solución si se multiplica como si se divide y cada una de estas opciones para cada uno de estos 4 bits va a parar a un multiplexor de 1 bit (4 en total), que dada la señal de control $S$, selecciona la opción que corresponde a la multiplicación o división. En este caso, si $S = 0$ se multiplica y si $S = 1$ se divide. Luego, para cada bit de output, se obtiene la solución correspondiente determinada por la señal $S$ seleccionada.