UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Práctica 5

Construcción de bloques estructurados por mapeo de puertos

ALUMNO

Ríos Lira, Gamaliel

PROFESOR

Flores Olvera, Vicente

ASIGNATURA

Laboratorio de Diseño Digital Moderno

GRUPO

1

8 de mayo de 2023

Índice

1. Objetivo

El alumno construirá funciones aritmético-lógicas estructuradas, utilizando bloques lógicos (sumadores, multiplexores y decodificadores) utilizando la plataforma *Quartus* y el *DLP*.

2. Materiales

Equipo del laboratorio

- Computadora

Software

- *Quartus Prime Lite* 18 (o superior)

3. Introducción

Multiplexor. Un multiplexor es un circuito combinacional que selecciona información binaria de una de muchas líneas de entrada y la envía a una sola línea de salida. La selección de una línea de entrada dada se controla con un conjunto de líneas de selección. Normalmente, hay 2^n líneas de entrada y n líneas de selección cuyas combinaciones de bits determinan cuál entrada se selecciona.

Decodificador. Un decodificador es un circuito combinacional que convierte información binaria de n líneas de entrada a un máximo de $2n$ líneas de salida distintas. Si la información codificada en n bits tiene combinaciones que no se usan, el decodificador podría tener menos de $2n$ salidas.

Full Adder. Un sumador completo es un circuito combinacional que forma la suma aritmética de tres bits. Tiene tres entradas y dos salidas. Dos de las variables de entrada, denotadas por x y y, representan los dos bits significativos que se sumarán. La tercera entrada, z, representa el acarreo previo.

Durante el desarrollo de esta práctica, se hace uso de estos componentes con la finalidad de solucionar algunos problemas asociados a la construicción de algunos circuitos digitales, tales como un sumador serial, una Unidad Aritmético-Lógica y un circuitor Desplazador-Rotador.

4. Desarrollo

4.1. Previo

Pregunta 1

Un circuito desplazador-rotador de 4 bits es un módulo combinacional que tiene como entrada una palabra de 4 bits $X = X_3X_2X_1X_0$, una palabra $Z = Z_3Z_2Z_1Z_0$ de 4 bits como salida, y tres entradas de control: s, d y r, que actúan como se indica a continuación.

Si $s = 0$, la salida refleja la entrada. Si $s = 1$, entonces la entrada es desplazada en 1 bit en la dirección indicada por d. Si $d = 0$ y $s = 1$, entonces el circuito desplaza la entrada 1 bit a la derecha. Si $d = 1$ y $s = 1$, la entrada es desplazada a la izquierda. El bit r indica si el circuito actúa como desplazador o como rotador. Es decir, si $sdr = 100$, la salida corresponde a la entrada desplazada a la derecha, y el nuevo bit $Z_3 = 0$. En cambio, si $sdr = 101$, el nuevo bit Z_3 corresponde al bit X_0. Asimismo, si $sdr = 110$, la salida corresponde a la entrada desplazada a la izquierda, y el nuevo bit Z_0 es 0. En cambio, si $sdr = 111$, el nuevo bit Z_0 corresponde al bit X_3.

*Diseñe este circuito usando **sólo multiplexores de 4 entradas**. Utilice tantos como encuentre necesario.*

Podemos ver que este problema se trata de un problema de selección de bits, por lo cual es posible llevarlo a cabo hacienod uso únicamente de multiplexores. Existen diversas aproximaciones, pero la más eficiente que se encontró fue la siguiente. Para cada bit z_n de salida, se tiene que existen los siguientes casos en función de los bits de entrada x_n:

s	d	r	z_3	z_2	z_1	z_0
0	0	0	x_3	x_2	x_1	x_0
0	0	1	x_3	x_2	x_1	x_0
0	1	0	x_3	x_2	x_1	x_0
0	1	1	x_3	x_2	x_1	x_0
1	0	0	0	x_3	x_2	x_1
1	0	1	x_0	x_3	x_2	x_1
1	1	0	x_2	x_1	x_0	0
1	1	1	x_2	x_1	x_0	x_3

Tabla 1: Funcionamiento (Desplazador-Rotador)

Los casos que devuelven 0 se encuentran en los extremos ya que únicamente son los únicos que tienen modificaciones con respecto al bit r. Casi todas las salidas se definen en función de s y d; a excepción de los casos que devuelven 0 directamente.

Primero, se crean utilizan dos multiplexores que se cierta forma trabajan de forma auxiliar. Se basan en los valores de entrada d y r.

- Para el primer caso, lo que se requiere es que devuelva 0 cuando $dr = 10$, de lo contrario,

que devuelva el valor x_3. Con esto, se selecciona el bit que irá hasta la izquierda dependiendo si se trata de una rotación o un desplazamiento. La salida se puede llamar k_0.

- $00 \rightarrow x_3$
- $01 \rightarrow x_3$
- $10 \rightarrow 0$
- $11 \rightarrow x_3$

- Para el segundo caso, lo que se requiere es que devuelva 0 cuando $dr = 00$, de lo contrario, que devuelva el valor x_0. Con esto, se selecciona el bit que irá hasta la izquierda dependiendo si se trata de una rotación o un desplazamiento. La salida se puede llamar k_3.

