1. Ejercicio 1

Al representar números reales con una cantidad finita de símbolos, es importante tener en cuenta las limitaciones de la forma de representación elegida. Según la cantidad de símbolos y qué represente cada posición, quedarán definidos un número máximo representable, uno mínimo, y una mínima diferencia entre dos números consecutivos.

En este ejercicio estudiaremos el rango y la resolución de números en formato punto fijo binario. Estos dos parámetros se definen de la siguiente manera:

- Rango: diferencia entre el máximo y el mínimo valor representable, siendo el mínimo no el menor en módulo sino el menor número.
- Resolución: mínima diferencia entre dos números consecutivos.

Para definir estos parámetros en punto fijo, si se esta trabajando en signado o no signado es irrelevante, puesto que esto sólo aplicaria un *offset* de -2^N, donde N es el número de bits de la parte entera (incluso si es $N = 0$).

Llamando n al número de bits de la parte fraccionaria, resolución y rango quedan definidos como:

$$\text{Res} = 2^{-n} \tag{1}$$

$$\text{Ran} = 2^N - 2^{-n} = 2^N - \text{Res} \tag{2}$$

Estas expresiones son válidas para cualquier valor de n y N pertenecientes a $\mathbb{N}_0$. La única excepción es cuando ambos valen cero, en cuyo caso no tiene ningún sentido el análisis de un número de 0 dígitos binarios. Este caso se considera entonces no válido.

Se implementó en $C++$ que calcula rango y resolución con estas fórmulas, recibiendo por línea de comando tres parámetros: signo (1 o 0), N y n en ese orden. Sólo se consideran *inputs* válidos para estos dos últimos números enteros menores a 128 (puesto que números mayores resultaban en $2^N = \infty$ o $2^{-n} = 0$). Cualquier número con punto decimal se considera flotante, incluso si todos los dígitos después del mismo son 0.

El programa imprime rango y resolución si el *input* era adecuado, y `ERROR` si no. Para parsear la línea de comandos se reutilizó codigo escrito para la materia *22.08 - Algoritmos y estructura de datos*. En cuanto a la implementación de las ecuaciones 1 y 2, se utilizó la función est'andar de $C++$ de potencia, puesto que el *shifting* sólo funciona si ni 2^N ni 2^n exceden el tamaño de un *int*.

El programa respondió satisfactoriamente al siguiente banco de pruebas:

```
{"bad arguments", "dont", "write-", "909WORDS!"},
{"bad sign", "2", "4", "4"},
{"negative size int", "1", 4", "3"},
{"negative size frac", "0", "8", 1"},
{"negative sign", 55", "4", "5"},
{"float in size int", "0", "4.4", "3"},
{"float in size frac", "1", "8", "1.5"},
{"too large size int", "0", "128", "5"},
{"too large size frac", "1", "2", "128"},
{"zeros", "0", "0", "0"},
{"largest size", "1", "127", "127"},
{"zero int", "0", "0", "21"},
{"zero frac", "1", "12", "0"},
```

Figura 1: banco de pruebas.

También se comprobó que se devolviese `ERROR` para una cantidad no válida de argumentos.

2. Ejercicio 2

Se tiene la siguiente expresión en maxtérminos $f(d;c;b;a) = \prod(M_0; M_1; M_5; M_7; M_8; M_{10}; M_{14}; M_{15})$

De aquí se desprende la siguiente tabla de verdad, de la cual se derivan las expresiones completas en función de las entradas.

d	c	b	a	-	f	M_i
0	0	0	0	-	0	M_0
0	0	0	1	-	0	M_1
0	0	1	0	-	1	M_2
0	0	1	1	-	1	M_3
0	1	0	0	-	1	M_4
0	1	0	1	-	0	M_5
0	1	1	0	-	1	M_6
0	1	1	1	-	0	M_7
0	0	0	0	-	0	M_8
0	0	0	1	-	1	M_9
0	0	1	0	-	0	M_{10}
0	0	1	1	-	1	M_{11}
0	1	0	0	-	1	M_{12}
0	1	0	1	-	1	M_{13}
0	1	1	0	-	0	M_{14}
0	1	1	1	-	0	M_{15}

Expandiendo la expresión, f $=(d+c+b+a)\cdot(d+c+b+\overline{a})\cdot(d+\overline{c}+b+\overline{a})\cdot(d+\overline{c}+\overline{b}+\overline{a})\cdot(\overline{d}+c+b+a)\cdot(\overline{d}+c+\overline{b}+a)\cdot(\overline{d}+\overline{c}+\overline{b}+a)\cdot(\overline{d}+\overline{c}+\overline{b}+\overline{a})$

