Universidad de Buenos Aires
Facultad De Ingeniería
Año 2023 – 2.^{do} Cuatrimestre

(86.41) Sistemas digitales

Trabajo práctico N.º 2

Docentes a cargo:
Alvarez, Nicolás
Alpago, Octavio

Integrante	Padrón	Correo electrónico
Lützelschwab, Nahila	100686	nlützelschwab@fi.uba.ar

1. Introducción

El presente trabajo práctico tiene como objetivo simular, sintetizar e implementar en FPGA algunas funciones de una unidad aritmética de punto flotante aplicando el lenguaje de descripción de hardware VHDL.

2. Desarrollo

Se implementaron las funciones de multiplicación y de suma/resta teniendo en cuenta las siguientes especificaciones:

- El método de redondeo utilizado es truncamiento.
- El tamaño de los campos significando y exponente serán genéricos (NF y NE respectivamente).
- No se consideraron números denormales como así tampoco casos especiales (NaN, $\pm\ \infty$).
- Al no considerar denormales, el caso de todos los campos '0' mientras que el bit de signo sea '0' o '1' se consideraron como el número cero.
- Al no considerar NaN y $\pm\infty$, si el resultado excede el rango de operación entonces se aplicó saturación, es decir se devuelve a la salida el máximo (o mínimo) representable.

2.1. Multiplicador

Se implementó el siguiente circuito:
La implementación en VHDL es la siguiente:

```
-- Nahila Lutzelschwab - TP2 - padron: 100686 - multiplication

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity mul_fp is
    generic(
            N : natural := 30;
            NE : natural := 8
    );
    port (
        rst : in std_logic;
        clk : in std_logic;
        X   : in std_logic_vector(N-1 downto 0);
        Y   : in std_logic_vector(N-1 downto 0);
        Z   : out std_logic_vector(N-1 downto 0)
    );
end mul_fp;

architecture behavioral of mul_fp is

    -- Declaration of constants to use
    constant NF         : natural := N-NE-1;
    constant EXC        : natural := 2**(NE-1)-1;
    constant EXCESS     : unsigned(NE+1 downto 0) := to_unsigned(EXC, NE+2);

    -- Minimum and maximum exponent
    constant E_MIN      : unsigned(NE-1 downto 0) := to_unsigned(0, NE);
    constant E_MAX      : unsigned(NE-1 downto 0) := to_unsigned(2**(NE)-2, NE);
    -- Constants for NULL operand
        constant E_NULL     : unsigned(NE+1 downto 0) := to_unsigned(0, NE+2);
    constant F_NULL     : unsigned(NF-1 downto 0) := to_unsigned(0, NF);
    constant Z_NULL                 : unsigned(N-2 downto 0)  := to_unsigned(0,N-1);