  - $00 \rightarrow 0$
  - $01 \rightarrow x_0$
  - $10 \rightarrow x_0$
  - $11 \rightarrow x_0$

Lo anterior debe ser combinado con un comportamiento adicional para que pueda funcionar de forma adecuada. k_0 es el bit correspondiente a z_0 en caso de que se tenga $sd = 11$, mientras que k_3 es el bit correspondiente a z_3 en caso de que se tenga $sd = 10$. Hace falta tomar en cuenta las demás combinaciones, pero es posible simplificar un poco la tabla anterior con ayuda de los valores auxiliares (k_0 y k_3) que calculamos previamente.

s	d	r	z_3	z_2	z_1	z_0
0	0	0	x_3	x_2	x_1	x_0
0	0	1	x_3	x_2	x_1	x_0
0	1	0	x_3	x_2	x_1	x_0
0	1	1	x_3	x_2	x_1	x_0
1	0	0	k_3	x_3	x_2	x_1
1	0	1	k_3	x_3	x_2	x_1
1	1	0	x_2	x_1	x_0	k_0
1	1	1	x_2	x_1	x_0	k_0

Lo siguiente es similar para todos los términos. Dependiendo del valor de los bits s y d, se selecciona el bit indicado por la tabla anterior. Se tienen entonces otros cuatro multiplexores.

- Para el caso de z_0, se tiene que:

  - $00 \rightarrow x_0$
  - $01 \rightarrow x_0$
  - $10 \rightarrow x_1$
  - $11 \rightarrow k_0$

- Para el caso de z_1, se tiene que:
  - $00 \rightarrow x_1$
  - $01 \rightarrow x_1$
  - $10 \rightarrow x_2$
  - $11 \rightarrow x_0$
- Para el caso de z_2, se tiene que:
  - $00 \rightarrow x_2$
  - $01 \rightarrow x_2$
  - $10 \rightarrow x_3$
  - $11 \rightarrow x_1$
- Para el caso de z_3, se tiene que:
  - $00 \rightarrow x_3$
  - $01 \rightarrow x_3$
  - $10 \rightarrow k_3$
  - $11 \rightarrow x_2$

Combinando los comportamientos anteriores se obtiene el circuito deseado, tal como se muestra en la Figura 1. En donde en total se hizo uso de **seis multiplexores**.

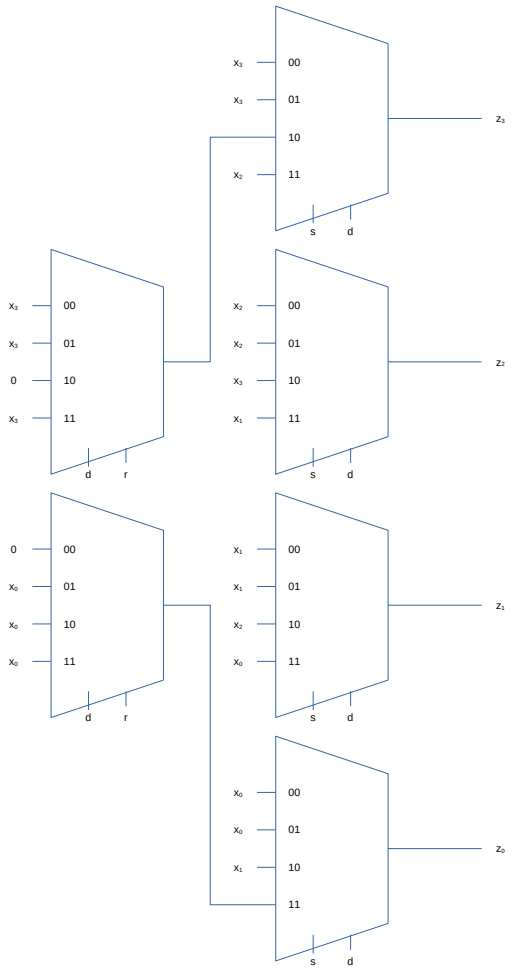


Figura 1: Circuito desplazador-rotador (Previo)

Pregunta 2

Construir un convertidor a siete segmentos de su respectivo número de cuenta

Para realizar este convertidor, se tienen que considerar únicamente dígitos decimales (0 al 9).

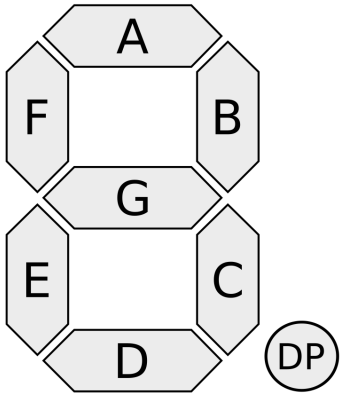


Figura 2: Diagrama de un display de 7 segmentos

Considerando el diagrama establecido en la Figura 2, se puede establecer la siguiente tabla de verdad. En la cual se toma una palabra de 4 bits y se devuelve una palabra de 8 bits (los 7 segmentos y el DP).

x_3	x_2	x_1	x_0	p	g	f	e	d	c	b	a
0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	1
0	0	1	0	1	0	1	0	0	1	0	0
0	0	1	1	1	0	1	1	0	0	0	0
0	1	0	0	1	0	0	1	1	0	0	1
0	1	0	1	1	0	0	1	0	0	1	0
0	1	1	0	1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1	1	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	1	0	0	0	0

Tabla 2: Funcionamiento (Display de 7 segmentos)

Con lo cual el decodificador nesesario para convertir un número de cuenta a 7 segmentos queda implementado.

4.2. Parte A

Diseñar mediante programación estructurada la implementación de un sumador binario de dos palabras de 4 bits.

