



MASTER THESIS

# DESIGN OF SINGLE PRECISION FLOAT ADDER (32-BIT NUMBERS) ACCORDING TO IEEE 754 STANDARD USING VHDL

Arturo Barrabés Castillo

Bratislava, April 25th 2012

Supervisors: Dr. Roman Zálusky
Prof. Viera Stopjaková
Fakulta Elecktrotechniky a Informatiky
Slovenská Technická Univerzita v Bratislave

# INDEX

Index .....3
Resum .....5
Zhrnutie .....5
Abstract .....5
**Chapter 1: Introduction ..... 7**
1.1. Floating Point Numbers .....7
1.2. The Standard IEEE 754 .....8
1.2.1. Overview .....8
1.2.2. Binary Interchange Format Encodings .....9
1.2.3. Precision and Rounding .....10
**Chapter 2: Code Development ..... 13**
2.1. 32-bits Floating Point Adder Design .....13
2.1.1. Addition/Subtraction Steps .....13
2.1.2. Block Diagram .....15
2.2. Blocks Design .....17
2.2.1. Pre-Adder Design .....17
2.2.2. Adder Design .....17
2.2.3. Standardizing Design .....19
**Chapter 3: Pre-Adder ..... 21**
3.1. Special Cases .....21
3.1.1. n_case Block .....21
3.2. Subnormal Numbers .....25
3.2.1. n_subn Block .....25
3.3. Mixed Numbers .....27
3.3.1. comp Block .....27
3.3.2. zero Block .....30
3.3.3. shift_left/shift Block .....32
3.3.4. norm Block .....35
3.4. Normal Numbers .....38
3.4.1. comp_exp Block .....38
3.4.2. shift Block .....41
3.4.3. n_normal Block .....41

3.5. Pre-Adder ..... 44
3.5.1. selector Block ..... 44
3.5.2. MUX/DEMUX Blocks ..... 48
3.5.3. preadder Block ..... 50
**Chapter 4: Adder ..... 55**
4.1. Adder ..... 55
4.1.1. Signout Block ..... 55
4.1.2. Adder Block ..... 59
4.1.3. Block_Adder Block ..... 62
4.2. Standardizing Block ..... 65
4.2.1. round Block ..... 65
4.2.2. shift_left/zero Block ..... 65
4.2.3. block_norm Block ..... 67
4.2.4. vector Block ..... 70
**Chapter 5: 32-Bits Floating Point Adder ..... 73**
5.1. Floating Point Adder ..... 74
5.1.1. Mux_fpadder Block ..... 74
5.1.2. fpadder Block ..... 74
5.2. Simulations ..... 77
5.2.1. Special Cases ..... 77
5.2.2. Normal Numbers ..... 80
5.2.3. Subnormal Numbers ..... 81
5.2.4. Mixed Numbers ..... 82
**Chapter 6: Results ..... 83**
6.1. Errors ..... 83
6.1.1. Gap between Numbers ..... 83
6.1.2. Rounding or Truncation ..... 85
6.1.3. Floating Point Addition ..... 86
6.2. Results analysis ..... 86
6.2.1. Subnormal Numbers ..... 86
6.2.2. Mixed Numbers ..... 88
6.2.3. Normal Numbers ..... 89
6.3. Conclusions ..... 91
**Chapter 7: Bibliography ..... 93**
**Annex: VHDL Code ..... 95**

# RESUM

La aritmètica de punt flotant és, amb diferència, el mètode més utilitzat d'aproximació a la aritmètica amb nombres reals per realitzar càlculs numèrics per ordinador.

Durant molt temps cada màquina presentava una aritmètica diferent: bases, mida dels significants i exponents, formats, etc. Cada fabricant implementava el seu propi model ,fet que dificultava la portabilitat entre diferents equips, fins que va aparèixer la norma IEEE 754 que definia un estàndard únic per a tothom.

L'objectiu d'aquest projecte és, a partir del estàndard IEEE 754, implementar un sumador/restador binari de punt flotant de 32 bits emprant el llenguatge de programació hardware VHDL.

