
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 Joanellie Hernández Alvarado 19121121

Pedro Eduardo Rojo Carrillo 19121155

Profesor: Arturo Méndez Patiño

Morelia Mich; a 9 de octubre del 2023.

OBJETIVO

- 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.

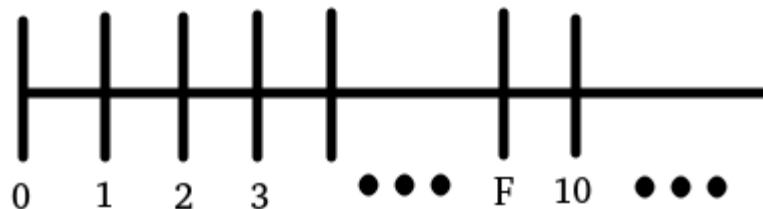


Figura 2.1: Recta numérica hexadecimal

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

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

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

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[], 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.

$$.D_{-1}B^{-1}+D_{-2}B^{-2}+...+D_{-n}B^{-n}$$

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. 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 media del número, lo que quiere decir que se contará con 4 bits de parte entera, 4 bits de parte fraccionaria y para el resultado 8 y 8 bits.

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 las que se usarán en la práctica serán 4 los codificadores que se instanciarán. La aparición de estos componentes instanciados se puede apreciar en la figura 2.2.

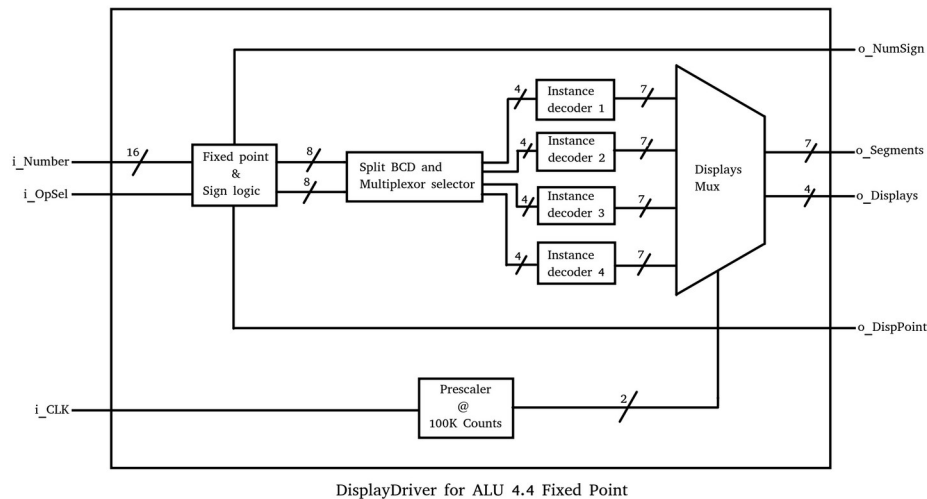


Figura 2.2: RTL Aproximado del "Display Driver"

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 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. 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 para las operaciones internas y otro para el despliegue del resultado. Para las operaciones internas 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 "-". Sin embargo para la interpretación del resultado se escogió magnitud signada gracias a que le facilita al humano por mucho la comprensión de los resultados. Por lo que para convertir de

complemento A2 a magnitud signada se deben de realizar unos ligeros ajustes, dichos ajustes se pueden 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.

Digital Tube

共阳数码管

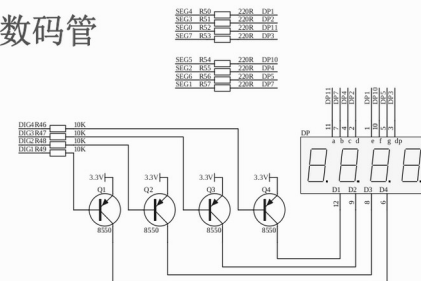


Figura 2.3: Conexión de los displays en la placa de desarrollo

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, cabe destacar que dentro de la entidad de la ALU se realiza un ligero ajuste en la información que el display driver recibe, dicho ajuste se tratará con mayor detalle en las secciones de código y problemas.