```

```
    -- Declaration of signals to use

    -- Flag for NULL operand
    signal NULL_flag  : std_logic := '0';

    -- Signs
    signal Sx   : std_logic;
    signal Sy   : std_logic;
    signal Sz   : std_logic;

    -- Exponent
    signal Ex       : unsigned(NE-1 downto 0) := (others => '0');
    signal Ey       : unsigned(NE-1 downto 0) := (others => '0');
    -- Exponent extensions, to make the multiplication NE+1 bits are needed -> Ex, Ey NE+2 bits
    signal Ex_ext   : unsigned(NE+1 downto 0) := (others => '0');
    signal Ey_ext   : unsigned(NE+1 downto 0) := (others => '0');
    signal Ez       : unsigned(NE-1 downto 0) := (others => '0');
    signal Ez_p     : unsigned(NE+1 downto 0) := (others => '0');
    signal Ez_pp    : unsigned(NE+1 downto 0) := (others => '0');
    signal Ez_ppp   : unsigned(NE+1 downto 0) := (others => '0');


    -- Mantissas
    signal Fx       : unsigned(NF-1 downto 0) := (others => '0');
    signal Fy       : unsigned(NF-1 downto 0) := (others => '0');
    signal Fz       : unsigned(NF-1 downto 0) := (others => '0');
    signal Fz_p     : unsigned(NF-1 downto 0) := (others => '0');
    signal Fz_pp    : unsigned(NF-1 downto 0) := (others => '0');

    signal Mx        : unsigned(NF downto 0) := (others => '0');
    signal My        : unsigned(NF downto 0) := (others => '0');
    signal Mz        : unsigned(2*NF+1 downto 0) := (others => '0');


--------------------------------------------------------------------------------

begin
    -- Assigning signs, exponentials and mantissa values
    Sx <= X(NE+NF);
    Sy <= Y(NE+NF);
    Ex <= unsigned(X(NF+NE-1 downto NF));
    Ey <= unsigned(Y(NF+NE-1 downto NF));
    Fx <= unsigned(X(NF-1 downto 0));
    Fy <= unsigned(Y(NF-1 downto 0));

    --The sign of the result will be the xor of the signs of the operands
    Sz <= Sx xor Sy;

    -- In case one of the operands is NULL
    -- the result will be the other operand
    NULL_flag <= '1' when ( (Ex = E_NULL) and (Fx = F_NULL) ) else
                 '1' when ( (Ey = E_NULL) and (Fy = F_NULL) ) else
                 '0';
    -- Adding two bits to Ex and Ey to make multiplication
    -- and assigning them the value of the exponent
    Ex_ext <= '0' &'0' & Ex;
    Ey_ext <= '0' &'0' & Ey;

    Ez_p <= Ex + Ey - EXCESS;

          -- Denormal numbers will not be considered
    -- Adding implicit '1' to the left of Fx and Fy
    -- ending up with Mx, My of NF+1 bits
    Mx <= '1' & Fx;
    My <= '1' & Fy;

         -- The unsigned multiplication operator is already overloaded
    -- so it's not necessary to add another zero.
```

```
105
106     -- Mz is the multiplication of the mantissa
107     -- of the operands, ending up being of 2*NF+2 bits.
108     Mz <= Mx * My;
109
110
111     -- The mantissa determines whether it's needed to be added
112     -- a '1' or not depending on the MSB of Mz
113     Ez_pp <= (Ez_p + 1) when Mz(2*NF+1) = '1' else Ez_p;
114
115     -- Saturation of the exponent
116     Ez_ppp <= E_MAX(NE-1 downto 0) when Ez_pp(NE) = '1' else
117               E_MIN(NE-1 downto 0) when Ez_pp(NE) = '1' else
118               Ez_pp(NE-1 downto 0);
119
120     -- Matissa rounding
121     Fz_p <= Mz(2*NF downto NF+1) when Mz(2*NF+1) = '1' else
122             Mz(2*NF-1 downto NF);
123
124     -- Mantissa saturation
125     Fz_p <= (others => '1') when Ez_pp(NE) = '1' else
126             (others => '0') when Ez_pp(NE) = '1' else
127             Fz_p;
128
129     -- If one of the operands is NULL the result will be NULL
130     Ez <= E_NULL when (NULL_flag = '1') else Ez_ppp;
131     Fz <= E_NULL when (NULL_flag = '1') else Fz_pp;
132
133     Z <= std_logic_vector(Sz & Z_NULL) when ( NULL_flag = '1' ) else
134          std_logic_vector(Sz & Ez & Fz);
135
136 end architecture behavioral;
137
```

2.1.1. Simulación - Test bench

Se simularon todas las unidades aritméticas de forma automatizada con los archivos de prueba provistos por la cátedra.

```
1
2  -- Nahila Lutzelschwab - TP2 - padron: 100686 - multiplier Testbench
3  library IEEE;
4  use IEEE.std_logic_1164.all;
5  use IEEE.numeric_std.all;
6  use std.textio.all;
7
8  entity tb_mul_fp is
9  end entity tb_mul_fp;
10
11 architecture tb_arch of tb_mul_fp is
12
13     constant FILE_PATH  : string := "test_mul_float_30_8.txt";
14     constant TCK        : time := 20 ns; -- Periodo de reloj
15     constant WORD_SIZE  : natural := 30; -- Tamaño de datos
16     constant EXP_SIZE   : natural := 8;  -- Tamaño del exponente
17     constant F_SIZE     : natural := WORD_SIZE- EXP_SIZE - 1; -- TamaÃ±o de mantisa
18
19     signal tb_rst   : std_logic;
20     signal tb_clk   : std_logic := '0';
21     signal x_file   : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');
22     signal y_file   : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');
23     signal z_file   : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');
24     signal z_duv    : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');
25
26     signal ciclos   : integer := 0;
27     signal errores  : integer := 0;
28
```

```
    file datos : text open read_mode is FILE_PATH;

begin
    tb_rst <= '0', '1' after 1 ns, '0' after 20 ns;
    tb_clk <= not(tb_clk) after TCK/2; -- Reloj

    Test_Sequence: process
        variable l   : line;
        variable ch  : character := ' ';
        variable aux : integer;
    begin
        while not(endfile(datos)) loop
            wait until rising_edge(tb_clk);
            -- Solo para debugging
            ciclos <= ciclos + 1;
            -- Se lee una linea del archivo de valores de prueba
            readline(datos, l);
            -- Se extrae un entero de la linea
            read(l, aux);
            -- Se carga el valor del operando X
            x_file <= std_logic_vector(to_unsigned(aux, WORD_SIZE));
            -- Se lee un caracter (el espacio)
            read(l, ch);
            -- Se lee otro entero de la linea
            read(l, aux);
            -- Se carga el valor del operando Y
            y_file <= std_logic_vector(to_unsigned(aux, WORD_SIZE));
            -- Se lee otro caracter (el espacio)
            read(l, ch);
            -- Se lee otro entero
            read(l, aux);
            -- Se carga el valor de la salida (resultado)
            z_file <= std_logic_vector(to_unsigned(aux, WORD_SIZE));
        end loop;

        file_close(datos); -- Se cierra el archivo


        -- Se aborta la simulacion (fin del archivo)
        assert false report
            "Fin de la simulacion" severity failure;

    end process Test_Sequence;

    -- Instanciacion del DUV
    DUV: entity work.mul_fp(behavioral)
    generic map(
        N => WORD_SIZE,
        NE => EXP_SIZE
    )
    port map(
        rst  => tb_rst,
        clk  => tb_clk,
        X    => x_file,
        Y    => y_file,
        Z    => z_duv
    );


    verificacion: process(tb_clk)
    begin
        if rising_edge(tb_clk) then
            assert to_integer(unsigned(z_file)) = to_integer(unsigned(z_duv)) report
                "Error: Salida del DUV no coincide con referencia (salida del DUV = " &
                integer'image(to_integer(unsigned(z_duv))) &
                ", salida del archivo = " &
                integer'image(to_integer(unsigned(z_file))) & ")"
                severity warning;

