

---

**TECNOLÓGICO NACIONAL DE MÉXICO**  
**INSTITUTO TECNOLÓGICO DE MORELIA**

---



DEPARTAMENTO DE INGENIERÍA ELÉCTRICA-ELECTRÓNICA

ACADEMIA DE INGENIERÍA ELECTRÓNICA

MATERIA DE:

**PROCESAMIENTO EN HARDWARE**

PRÁCTICA #2

**“ALU DE 8 BITS CON PUNTO FIJO”**

**Alumnos:** Andrea Joanelle Hernández Alvarado 19121121

Pedro Eduardo Rojo Carrillo 19121155

**Profesor:** Arturo Méndez Patiño

Morelia Mich; a 18 de Noviembre del 2023.

## OBJETIVOS

- Realizar la descripción de una unidad lógica aritmética de 8 bits en VHDL con punto fijo, que pueda realizar la suma, resta y multiplicación de operandos de 8 bits con punto fijo.
- La ALU debe de mostrar el resultado de la operación en Hexadecimal y en los displays de 7 segmentos de la tarjeta de desarrollo de la Cyclone IV.
- Comprender y reafirmar el conocimiento sobre la interpretación del punto en los sistemas digitales así como sus ventajas y desventajas.
- Reafirmar el conocimiento acerca de las codificaciones de los números signados.
- Elegir el sistema de codificación de signo más adecuado según las necesidades de los alumnos.

## INTRODUCCIÓN

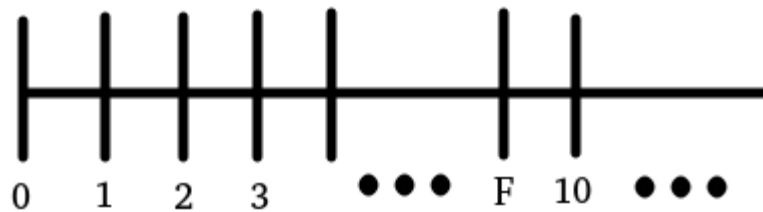
Los números en sí, son estructuras lógico matemáticas abstractas que expresan cantidades y están sujetos a una codificación para que un intérprete pueda procesarlos; Dicha cantidad abstracta puede o no ser signada, es decir, puede o no representar una sustracción de si mismo o una adición de si mismo. Para que un intérprete pueda comprender dichas estructuras, estas deben de tener una base, es decir, una cantidad de cifras o dígitos reglados por un conjunto de símbolos y letras que permitan la correcta codificación y comprensión de dicho número.

Para poder poder operar entre números se requiere de la habilidad de poder interpretar su codificación, es decir, la base en la que se encuentran y que reglas impone dicha base, e.g. los números de base 6 sólo pueden tener como dígitos del 0 al 5 y que caso de que un número supere esa magnitud deberá usarse más de un dígito. A continuación en la ecuación 2.1 se muestra la notación polinomial general de los números

$$\sum_{i=m}^{n-1} D_i B^i$$

*Ecuación 2.1 Notación Polinomial General*

Dónde “n” es la cantidad de dígitos entera, “m” es la cantidad de dígitos de parte fraccionaria y D es el dígito que va desde 0 hasta B-1 y B es la base. Una vez comprendiendo dichos conceptos podemos comenzar a revisar como es que un intérprete puede operar los números, comenzando por la adición; en caso de que los números que se van a operar se encuentren con distinta codificación (base) es necesario pasarlos a la misma base para poder operarlos de forma adecuada. Una vez que dichos números se pueden operar, para sumarlos; se debe de avanzar en la recta numérica la cantidad de veces que la magnitud del operando indica, e.g.



*Figura 2.1: Recta numérica hexadecimal*

En la figura 2.1 se puede apreciar que si se suma 1 con 1 entonces la magnitud del número resultante será 2; y si se suma 1 con F entonces el resultado será 10. Observe que en el momento en el que supera la cantidad máxima representada por la base, se cae en la necesidad de utilizar otro dígito, que comienza desde el valor entero más pequeño distinto de cero.

Para la sustracción tan sólo se debe de aplicar la diferencia entre dichas magnitudes, lo que significa retroceder en la recta numérica el número de veces que representa la magnitud negativa operada.

Hasta este punto todo resulta relativamente sencillo, sin embargo, en el momento en el que el resultado de una sustracción es menor que cero, caemos en la necesidad de representar aquellas magnitudes que significan en sí mismas una sustracción, es decir, en representar los números negativos.

Matemáticamente hablando existe una cantidad infinita de posibles formas para poder representar los números negativos, sin embargo, en este reporte revisaremos algunas de las más populares y utilizadas.

### Complementos

[1] [2] [3] Se le denomina complemento de una cifra, a la diferencia entre la base y dicha cifra, e.g. en sistema octal (de base 8) el complemento del número 3 sería el 5 porque dichos números sumados forman a su base que es 8. Esta representación permite la aplicación de uno de los axiomas matemáticos que indica que la sustracción no es más que la suma del inverso aditivo, es decir,  $A - B = A + (-B)$ . Permitiendo así que dichos operandos puedan ser procesados por el intérprete como una simple adición. Como se puede observar esta representación permite conservar una aritmética mecanizada la cuál facilita el algoritmo del cálculo de las operaciones, dicha propiedad será analizada más adelante.

### Complemento a la base disminuida

[3] [4] Una alternativa al sistema de complemento a la base es el complemento a la base menos 1, donde en este sistema un número negativo de  $n$  dígitos en base  $r$  se representa por lo que le falta para llegar a  $r^n - 1$ , e.g. El complemento natural de la base decimal sería el complemento A10 mientras que el complemento a la base disminuida sería Complemento A9.

### Magnitud signada

[5] Otra representación muy utilizada es la magnitud signada, la cuál consiste en añadir un símbolo extra a los números para poder indicar su polaridad. Este método es el que utilizan los seres humanos.

Como se mencionó anteriormente la base indica al intérprete como es que este debe de procesar y operar al número, lo cuál nos lleva a realizarnos la siguiente pregunta ¿Existen bases que sean más fáciles de procesar para algunos intérpretes? La respuesta es sí, si el intérprete es un ser humano, entonces le será más fácil operar dichos números en base 10, sin embargo, si el intérprete es una máquina le será más sencillo procesar dichos número en base 2, esto sucede debido a la naturaleza de los mismos. La razón de ser así se puede encontrar en la antigua Mesopotamia, dónde se llegó a utilizar el sistema duodecimal [6], derivado de contar el número de falanges en los dedos de la mano, sin embargo, debido a que el ser humano cuenta con 10 dedos en total, se optó por cambiar ese sistema al sistema decimal o de base 10. En el caso de las máquinas la circuitería sólo puede detectar con la discretización de un valor analógico de voltaje (ignorando codificaciones complejas

e.g. de desplazamiento como PSK, ASK, etc.), un 1 lógico o un 0 lógico (apagado o encendido), lo que las lleva a implementar por naturaleza el sistema de base 2.

Los sistemas digitales implementan en distintas aplicaciones las tres representaciones de números con signo que se mencionaron anteriormente; Complemento a la base (Complementos), Magnitud signada, Excesos, entre otros. Sin embargo en esta ocasión sólo se tratarán las representaciones de complemento a la base y magnitud signada.

### Complemento A1

Este complemento se puede interpretar como el complemento a la base disminuida de la base que manejan los sistemas digitales i.e. base 2; El complemento A1 implica de manera práctica manejar los números en cifras de base 1, sin embargo, debido a que resulta imposible manejar un sistema numérico cuya base sea 1, es decir, que todos sus dígitos sean cero, entonces por definición se concretó que este complemento sería la inversión de todos los dígitos del número en base 2, lo que significa que con una compuerta lógica not (o el conjunto de estas) es posible obtener el resultado del complemento A1 de un número de base 2. Debido a que esta representación posee ciertos problemas y lagunas matemáticas no es muy utilizado en los sistemas digitales; algunos de estos problemas son:

- La existencia de un cero positivo y un cero negativo: Este error aparece si nosotros buscamos codificar el cero positivos, es decir, si se le aplica una compuerta not a 0, esta daría como resultado puros unos, lo que significa que existe un 0 negativo, y por ende un desperdicio de espacio en la recta numérica.
- El error de las operaciones cuándo se cruza el "0 negativo": Si el resultado de una operación es negativo para poder llegar a dicho número no sólo se tendría que pasar por el cero, ya que dicha resta debe de pasar también por el cero negativo, generando un error en la operación y obteniendo un resultado que se encuentra una unidad por debajo del resultado real.

El rango de números que se pueden representar en binario por medio de este complemento es de  $[-2^{n-1}+1, 2^{n-1}-1]$  donde n es el número de bits con los que se cuentan, e.g. Si se tiene un registro de 8 bits codificado en complemento A1 el número más pequeño que este registro puede almacenar sería el -127 y el número más grande sería 127.

### Complemento A2

Al ser el complemento natural de la base binaria esta representación permite que las operaciones puedan ser efectuadas sin error alguno, por lo que se puede inferir que esta representación elimina al cero negativo que el complemento anterior tenía. Gracias a esto el complemento A2 se ha convertido en la principal codificación de números con signo en los sistemas digitales. Para poder encontrar el complemento A2 de un número en binario es posible usar dos métodos: convertir mediante la resta de la base con el número a convertir, ó utilizar el método más común en la electrónica digital; sumarle 1 al complemento A1 del número. Esto sucede debido a que (como se ha mencionado anteriormente): el complemento A2 elimina el cero negativo que el complemento A1 posee. De tal manera que sus rangos están dados por:  $[-2^{n-1}, 2^{n-1}-1]$ . Como se puede observar la eliminación

del cero negativo resulta en la adición de una unidad en el límite de la misma naturaleza, permitiendo así que e.g. un registro de 8 bits que almacene un número codificado en esta representación pueda almacenar como número más pequeño el -128 y como número más grande el 128.