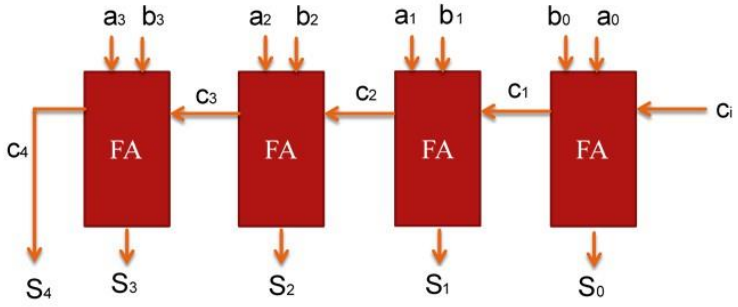


Figura 3: Circuito de ejemplo (Parte A)

Para comenzar, primero se debe implementar el componente *Full Adder (FA)*. El cual se comporta según la siguiente tabla de verdad.

a	b	c	s	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabla 3: Tabla de verdad de un *Full Adder*

Obteniendo las expresiones para el cálculo de s y c_o, se tiene que:

$$\begin{aligned} s(abc) &= \sum_m (1,2,4,7) \\ &= \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc \\ &= \bar{a}(\bar{b}c + b\bar{c}) + a(\bar{b}\bar{c} + bc) \\ &= \bar{a}(b \oplus c) + a\overline{(b \oplus c)} \\ &= a \oplus (b \oplus c) \end{aligned}$$

$$\boxed{\therefore s(abc) = a \oplus (b \oplus c)}$$

Por otra parte, se tiene que:

$$\begin{aligned} c_o(abc) &= \sum_m (3,5,6,7) \\ &= \bar{a}bc + a\bar{b}c + ab\bar{c} + abc \\ &= c(\bar{a}b + a\bar{b}) + ab(\cancel{\bar{c} + c}) \\ &= c(a \oplus b) + ab \end{aligned}$$

$$\boxed{\therefore c_o(abc) = c(a \oplus b) + ab}$$

De esta forma, a través del uso del lenguaje *VHDL*, se implementó la entidad `FA`, la cual toma un vector de entrada `ent` y un vector de salida `res`.

```
-- Implementacion: Full Adder (Parte A)
entity FA is
  port(
    ent :in   std_logic_vector(2 downto 0);
    res :out  std_logic_vector(1 downto 0)
  );
end entity;

architecture Behave of FA is
  signal a, b, ci :std_logic;
begin
  a   <= ent(2);
  b   <= ent(1);
  ci  <= ent(0);
  res <= (
    a xor b xor ci,
    (ci and (a xor b)) or (a and b)
  );
end;
```

El vector de entrada de `ent` se representa de la siguiente forma:

- $a \rightarrow ent(2)$
- $b \rightarrow ent(1)$
- $c \rightarrow ent(0)$

Mientras que las salidas `res` se representa de la siguiente forma:

- $s \rightarrow res(1)$
- $c_o \rightarrow res(0)$

Posteriormente, se hizo la integración completa del sumador binario de dos palabras de 4 bits en una nueva entidad, la cual se llamó `Adder4`. Es importante considerar que se debe de realizar la operación en forma de cadena.

- Para el bit 0, se toma como entrada $a(0)$, $b(0)$ y ci. Se devuelve el primer bit del resultado $s(0)$ y un bit de acarreo, el cual se almacena dentro de una señal $c(0)$.
- Para el bit 1, se toma como entrada $a(1)$, $b(1)$ y $c(0)$. Se devuelve el segundo bit del resultado $s(1)$ y un bit de acarreo, el cual se almacena dentro de una señal $c(1)$.
- Para el bit 1, se toma como entrada $a(2)$, $b(2)$ y $c(1)$. Se devuelve el segundo bit del resultado $s(2)$ y un bit de acarreo, el cual se almacena dentro de una señal $c(2)$.
- Para el bit 1, se toma como entrada $a(3)$, $b(3)$ y $c(2)$. Se devuelve el segundo bit del resultado $s(3)$ y un bit de acarreo, a su vez conforma el bit más significativo del resultado, que se almacena directamente en $s(4)$.

Lo anterior se resume en la siguiente entidad:

```
-- Implementacion: Suamador 4 bits (Parte A)
entity Adder4 is
  port(
    a,b   :in std_logic_vector(3 downto 0);
    ci    :in std_logic;
    s     :out std_logic_vector(4 downto 0)
  );
end;

architecture Behave of Adder4 is
  component FA is
    port(
      ent :in std_logic_vector(2 downto 0);
      res :out  std_logic_vector(1 downto 0)
    );
  end component;
  signal c :std_logic_vector(2 downto 0);
begin
  adder_0 :FA port map(
    ent => (a(0), b(0), ci),
    res(1) => s(0),
    res(0) => c(0)
  );
  adder_1 :FA port map(
    ent => (a(1), b(1), c(0)),
    res(1) => s(1),
    res(0) => c(1)
  );
  adder_2 :FA port map(
    ent => (a(2), b(2), c(1)),
    res(1) => s(2),
    res(0) => c(2)
  );
  adder_3 :FA port map(
    ent => (a(3), b(3), c(2)),
    res(1) => s(3),
    res(0) => s(4)
  );
```

```
39 end;
```

Ejecución en Quartus II

Para comprobar el funcionamiento del circuito implementado, se puede realizar una simulación a través de un archivo *University Program VWF*. Se realizaron 16 casos en los que los valores de a se generaron secuencialmente, mientras que los valores de b se generaron de forma aleatoria. Los primeros ocho casos de prueba tienen $c_i = 0$, y los demás tienen $c_i = 1$. El resultado es el siguiente:

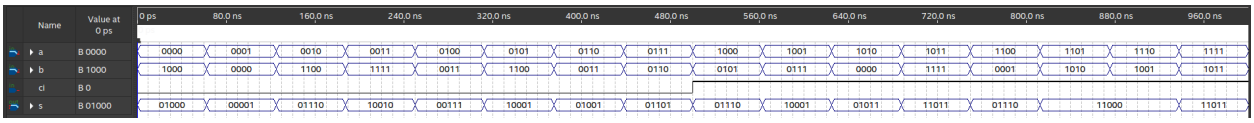


Figura 4: Simulación del circuito (Parte A)

En la Figura 4 se puede comprobar el correcto funcionamiento del sumador.