```

```
98              if to_integer(unsigned(z_file)) /= to_integer(unsigned(z_duv)) then
99                  errores <= errores + 1;
100             end if;
101         end if;
102     end process;
103
104 end architecture tb_arch;
105
```

Los resultados finales de todos los archivos simulados arrojan errores debido al diseño implementado, siendo que se tomaron algunas simplificaciones como no se consideran los casos especiales (NaN, $\pm\infty$) ni denormales.

Se puede observar en la figura 1 la simulación de la multiplicación a partir del archivo "test_mul_float_30_8.txt".

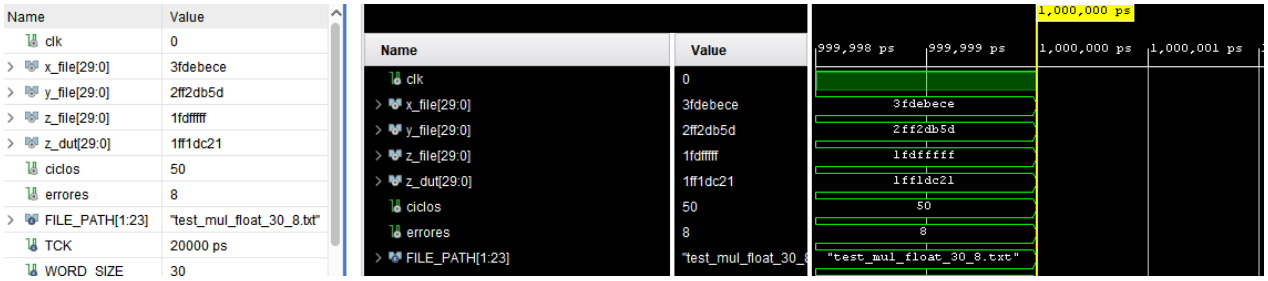

Figura 1

2.1.2. Implementación

Se realizó la implementación sobre el dispositivo FPGA xc7a15tftg256-1 con el software Vivado, se puede observar en la figura 2 que es muy grandes.

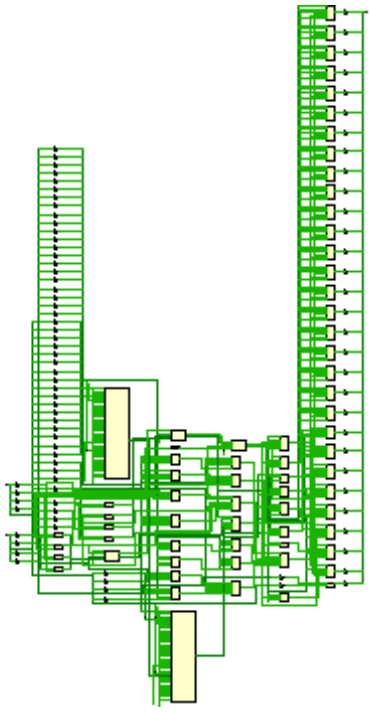

Figura 2

2.2. Suma/resta

Se implementó el siguiente circuito:

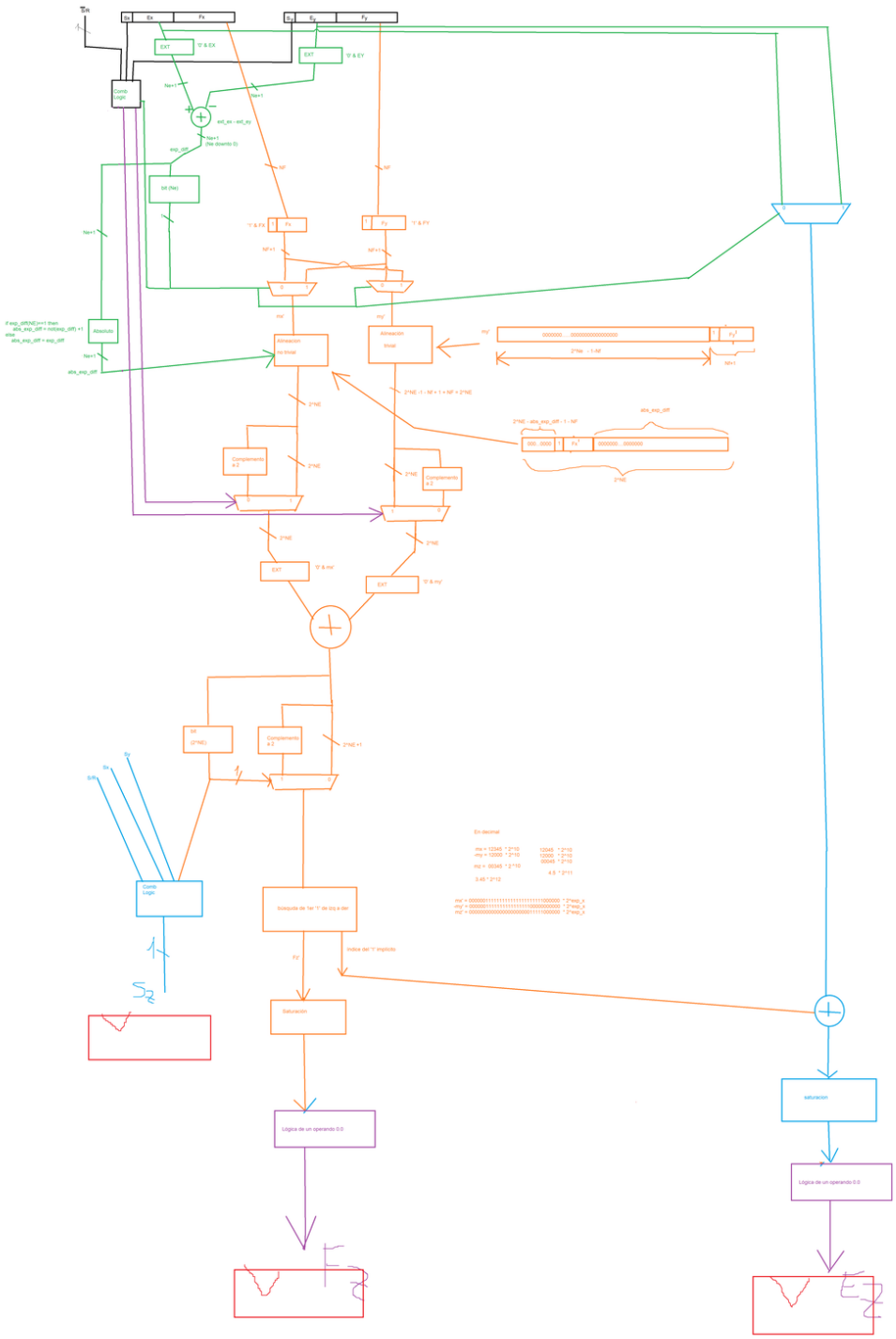

Figura 3

La implementación de la suma/resta en VHDL es la siguiente:

```
1  -- Nahila Lutzelschwab - TP2 - padron: 100686 - addition/subtracion
2
3  -- Declaration of common library and use
4  library IEEE;
5  use IEEE.std_logic_1164.all;
```

```vhdl
use IEEE.numeric_std.all;


entity fp_add_subtract is
    generic(N : natural := 80;
            Ne : natural := 8 -- exponent bits
    );
    port (
            -- Z = X +/- Y
         X  : in std_logic_vector(N-1 downto 0);
         Y  : in std_logic_vector(N-1 downto 0);
         Z  : out std_logic_vector(N-1 downto 0);

        -- indicator whether it is '0' for addition or '1' for subtraction
        operation_type : in std_logic
    );
end fp_add_subtract;



architecture behavioral of fp_add_subtract is

        -- Declaration of constants to use

    constant NF     : natural := N-NE-1; -- mantissa bits
    constant EXC    : natural := 2**(NE-1)-1; -- excess
    constant EXCESS : signed(NE-1 downto 0) := to_signed(EXC, NE);

    -- minimum and maximum exponent
    constant E_MIN  : signed(NE downto 0) := to_signed(0, NE+1);
    constant E_MAX  : signed(NE downto 0) := to_signed(2**(NE)-2, NE+1);

    constant index_first_one    : natural := 0;

    -- Declaration of signals to use
    signal Ex   : unsigned(NE-1 downto 0) := (others => '0');
    signal Ey   : unsigned(NE-1 downto 0) := (others => '0');
    -- Exponential for Z and Z'
    signal Ez           : signed(NE-1 downto 0) := (others => '0');
    signal Ez_p  : signed(NE downto 0) := (others => '0');

    -- subtraction Ex Ey to see if they are equal or not -> NE+1 bits -> Ex, Ey NE+1 bits
    signal Ex_ext       : unsigned(NE downto 0) := (others => '0');
    signal Ey_ext       : unsigned(NE downto 0) := (others => '0');
    signal E_diff       : signed(NE downto 0) := (others => '0');
    signal E_diff_abs   : unsigned(NE downto 0) := (others => '0');

    signal Y_aux    : std_logic_vector(N-1 downto 0) := (others => '0');

    -- X and Y prime
    signal X_p  : unsigned(N-1 downto 0) := (others => '0');
    signal Y_p  : unsigned(N-1 downto 0) := (others => '0');

    signal aux : std_logic_vector(1 downto 0);

    -- Signs of X , Y and E_diff
    signal Sx       : std_logic;
    signal Sy       : std_logic;
    signal Sz       : std_logic;
    signal S_Ediff  : std_logic;
    signal S_add    : std_logic;

    -- Mantissas
    signal Fx     : unsigned(NF-1 downto 0) := (others => '0');
    signal Fy     : unsigned(NF-1 downto 0) := (others => '0');
    signal Fz     : unsigned(2**(NE)-1 downto 0) := (others => '0');
    signal Fx_p   : unsigned(NF downto 0) := (others => '0');
    signal Fy_p   : unsigned(NF downto 0) := (others => '0');
    signal Fz_p   : unsigned(2**(NE)-1 downto 0) := (others => '0');
```

```
75
76      -- Mantissas for X' and Y'
77      signal Mx_p     : unsigned(NF downto 0) := (others => '0');
78      signal My_p     : unsigned(NF downto 0) := (others => '0');