### Magnitud signada

Esta representación es la más sencilla de comprender, ya que sólo consta de añadir un bit más del lado izquierdo del número, el bit de signo, si el número es una cantidad negativa entonces el bit toma el valor de 1 lógico; por otro lado si el número representa una cantidad positiva dicho bit toma el valor de 0 lógico. Similar al complemento A1 esta representación tiene el siguiente rango de números  $[-2^{n-1}+1, 2^{n-1}-1]$ , e.g. si se cuenta con un registro de 8 bits entonces se contarían sólo con 7 bits para representar nuestro número, ya que se tendría que dejar uno para el bit de signo, teniendo para este caso en específico como número más negativo el -127 y como número más positivo o más grande el 127.

Ahora que se mencionó el bit de signo en la magnitud signada, cabe destacar que las otras dos representaciones tienen también un bit de signo, el cuál se encuentra en el bit n si se piensa que se tiene una “n” cantidad de bits, sin embargo, la diferencia entre el bit de signo de la magnitud signada con el del complemento A2 por ejemplo es que el del complemento A2 tiene un peso en el número, mientras que el de la magnitud signada dicho bit no representa ninguna cantidad o peso. Por lo que es considerable pensar que lo único en lo que dichos bits de signo concuerdan, es en la posición en la que se encuentran y la representación de la polaridad del número.

Hasta el momento se ha pensado en los números que únicamente representan una magnitud entera, pero no en aquellos que representan una cantidad fraccionada. Dicha cantidad tiene sus propias reglas de la simbología, para ello se coloca un punto a fin de marcar los límites de donde se encuentran las cifras enteras y donde se encuentran las cifras fraccionarias. De tal forma que la notación polinomial general de la parte fraccionaria de cualquier número se encuentra en la ecuación 2.2

$$\sum_{i=-n}^{-1} D_i B^i$$

### *Ecuación 2.2: Notación polinomial general fraccionaria*

Las variables son las mismas que en la ecuación 2.1 y la expansión de dicha expresión está dada por la ecuación 2.3.

$$\cdot D_{-1}B^{-1} + D_{-2}B^{-2} + \dots + D_{-n}B^{-n}$$

*Ecuación 2.3: Expansión de la ecuación 2.2*

En esta ecuación puede apreciarse que la base se eleva hacia potencias negativas, lo cuál representa de forma exacta a una división o una fracción. En cuánto al punto, podemos observar que se encuentra hasta la izquierda, lo que quiere decir que todos los otros números más allá del punto, son enteros.

Cuándo se quiere operar dos número con punto lo que se suele hacer es colocar dichos puntos en la misma posición para después realizar la operación sin considerar el punto, lo que quiere decir que a nivel operativo la aparición del punto sólo modifica el acomodo de los números antes de ser operados, y sólo en el momento en el que se busca interpretar el resultado es que importa la posición del punto. A este mecanismo de interpretación se le conoce como punto fijo, y es el que la mayoría de seres humanos emplea para poder realizar operaciones entre números. Esta notación tiene sus ventajas y desventajas sin embargo eso será tratado en otra práctica.

Una vez aclarado eso, se puede llegar a la conclusión de que una ALU que esté diseñada para operar números con punto fijo será igual a una ALU que fue pensada para operar números enteros, ya que no existe diferencia alguna. [7] La diferencia sólo llega al momento de interpretar el resultado de la operación, ya que en ese momento si importa la posición del punto y que cifras están antes y después.

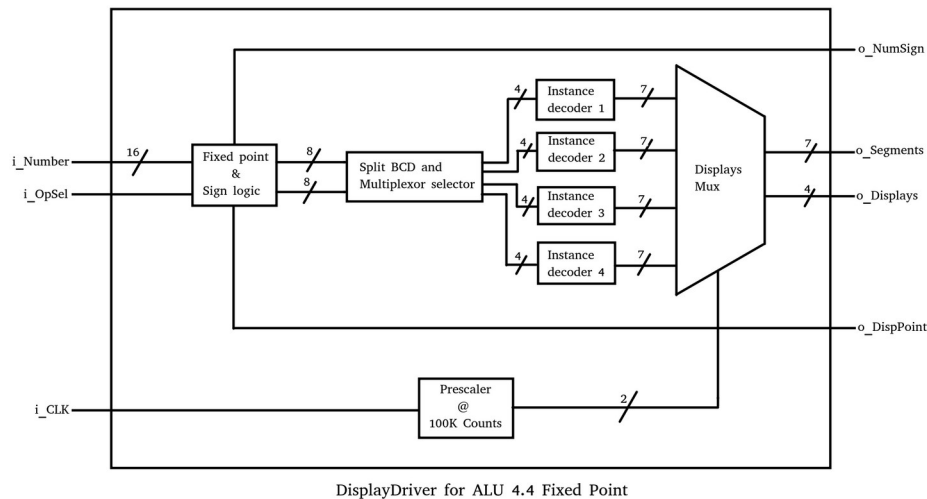
### **RTL aproximado**

A continuación se presentará como fue pensado y diseñado el hardware por lo alumnos para poder llevar a cabo el diseño en VHDL, sin embargo, esta representación no es exactamente lo que hay en el código, es sólo un diagrama que facilita la comprensión del sistema digital en general, y explica de forma simple como fue desarrollado.

El componente a desarrollar se encontrará dividido en 3 sub componentes o instancias VHDL que permitirán la reutilización de código y el encapsulamiento de los sistemas, de tal forma que se facilitarán tanto la comprensión del circuito como la solución de errores.

Para poder representar una multiplicación de 8 bits por 8 bits hace falta tener la capacidad de representar cantidades de 16 bits, por lo que será necesario utilizar 4 pantallas para poder imprimir 4 números hexadecimales, el punto fijo se encontrará en la parte que el DIP Switch de la placa indique.

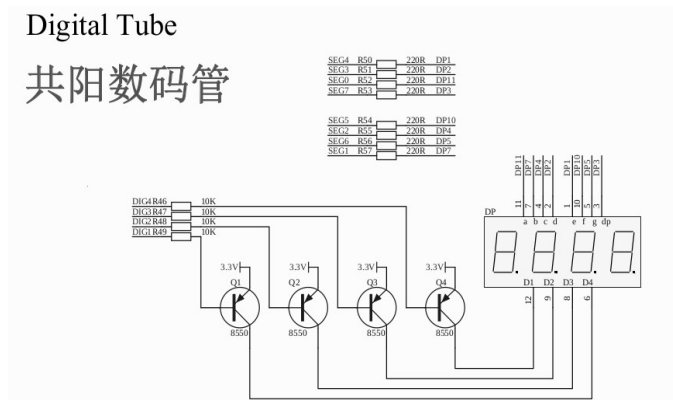
El primer componente y el más básico es un decodificador BCD a 7 segmentos que permitirá ir colocando en las pantallas un nibble (conjunto de 4 bits) específico y designado, dado que son 4 pantallas, serán 4 los codificadores que se instanciarán. La aparición de estos componentes instanciados se puede apreciar en la figura 2.2.



**Figura 2.2: RTL Aproximado del "Display Driver"**

Como segundo componente y que contendrá al primero, se tiene un driver para los displays de 7 segmentos, ya que como se puede apreciar en la figura 2.3 los segmentos de dichos displays están conectados a las mismas terminales, y sólo los ánodos comunes son los que están con conexiones distintas, por lo que dicho driver deberá de multiplexar los displays al menos a 10Hz, ya que poco menos de 100ms es el tiempo que un destello de luz dura. En general, las personas no pueden distinguir estímulos tenues, cortos, brillantes y largos en una décima de segundo [8]. Por lo que para este propósito se utilizará la frecuencia interna que la FPGA brinda en el pin 23 y será preescalada a 100K cuentas, dando como resultado una frecuencia de 500Hz (ya que el cristal de la FPGA oscila a 50MHz). Como se habló en la sección anterior el driver también deberá de trabajar la interpretación del punto y por ende, deberá escogerse una codificación de signo. En este caso se escogió el complemento A2 debido a que respeta las operaciones y las da de forma correcta, además es el que trabaja la librería estándar de la IEEE en VHDL al usar el operando "-". Como se puede apreciar en la figura 2.2, la lógica de signo y de punto fijo separan el resultado que llega en dos vectores de 8 bits para que la lógica de separación BCD y multiplexado pueda utilizar las pantallas.