RTL

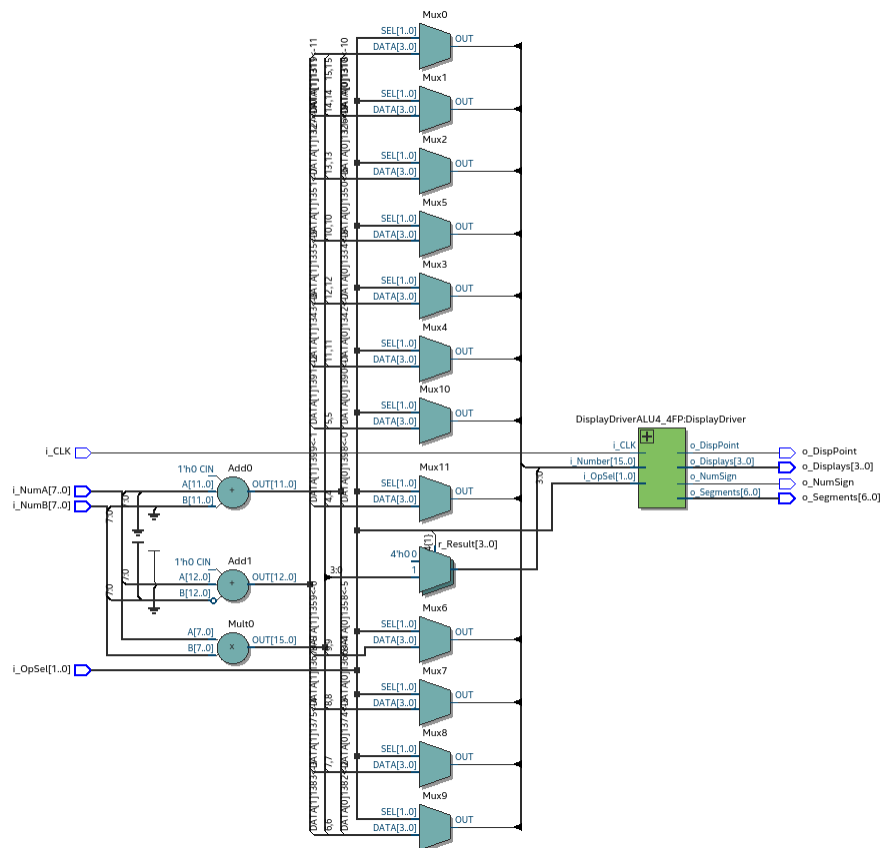


Figura 2.4: RTL del diseño general

Como se puede apreciar en la figura 2.4, al comienzo los operandos son llevados a un sumador (Add0) cuyo acarreo de entrada está desconectado (el vector de entrada es un operando y no el resultado de una operación por lo que no se necesita), otro sumador (Add1) el cuál es en realidad un restador, sin embargo, 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 que arroja su resultado a un vector de 16 bits. 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 si tomará el bit de signo del bit 8 (para el caso de una suma o resta) o del bit 15 (para el caso de una multiplicación). Es de importancia notar como aquellos vectores que entran a los operadores antes descritos, son de 12 bits, esto es para que los sumadores entreguen el resultado en 12 bits y no trunquen el acarreo salido de la operación (en el caso de la suma y resta) y que también se pueda multiplexar en nibbles sin mayor dificultad. Para que los operandos puedan ser de 12 bits algunas partes del vector tienen que ir conectados directo a tierra (hacer que siempre sean cero), justo como lo marca el RTL.

Por el lado de eficiencia operativa y energética el hecho de realizar las operaciones a pesar de que no sean seleccionadas no implica ningún gasto de energía extra, 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 y por ende no afecta el rendimiento.

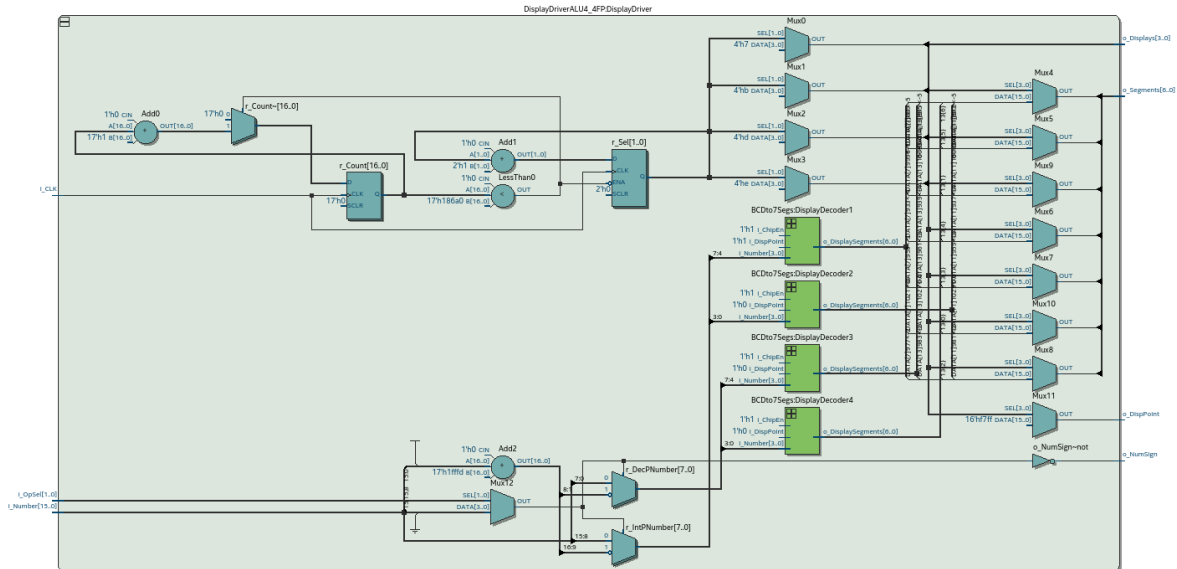


Figura 2.5: RTL del DisplayDriver

En la figura 2.5 puede apreciarse que el RTL aproximado termino siendo algo parecido al RTL aproximado mostrado en la figura 2.2, puede apreciarse que el preescalador fue 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.