79
80      -- Auxiliar mantissas
81      signal Mx_pp        : unsigned(2**(NE)-1 downto 0) := (others => '0');
82      signal My_pp        : unsigned(2**(NE)-1 downto 0) := (others => '0');
83      signal Mx_ppp       : unsigned(2**(NE) downto 0) := (others => '0');
84      signal My_ppp       : unsigned(2**(NE) downto 0) := (others => '0');
85      signal addition     : signed(2**(NE) downto 0) := (others => '0');
86      signal addition_p   : signed(2**(NE) downto 0) := (others => '0');
87
88
89       -- Find first '1' from left to right
90      function find_first_one (x_signal: std_logic_vector) return natural is
91          variable index       : natural;
92          variable one_reached : boolean := False;
93
94      begin
95          for i in x_signal'length-1 downto 0 loop
96              if x_signal(i) = '1' and one_reached = False then
97                  one_reached := True;
98                  index := i;
99              end if;
100         end loop;
101
102         if index < 0 then
103             index := 0;
104         end if;
105
106         return index;
107     end function;
108
109
110 begin
111     -- In case it's subtraction (operation_type = 1) adjust Y changing sign
112     --Y_aux <= not(Y(N-1)) & Y(N-2 downto 0) when operation_type = '1' else Y;
113
114
115     -- Evaluating whether Ex is equal or different from Ey
116
117     -- Adding a bit to Ex and Ey to make subtraction
118     -- and assigning them the value of the exponent
119     Ex_ext <= '0' & unsigned(X(NF+NE-1 downto NF));
120     Ey_ext <= '0' & unsigned(Y_aux(NF+NE-1 downto NF));
121     E_diff <=  signed(Ex_ext - Ey_ext);
122
123     E_diff_abs <= unsigned(not(E_diff)+1) when (E_diff(NE) = '1') else unsigned(E_diff);
124     S_Ediff <= E_diff(NE);
125
126     -- Assigning signs and exponential values
127     Sx <= X(NE+NF);
128     Sy <= Y(NE+NF);
129     Ex <= unsigned(X(NF+NE-1 downto NF));
130     Ey <= unsigned(Y(NF+NE-1 downto NF));
131     Fx <= unsigned(X(NF-1 downto 0));
132     Fy <= unsigned(Y(NF-1 downto 0));
133
134     -- Denormal numbers will not be considered
135     -- Adding '1' to the left of Fx and Fy
136     -- ending up with Mx, My of NF+1 bits
137     Fx_p <= '1' & Fx;
138     Fy_p <= '1' & Fy;
139
140     -- Swapping operands in order to align only one of them
141     -- (only the one with highest exponent)
142     Mx_p <= Fy_p when (E_diff(NE) = '1') else Fx_p;
143     My_p <= Fx_p when (E_diff(NE) = '1') else Fy_p;
```

```
144
145     -- Aligning operands
146     -- |Ex|>|Ey|
147
148     -- Shift (2**(NE)-NF-1) zeros to the right
149     -- leaving E_diff_abs bits to the left
150    Mx_pp <= (Mx_p sll natural(2**(NE)-NF-1));
151
152     -- My' has trivial alignment (leading zeros)
153     My_pp <= (My_p sll to_integer(E_diff_abs));
154
155
156     -- Complement
157     process (operation_type, Sx, Sy, Mx_pp, My_pp)
158       begin
159         aux <= Sx & Sy;