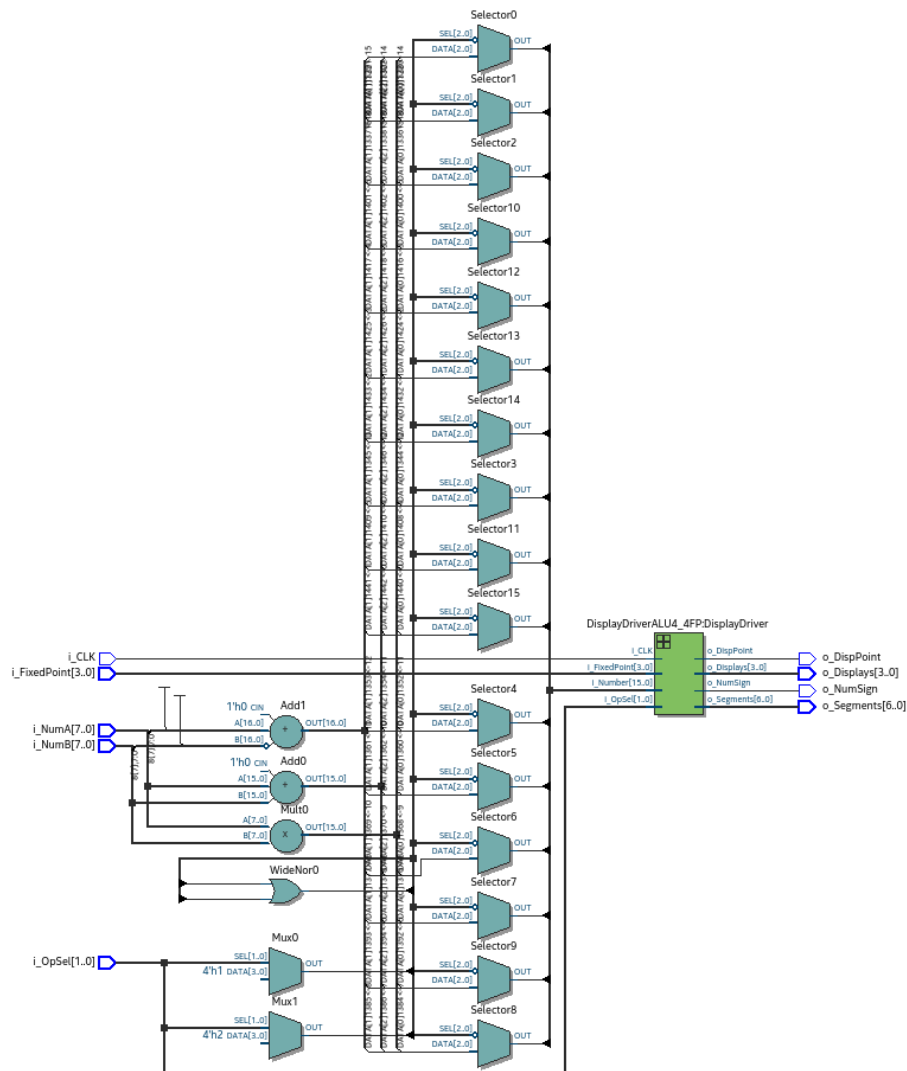
Por último y como entidad principal (no aparece en la figura 2.2) se tiene la ALU que instanciará al display driver y le entregará el resultado al mismo.



**Figura 2.3: Conexión de los displays en la placa de desarrollo**



## RTL



*Figura 2.4: RTL del diseño general*

Como se puede apreciar en la figura 2.4, al comienzo los operandos son llevados a un sumador (Add0), otro sumador (Add1) el cuál es en realidad un restador, debido a que el operando B está negado, lo que implica que está obteniendo el complemento a 2 del operando para efectuar la resta (recuerde que la resta es la suma del inverso aditivo). Por último los operandos también se envían a un multiplicador embebido, así mismo el acarreo de entrada se encuentra una simbología escrita como 1'h0, lo que significa que a esa entrada se le está asignado un vector de 1 bit de longitud con un valor de cero hexadecimal, por lo que se puede inferir que el bit de acarreo de entrada no se está utilizando. Nótese que todos los vectores de entrada de los operadores son de 16 bits y no de 8 (Como están declaradas las entradas), debido a que se realizó una copia del bit de signo de la entrada de 8 bits, en la parte más alta del vector de 16 bits, esto se hizo porque la multiplicación arroja un vector de 16 bits y el Display driver espera recibir la misma cantidad para emplear los 4 displays. Las operaciones en sí se efectúan sin importar si están seleccionadas o no y

después por medio del selector, toda la red combinacional de multiplexores elegirá que resultado es el que se enviará al display driver (para ser mostrado), dicho valor de selección también se manda al display driver, ya que este debe de saber como debe recorrer o no el punto según la operación.

Por el lado de eficiencia operativa y energética el hecho de realizar las operaciones a pesar de que no sean seleccionadas no implica un gasto de energía significativo, debido a que la salidas de estos operandos no se utilizan y al no encontrar ningún camino de salida y quedarse dicho resultado en los multiplexores no hay flujo de corriente hacia las salidas.

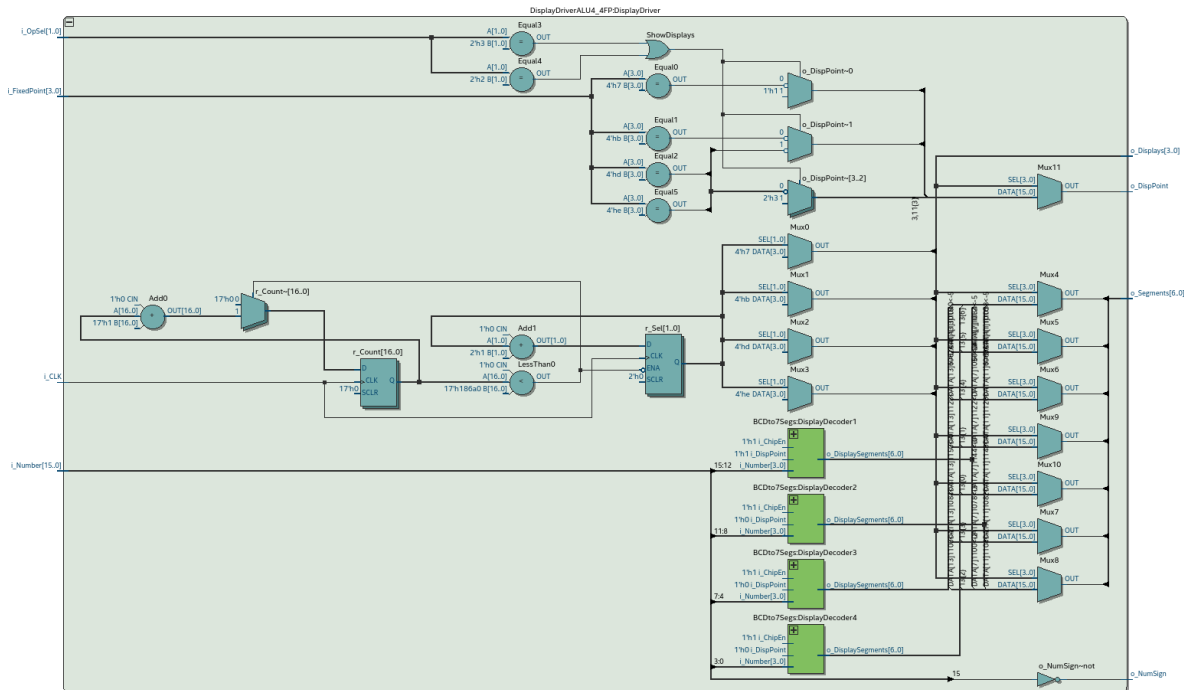


Figura 2.5: RTL del DisplayDriver

En la figura 2.5 puede apreciarse que el RTL termino siendo algo parecido al RTL aproximado mostrado en la figura 2.2, puede apreciarse que el preescalador termino siendo un conjunto de flipflops D, sumadores y comparadores, donde, como se verá en código, el valor del registro de cuentas irá aumentando hasta que se llegue a una cuenta establecida por un generic y será en donde el selector cambiará de valor, cambiando así la pantalla que muestra el resultado. Aquellas entidades verdes que se aprecian son los decodificadores de 7 segmentos.

La decodificación de instrucción (Selección de operación aritmética) se convirtió en un conjunto de comparadores con compuertas lógicas para poder decodificar que operación fue a que se efectuó y de esa forma modificar la posición del punto en caso de que sea necesario.

## CÓDIGO

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_signed.all;
4 use IEEE.numeric_std.all;
5
6 entity ALU8BitsDisplaysFP is
7     generic(
8         g_PSC_COUNTS : integer := 100000
9     );
10    port (
11        i_CLK      : in std_logic;
12        i_NumA      : in signed(7 downto 0);
13        i_NumB      : in signed(7 downto 0);
14        i_FixedPoint : in std_logic_vector(3 downto 0);
15        i_OpSel     : in std_logic_vector(1 downto 0);
16        o_Displays  : out std_logic_vector(3 downto 0);
17        o_Segments  : out std_logic_vector(6 downto 0);
18        o_DisPoint  : out std_logic;
19        o_NumSign   : out std_logic
20    );
21 end entity ALU8BitsDisplaysFP;
22
23 architecture rtl of ALU8BitsDisplaysFP is
24
25     component DisplayDriverALU4_4FP is
26         generic(
27             g_PSC_COUNTS : integer := 100000
28         );
29         port (
30             i_CLK      : in std_logic;
31             i_OpSel     : in std_logic_vector(1 downto 0);
32             i_Number    : in std_logic_vector(15 downto 0);
33             i_FixedPoint : in std_logic_vector(3 downto 0);
34             o_Displays  : out std_logic_vector(3 downto 0);
35             o_Segments  : out std_logic_vector(6 downto 0);
36             o_DisPoint  : out std_logic;
37             o_NumSign   : out std_logic
38         );
39     end component DisplayDriverALU4_4FP;
40
41     type t_Operations is (t_Sume, t_Subst, t_Mult);
42
43     signal r_Op      : t_Operations;
44     signal r_Result   : signed(15 downto 0);
45     signal r_AdtA     : signed(15 downto 0) := (others => '0');
46     signal r_AdtB     : signed(15 downto 0) := (others => '0');
47
```