Es importante recalcar el uso de las nuevas herramientas que vimos en clase. Primero, se puede ver la utilización de una entidad dentro de otra entidad (como es el caso de `FA` dentro de `Addder4`). Fue importante utilizar la declaración del componente *FA* dentro de `Adder4` (similar a una importación en otros lenguajes de programación). Se deben crear varias instanicias de este mismo tipo (`adder_0`, `adder_1`, `adder_2` y `adder_3`), cada una de las cuales se encarga de calcular un bit en específico. Para poder declarar cada una de ellas de forma adecuada, es necesario hacer un mapeo de puertos a través de la instrucción `port map`. En la cual se tienen que asignar los valores de las entradas y también, a través de este, se asignan las salidas.

4.3. Parte B

Diseñar mediante la programación estructurada la implementación de una unidad lógica-aritmética binaria de dos palabras de 4 bits, como se muestra en la siguiente tabla:

s_1	s_0	c_i	Aritmético ($M = 0$)	Lógico ($M = 1$)
0	0	0	A	$A \cdot B$
0	0	1	$A + 1$	$A + B$
0	1	0	$A + B$	$\bar{A}$
0	1	1	$A + B + 1$	$A \oplus B$
1	0	0	$A + \bar{B}$	$\overline{A + B}$
1	0	1	$A + \bar{B} + 1(A - B)$	$\overline{A \cdot B}$
1	1	0	$A - 1$	B
1	1	1	A	$\bar{A} \cdot B$

Tabla 4: Tabla de verdad del funcionamiento de la ALU

Para comenzar esta sección, fue importante hacer uso de las siguientes tablas (poroporcionadas en el planteamiento de la práctica) para llevar a cabo la impelemntación de la unidad aritmética.

Selector			**Salida Ni**	**Función**	**Descripción**
s_1	s_0	c_i	N_i	F	
0	0	0	0	A	Transferir A
0	0	1	0	$A + 1$	Incrementar A
0	1	0	B	$A + B$	Sumar o agregar B a A
0	1	1	B	$A + B + 1$	Suma con acarreo
1	0	0	0	$A + \bar{B}$	Agregar $C_1(B)$ a A
1	0	1	0	$A + \bar{B} + 1$	Agregar $C_2(B)$ a A
1	1	0	Todos 1	$A - 1$	Decrementar A
1	1	1	Todos 1	A	Transferir A

Tabla 5: Tabla de verdad del funcionamiento de la UL

Simplificando un poco la información brindada por la tabla anterior,se tiene que:

s_1	s_0	N_i
0	0	0
0	1	B
1	0	$\bar{B}$
1	1	1

Con lo cual se tiene la expresión:

$$\begin{aligned}
N_i(s_1s_0) &= \cancel{\bar{s_1}\bar{s_0}(0)} + \bar{s_1}s_0(B) + s_1\bar{s_0}(\bar{B}) + s_1s_0(1) \\
&= \bar{s_1}s_0B + s_1\bar{s_0}\bar{B} + s_1s_0 \\
&= (\bar{s_1}s_0B + s_1s_0) + (s_1\bar{s_0}\bar{B} + s_1s_0) \\
&= s_0(\bar{s_1}B + s_1) + s_1(\bar{s_0}\bar{B} + s_0) \\
&= s_0\cancel{(\bar{s_1} + s_1)}(B + s_1) + s_1\cancel{(\bar{s_0} + s_0)} \\
&= s_0(B + s_1) + s_1(\bar{B} + s_0) \\
&= s_0B + s_1\bar{B} + s_0s_1 \\
&= s_0B + s_1\bar{B} + s_0s_1(B + \bar{B}) \\
&= s_0B + s_1\bar{B} + s_0s_1B + s_0s_1\bar{B} \\
&= s_0B(s_1 + 1) + s_1\bar{B}(s_0 + 1) \\
&= s_0B + s_1\bar{B}
\end{aligned}$$

$$\boxed{\therefore N_i(s_1s_0) = s_0B + s_1\bar{B}}$$

Entonces, se contruyó la Unidad Aritmética haciendo uso de cuatro *Full Adder*, para los cuales se operó por un lado con a_i y por el otro con el resultado de $N_i = s_0b_i + s_1\bar{b_i}$, tal como se muestra en el siguiente fragmento de código.

```
-- Implementacion de la Unidad Aritmetica
library ieee;
use ieee.std_logic_1164.all;

entity UA is
  port(
    a,b   :in std_logic_vector(3 downto 0);
    sel   :in std_logic_vector(2 downto 0);
    res   :out  std_logic_vector(4 downto 0)
  );
end entity;

architecture Behave of UA is
  component FA is
    port(
      ent :in std_logic_vector(2 downto 0);
      res :out  std_logic_vector(1 downto 0)
    );
  end component;

  signal c :std_logic_vector(2 downto 0);
  signal n  :std_logic_vector(3 downto 0);
begin
  n <= (
    (sel(1) and b(3)) or (sel(2) and not b(3)),
    (sel(1) and b(2)) or (sel(2) and not b(2)),
    (sel(1) and b(1)) or (sel(2) and not b(1)),
    (sel(1) and b(0)) or (sel(2) and not b(0))
  );
```

```
30
31    op_0  :FA port map(
32      ent => (a(0), n(0), sel(0)),
33      res(1) => res(0),
34      res(0) => c(0)
35    );
36
37    op_1  :FA port map(
38      ent => (a(1), n(1), c(0)),
39      res(1) => res(1),
40      res(0) => c(1)
41    );
42
43    op_2  :FA port map(
44      ent => (a(2), n(2), c(1)),
45      res(1) => res(2),
46      res(0) => c(2)
47    );
48
49    op_3  :FA port map(
50      ent => (a(3), n(3), c(2)),
51      res(1) => res(3),
52      res(0) => res(4)
53    );
54  end;
```

Tal como se puede ver, se tienen las entradas a y b (con las cuales se operará) y una configuración de selección $sel(3)$, la cual es un vector que almacena la entrada de selección mostrada en la Tabla 5. Se hace uso de una señan n para calcular los valores de las respectivas N_i y posteriormente se hace udo de los **Full Adder** op_0, op_1, op_2 y op_3 para conformar el resultado de la operación seleccionada, el cual se almacena en una palabra de cinco bits: $res(5)$.