160         case operation_type is
161           -- Addition
162           when '0' =>
163             case aux is
164               when "00" =>  -- Mx_pp>0, My_pp>0
165                 Mx_pp <= Mx_pp;
166                 My_pp <= My_pp;
167
168               when "01" =>  -- Mx_pp>0, My_pp<0
169                 My_pp <= (not(My_pp)+1);
170
171               when "10" =>  -- Mx_pp<0, My_p>0
172                 Mx_pp <= (not(Mx_pp)+1);
173
174               when others =>  -- Mx_pp<0, My_p<0
175                 Mx_pp <= (not(Mx_pp)+1);
176                 My_pp <= (not(My_pp)+1);
177             end case;
178
179           -- Subtraction
180           when '1' =>
181             case aux is
182               when "00" =>  -- Mx_pp>0, My_pp>0 -> Mx_pp + (-My_pp) -> My_pp<0
183                 My_pp <= (not(My_pp)+1);
184
185               when "01" =>  -- Mx_pp>0, My_pp<0 -> Mx_pp + -(-My_pp) -> My_pp>0
186                 Mx_pp <= Mx_pp;
187                 My_pp <= My_pp;
188
189               when "10" =>  -- Mx_pp<0, My_pp>0 -> Mx_pp + (-My_pp) -> My_pp<0
190                 Mx_pp <= (not(Mx_pp)+1);
191                 My_pp <= (not(My_pp)+1);
192
193               when others =>  -- Mx_pp<0, My_pp<0 -> Mx_pp + -(-My_pp) -> My_pp>0
194                 Mx_pp <= (not(Mx_pp)+1);
195             end case;
196
197         end case;
198      end process;
199
200
201     Mx_ppp <= '0' & My_pp;
202     Mx_ppp <= '0' & My_pp;
203
204     -- Efective addition/subtraction
205     addition_p <= signed(Mx_ppp + My_ppp);
206
207     S_add <= addition_p(2**NE);
208
209     -- Final addition/subtraction
210     addition <= addition_p when S_add = '0' else (not(addition_p)+1);
211
212     -- Find first one from left to right
```

```
    --index_first_one := find_first_one(x_signal <= std_logic_vector(addition));

    -- Z sign
    Sz <= S_add;

    -- Z' exponent
    Ez_p <= to_signed(find_first_one(x_signal => std_logic_vector(addition)) - NF, NE+1);
    -- Saturation of the exponent
    -- If the calculated exponent is higher than the maximum
    -- Ez will saturate at E_MAX, same consideration for the minimum

    Ez <= to_signed(to_integer(E_MAX), NE) when (to_integer(Ez_p) > to_integer(E_MAX)) else
          to_signed(to_integer(E_MIN), NE) when (to_integer(Ez_p) < to_integer(E_MIN)) else
          Ez_p(NE-1 downto 0);


    -- Find first one from left to right
    -- discard the first one for being implicit
    -- F_z would be NF bits to the right
    Fz_p <= unsigned(addition((index_first_one - 1) downto 0));

    --  Saturation of the mantissa
    -- If the calculated mantissa is higher than the maximum
    -- it will saturate at E_MAX, same consideration for the minimum
    Fz <= (others => '1') when ( to_integer(Ez_p) > to_integer(E_MAX) ) else
          (others => '0') when ( to_integer(Ez_p) < to_integer(E_MIN) ) else
          Fz_p;

    -- Z
    Z <= Sz & std_logic_vector(Ez) & std_logic_vector(Fz);

end architecture behavioral;