```

48 begin
49
50   DisplayDriver : DisplayDriverALU4_4FP
51   generic map (
52     g_PSC_COUNTS => g_PSC_COUNTS
53   )
54   port map(
55     i_CLK      => i_CLK,
56     i_OpSel    => i_OpSel,
57     i_Number   => std_logic_vector(r_Result),
58     i_FixedPoint => i_FixedPoint,
59     o_Displays => o_Displays,
60     o_Segments => o_Segments,
61     o_Dispoint => o_Dispoint,
62     o_NumSign  => o_NumSign
63   );
64
65   with i_OpSel select r_Op <=
66     t_Sume when "00",
67     t_Subs when "01",
68     t_Mult when others;
69
70   --A
71   r_AdtA(7 downto 0) <= i_NumA;
72
73   with i_NumA(7) select r_AdtA(15 downto 8) <=
74     (others => '1') when '1',
75     (others => '0') when others;
76
77   --B
78   r_AdtB(7 downto 0) <= i_NumB;
79
80   with i_NumB(7) select r_AdtB(15 downto 8) <=
81     (others => '1') when '1',
82     (others => '0') when others;
83
84   --Operations
85   with r_Op select r_Result <=
86     (r_AdtA + r_AdtB) when t_Sume,
87     (r_AdtA - r_AdtB) when t_Subs,
88     (r_AdtA(7 downto 0) * r_AdtB(7 downto 0)) when others;
89
90 end architecture rtl;

```

*Figura 2.6 Código de la ALU de 8 bits*

Desde la línea 6 a la 21 se declaran los puertos de la entidad, dónde i\_CLK es la señal de reloj que llegará al display driver y la usará para multiplexar las pantallas, i\_NumA es el operando A e i\_NumB es el operando B, i\_OpSel es el selector de operación, o\_Displays es el vector de salida que se encargará de decidir que pantalla se apaga y que pantalla se

enciende, o\_Segments es el vector que decidirá que segmentos del display se encienden y que segmentos se apagan según el decodificador BCD a 7 segmentos, o\_DispPoint es un std\_logic que indicará donde se encuentra el punto de fracción en las pantallas, o\_NumSign es un std\_logic que por medio de un LED se encargará de mostrar al usuario el signo del resultado.

De las líneas 23 a 48 se encuentran las declaraciones de lo que se utilizará en la arquitectura de la entidad; el componente "DisplayDriver" y los registros o signals que se utilizarán dentro de la entidad. r\_Result es el registro que guarda el resultado de la operación, r\_AdtA y r\_AdtB son los registros que hacen posible la multiplicación de 8 bits por 8 bits, su nomenclatura significa "Additional A" y "Additional B", haciendo alusión a que se tienen bits adicionales para la compatibilidad con los demás componentes del diseño. Para facilitar el uso y prueba del componente se declaró un nuevo tipo de dato, el cuál es t\_Operations que contiene a t\_Sume, t\_Subst, t\_Mult.

De las líneas 48 a 63 se encuentra la instanciación del componente "DisplayDriver" y sus respectivas conexiones.

De las líneas 65 a 68 se define la codificación de instrucciones, 00 para suma, 01 para resta y 1X para multiplicación.

De las líneas 71 a 75 se define la parte más baja del registro r\_AdtA para después en la parte más alta del registro hacer una copia del bit de signo, lo mismo sucede para el operando r\_AdtB en las líneas 78 a 82.

Por último en las líneas 85 a 88 se efectúan las operaciones según sean indique el registro r\_Op, cabe destacar que para el caso específico de la multiplicación se toma la parte más baja del registro para efectuarla, afín de que el resultado pueda guardarse en un registro de 16 bits.

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_unsigned.all;
4 use IEEE.numeric_std.all;
5
6 entity DisplayDriverALU4_4FP is
7     generic(
8         g_PSC_COUNTS : integer := 100000
9     );
10    port (
11        i_CLK      : in std_logic;
12        i_OpSel     : in std_logic_vector(1 downto 0);
13        i_Number    : in std_logic_vector(15 downto 0);
14        i_FixedPoint : in std_logic_vector(3 downto 0);
15        o_Displays  : out std_logic_vector(3 downto 0);
16        o_Segments  : out std_logic_vector(6 downto 0);
17        o_DispPoint : out std_logic;
18        o_NumSign   : out std_logic
19    );
20 end entity DisplayDriverALU4_4FP;
21
22 architecture rtl of DisplayDriverALU4_4FP is
23
24     component BCDTo7Segs is
25         port (
26             i_DispPoint : in std_logic; --Display point enable
27             i_ChipEn     : in std_logic; --Chip enable (works with PNP transistors)
28             i_Number     : in std_logic_vector(3 downto 0); --Number that will be
displayed
29             o_DisplaySegments : out std_logic_vector(6 downto 0) --Display segments
vector
30         );
31     end component BCDTo7Segs;
32
33     --constant DISP_NEGATIVE_SIGN : std_logic_vector(6 downto 0) := "0111111";
34     --constant DISP_POSITIVE_SIGN : std_logic_vector(6 downto 0) := "1111111";
35     --Multiplexor Registers
36     signal r_Sel : std_logic_vector(1 downto 0) := "00";
37     signal r_Show : std_logic_vector(3 downto 0) := "0000";
38     signal r_Count : integer range 0 to 100000;
39     --Integer Part Registers
40     signal r_Segs1, r_Segs2 : std_logic_vector(6 downto 0);
41     signal r_SegsConn1, r_SegsConn2 : std_logic_vector(6 downto 0);
42     --Decimal Part Registers
43     signal r_Segs3, r_Segs4 : std_logic_vector(6 downto 0);
44     signal r_SegsConn3, r_SegsConn4 : std_logic_vector(6 downto 0);
45     --Signed Logic Registers
46     signal r_NumberSign : std_logic;

```

```

47  signal r_IntPNumber, r_DecPNumber : std_logic_vector(7 downto 0);
48
49  begin
50      --Integer part displays
51      DisplayDecoder1 : BCDTo7Segs port map (
52          i_Dispoint    => '1',
53          i_ChipEN       => '1',
54          i_Number       => r_IntPNumber(7 downto 4),
55          o_DisplaySegments => r_SegsConn1
56      );
57      DisplayDecoder2 : BCDTo7Segs port map (
58          i_Dispoint    => '0',
59          i_ChipEN       => '1',
60          i_Number       => r_IntPNumber(3 downto 0),
61          o_DisplaySegments => r_SegsConn2
62      );
63      --Decimal part displays
64      DisplayDecoder3 : BCDTo7Segs port map (
65          i_Dispoint    => '0',
66          i_ChipEN       => '1',
67          i_Number       => r_DecPNumber(7 downto 4),
68          o_DisplaySegments => r_SegsConn3
69      );
70      DisplayDecoder4 : BCDTo7Segs port map (
71          i_Dispoint    => '0',
72          i_ChipEN       => '1',
73          i_Number       => r_DecPNumber(3 downto 0),
74          o_DisplaySegments => r_SegsConn4
75      );
76
77      clkCount : process(i_CLK)
78      begin
79          if rising_edge(i_CLK) then
80              if r_Count < g_PSC_COUNTS then
81                  r_Count <= r_Count + 1;
82              else
83                  r_Sel <= r_Sel + 1;
84                  r_Count <= 0;
85              end if;
86          end if;
87      end if;
88  end process;
89
90  ShowDisplays : process(r_Sel)
91  begin
92      r_Segs1 <= r_SegsConn1;
93      r_Segs2 <= r_SegsConn2;
94      r_Segs3 <= r_SegsConn3;
95      r_Segs4 <= r_SegsConn4;
96

```

```

97     case r_Sel is
98         when "00" =>
99             r_Show <= "1110";
100        when "01" =>
101            r_Show <= "1101";
102        when "10" =>
103            r_Show <= "1011";
104        when "11" =>
105            r_Show <= "0111";
106        when others =>
107            r_Show <= "1111";
108    end case;
109
110    case r_Show is
111        when "0111" => --Display 1 MSB
112            o_Segments <= r_Segs1;
113            if i_OpSel = "11" OR i_OpSel = "10" then
114                o_Dispoint <= '1';
115            else
116                if i_FixedPoint = "0111" then
117                    o_Dispoint <= '0'; --Turn on the point segment
118                else
119                    o_Dispoint <= '1'; --Turn off the point segment
120                end if;
121            end if;
122        when "1011" => --Display 2
123            o_Segments <= r_Segs2;
124            if i_OpSel = "11" OR i_OpSel = "10" then
125                if i_FixedPoint = "1101" then
126                    o_Dispoint <= '0';
127                else
128                    o_Dispoint <= '1';
129                end if;
130            else
131                if i_FixedPoint = "1011" then
132                    o_Dispoint <= '0'; --Turn on the point segment
133                else
134                    o_Dispoint <= '1'; --Turn off the point segment
135                end if;
136            end if;
137        when "1101" => --Display 3
138            o_Segments <= r_Segs3;
139            if i_OpSel = "11" OR i_OpSel = "10" then
140                o_Dispoint <= '1';
141            else
142                if i_FixedPoint = "1101" then
143                    o_Dispoint <= '0'; --Turn on the point segment
144                else
145                    o_Dispoint <= '1'; --Turn off the point segment
146                end if;
147            end if;

```



```

148     when others => --Display 4 LSB
149         o_Segments <= r_Segs4;
150         if i_OpSel = "11" OR i_OpSel = "10" then
151             o_DisPoint <= '1';
152         else
153             if i_FixedPoint = "1110" then
154                 o_DisPoint <= '0'; --Turn on the point segment
155             else
156                 o_DisPoint <= '1'; --Turn off the point segment
157             end if;
158         end if;
159     end case;
160 end process;
161
162 r_NumberSign <= i_Number(15);
163
164 --Integer Part
165 r_IntPNumber <= i_Number(15 downto 8);
166 --Decimal Part
167 r_DecPNumber <= i_Number(7 downto 0);
168
169 o_Displays <= r_Show;
170
171 o_NumSign <= not r_NumberSign;
172
173 end architecture rtl;

```

*Figura 2.7 Código del display driver*

En las líneas 6 a 20 se declaran los puertos de la entidad de los cuáles las más notables son: i\_CLK es la señal de reloj proveniente del pin 23, i\_OpSel es un vector entregado por la ALU para saber que operación se realizó. Por encima de estas declaraciones se encuentra un generic, que no es más que una constante que controla el número de cuentas en el preescalador, esta constante se maneja de esa manera para poder cambiar su valor en el testbench y no esperar 100K señales de reloj para ver que un resultado cambie.