Es importante mencionar que la entrada de selección ya incluye el bit de acarrero de entrada, el cual será el bit menos significativo, acorde a lo establecido en la Tabla 5.

Por otra parte, la implementación de la Unidad Lógica fue sumamente más sencilla ya que cada caso se traduce a una operación lógica en VHDL. Nada más se fijaron los casos basándonos en la Tabla 4 a través de una estructura *with-select*.

```
1  -- Implementacion de la Unidad Logica
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity UL is
6    port(
7      a,b   :in std_logic_vector(3 downto 0);
8      sel   :in std_logic_vector(2 downto 0);
9      res   :out  std_logic_vector(3 downto 0)
10   );
11 end entity;
12
13 architecture Behave of UL is
14 begin
```

```
  with sel select
  res <=
    a and b       when "000",
    a or b        when "001",
    not a         when "010",
    a xor b       when "011",
    not (a or b)  when "100",
    not (a and b) when "101",
    b             when "110",
    not a and b   when "111";
end;
```

Las entradas se manejan de igual forma que en la *Unidad Aritmética*. Sin embargo, para este caso se tiene que la salida no corresponde a 5 bits, sin o a 4 bits. Esto debido a que como tal las operaciones lógicas no generan un acarreo. Por cada par de bits que se opera, sólo existirá un único bit de salida.

Para cumplir con el objetivo de la práctica, fue necesario juntar los módulos anteriores dentro de una misma entidad, las cual se denominó *ALU* (Unidad Aritmético-Lógica por sus siglas en inglés). Dentro de ella se creó una entidad de tipo *UL*, la cual tiene una salida g de 4 bits, y una unidad *UA*, la cual tiene una salida f de 5 bits. Las entradas del sistema son dos palabras de 4 bits (a y b), junto con cuatro entradas de selección: las mismas tres mencionadas en las secciones anteriores (s_1, s_0, c_i) más una adicional para seleccionar el tipo de operación a utilizar (m).

Sumado a lo anterior se tuv que implementar una estructura similar a un multiplexor pero que permitiera seleccionar una palabra de entre un conjunto de dos palabras de 5 bits. Este se implementó de forma muy sencilla haciendo uso de la estructura *with-select* que ya se ha utilizado anteriormente.

```
-- Multplexor 2:1 de 5 bits
library ieee;
use ieee.std_logic_1164.all;

entity Mux2 is
  port(
    f :in std_logic_vector(4 downto 0);
    g :in std_logic_vector(4 downto 0);
    s :in std_logic;
    z :out std_logic_vector(4 downto 0)
  );
end entity;

architecture Behave of Mux2 is
begin
  with s select
  z <=
    f when '0',
    g when '1';
end;
```

La integración de todo lo descrito anteriormente dentro de la Unidad Arirtmético-Lógica se muestra a continuación:

```vhdl
-- Implementacion de la Unidad Aritmetico-Logica
library ieee;
use ieee.std_logic_1164.all;

entity ALU is
  port(
    a,b :in std_logic_vector(3 downto 0);
    sel :in std_logic_vector(2 downto 0);
    m   :in std_logic;
    res :out  std_logic_vector(4 downto 0)
  );
end entity;

architecture Behave of ALU is
  component UA is
    port(
      a,b   :in std_logic_vector(3 downto 0);
      sel   :in std_logic_vector(2 downto 0);
      res   :out  std_logic_vector(4 downto 0)
    );
  end component;

  component UL is
    port(
      a,b   :in std_logic_vector(3 downto 0);
      sel   :in std_logic_vector(2 downto 0);
      res   :out  std_logic_vector(3 downto 0)
    );
  end component;

  component Mux2 is
    port(
      f :in std_logic_vector(4 downto 0);
      g :in std_logic_vector(4 downto 0);
      s :in std_logic;
      z :out std_logic_vector(4 downto 0)
    );
  end component;

  signal f,g    :std_logic_vector(4 downto 0);
begin
  l_unit  : UL port map(
    a => a,
    b => b,
    sel => sel,
    res(0) => f(0),
    res(1) => f(1),
    res(2) => f(2),
    res(3) => f(3)
  );

  f(4) <= '0';

  a_unit  : UA port map(
```

```
    a => a,
    b => b,
    sel => sel,
    res => g
  );

  mux   : Mux2 port map(
    f => f,
    g => g,
    s => m,
    z => res
  );
end;
```

Además, el siguiente requerimiento del problema era mostrar el resultado en un display de 7 segmentos. Para ello, se creó el componente *Deco7Seg*, el cual se fijó de la siguiente forma:

```
-- Implementacion de un decodificador de 7 segmentos
library ieee;
use ieee.std_logic_1164.all;

entity Deco7Seg is
  port (
    num :in std_logic_vector(3 downto 0);
    sal :out std_logic_vector(7 downto 0)
  );
end entity;

architecture Behave of Deco7Seg is
begin
  with num select
  sal <=  "11000000" when "0000",
          "11111001" when "0001",
          "10100100" when "0010",
          "10110000" when "0011",
          "10011001" when "0100",
          "10010010" when "0101",
          "10000010" when "0110",
          "11111000" when "0111",
          "10000000" when "1000",
          "10010000" when "1001",
          "10001000" when "1010",
          "10000011" when "1011",
          "10100111" when "1100",
          "10100001" when "1101",
          "10000100" when "1110",
          "10001110" when "1111";
end;
```