CÓDIGO

```
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 ALU8BitsDisplaysFP is
7     generic(
8         g_PSC_COUNTS : integer := 100000
9     );
10    port (
11        i_CLK          : in std_logic;
12        i_NumA          : in std_logic_vector(7 downto 0);
13        i_NumB          : in std_logic_vector(7 downto 0);
14        i_OpSel         : in std_logic_vector(1 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 ALU8BitsDisplaysFP;
21
22 architecture rtl of ALU8BitsDisplaysFP is
23
24     component DisplayDriverALU4_4FP is
25         generic(
26             g_PSC_COUNTS : integer := 100000
27         );
28         port (
29             i_CLK          : in std_logic;
30             i_OpSel        : in std_logic_vector(1 downto 0);
31             i_Number       : in std_logic_vector(15 downto 0); --Number with fixed point in
the middle (0x00.00)
32             o_Displays     : out std_logic_vector(3 downto 0);
33             o_Segments     : out std_logic_vector(6 downto 0);
34             o_DispPoint    : out std_logic;
35             o_NumSign      : out std_logic
36         );
37     end component DisplayDriverALU4_4FP;
38
39     signal r_Result      : std_logic_vector(15 downto 0);
40     signal r_Mult         : std_logic_vector(15 downto 0);
41     signal r_Sume        : std_logic_vector(11 downto 0);
42     signal r_Sub          : std_logic_vector(11 downto 0);
43     signal r_AdtA        : std_logic_vector(11 downto 0) := (others => '0');
44     signal r_AdtB        : std_logic_vector(11 downto 0) := (others => '0');
45
46 begin
47
48
```

```

49 DisplayDriver : DisplayDriverALU4_4FP
50 generic map (
51     g_PSC_COUNTS => g_PSC_COUNTS
52 )
53 port map(
54     i_CLK      => i_CLK,
55     i_OpSel    => i_OpSel,
56     i_Number   => r_Result,
57     o_Displays => o_Displays,
58     o_Segments => o_Segments,
59     o_DispPoint => o_DispPoint,
60     o_NumSign  => o_NumSign
61 );
62
63 r_AdtA(7 downto 0) <= i_NumA;
64
65 r_AdtB(7 downto 0) <= i_NumB;
66
67 r_Sume <= r_AdtA + r_AdtB;
68
69 r_Subst <= r_AdtA - r_AdtB;
70
71 r_Mult <= i_NumA * i_NumB;
72
73 with i_OpSel select r_Result(15 downto 12) <=
74     r_Sume(11 downto 8) when "00", --Sume
75     r_Subst(11 downto 8) when "01", --Substraction
76     r_Mult(15 downto 12) when others; --Multiplication
77 with i_OpSel select r_Result(11 downto 8) <=
78     r_Sume(7 downto 4) when "00", --Sume
79     r_Subst(7 downto 4) when "01", --Substraction
80     r_Mult(11 downto 8) when others; --Multiplication
81 with i_OpSel select r_Result(7 downto 4) <=
82     r_Sume(3 downto 0) when "00", --Sume
83     r_Subst(3 downto 0) when "01", --Substraction
84     r_Mult(7 downto 4) when others; --Multiplication
85 with i_OpSel select r_Result(3 downto 0) <=
86     (others => '0') when "00", --Sume
87     (others => '0') when "01", --Substraction
88     r_Mult(3 downto 0) when others; --Multiplication
89
90
91 end architecture rtl;

```

Figura 2.6 Código de la ALU de 8 bits

Desde la línea 6 a la 20 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 22 a 46 se encuentran las declaraciones de lo que se utilizará en la arquitectura de la entidad; la declaración del componente “DisplayDriver” y de los registros o signals que se utilizarán dentro de la entidad. r_Result es el registro que guarda el resultado de la operación, r_Mult, r_Sume y r_Subs son los registros que almacenarán los resultados de la multiplicación, suma y resta respectivamente; r_AdtA y r_AdtB son los registros que hacen posible el comportamiento descrito en el RTL (hacer que la suma y resta guarden el acarreo o adeudo), 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.

De las líneas 48 a 60 se encuentra la instanciación del componente “DisplayDriver” y sus respectivas conexiones.

De las líneas 62 a 70 se hacen las operaciones y de la línea 72 a la 87 se realiza el ajuste de los números según sea la operaciones; a continuación se explicará el porque de dicho ajuste.