De las líneas 22 a 49 podemos encontrar la lista de declaraciones de la arquitectura. Como primera declaración se puede ver el componente que hará la decodificación para los displays "BCDTo7Segs", dos constantes que contienen la información para imprimir una pantalla el signo menos ("DISP\_NEGATIVE\_SIGN") o dejar la pantalla apagada en el caso de un número positivo ("DISP\_POSITIVE\_SIGN"). A pesar de que estas constantes no se utilizan en el código se dejaron listas para poder hacer este código lo más reutilizable posible.

De las líneas 36 a 47 se puede observar las declaraciones de los registros para el multiplexado, donde r\_Sel es el registro que se encargará de ir seleccionando que pantalla se va a mostrar, r\_Show es un registro que guardará la secuencia de bits que deben de encenderse y apagarse para controlar las pantallas e.g. si se tiene un "0111" se encenderá el display de la izquierda (viendo la FPGA en plano cenital). El registro r\_Count se encargará de almacenar las cuentas del preescalador para de ese modo hacer que la

frecuencia de actualización no sea tan rápida y que no se pueda interpretar el valor impreso en la pantalla. Los registros Segs1 y Segs2 entregarán dicho valor a la salida o\_Segments dentro del proceso de impresión; los registros SegsConn1 y 2 almacenarán el valor decodificado por los componentes BCDDto7Segs para dentro del proceso entregarle ese valor a los registros r\_Segs1 y r\_Segs2. Lo mismo sucede para Segs3 y 4 pero con la parte más baja del número, clasificada como parte decimal, que aunque dependiendo de la posición del punto puede o no ser la parte decimal, se denominó de esa forma por como se pensó el código y la estructura del mismo, poco después se añadió la compatibilidad con el movimiento del punto fijo.

r\_NumberSign es el registro que guardará el bit de signo y r\_IntPNumber, r\_DecPNumber almacenarán la parte más alta y baja respectivamente del número entrante a imprimir

Podemos comenzar a ver el comportamiento de la arquitectura en las siguientes líneas de código en donde se hace la declaración de los 4 decodificadores BCD a 7 segmentos, dónde el decoder 1 es para el nibble más significativo, el decoder 2 para el nibble que le sigue y así sucesivamente hasta llegar al decoder 4 que tiene el nibble menos significativo.

Nótese que no todas las salidas de las instancias son conectadas o utilizadas, debido a que este código fue hecho para manejarse con 7 segmentos para cada display (y no compartidos), por lo que para este caso en específico no se requieren.

A partir de la línea 78 podemos ver el primer proceso del código, este se encarga del preescalado de la señal de reloj al mismo tiempo que manda la señal de cambio al multiplexor de las pantallas. En la línea 90 principia el código del proceso más importante del componente, ya que este recibe los registros que los decodificadores de 7 segmentos mandan en registros internos que el proceso puede leer (no se puede compartir registros entre un proceso secuencial y una instancia).

En el primer case se elige que valor tomará el registro r\_Show según sea el valor del registro r\_Sel, codificando como "00" la pantalla de hasta la derecha, "01" la pantalla que le sigue a su izquierda y así sucesivamente hasta llegar a la última pantalla. Una vez que el código definió que display es el que va a imprimir un carácter entra a otro case para elegir ahora que segmentos se encenderán y que segmentos se apagarán, esto sucede según el valor dado por los decodificadores de 7 segmentos y el display que se mostrará. Una vez que se entra a un case, es decir, que se definió a que pantalla le imprimir el carácter se pregunta que operación se realizó, debido a que si la operación fue una suma o una resta, la posición del punto no debe de moverse tras la operación, sin embargo cuándo se realiza una multiplicación el punto debe de moverse el número de dígitos que tiene de parte fraccionaria, es decir, si se tiene un 0xFE.80 por 0xE8.49, el punto deberá de recorrerse dos posiciones a la derecha, para terminar con el resultado 0xE6EC.928.

A partir de la línea 160 se terminan todos los procesos con lógica secuencial y en la 162 se da paso a los de lógica combinatorial, como primer elemento se tiene la identificación del bit de signo, y la partición del registro de entrada en dos bytes para poder operarlos dentro del componente. La asignación del registro r\_Show hacia o\_Displays para poder controlar el encendido de los transistores PNP que controlan los displays y la salida negada del registro del bit de signo para de esa forma encender un led si el número es negativo.

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 --Designed only for CycloneIV displays
5 entity BCDto7Segs is
6     port (
7         i_DispPoint    : in std_logic;           --Display point enable
8         i_ChipEn        : in std_logic;           --Chip enable (works with PNP
transistors)
9         i_Number        : in std_logic_vector(3 downto 0); --Number that will be
displayed
10        o_DisplaySegments : out std_logic_vector(6 downto 0); --Display segments
vector
11        o_DispEnable      : out std_logic;           --Out pin to enable the display
(work with i_ChipEn)
12        o_DispPoint       : out std_logic           --Turns up or down the display point
(work with i_DispPoint)
13    );
14 end entity BCDto7Segs;
15
16 architecture rtl of BCDto7Segs is
17     --Segments connection on board
18     --a - 128
19     --b - 121
20     --c - 125
21     --d - 129
22     --e - 132
23     --f - 126
24     --g - 124
25     --dp - 127
26     --Enable Disp 1 - 133
27     --Enable Disp 2 - 135
28     --Enable Disp 3 - 136
29     --Enable Disp 4 - 137
30 begin
31
32     with i_Number select
33         o_DisplaySegments <= "1000000" when "0000",
34                                "1111001" when "0001",
35                                "0100100" when "0010",
36                                "0110000" when "0011",
37                                "0011001" when "0100",
38                                "0010010" when "0101",
39                                "0000010" when "0110",
40                                "1111000" when "0111",
41                                "0000000" when "1000",
42                                "0011000" when "1001",
43                                "0001000" when "1010",
44                                "0000011" when "1011",
45                                "1000110" when "1100",

```