Que de forma general toma un número en binario a 4 bits y devuelve un vector de 8 bits con la siguiente estructura:

$$(p, g, f, e, d, c, b, a)$$

Finalmente, la entidad que abarca todo el funcionamiento es *ParteB*, la cual toma dos palabras de

entrada (a y b) de 4 bits; y a través de un selector de 3 bits (sel) y la entrada m calculan un valor con la ALU que revuelve una respuesta z de 5 bits. La respuesta se divide en dos palabras de 4 bits (la parte alta y la parte alta). De esta forma, se puede conformar la salida en hexadecimal haciendo uso de dos display por separado; cada uno de ellos calculado por separado a través del componente *Deco7Seg*. El código de esta entidad se muestra a continuación.

```
-- Implementacion de la sulucion completa
library ieee;
use ieee.std_logic_1164.all;

entity ParteB is
  port (
    x, y  :in std_logic_vector(3 downto 0);
    sel :in std_logic_vector(2 downto 0);
    m   :in std_logic;
    z_bcd_l :out std_logic_vector(7 downto 0);
    z_bcd_h :out std_logic_vector(7 downto 0)
  );
end entity;

architecture Behave of ParteB is
  component ALU is
    port(
      a,b :in std_logic_vector(3 downto 0);
      sel :in std_logic_vector(2 downto 0);
      m   :in std_logic;
      res :out  std_logic_vector(4 downto 0)
    );
  end component;

  component Deco7Seg is
    port (
      num :in std_logic_vector(3 downto 0);
      sal :out std_logic_vector(7 downto 0)
    );
  end component;

  signal z  :std_logic_vector(4 downto 0);
begin

  alu_inst  :ALU port map(
    a => x,
    b => y,
    sel => sel,
    m => m,
    res => z
  );

  deco_l: Deco7Seg port map(
    num => z(3 downto 0),
    sal => z_bcd_l
  );

  deco_h: Deco7Seg port map(
```

```
49      num => ('0', '0', '0', z(4)),
50      sal => z_bcd_h
51    );
52  end;
```

Ejecución en Quartus II

Para facilitar la visualización del funcionamiento del circuito, se simuló úniicamente la sección referente a la ALU, y posteriormente el código generado se cargó a la tarjeta *DE10-Lite*.

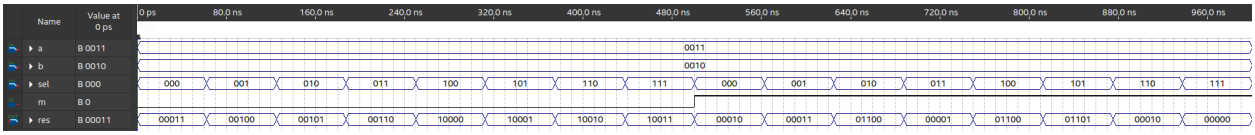
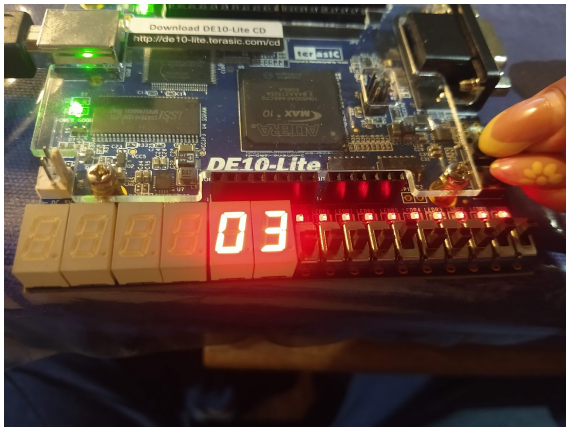


Figura 5: Simulación del circuito (Parte B)

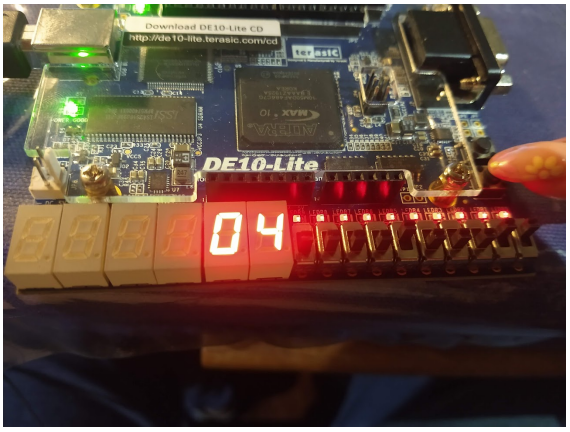
Funcionamiento en la tarjeta de desarrollo

Haciendo uso de una correcta asignación de pines en la sección de *Pin Planner*, se puede cargar el programa a la tarjeta a través del modo *Programmer*.

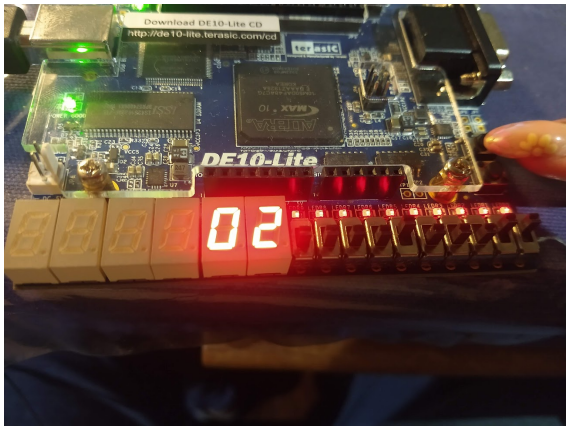
Se tomaron foto de algunos algunos casos de pueba significativos y referentes a la simulación de la Figura 5. El primero que se tiene es el de $sel(2) = 0$, $sel(1) = 0$ y se varían $sel(0)$ y m.