El display driver posee la convención de bits mostrado en la figura 2.7 y que será explicado a continuación

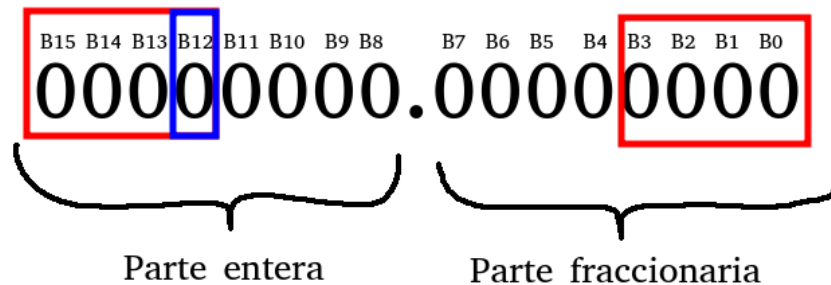


Figura 2.7: Acomodo del punto fijo en la ALU

Aquellos bits encerrados en el recuadro rojo son los bits que tendrán uso sólo cuándo la multiplicación se esté efectuando, el bit 12 encerrado en el cuadro azul se utilizará sólo cuando se presente un acarreo o adeudo en las operaciones así como también en la multiplicación y aquellos bits que no están encerrados son los que siempre estarán en uso ya que se son los que tomarán el valor de la suma o de la resta. Debido a este acomodo especial es necesario indicarle al DisplayDriver que operación es la que se está llevando acabo ya que el bit de signo cuando se efectúa una resta es el B12 pero cuándo se está efectuando una multiplicación entonces el bit de signo es el B15.

Cuándo se efectúa una operación el RTL por defecto deja dicho resultado en el bit 0 del vector y partir de ahí es que se comienzan a poner los demás bits, sin embargo, para este

caso en específico eso generaría errores, ya que dependiendo de la operación el LSB se colocará en la posición 4 o 0. Por lo que eso llevó a un problema que se tratará de manera detallada en la sección de problemas resueltos, en esta sección vamos a analizar como funcionan estas líneas de código.

Como se puede apreciar en las líneas 72 a 87 de la figura 2.6 el registro que enviará el resultado de la operación al display driver irá tomando su valor de nibble en nibble, ya que de este modo se puede controlar que bits aparecerán en cada parte del vector, el nibble más significativo (15 downto 12) tomará el valor del nibble de acarreo o adeudo resultante de la suma y la resta (nótese que los 3 bits más significativos del nibble siempre serán 0), mientras que si la operación es la multiplicación entonces este nibble tomará el valor del nibble más alto de la multiplicación; para el nibble subsecuente (11 downto 8) se toma el resultado entero de la suma o resta, y si la operación es multiplicación entonces se toma el segundo nibble de la parte entera del resultado; algo similar sucede con el siguiente nibble (7 downto 4) sólo que para el caso de suma y resta es el nibble fraccionario y para la multiplicación es tan sólo el primer nibble fraccionario; por último el nibble (3 downto 0), en este la suma y la resta nunca toman un valor, es decir, siempre son cero, ya que estas operaciones nunca utilizan ese nibble para operar, pues son números de 4.4 bits (4 bits para entero y 4 bits para fraccionario en punto fijo), caso contrario sucede con la multiplicación, pues en esta si se utilizan dichos nibbles.

```
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); --Number with fixed point in the
middle (0x00.00)
14        o_Displays      : out std_logic_vector(3 downto 0);
15        o_Segments      : out std_logic_vector(6 downto 0);
16        o_DispPoint     : out std_logic;
17        o_NumSign       : out std_logic
18    );
19 end entity DisplayDriverALU4_4FP;
20
21
22
23
24
25
26
27
```

```

28
29 architecture rtl of DisplayDriverALU4_4FP is
30
31     component BCDTo7Segs is
32     port (
33         i_DispPoint      : in std_logic;           --Display point enable
34         i_ChipEn         : in std_logic;           --Chip enable (works with PNP
transistors)
35         i_Number         : in std_logic_vector(3 downto 0); --Number that will be
displayed
36         o_DisplaySegments : out std_logic_vector(6 downto 0); --Display segments
vector
37         o_DispEnable     : out std_logic;           --Out pin to enable the display
(works with i_ChipEn)
38         o_DispPoint      : out std_logic           --Turns up or down the display
point (works with i_DispPoint)
39     );
40     end component BCDTo7Segs;
41
42     constant DISP_NEGATIVE_SIGN : std_logic_vector(6 downto 0) := "0111111";
43     constant DISP_POSITIVE_SIGN : std_logic_vector(6 downto 0) := "1111111";
44     --Multiplexor Registers
45     signal r_Sel                : std_logic_vector(1 downto 0) := "00";
46     signal r_Show               : std_logic_vector(3 downto 0) := "0000";
47     signal r_Number             : std_logic_vector(15 downto 0);
48     signal r_Count              : integer range 0 to 100000;
49     --Integer Part Registers
50     signal r_Segs1, r_Segs2      : std_logic_vector(6 downto 0);
51     signal r_SegsConn1, r_SegsConn2 : std_logic_vector(6 downto 0);
52     --Decimal Part Registers
53     signal r_Segs3, r_Segs4      : std_logic_vector(6 downto 0);
54     signal r_SegsConn3, r_SegsConn4 : std_logic_vector(6 downto 0);
55     --Signed Logic Registers
56     signal r_NumberSign          : std_logic;
57     signal r_IntPNumber, r_DecPNumber : std_logic_vector(7 downto 0);
58
59 begin
60     --Integer part displays
61     DisplayDecoder1 : BCDTo7Segs port map (
62         i_DispPoint  => '1',
63         i_ChipEN     => '1',
64         i_Number     => r_IntPNumber(7 downto 4),
65         o_DisplaySegments => r_SegsConn1
66     );
67
68     DisplayDecoder2 : BCDTo7Segs port map (
69         i_DispPoint  => '0',
70         i_ChipEN     => '1',
71         i_Number     => r_IntPNumber(3 downto 0),
72         o_DisplaySegments => r_SegsConn2
73     );