1. Simplificamos mediante el uso del álgebrea booleana:

   Agrupando a los maxtérminos anteriores de a dos y en orden, aplicando la propiedad 14.b del álgebra de Boole de la página del libro "Fundamentals of Digital Logic with Verilog Design" propuesto por la cátedra, la expresión queda simplificada a:

   f $=(d+c+b)\cdot(d+\overline{c}+\overline{a})\cdot(\overline{d}+c+a)\cdot(\overline{d}+\overline{c}+\overline{b})$

   Luego, aplicando los siguientes cambios de variable y la propiedad 17.a del libro:

$$\begin{cases} \mathrm{y} = \mathrm{c} + \mathrm{b} \\ \mathrm{z} = \overline{\mathrm{c}} + \overline{\mathrm{a}} \\ \mathrm{y}^{'} = \mathrm{c} + \mathrm{a} \\ \mathrm{z}^{'} = \overline{\mathrm{c}} + \overline{\mathrm{b}} \end{cases} \tag{3}$$

   f $=(d+y\cdot z)\cdot(\overline{d}+y^{'}\cdot z^{'})$

   Aplicamos propiedad distributiva y de nuevo la propiedad 17.a para llegar a:

   f $=d\cdot y^{'}\cdot z^{'}+\overline{d}\cdot y\cdot z+y^{'}\cdot z^{'}\cdot y\cdot z$

   f $=d\cdot y^{'}\cdot z^{'}+\overline{d}\cdot y\cdot z$

   Volviendo a las variables originales:

   f $=a\cdot d\cdot()+\overline{a}\overline{d}\cdot(c+b)+d\cdot c\cdot\overline{b}+\overline{d}\cdot\overline{c}\cdot b$

   que resulta ser la mínima expresión de f.

2. Simplificamos mediante el uso de mapas de Karnaugh:

$c,d \vert a,\ b$	00	01	11	10
00	M_0	M_2	M_3	M_1
01	M_8	M_{10}	M_{11}	M_9
11	M_{12}	M_{14}	M_{15}	M_{13}
10	M_4	M_6	M_7	M_5

- f

$c,d \vert a,\ b$	00	01	11	10
00	0	1	1	0
01	0	0	1	1
11	1	0	0	1
10	1	1	0	0

3. Ejercicio 3

3.1. Decoder

El decoder es un circuito lógico que permite convertir información binaria (n bits), a 2^n salidas . Se implementó un decoder de de 2 entradas y cuatro salidas, de la siguiente manera:

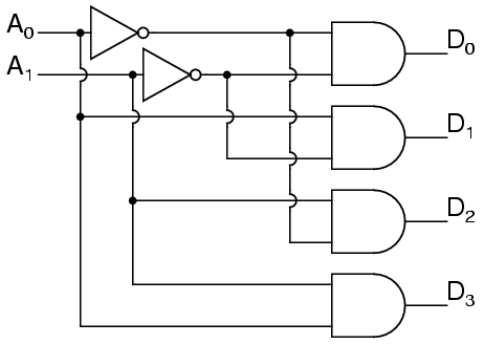

Figura 2: Implementasión Decoder 4 salidas

Cuya tabla de verdad es:

A_0	A_1	D_0	D_1	D_2	A_3
0	0	1	0	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	1	0	0	0	1

Cuadro 1: Tabla de verdad del Decoder

3.2. Mux

Los mux son circuitos lógicos que permiten poner a la salida una de las entradas. Se pidió la implementación de un mux de 4 entradas, para ellos se realizó un mux de 2 entradas y a partir de él, se construyó el de cuatro salidas.