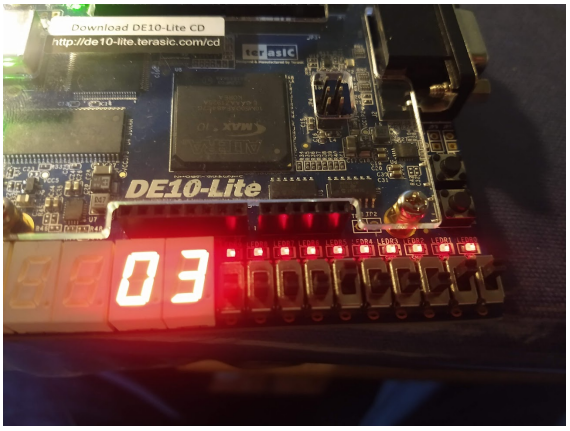
(a) Caso de prueba $sel = 000$, $m = 0$



(b) Caso de prueba $sel = 001$, $m = 0$



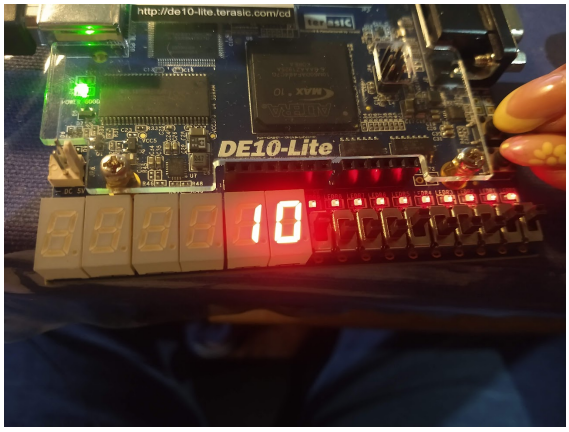
(c) Caso de prueba $sel = 000$, $m = 1$



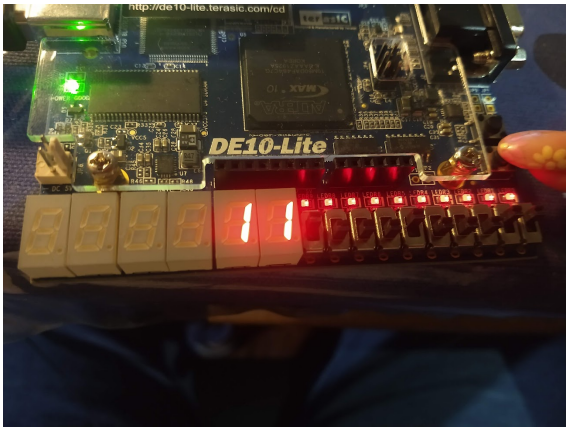
(d) Caso de prueba $sel = 001$, $m = 1$

Figura 6: Primer caso de prueba (Sección B)

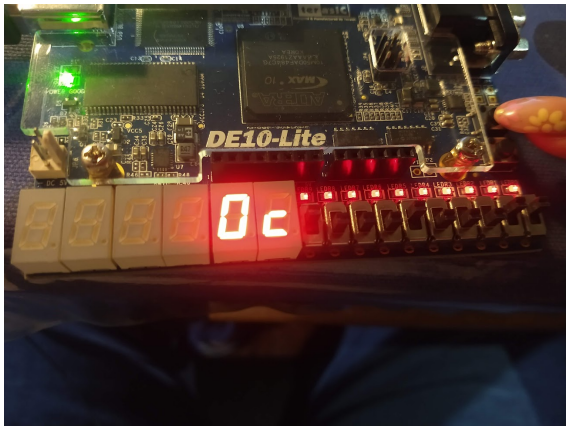
Los resultados coinciden con la simulación (tomando en cuenta el hexadecimal, claeramente). Se realizó otro caso de prueba adicional para tener en consideración más posibilidades. El caso $sel(2) = 1$, $sel(1) = 0$ y se varían $sel(0)$ y m.



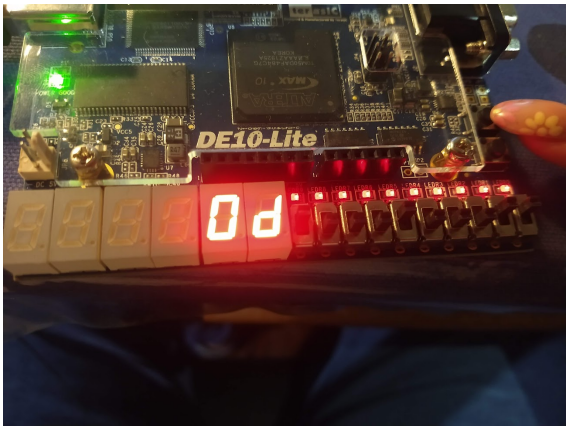
(a) Caso de prueba $sel = 100$, $m = 0$



(b) Caso de prueba $sel = 101$, $m = 0$



(c) Caso de prueba $sel = 100$, $m = 1$



(d) Caso de prueba $sel = 101$, $m = 1$

Figura 7: Segundo caso de prueba (Sección B)

Nuevamente, los resultados coinciden con los de la Figura 5, con lo cual se puede concluir que el problema se solucionó de forma exitosa.