```



```

74  --Decimal part displays
75  DisplayDecoder3 : BCDTo7Segs port map (
76      i_Dispoint    => '0',
77      i_ChipEN      => '1',
78      i_Number       => r_DecPNumber(7 downto 4),
79      o_DisplaySegments => r_SegsConn3
80  );
81
82  DisplayDecoder4 : BCDTo7Segs port map (
83      i_Dispoint    => '0',
84      i_ChipEN      => '1',
85      i_Number       => r_DecPNumber(3 downto 0),
86      o_DisplaySegments => r_SegsConn4
87  );
88
89  clkCount : process(i_CLK)
90  begin
91      if rising_edge(i_CLK) then
92          if r_Count < g_PSC_COUNTS then
93              r_Count <= r_Count + 1;
94          else
95              r_Sel <= r_Sel + 1;
96              r_Count <= 0;
97          end if;
98      end if;
99  end process;
100
101  ShowDisplays : process(r_Sel)
102  begin
103      r_Segs1 <= r_SegsConn1;
104      r_Segs2 <= r_SegsConn2;
105      r_Segs3 <= r_SegsConn3;
106      r_Segs4 <= r_SegsConn4;
107
108      case r_Sel is
109          when "00" =>
110              r_Show <= "1110";
111          when "01" =>
112              r_Show <= "1101";
113          when "10" =>
114              r_Show <= "1011";
115          when "11" =>
116              r_Show <= "0111";
117          when others =>
118              r_Show <= "1111";
119      end case;
120
121      case r_Show is
122          when "0111" => --Display 1
123              o_Segments <= r_Segs1;
124              o_Dispoint <= '1'; --Turn off the point led

```

```

125     when "1011" => --Display 2
126         o_Segments <= r_Segs2;
127         o_Dispoint <= '0'; --Turn on the point led
128     when "1101" => --Display 3
129         o_Segments <= r_Segs3;
130         o_Dispoint <= '1'; --Turn off the point led
131     when others => --Display 4
132         o_Segments <= r_Segs4;
133         o_Dispoint <= '1'; --Turn off the point led
134     end case;
135 end process;
136
137 --Selecting the sign bit
138 with i_OpSel select r_NumberSign <=
139     '0' when "00",
140     i_Number(12) when "01",
141     i_Number(15) when others; --Multiplication
142
143 --Coverting A2 into Signed Magnitude
144 with r_NumberSign select r_Number <=
145     not(i_Number-1) when '1',
146     i_Number      when others;
147
148 --Integer Part
149 r_IntPNumber <= r_Number(15 downto 8);
150 --Decimal Part
151 r_DecPNumber <= r_Number(7 downto 0);
152
153 o_Displays <= r_Show;
154
155 o_NumSign <= not r_NumberSign;
156
157 end architecture rtl;

```

Figura 2.8 Código del display driver

En las líneas 6 a 19 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, esto es necesario para saber donde se encuentra el bit de signo el cuál es el 15 para la multiplicación, 12 para la resta y en el caso de la suma no hay debido a que Los operandos son números sin signo. 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 manejo 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 21 a 52 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 37 a 41 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). `r_Number` es el registro que guardará el número ya descomplementado (en caso de que llegue un número negativo), `r_Count` es el registro que guardará las cuentas del preescalador. Las siguientes declaraciones son para la parte entera del número, fraccionaria (llamada en código "decimal") y los registros de la lógica de signos.

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, donde el decoder 1 es para el nibble más significativo de la parte entera y el decoder 2 para el nibble menos significativo, en los análogamente en los decoders 3 y 4 sucede lo mismo.

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 81 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 mutliplexor de las pantallas. En la línea 93 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. Cabe destacar que en este case se decide prender siempre el punto en el display 2, ya que este es display que representa la cifra hexadecimal menos significativa de la parte entera.

A partir de la línea 127 se terminan todos los procesos con lógica secuencial y en la 130 se da paso a los de lógica combinacional, como primer elemento se tiene una sentencia "with select" para poder escoger el signo, en este caso se hace a partir de la entrada "OpSel" y se puede apreciar que si dicha entrada es una suma "00" entonces se asume siempre un bit de signo 0, es decir, siempre un número positivo; si es una resta "01" se selecciona el bit numero 12 del número, esto se puede ver en el recuadro azul de la figura 2.7 y por último si la operación es una multiplicación "10" o "11" entonces el bit de signo es el 15.

Como siguiente elemento tenemos la des complementación del número (en caso de que este sea negativo), si el bit de signo es 1, entonces se resta el número – 1 y luego se niega, de tal manera que se regresa al número positivo.

En las siguientes líneas (141 a 143) se le proporciona al registro de parte entera los 8 bits más significativos y al registro de parte decimal los 8 bits menos significativos, como se mencionó anteriormente dicha partición hace que la ALU tenga que ajustar los bits de tal forma que si la operación no requerirá más allá de 9 bits (8 y 1 de acarreo) entonces debe de colocar el número antes corrido hacia la izquierda 4 posiciones.