```

2.2.1. Simulación - Test bench

Para simular el circuito se reemplazaron los valores por otros más chicos para que los resultados sean apreciables y se realizó mediante el siguiente test bench:

```
-- Nahila Lutzelschwab - TP2 - padron: 100686 - Testbench addition/subtracion
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity tb_fp_add_subtract is
end entity tb_fp_add_subtract;

architecture tb_arch of tb_fp_add_subtract is

    constant FILE_PATH  : string := "test_sum_float_30_8.txt";
    constant TCK        : time := 20 ns; -- Periodo de reloj
    constant WORD_SIZE  : natural := 30; -- Tamaño de datos
    constant EXP_SIZE   : natural := 8;  -- Tamaño del exponente
    constant F_SIZE     : natural := WORD_SIZE- EXP_SIZE - 1; -- Tamaño de mantisa


    signal tb_rst             : std_logic;
    signal tb_clk                   : std_logic := '0';
    signal operation_type_tb  : std_logic := '0';

    signal x_file       : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');
    signal y_file       : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');
    signal z_file       : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');
    signal z_dut        : std_logic_vector(WORD_SIZE-1 downto 0) := (others => '0');

    signal ciclos   : integer := 0;
```

```
    signal errores  : integer := 0;

    file datos : text open read_mode is FILE_PATH;

begin
    tb_rst <= '0', '1' after 1 ns, '0' after 20 ns;
    tb_clk <= not(tb_clk) after TCK/2; -- Reloj

    Test_Sequence: process
        variable l   : line;
        variable ch  : character := ' ';
        variable aux : integer;
    begin
        while not(endfile(datos)) loop
            wait until rising_edge(tb_clk);
            -- Solo para debugging
            ciclos <= ciclos + 1;
            -- Se lee una linea del archivo de valores de prueba
            readline(datos, l);
            -- Se extrae un entero de la linea
            read(l, aux);
            -- Se carga el valor del operando X
            x_file <= std_logic_vector(to_unsigned(aux, WORD_SIZE));
            -- Se lee un caracter (el espacio)
            read(l, ch);
            -- Se lee otro entero de la linea
            read(l, aux);
            -- Se carga el valor del operando Y
            y_file <= std_logic_vector(to_unsigned(aux, WORD_SIZE));
            -- Se lee otro caracter (el espacio)
            read(l, ch);
            -- Se lee otro entero
            read(l, aux);
            -- Se carga el valor de la salida (resultado)
            z_file <= std_logic_vector(to_unsigned(aux, WORD_SIZE));
        end loop;

        file_close(datos); -- Se cierra el archivo


        -- Se aborta la simulacion (fin del archivo)
        assert false report
            "Fin de la simulacion" severity failure;

    end process Test_Sequence;

    -- Instanciacion del DUT
    DUT: entity work.fp_add_subtract(behavioral)
    generic map(
        N => WORD_SIZE,
        NE => EXP_SIZE
    )
    port map(
        rst             => tb_rst,
        clk             => tb_clk,
        X               => x_file,
        Y               => y_file,
        Z               => z_dut,
        operation_type  => operation_type_tb
    );


    verificacion: process(tb_clk)
    begin
        if rising_edge(tb_clk) then
            assert to_integer(unsigned(z_file)) = to_integer(unsigned(z_dut)) report
                "Error: Salida del DUT no coincide con referencia (salida del DUT = " &
                integer'image(to_integer(unsigned(z_dut))) &
                ", salida del archivo = " &
```

```
98                  integer'image(to_integer(unsigned(z_file))) & ")"
99                  severity warning;
100
101             if to_integer(unsigned(z_file)) /= to_integer(unsigned(z_dut)) then
102                 errores <= errores + 1;
103             end if;
104         end if;
105     end process;
106
107 end architecture tb_arch;
```

A diferencia del multiplicador, los resultados de las diferentes simulaciones son prácticamente nulas.
Se puede observar en las figuras 4, 5 la simulación de la suma y la resta a partir del archivo "test_mul_float_30_8.txt".