```

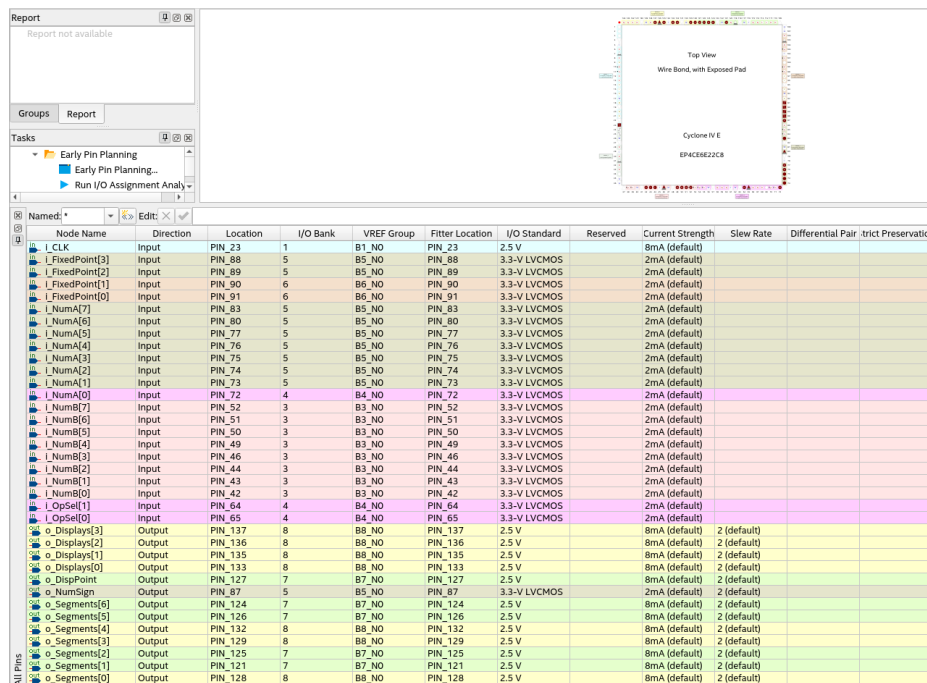
46      "0100001" when "1101",
47      "0000110" when "1110",
48      "0001110" when others;
49
50  o_Disable <= not i_ChipEn;
51
52  o_DisPoint <= i_DisPoint;
53
54  end architecture;

```

Figura 2.8 Código del decodificador de BCD a 7 segmentos

Este código muy simple ya que sólo es un tabla de igualaciones combinacional con una sentencia “with select”.

## ASIGNACIÓN DE PINES EN LA FPGA



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Tri-state Preservation
i_CLK	Input	PIN_23	1	B1_NO	PIN_23	2.5 V		8mA (default)			
i_FixedPoint[3]	Input	PIN_88	5	B5_NO	PIN_88	3.3-V LVCMOS		2mA (default)			
i_FixedPoint[2]	Input	PIN_89	5	B5_NO	PIN_89	3.3-V LVCMOS		2mA (default)			
i_FixedPoint[1]	Input	PIN_90	6	B6_NO	PIN_90	3.3-V LVCMOS		2mA (default)			
i_FixedPoint[0]	Input	PIN_91	6	B6_NO	PIN_91	3.3-V LVCMOS		2mA (default)			
i_Numa[7]	Input	PIN_83	5	B5_NO	PIN_83	3.3-V LVCMOS		2mA (default)			
i_Numa[6]	Input	PIN_80	5	B5_NO	PIN_80	3.3-V LVCMOS		2mA (default)			
i_Numa[5]	Input	PIN_77	5	B5_NO	PIN_77	3.3-V LVCMOS		2mA (default)			
i_Numa[4]	Input	PIN_76	5	B5_NO	PIN_76	3.3-V LVCMOS		2mA (default)			
i_Numa[3]	Input	PIN_75	5	B5_NO	PIN_75	3.3-V LVCMOS		2mA (default)			
i_Numa[2]	Input	PIN_74	5	B5_NO	PIN_74	3.3-V LVCMOS		2mA (default)			
i_Numa[1]	Input	PIN_73	5	B5_NO	PIN_73	3.3-V LVCMOS		2mA (default)			
i_Numa[0]	Input	PIN_72	4	B4_NO	PIN_72	3.3-V LVCMOS		2mA (default)			
i_Numb[7]	Input	PIN_52	3	B3_NO	PIN_52	3.3-V LVCMOS		2mA (default)			
i_Numb[6]	Input	PIN_51	3	B3_NO	PIN_51	3.3-V LVCMOS		2mA (default)			
i_Numb[5]	Input	PIN_50	3	B3_NO	PIN_50	3.3-V LVCMOS		2mA (default)			
i_Numb[4]	Input	PIN_49	3	B3_NO	PIN_49	3.3-V LVCMOS		2mA (default)			
i_Numb[3]	Input	PIN_46	3	B3_NO	PIN_46	3.3-V LVCMOS		2mA (default)			
i_Numb[2]	Input	PIN_44	3	B3_NO	PIN_44	3.3-V LVCMOS		2mA (default)			
i_Numb[1]	Input	PIN_43	3	B3_NO	PIN_43	3.3-V LVCMOS		2mA (default)			
i_Numb[0]	Input	PIN_42	3	B3_NO	PIN_42	3.3-V LVCMOS		2mA (default)			
i_OpSel[1]	Input	PIN_64	4	B4_NO	PIN_64	3.3-V LVCMOS		2mA (default)			
i_OpSel[0]	Input	PIN_65	4	B4_NO	PIN_65	3.3-V LVCMOS		2mA (default)			
o_Displays[3]	Output	PIN_137	8	B8_NO	PIN_137	2.5 V		8mA (default)	2 (default)		
o_Displays[2]	Output	PIN_136	8	B8_NO	PIN_136	2.5 V		8mA (default)	2 (default)		
o_Displays[1]	Output	PIN_135	8	B8_NO	PIN_135	2.5 V		8mA (default)	2 (default)		
o_Displays[0]	Output	PIN_133	8	B8_NO	PIN_133	2.5 V		8mA (default)	2 (default)		
o_DisPoint	Output	PIN_127	7	B7_NO	PIN_127	2.5 V		8mA (default)	2 (default)		
o_NumSign	Output	PIN_87	5	B5_NO	PIN_87	3.3-V LVCMOS		2mA (default)	2 (default)		
o_Segments[6]	Output	PIN_124	7	B7_NO	PIN_124	2.5 V		8mA (default)	2 (default)		
o_Segments[5]	Output	PIN_126	7	B7_NO	PIN_126	2.5 V		8mA (default)	2 (default)		
o_Segments[4]	Output	PIN_132	8	B8_NO	PIN_132	2.5 V		8mA (default)	2 (default)		
o_Segments[3]	Output	PIN_129	8	B8_NO	PIN_129	2.5 V		8mA (default)	2 (default)		
o_Segments[2]	Output	PIN_125	7	B7_NO	PIN_125	2.5 V		8mA (default)	2 (default)		
o_Segments[1]	Output	PIN_121	7	B7_NO	PIN_121	2.5 V		8mA (default)	2 (default)		
o_Segments[0]	Output	PIN_128	8	B8_NO	PIN_128	2.5 V		8mA (default)	2 (default)		

Figura 2.9: Asignación de pines en la FPGA

Cabe aclarar que los pines que salen hacia los displays de 7 segmentos se declararon como 2.5V (definición estándar de la FPGA) para de esta forma reducir la intensidad luminosa de los displays. (esto se hizo por comodidad del alumno al programar de noche)

## ESQUEMÁTICO DE CONEXIÓN

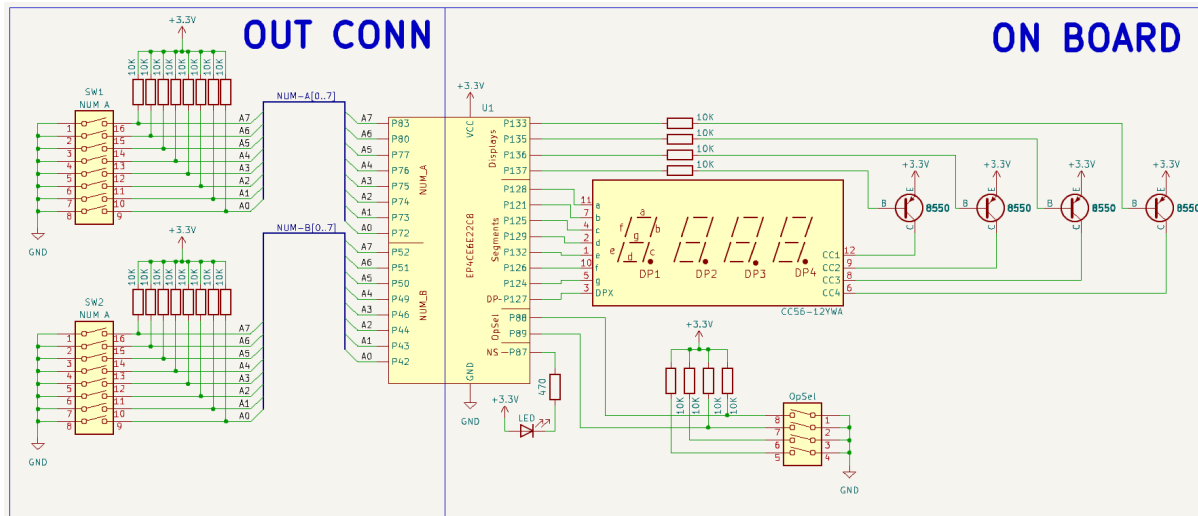


Figura 2.10: Esquemático de conexión

Como se puede observar en el esquemático de la figura 2.11, el cuadro de la izquierda es lo que se encuentra fuera de la placa de desarrollo de la FPGA y el cuadro de la derecha lo que internamente ya está conectado.

## REPORTE DE TIEMPOS

	Data Port	Clock Port	Rise ▲	Fall	Clock Edge	Clock Reference
1	o_DisPpoint	i_CLK	11.336	11.078	Rise	i_CLK
2	▼ o_Segments[*]	i_CLK	10.984	10.872	Rise	i_CLK
1	o_Segments[4]	i_CLK	10.984	10.872	Rise	i_CLK
2	o_Segments[2]	i_CLK	10.548	10.305	Rise	i_CLK
3	o_Segments[3]	i_CLK	10.452	10.201	Rise	i_CLK
4	o_Segments[6]	i_CLK	10.331	10.131	Rise	i_CLK
5	o_Segments[0]	i_CLK	10.315	10.028	Rise	i_CLK
6	o_Segments[5]	i_CLK	10.225	10.035	Rise	i_CLK
7	o_Segments[1]	i_CLK	9.711	9.547	Rise	i_CLK
3	▼ o_Displays[*]	i_CLK	10.640	10.587	Rise	i_CLK
1	o_Displays[2]	i_CLK	10.640	10.587	Rise	i_CLK
2	o_Displays[3]	i_CLK	9.823	9.578	Rise	i_CLK
3	o_Displays[0]	i_CLK	9.326	9.172	Rise	i_CLK
4	o_Displays[1]	i_CLK	9.167	9.013	Rise	i_CLK

Figura 2.11: Reporte de propagación del retraso

Como se puede observar en la figura 2.11 el tiempo más grande de reacción al ciclo de reloj es de 11.336ns por lo que este sería el mínimo periodo que esta aplicación puede trabajar.

Es decir que esta aplicación no podría trabajar con un reloj mayor a 90MHz (1/11.336ns).

## REPORTE DE ÁREA

Flow Summary	
<<Filter>>	
Flow Status	Successful - Sat Nov 18 20:17:43 2023
Quartus Prime Version	22.1std.2 Build 922 07/20/2023 SC Lite Edition
Revision Name	ALU8BitsDisplaysFP
Top-level Entity Name	ALU8BitsDisplaysFP
Family	Cyclone IV E
Device	EP4CE6E22C8
Timing Models	Final
Total logic elements	120 / 6,272 ( 2 % )
Total registers	19
Total pins	36 / 92 ( 39 % )
Total virtual pins	0
Total memory bits	0 / 276,480 ( 0 % )
Embedded Multiplier 9-bit elements	1 / 30 ( 3 % )
Total PLLs	0 / 2 ( 0 % )

*Figura 2.12: Reporte de área*

En este reporte se puede apreciar que el número de elementos lógicos que el diseño necesita son 120, lo cuál es muy poco, como lo indica la figura 2.12, 2% de la capacidad total de la FPGA y el 39% de sus pines.

Por último cabe destacar que se está haciendo uso de un multiplicador embebido de 9 bits, el cuál según la figura 2.12 representaría un 3% de toda la capacidad de multiplicadores.

## CÓDIGO DEL TESTBENCH

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_unsigned.all;
4 use IEEE.numeric_std.all;
5
6 entity tb_ALU8BitsDisplaysFP is
7 end entity tb_ALU8BitsDisplaysFP;
8
9 architecture rtl of tb_ALU8BitsDisplaysFP is
10
11     component ALU8BitsDisplaysFP is
12         generic(
13             g_PSC_COUNTS : integer := 100000
14         );
15         port (
16             i_CLK      : in std_logic;
17             i_NumA      : in signed(7 downto 0);
18             i_NumB      : in signed(7 downto 0);
19             i_FixedPoint : in std_logic_vector(3 downto 0);
20             i_OpSel     : in std_logic_vector(1 downto 0);
21             o_Displays  : out std_logic_vector(3 downto 0);
22             o_Segments  : out std_logic_vector(6 downto 0);
23             o_DispPoint : out std_logic;
24             o_NumSign   : out std_logic
25         );
26     end component ALU8BitsDisplaysFP;
27
```

```

28  type t_Nums is (t_Num1, t_Num2, t_Num3, t_Num4, t_Num5);
29  type t_Ops is (t_Sum, t_Sub, t_Mult);
30
31  signal i_CLK      : std_logic := '0';
32  signal i_NumA     : signed(7 downto 0) := (others => '0');
33  signal i_NumB     : signed(7 downto 0) := (others => '0');
34  signal i_FixedPoint : std_logic_vector(3 downto 0) := (others => '0');
35  signal i_OpSel     : std_logic_vector(1 downto 0) := (others => '0');
36  signal o_Displays  : std_logic_vector(3 downto 0) := (others => '0');
37  signal o_Segments  : std_logic_vector(6 downto 0) := (others => '0');
38  signal o_DisPoint  : std_logic := '0';
39  signal o_NumSign   : std_logic := '0';
40  signal r_Nums      : t_Nums := t_Num1;
41  signal r_i         : t_Nums := t_Num1;
42  signal r_Ops       : t_Ops := t_Sum;
43
44  begin
45
46    U1 : ALU8BitsDisplaysFP
47      generic map(
48        (1)
49      )
50      port map(
51        i_CLK      => i_CLK,
52        i_NumA     => i_NumA,
53        i_NumB     => i_NumB,
54        i_FixedPoint => i_FixedPoint,
55        i_OpSel     => i_OpSel,
56        o_Displays  => o_Displays,
57        o_Segments  => o_Segments,
58        o_DisPoint  => o_DisPoint,
59        o_NumSign   => o_NumSign
60      );
61
62    clk_proc : process
63    begin
64      i_CLK <= '1';
65      wait for 1 ps;
66      i_CLK <= '0';
67      wait for 1 ps;
68    end process;
69
70    stim_proc : process
71    begin
72      for r_i in t_Nums loop
73        r_Nums <= r_i;
74        r_Ops <= t_Sum;
75        wait for 10 ps;
76        r_Ops <= t_Sub;
77        wait for 10 ps;
78        r_Ops <= t_Mult;