Por último de la 145 en adelante se hace la conexión entre la salida de los displays y el registro r_Show y la asignación a la salida del bit de signo con su registro, dicha salida será conectada a un led en la placa para mostrar el signo, la salida se niega debido a que el ánodo del led en la placa está conectado a VCC (esto fue mostrado en la práctica anterior)

```
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
```

```

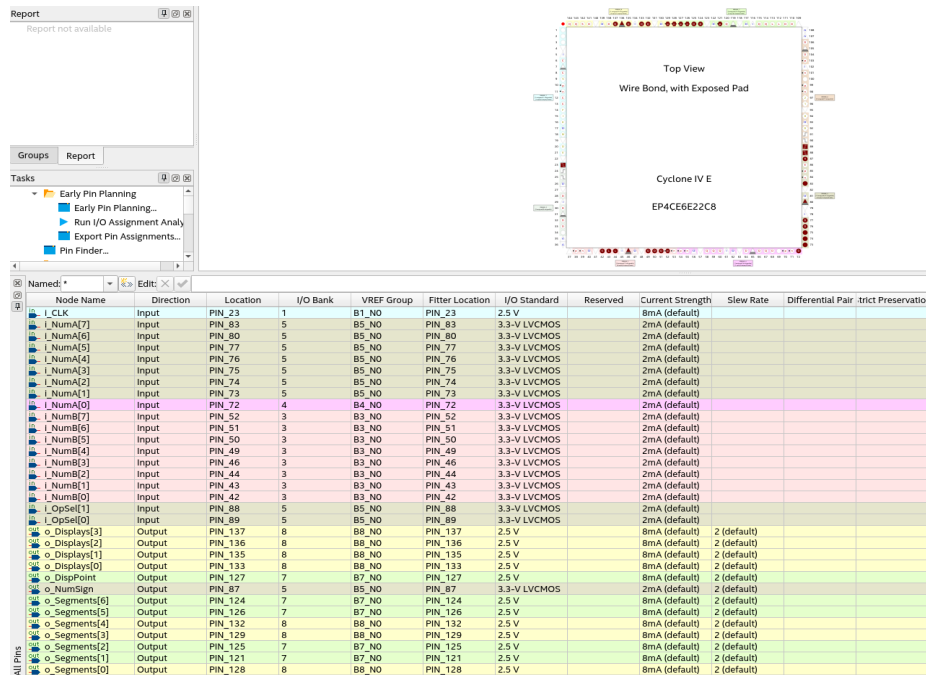
32 with i_Number select
33   o_DisplaySegments <= "1000000" when "0000",
34   "1111001" when "0001",
35   "0100100" when "0010",
36   "0110000" when "0011",
37   "0011001" when "0100",
38   "0010010" when "0101",
39   "0000010" when "0110",
40   "1111000" when "0111",
41   "0000000" when "1000",
42   "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 2.9 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_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_88	5	B5_NO	PIN_88	3.3-V LVCMOS		2mA (default)			
i_OpSel[0]	Input	PIN_89	5	B5_NO	PIN_89	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 2.10: Asignación de pines en la FPGA

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.

22

REPORTE DE TIEMPOS

	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	o_Dispoint	i_CLK	9.899	9.985	Rise	i_CLK
2	▼ o_Displays[*]	i_CLK	11.497	11.420	Rise	i_CLK
1	o_Displays[0]	i_CLK	10.337	10.139	Rise	i_CLK
2	o_Displays[1]	i_CLK	10.111	9.862	Rise	i_CLK
3	o_Displays[2]	i_CLK	11.497	11.420	Rise	i_CLK
4	o_Displays[3]	i_CLK	9.429	9.344	Rise	i_CLK
3	▼ o_Segments[*]	i_CLK	12.231	11.965	Rise	i_CLK
1	o_Segments[0]	i_CLK	9.910	9.781	Rise	i_CLK
2	o_Segments[1]	i_CLK	11.077	10.888	Rise	i_CLK
3	o_Segments[2]	i_CLK	10.622	10.618	Rise	i_CLK
4	o_Segments[3]	i_CLK	10.954	10.712	Rise	i_CLK
5	o_Segments[4]	i_CLK	10.833	10.712	Rise	i_CLK
6	o_Segments[5]	i_CLK	12.231	11.965	Rise	i_CLK
7	o_Segments[6]	i_CLK	10.978	10.799	Rise	i_CLK

Figura 2.12: Reporte de propagación del retraso

Como se puede observar en la figura 2.12 el tiempo más grande de reacción al ciclo de reloj es de 12.231ns por lo que este sería el mínimo periodo que esta aplicación puede trabajar.

REPORTE DE ÁREA


Flow Summary	
 <<Filter>>	
Flow Status	Successful - Sun Oct 8 18:21:23 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	145 / 6,272 (2 %)
Total registers	19
Total pins	32 / 92 (35 %)
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 2.13: Reporte de área

En este reporte se puede apreciar que el número de elementos lógicos que el diseño necesita son 145, lo cuál es muy poco, como lo indica la figura, 2% de la capacidad total de la FPGA y el 35% de sus pines.

Por último cabe destacar que se está haciendo uso de un multiplicador embebido de 9 bits, el cuál según la figura 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 std_logic_vector(7 downto 0);
18             i_NumB      : in std_logic_vector(7 downto 0);
19             i_OpSel     : in std_logic_vector(1 downto 0);
20             o_Displays  : out std_logic_vector(3 downto 0);
21             o_Segments  : out std_logic_vector(6 downto 0);
22             o_DisPoint  : out std_logic;
23             o_NumSign   : out std_logic
24         );
25     end component ALU8BitsDisplaysFP;
26
27     signal i_CLK      : std_logic := '0';
28     signal i_NumA      : std_logic_vector(7 downto 0) := (others => '0');
29     signal i_NumB      : std_logic_vector(7 downto 0) := (others => '0');
30     signal i_OpSel     : std_logic_vector(1 downto 0) := (others => '0');
31     signal o_Displays  : std_logic_vector(3 downto 0) := (others => '0');
32     signal o_Segments  : std_logic_vector(6 downto 0) := (others => '0');
33     signal o_DisPoint  : std_logic := '0';
34     signal o_NumSign   : std_logic := '0';
35
36 begin
37
38     U1 : ALU8BitsDisplaysFP
39     generic map(
40         (1)
41     )
42     port map(
43         i_CLK      => i_CLK,
44         i_NumA      => i_NumA,
45         i_NumB      => i_NumB,
46         i_OpSel     => i_OpSel,
47         o_Displays  => o_Displays,
48         o_Segments  => o_Segments,
49         o_DisPoint  => o_DisPoint,
50         o_NumSign   => o_NumSign
```



```

51 );
52
53 clk_proc : process
54 begin
55     i_CLK <= '1';
56     wait for 1 ns;
57     i_CLK <= '0';
58     wait for 1 ns;
59 end process;
60
61 stim_proc : process
62 begin
63     for i in 0 to 255 loop
64         for j in 0 to 255 loop
65             for k in 0 to 4 loop
66                 i_NumA <= std_logic_vector(to_unsigned(i, i_NumA'length));
67                 i_NumB <= std_logic_vector(to_unsigned(j, i_NumB'length));
68                 i_OpSel <= std_logic_vector(to_unsigned(k, i_OpSel'length));
69                 wait for 1 ns;
70             end loop;
71         end loop;
72     end loop;
73     wait;
74 end process;
75 end architecture rtl;