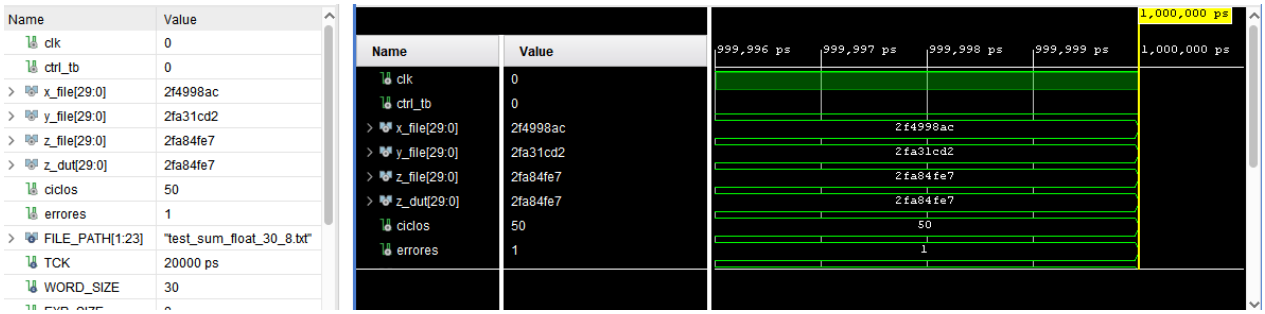

Figura 4: Suma

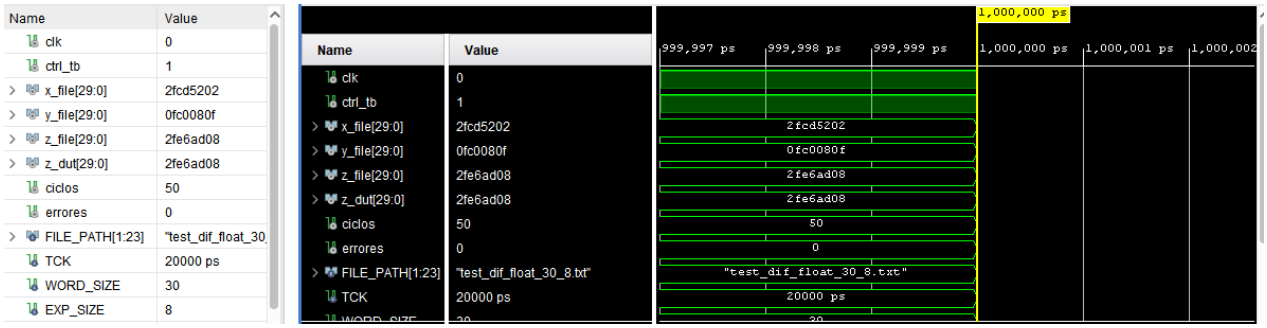

Figura 5: Resta

2.2.2. Implementación

Se realizó la implementación sobre el dispositivo FPGA xc7a15tftg256-1 con el software Vivado. Se puede observar en la figura 6 que dicha implementación es bastante grande.

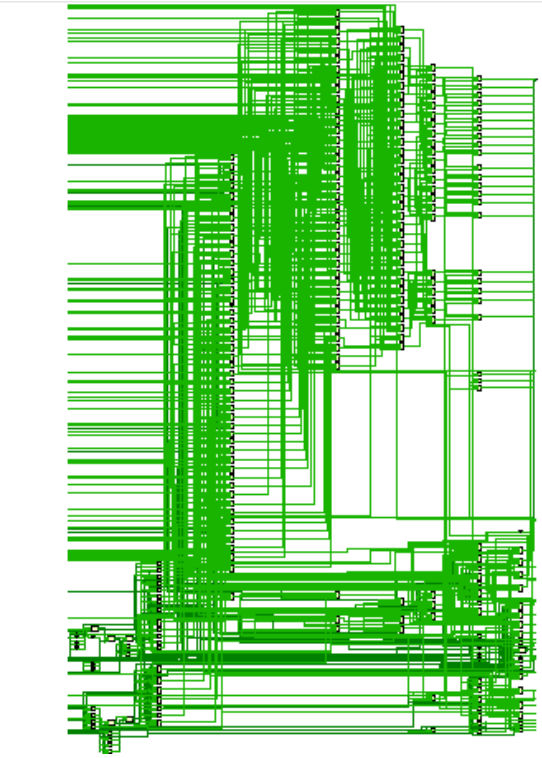

Figura 6

3. Conclusión

Se puede concluir que el presente trabajo práctico permitió entender el funcionamiento de una unidad de punto flotante de forma combinacional. A su vez, Vivado que permitió realizar su implementación sobre un dispositivo FPGA específico.