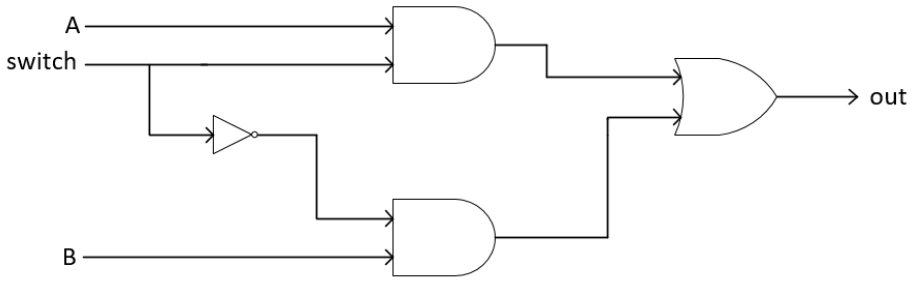

Figura 3: Implementasión mux 2 entradas

Switch	Out
0	B
1	A

Cuadro 2: Tabla de verdad del Decoder

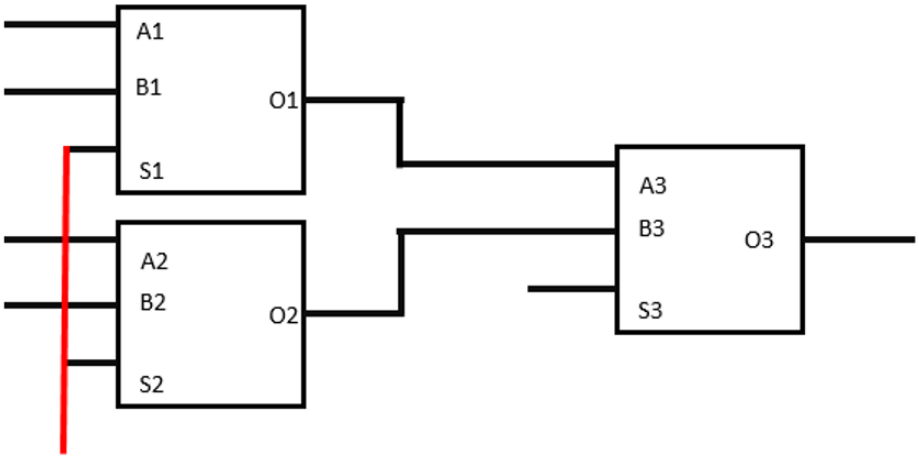

Figura 4: Implementasión mux 4 entradas, las cajas son mux de dos entradas. Donde A y B son entrada y S es la selección

S_1 y S_2	S_3	O_3
0	0	B_2
0	1	B_1
1	0	A_2
1	1	A_1

Cuadro 3: Tabla de verdad del Mux de 4 entradas

4. Ejercicio 4

Se desea realizar un circuito que convierta un número binario de 4 bits en su complemento a dos.

1. Expresamos el valor de cada bit de salida en función de los mintérminos de los bits de entrada.

   Sean b_3, b_2, b_1 y b_0 los bits de entrada, donde b_3 es el bit más significativo y b_0 el menos significativo.

   A su vez, sean y_3, y_2, y_1 e y_0 los bits de salida (complemento a dos de la entrada), donde y_3 es el bit más significativo e y_0 el menos significativo.

   Luego, se considera cada bit de salida por separado como una función f de los bit de entrada de forma tal que y_j = f(b_3; b_2; b_1; b_0), con j = 3,2,1,0. Cada y_j tendrá 16 posibles valores, que serán identificados como $y_{j,i}$, con i=0;1;...;15

   Es sabido que cada y_j puede ser vista como una suma (operación lógica OR) de los mintérminos de los bits de entrada. Así, $y_j = \sum_{i=0}^{n} m_{j,i} \cdot y_{j,i}$, con n = 15 por ser 16 las posibles entradas de 4 bits

y siendo $m_{j,i}$ el mintérmino correspondiente al i-ésimo valor posible del j-ésimo bit de salida.

b_3	b_2	b_1	b_0	-	y_3	y_2	y_1	y_0	
0	0	0	0	-	0	0	0	0	$m_{j,0}$
0	0	0	1	-	1	1	1	1	$m_{j,1}$
0	0	1	0	-	1	1	1	0	$m_{j,2}$
0	0	1	1	-	1	1	0	1	$m_{j,3}$
0	1	0	0	-	1	1	0	0	$m_{j,4}$
0	1	0	1	-	1	0	1	1	$m_{j,5}$
0	1	1	0	-	1	0	1	0	$m_{j,6}$
0	1	1	1	-	1	0	0	1	$m_{j,7}$
1	0	0	0	-	1	0	0	0	$m_{j,8}$
1	0	0	1	-	0	1	1	1	$m_{j,9}$
1	0	1	0	-	0	1	1	0	$m_{j,10}$
1	0	1	1	-	0	1	0	1	$m_{j,11}$
1	1	0	0	-	0	1	0	0	$m_{j,12}$
1	1	0	1	-	0	0	1	1	$m_{j,13}$
1	1	1	0	-	0	0	1	0	$m_{j,14}$
1	1	1	1	-	0	0	0	1	$m_{j,15}$

De la tabla anterior, se observa que:

$$\begin{cases} \mathrm{y_3 = m_{3,1} + m_{3,2} + m_{3,3} + m_{3,4} + m_{3,5} + m_{3,6} + m_{3,7} + m_{3,8}} \\ \mathrm{y_2 = m_{2,1} + m_{2,2} + m_{2,3} + m_{2,4} + m_{2,9} + m_{2,10} + m_{2,11} + m_{2,12}} \\ \mathrm{y_1 = m_{1,1} + m_{1,2} + m_{1,5} + m_{1,6} + m_{1,9} + m_{1,10} + m_{1,13} + m_{1,14}} \\ \mathrm{y_0 = m_{0,1} + m_{0,3} + m_{0,5} + m_{0,7} + m_{0,9} + m_{0,11} + m_{0,13} + m_{0,15}} \end{cases} \tag{4}$$