```

Figura 2.14 Código del Testbench

De las líneas 9 a la 10 se puede encontrar la declaración del componente que se va a simular. De la 27 a la 34, 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 32 a 51 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 cambio de salida.

Seguido de ello está el proceso de reloj, y después el proceso de estímulos, en dónde a partir de ciclos for se le da a todas las combinaciones posibles al circuito, para que de este modo se pueda comprobar si los resultados arrojados son correctos.

CAPTURAS DE PANTALLA DE LA SIMULACIÓN

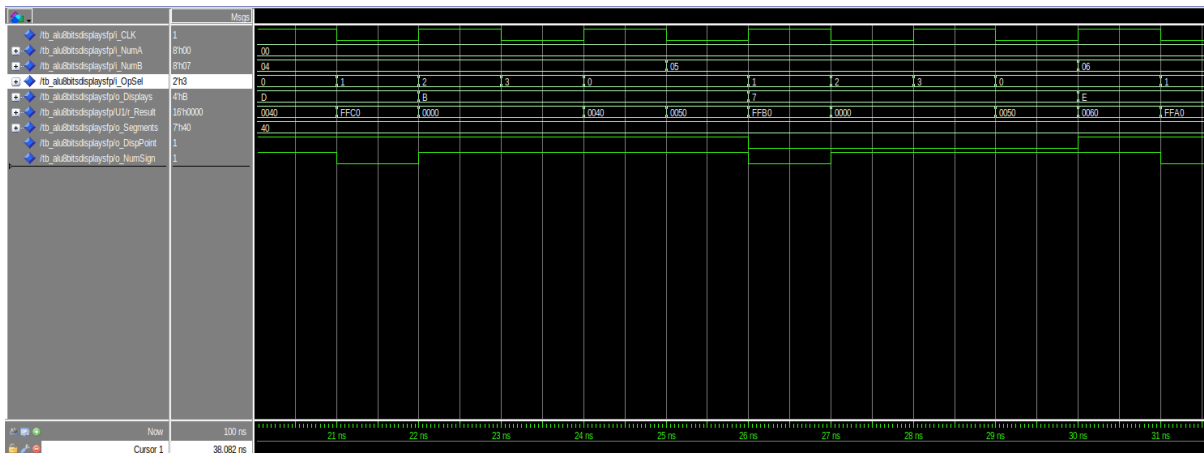


Figura 2.15 Captura 1 de simulación

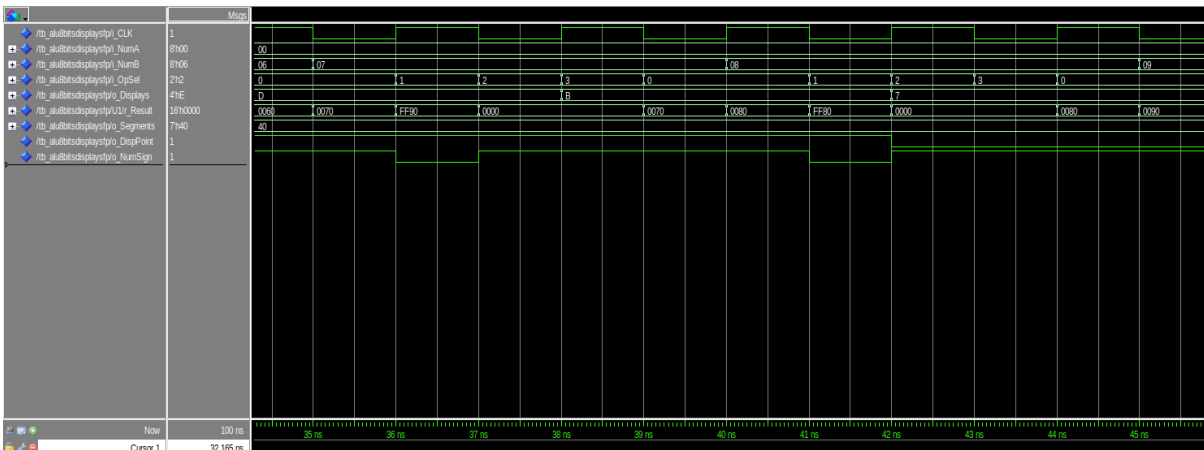


Figura 2.16 Captura 2 de simulación

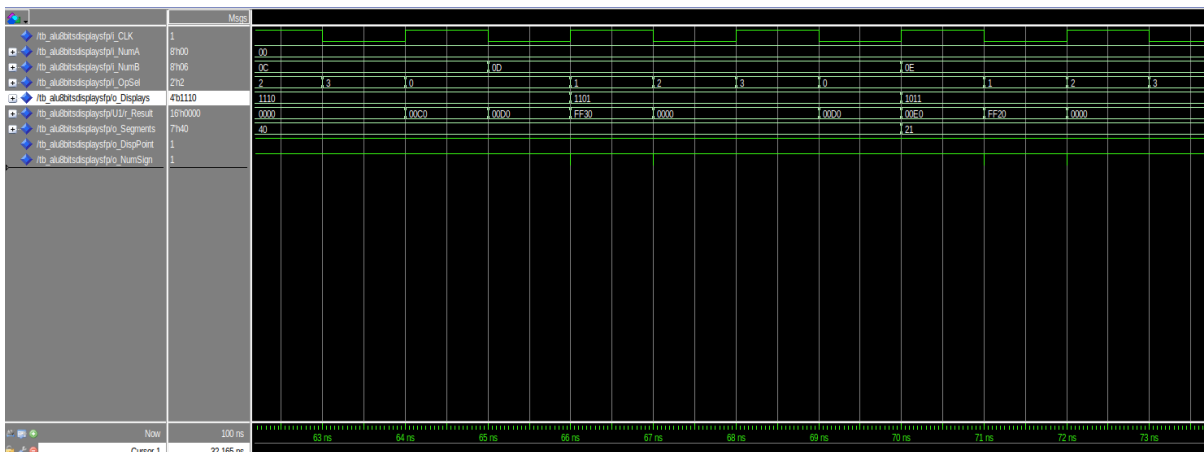


Figura 2.17 Captura 3 de simulación

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 [].

Solución: Esto sucedió debido a que procesos de naturaleza combinacional (negar con compuertas not y realizar 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 [] (10041 Inferred latch for “signal” at “file.vhd”), lo que se necesitó hacer fue mover este proceso de negaciones y restas (que fue el de descomplementar) 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