```

```

79     wait for 10 ps;
80 end loop;
81 wait;
82 end process;
83
84 declarations_proc : process(r_Ops, r_Nums)
85 begin
86     case r_Ops is
87         when t_Sum =>
88             i_OpSel <= "00";
89         when t_Sub =>
90             i_OpSel <= "01";
91         when t_Mult =>
92             i_OpSel <= "11";
93     end case;
94
95     case r_Nums is
96         when t_Num1 =>
97             i_NumA <= "11010100";
98             i_NumB <= "01010101";
99         when t_Num2 =>
100             i_NumA <= "11010100";
101             i_NumB <= "01010101";
102         when t_Num3 =>
103             i_NumA <= "11110100";
104             i_NumB <= "11110101";
105         when t_Num4 =>
106             i_NumA <= "11010010";
107             i_NumB <= "10011010";
108         when t_Num5 =>
109             i_NumA <= "10010001";
110             i_NumB <= "10010001";
111     end case;
112 end process;
113
114 end architecture rtl;

```

*Figura 2.14 Código del Testbench*

De las líneas 11 a la 26 se puede encontrar la declaración del componente que se va a simular. De la 28 a 29 se declaran dos type, uno para almacenar el valor de 4 número definidos y otro la operaciones, esto para facilitar dentro del proceso de estímulos la secuencia de datos que se le da al componente.

De la 31 a la 42, los signal que se encargarán de darle un valor legible al simulador, y todas ellas están inicializadas con ceros para que el simulador no tenga problemas al mostrar el resultado de las variables.

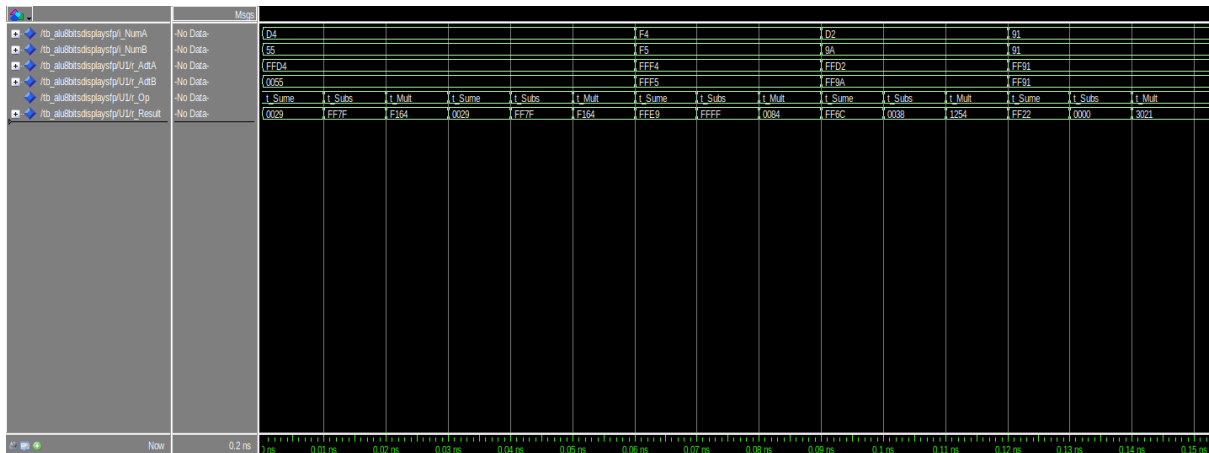
De las líneas 46 a 60 se hace la instanciación del componente en donde podemos ver que el generic se reemplaza con un 1, para que la señal de los displays no se preescale, y que como se mencionó anteriormente; no se tenga que esperar 100K pulsos de reloj para que el circuito muestre un cambio en la salida.



Seguido de ello está el proceso de reloj el cuál sólo simula una señal de reloj de 2 ps de periodo, luego está el proceso de estímulos con un for para poder darle todos los valores que se encuentran en t\_Nums a nuestro componente, así como la selección de operaciones.

Como tercer proceso se encuentran las tablas de conversiones para los type, si bien estos pudieron ser implementados de manera combinacional y con sentencias with select, durante todo el proceso de desarrollo fue evolucionando la región de código y antes si se empleaban cambios dentro del proceso, sin embargo, como legado terminó así y se decidió dejar de ese modo por si en una futura prueba se necesitan cambios secuenciales en las declaraciones.

## CAPTURAS DE PANTALLA DE LA SIMULACIÓN



*Figura 2.15 Captura de simulación*

En la simulación se le da un enfoque sólo al correcto funcionamiento de la ALU debido a que el multiplexado de displays fue mucho más práctico probarlo en la placa.

## PROBLEMAS ENCONTRADOS

### Problemas referentes a VHDL

#### 1. Interpretación de la signals al momento de hacer la síntesis

Hasta antes de la realización de esta práctica los alumnos creían que todas aquellas signals que formaban parte de un “process” eran interpretadas como flip-flops, debido a que estos procesos debían ser controlados por una señal de reloj o por el cambio del estado de alguna señal, sin embargo, esto no es del todo cierto; cuándo el ingeniero deja una sentencia sin un estado anterior claro, entonces el sintetizador interpreta esto como un efecto memoria, por lo que asocia este comportamiento al de los latches SR asíncronos, sin embargo, estos son nocivos para el correcto funcionamiento del diseño, esto es debido a que el comportamiento correcto del latch está asegurado sólo por medio ciclo de reloj [9].

Solución: Esto sucedió debido a que procesos de naturaleza combinacional (negar con compuertas not y realizar sumas o restas) fueron colocados dentro de procesos controlados por señales periódicas, es decir, “clocked process”, por lo que esto generaba que el sintetizador infiriera registros como latches lanzando la advertencia mostrada en [10] (10041

Inferred latch for “signal” at “file.vhd”), lo que se necesitó hacer fue mover este proceso fuera del process para de este modo ser trabajado como circuitería de procesamiento en paralelo y evitar incluso la necesidad de usar registros.