4.4. Parte C

Propuesta

Se propone llevar a cabo el sistema planteado en la Sección 4.1 haciendo uso del lenguaje VHDL, en donde se tiene un circuito ***Desplazador-Rotador*** *que toma como entrada una palabra de 4 bits y devuelve otra palabra de 4 bits. Esto, con la finalidad de llevar a la práctica el uso de los multiplexores y verificar el funcionamiento del circuito.*

Solución

Tal como ya se plenteó anteriormente, la solución al problema (al menos de forma conceptual) se puede ver en la Figura 1.

Para dar solución al problema, únicamente se requirieron implementar dos entidades. Por un lado, el multiplexor 4:1 (*Mux4*); y por otro, el circuito Desplazador-Rotador que por en sí mismo representa la solución al problema (*DesplazadorRotador*).

Dadas las cuatro entradas de un multiplexor ($x_3x_2x_1x_0$) y dadas las líneas de selección (s_1s_0), la salida se puede representar combinando una forma SOP con los valores de la línea correspondiente. Tal como se muestra a continuación:

$$z = \bar{s_1}\bar{s_0}x_0 + \bar{s_1}s_0x_1 + s_1\bar{s_0}x_2 + s_1s_0x_3$$

Por lo cual, el funcionamiento del multiplexor queda definido por:

```
-- Implementacion: Multiplexor (4:1)
library ieee;
use ieee.std_logic_1164.all;

entity Mux4 is
  port(
    x   :in std_logic_vector(3 downto 0);
    s   :in std_logic_vector(1 downto 0);
    z   :out std_logic
  );
end entity;

architecture Behave of Mux4 is
begin
  z <=   ((not s(1) and not s(0)) and x(0))
      or ((not s(1) and     s(0)) and x(1))
      or ((    s(1) and not s(0)) and x(2))
      or ((    s(1) and     s(0)) and x(3));
end;
```

Posteriormente, se creó la entidad *DesplazadorRotador* en donde se conectaron los seis multiplexores de la Figura 1 para calcular el valor de z. El código con el cual se implementó la entidad fue el siguiente:

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity DesplazadorRotador is
  port(
    x   :in std_logic_vector(3 downto 0);
    c   :in std_logic_vector(2 downto 0);
    z   :out std_logic_vector(3 downto 0)
  );
end entity;

architecture Behave of DesplazadorRotador is
  component Mux4 is
    port(
      x   :in std_logic_vector(3 downto 0);
      s   :in std_logic_vector(1 downto 0);
      z   :out std_logic
    );
  end component;

  signal s, d, r  :std_logic;
  signal k0, k3 :std_logic;
begin
  s <= c(2);
  d <= c(1);
  r <= c(0);

  Mux_K0  :Mux4 port map(
    x => (x(3), '0', x(1), x(1)),
    s => (d, r),
    z => k0
  );

  Mux_K3  :Mux4 port map(
    x => (x(2), x(2), x(0), '0'),
    s => (d, r),
    z => k3
  );

  Mux_Z0  :Mux4 port map(
    x => (k0, x(1), x(0), x(0)),
    s => (s, d),
    z => z(0)
  );

  Mux_Z1  :Mux4 port map(
    x => (x(0), x(2), x(1), x(1)),
    s => (s, d),
    z => z(1)
  );

  Mux_Z2  :Mux4 port map(
    x => (x(1), x(3), x(2), x(2)),
    s => (s, d),
```

```
55      z => z(2)
56    );
57
58    Mux_Z3  :Mux4 port map(
59      x => (x(2), k3, x(3), x(3)),
60      s => (s, d),
61      z => z(3)
62    );
63  end;
```

Con lo anterior, se puede pasar a realizar pruebas.

Ejecución en Quartus II

Se creó una simulación funcional y el resultado fue el siguiente:

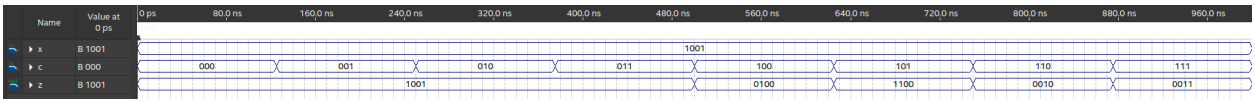


Figura 8: Simulación del circuito (Parte C)

Lo cual coincide con lo establecido en la Tabla 1. Uno de los aspectos importantes de mencionar es que para la primera mitad del funcionamiento no altera la salida (tal como se estableció en el planteamiento de la actividad). De ahí en fuera, los demás casos son:

- **Para `100`:** Se tiene un desplazador a la derecha.
- **Para `101`:** Se tiene un rotador a la derecha.
- **Para `110`:** Se tiene un desplazador a la izquierda.
- **Para `111`:** Se tiene un rotador a la izquierda.

Con lo cual queda comprobado el funcionamiento del circuito propuesto.

5. Conclusiones

Ríos Lira, Gamaliel. Con la realización de esta práctica, el objetivo planteado al principio de la misma se cumplió satisfactoriamente. Por una parte, se utilizaron componentes aritméticos para construir tanto el sumador serial como la ALU; multiplexores para la ALU y el circuito Desplazador-Rotador; y decodificadores para el ejercicio de la misma ALU. La descomposición del sumador en *Full Adders* fue de suma imporatancia para constuir varios de los demás componentes. Además, con la realización del previo comprendí de mejor manera el uso de los multplexores como una herramienta de "alto nivel" (ya implementada) que nos ayuda a seleccionar una u otra entrada evitando la implementación de "bajo nivel" del mismo (a través de compuertas lógicas o algún método similar).

Una parte que me pareció complicada de esta práctica fue la asignación de pines, ya que al ser tantos, cualquier error resultaba en un comportamiento no esperado.

6. Bibliografía

[1] M. Mano, *Digital Design*. Upper Saddle River, NJ, United States: Prentice Hall, 2002.