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, y este era el problema que se suscitaba en la figura 2.7, pues en esta se logra ver que cuándo la operación es una multiplicación el punto queda entre en el bit B8 y B7, debido a que la multiplicación arroja como máximo un resultado de 16 bits, sin embargo, cuándo se realizaba una suma y resta, por defecto el sistema coloca estos resultados en el B0 a Bn, ocasionando que el punto no siempre este entre la posición B8 y B7, pues en este caso se encontraría entre la posición B4 y B3, problema que si bien puede ser solucionado en el Display Driver, esto ocasionaría que no fuera un código tan portable del todo.

Solución: Colocar un pequeño ajuste en el diseño de la ALU para poder cambiar la posición del punto según la operación, por lo que estrictamente hablando el manejo del punto fijo no se encuentra sólo en el Display Driver, si no también en la ALU, esto se realizó así debido a que es más sencillo describir una ALU (en caso de que se necesite con otras necesidades o especificaciones) que describir un Display Driver, además de que este último tiene más utilidad (en cuánto a prácticas se refiere), ya que es un método para desplegar información y así depurar futuros componentes.

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 y en los resultados que la ALU arrojaba lo cuál era muy poco práctico.

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 2.11) 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.

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: Construir 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

Andrea Joanelle Hernández Alvarado

BIBLIOGRAFÍA