## **2. Existencia de los generics**

[11] Como se verá en la sección de problemas referentes a la simulación, el uso de los generics fue necesario para resolver ese problema, sin embargo, para ese momento también se desconocía como es que se usaban los generics y que implicaban en código vhdl.

Solución: Después de una investigación se descubrió que los generics son una forma de declarar de forma local una constante que puede ser asignada en el momento en el que dicho componente se instancia, por lo que este mecanismo hace posible que se puedan instanciar múltiples veces un componente de VHDL y asignar diferentes valores a sus “generics” para obtener distintos comportamientos y así también obtener código reutilizable.

### **Problemas referentes a la lógica de diseño y RTL**

#### **2. Manejo del punto fijo**

Como se comentó anteriormente en la sección de RTL y RTL aproximado, el manejo del punto fijo se confirió al componente “Display Driver”, sin embargo había un tema que dicho componente tenía complicaciones para ser programado, el punto fijo tiene que ser recorrido cuándo se realiza un multiplicación.

Solución: Que la ALU notifique al Display Driver que operación se seleccionó para de ese modo recorrer el punto o no dentro del Display Driver.

### **Problemas referentes a la simulación**

#### **1. Manejo de preescalamientos en simulación**

Cuándo se pretendía simular el código del display driver con la ALU para ver si este funcionaba de forma correcta se tenía el problema de había que esperar que la señal de reloj pasara 100K cuentas para poder ver un cambio en los displays.

Solución: Hacer uso de los generics en la ALU y en el display driver para poder controlar en la instancia del testbench el número de cuentas que el preescalador requiere para entregar un pulso o cambiar este, es decir, controlar por medio del generic la frecuencia del preescalador.

### **Problemas referentes a conexiones en físico**

#### **1. Conexiones de múltiples cables**

Debido a que se necesitaban dos números de 8 bits con punto fijo para poder operar en la ALU, se necesitaban dos DIP Switches (como se ve en la figura 2.10) y eso se traducía en 16 cables de conexión hacia la FPGA, más otros 16 para la conexión a tierra de los DIP y otros 16 para la conexión a VCC. Esto ocasionaba que el sistema en general fuera muy susceptible a errores de conexión que podían ocasionar malos diagnósticos.

Solución: Construir dos circuitos con DIP Switches y LEDs para poder visualizar con certeza que valor se estaba enviando a la FPGA y evitar errores por falso contacto de los cables en protoboard o en placa.

## **2. Lenta interpretación humana sobre la codificación binaria**

En comparación con las máquinas el ser humano es ineficiente y lento para realizar operaciones aritméticas, por lo que para poder interpretar los operandos que son enviados a la FPGA así como los resultados y la operación, se necesita de un tiempo muy largo (en comparación con las máquinas).

Solución: Programar otro sistema embebido que interprete los operandos y el código de la operación con un STM32F103C8T6 y una OLED SSD1306, para de este modo sólo interpretar el resultado que la FPGA arroja.

## **CONCLUSIONES**

### **Pedro Eduardo Rojo Carrillo**

El desarrollo de esta práctica permitió comprender y terminar de afirmar el conocimiento que se tenía sobre las FPGA's y la aritmética de punto fijo, permitió así mismo conocer sobre la respuesta del ojo humano, ya que para poder elegir la frecuencia de muestro o los "FPS" que los displays de 7 segmentos iban a mostrar se tuvo que investigar dicha información; se conoció mucho acerca de la interpretación del sintetizador sobre el código escrito en VHDL y el proceso que lleva a cabo para realizar la lógica de transferencia de los registros, ya que como se mencionó anteriormente si el ingeniero no especifica un comportamiento definido entonces el sintetizador infiere el registro como un Latch SR, cosa que, en código se puede ver cerrando los casos con un when others si se una sentencia case when o con un else si es que utiliza un if, también el alumno al adentrarse a temas como la representación de los número con signo comprendió el origen y el porqué de los complementos, se llegó a la conclusión de que los complementos son vistos de forma implícita en los primeros años de educación cuándo se ve la resta, porque cuándo se le incita al niño a restar se le propone calcular cuántos números faltan para llegar a n número desde x número efectuando así la resta, y eso es lo que son los complementos, el número complementario que debe de sumarse al número original para llegar a la base en el que el número está codificado. Otro gran conocimiento que el alumno pudo obtener es la interpretación de los RTL que salen como resultado de la programación en el lenguaje de descripción de hardware, aprender a interpretar que componentes fueron creados y como están conectados, aprender a interpretar los buses que están representados dentro de los esquemáticos y que sugieren simbologías como las que se pueden ver en la figura 2.4 o 2.5. el aprender que un restador es lo mismo que un sumador que primero calcula el complemento a 2 de alguno de los operandos y que dicha acción se simboliza con una negación a la entrada de los componentes.

### **Andrea Joanelle Hernández Alvarado**

Durante la práctica se revisó el funcionamiento del punto fijo y su aritmética a nivel de electrónica digital así como las implicaciones que tiene, por ejemplo la suma y la resta no tienen que mover el punto fijo dentro del resultado, y la multiplicación si; se reforzó el diseño de unidades lógicas aritméticas, ya que en este caso se tuvo que llevar a operar números de 8 bits. Una de las cosas que complicaron el código y la compresión del mismo fue la multiplicación, ya que como se mencionó anteriormente el producto de esa operación da como resultado un vector del doble del tamaño del operando, es decir, si se multiplican 8 bits por 8 bits da como resultado un número de 16 bits (cuándo son números grandes); algunos otros aspectos que aportaron al conocimiento de la alumna fueron el diseño del preescalador, ya que se comprendió a mayor profundidad el funcionamiento de los timers a partir de cuentas y la forma en que estos pueden proporcionar una señal preescalada, si bien no son exactamente lo mismo, el abstraer el conocimiento de los timers para poder reducir la frecuencia de actualización de los displays fue de alta utilidad. Esta práctica también permitió que se conociera a mayor profundidad la naturaleza de los process y los algoritmos de lógica secuencial, ya que antes de realizar esta práctica la alumna no poseía un conocimiento claro de estos.

## BIBLIOGRAFÍA

- [1] Instituto de Tecnologías Educativas. “Electrónica digital”. Insitituto de Eecnologías Educativas. Accedido el 18 de octubre de 2023. [En línea]. Disponible: [http://recursostic.educacion.es/secundaria/tecnologia/controladora/contenido/anexos/introduccion\\_electronica/electronica\\_digital/electronica\\_digital2.htm#:~:text=Se%20denomina%20Complemento%20de%20una,el%208-3=%205.](http://recursostic.educacion.es/secundaria/tecnologia/controladora/contenido/anexos/introduccion_electronica/electronica_digital/electronica_digital2.htm#:~:text=Se%20denomina%20Complemento%20de%20una,el%208-3=%205.)
- [2] F. Zuñiga. “Complementos a una base numérica”. SlideShare. Accedido el 19 de octubre de 2023. [En línea]. Disponible: <https://es.slideshare.net/fabianzunigav/complemento-a-las-bases>.
- [3] Timur. “Complementos numéricos”. PLANETCALC. Accedido el 19 de octubre de 2023. [En línea]. Disponible: <https://es.planetcalc.com/8574/#:~:text=El%20número%20del%20complemento%20es,a%203%20para%20formar%2010.>
- [4] R. Chen. “Complementos a la base disminuida”. SlideShare. Accedido el 19 de octubre de 2023. [En línea]. Disponible: <https://es.slideshare.net/robertochen56/complemento-a-la-base-disminuida>.
- [5] “Formato de signos”. uaemex. Accedido el 19 de octubre de 2023. [En línea]. Disponible: <http://ri.uaemex.mx/bitstream/handle/20.500.11799/66213/secme-28784.pdf?sequence=1>.
- [6] R. Ibañez. “El sistema duodecimal, o si los humanos hubiésemos tenido seis dedos en las manos”. Cuaderno de cultura científica. Accedido el 18 de octubre de 2023. [En línea]. Disponible: <https://culturacientifica.com/2022/01/26/el-sistema-duodecimal-o-si-los-humanos-hubiesemos-tenido-seis-dedos-en-las-manos/>.
- [7] S. Noriega. “Representación de números binarios en punto fijo y punto flotante.” Tesla - Facultad de Ingeniería - UNLP. Accedido el 21 de octubre de 2023. [En línea]. Disponible: <https://catedra.ing.unlp.edu.ar/electrotecnia/islyd/apuntes/apptofijoyflotante2003.pdf>.
- [8] Rodrigo Alonso. “¿Cuántos FPS puede ver realmente el ojo del ser humano?” HardZone. Accedido el 22 de octubre de 2023. [En línea]. Disponible: <https://hardzone.es/tutoriales/rendimiento/fps-ojo-humano/>.
- [9] J.Jensen. “Why latches are bad and how to avoid them - VHDLwhiz”. VHDLwhiz. Accedido el 23 de noviembre de 2023. [En línea]. Disponible: <https://vhdlwhiz.com/why-latches-are-bad/>.
- [10] “Quartus 20.1 and warnings about Latches”. Home - Intel Community. Accedido el 28 de octubre de 2023. [En línea]. Disponible: <https://community.intel.com/t5/Intel-Quartus-Prime-Software/Quartus-20-1-and-warnings-about-Latches/td-p/1522272>.
- [11] John. “Writing reusable VHDL code using generics and generate statements”. FPGA Tutorial. Accedido el 28 de octubre de 2023. [En línea]. Disponible: <https://fpgatutorial.com/vhdl-generic-generate/>.