2. Expresamos el valor de cada bit de salida en forma simplificada. El método elegido para realizar la simplificación es el de mapas de Karnaugh:

   Así, para cada y_j, el mapa de Karnaugh de 4 variables/bits queda definido como:

   - y_j

b_2,b_3|b_0, b_1	00	01	11	10
00	$m_{j,0}$	$m_{j,2}$	$m_{j,3}$	$m_{j,1}$
01	$m_{j,8}$	$m_{j,10}$	$m_{j,11}$	$m_{j,9}$
11	$m_{j,12}$	$m_{j,14}$	$m_{j,15}$	$m_{j,13}$
10	$m_{j,4}$	$m_{j,6}$	$m_{j,7}$	$m_{j,5}$

   Los siguientes mapas aparecerán con los valores de sus mintérminos reemplazados y los grupos ya formados:

   - y_3

b_2,b_3|b_0, b_1	00	01	11	10
00	0	1	1	1
01	1	0	0	0
11	0	0	0	0
10	1	1	1	1

Figura 5: Mapa de Karnaugh para y_3

De este mapa se puede obtener $y_3 = \overline{b_3} \cdot b_2 + \overline{b_3} \cdot b_1 + \overline{b_3} \cdot b_0 + b_3 \cdot \overline{b_2 \cdot b_1 \cdot b_0}$

Así, $y_3 = \overline{b_3} \cdot (b_2 + b_1 + b_0) + b_3 \cdot \overline{b_2 \cdot b_1 \cdot b_0}$

- y_2

b_2,b_3|b_0, b_1	00	01	11	10
00	0	1	1	1
01	0	1	1	1
11	1	0	0	0
10	1	0	0	0

Figura 6: Mapa de Karnaugh para y_2

De este mapa se puede obtener $y_2 = \overline{b_2} \cdot b_1 + \overline{b_2} \cdot b_0 + b_2 \cdot \overline{b_1} \cdot \overline{b_0}$

Así, $y_2 = \overline{b_2} \cdot (b_1 + b_0) + b_3 + b_2 \cdot \overline{b_1} \cdot \overline{b_0}$

- y_1

b_2,b_3|b_0, b_1	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

Figura 7: Mapa de Karnaugh para y_1

De este mapa se puede obtener $y_1 = \overline{b_1} \cdot b_0 + \overline{b_0} \cdot b_1$

Así, y_1 resulta ser la xor entre b_1 y b_0.

- y_0

b_2,b_3|b_0, b_1	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	0	1	1
10	0	0	1	1

Figura 8: Mapa de Karnaugh para y_0

De este mapa se puede obtener $y_0 = b_0$

Así, el bit menos significativo de la entrada resulta ser el bit menos significativo de la salida (conexión directa).

5. Ejercicio 5

El formato BCD (de sus siglas en inglés *binary coded decimal*, es decir decimal codificado en binario) es aquél que representa un dígito decimal por cada *nybble* binario. Si bien esto es menos eficiente que la representación binaria en cuanto a que quedan 6 combinaciones de *bits* sin usar por cada 16 (lo que normalmente serían los números del 10 al 15), simplifica la conversión de binario a decimal. A su vez, permite mayor precisión en algunos casos: por ejemplo, el número $0{,}1_{10}$ no puede re presentarse de forma exacta en binario con ninguna cantidad de *bits*, mientras que sí se puede con una convención adecuada de BCD.

En este ejercicio se implementó un sumador de números BCD con dos entradas y dos salidas de un dígito cada una: dos números del 1 al 9 en la entrada, y las decenas y unidades de su suma en la salida. Cada dígito consta de cuatro *bits*.

Para expresar el resultado en BCD, se utilizó un registro extra de 5 *bits* para calcular la suma, de forma tal que el resultado siempre pueda quedar contenido en él (puesto que el máximo *input* es $9 + 9 = 18 < 31 = 2^5 - 1$, e incluso si tomamos casos no válidos $15 + 15 < 31$). Si este resultado obtenido es menor que 10, se pone un cero en las decenas y la suma en las unidades. De lo contrario, las decenas valen 1, y las unidades, la suma menos 10.

A su vez, puesto que tener input de valores entre el 10 y el 15 no es válido a pesar de que los 4 *bits* lo permiten, si esta situación ocurre se indica error seteando en 1 todos los *bits* de la salida, de forma tal que no pueda interpretarse como un número en BCD.

Se verificó que funcionara adecuadamente con un banco de pruebas que hace todas las combinaciones posibles de entradas, tanto válidas como no válidas.

6. Ejercicio 6