# ZHRNUTIE

Práca s číslami s pohyblivou desatinnou čiarkou je najpoužívanejší spôsob pre vykonávanie aritmetických výpočtov s reálnymi číslami na moderných počítačoch. Donedávna, každý počítač využíval rôzne typy formátov: báza, znamienko, veľkosť exponentu, atď. Každá firma implementovala svoj vlastný formát a zabraňovala jeho prenosu na iné platformy pokiaľ sa nevymedzil jednotný štandard IEEE 754. Cieľom tejto práce je implementovanie 32-bitovej sčítačky/odčítačky pracujúcej s číslami s pohyblivou desatinnou čiarkou podľa štandardu IEEE 754 a to pomocou jazyka na opis hardvéru VHDL.

# ABSTRACT

Floating Point arithmetic is by far the most used way of approximating real number arithmetic for performing numerical calculations on modern computers.

Each computer had a different arithmetic for long time: bases, significant and exponents' sizes, formats, etc. Each company implemented its own model and it hindered the portability between different equipments until IEEE 754 standard appeared defining a single and universal standard.

The aim of this project is implementing a 32 bit binary floating point adder/subtractor according with the IEEE 754 standard and using the hardware programming language VHDL.

# CHAPTER 1: INTRODUCTION

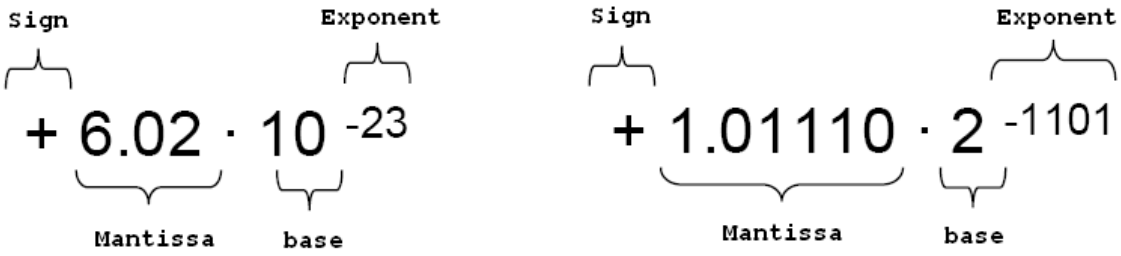
Many fields of science, engineering and finance require manipulating real numbers efficiently. Since the first computers appeared, many different ways of approximating real numbers on it have been introduced.

One of them, the floating point arithmetic, is clearly the most efficient way of representing real numbers in computers. Representing an infinite, continuous set (real numbers) with a finite set (machine numbers) is not an easy task: some compromises must be found between speed, accuracy, ease of use and implementation and memory cost.

Floating Point Arithmetic represent a very good compromise for most numerical applications.

## 1.1. Floating Point Numbers

The floating point numbers representation is based on the scientific notation: the decimal point is not set in a fixed position in the bit sequence, but its position is indicated as a base power.




All the floating point numbers are composed by three components:

- *Sign:* it indicates the sign of the number (0 positive and 1 negative)
- *Mantissa*: it sets the value of the number

- *Exponent*: it contains the value of the base power (biased)
- *Base*: the base (or radix) is implied and it is common to all the numbers (2 for binary numbers)

The free using of this format caused either designed their own floating point system. For example, Konrad Zuse did the first modern implementation of a floating point arithmetic in a computer he had built (the Z3) using a radix-2 number system with 14-bit significant, 7-bit exponents and 1-bit sign. On the other hand the PDP-10 or the Burroughs 570 used a radix-8 and the IBM 360 had radix-16 floating point arithmetic.

This led to the need for a standard which would make a clear and concise format to be used by all the developers.

## 1.2. The Standard IEEE 754

The first question that comes to mind is "*What's IEEE?"*. The *Institute of Electrical and Electronics Engineers* (IEEE) is a non-profit professional association dedicated to advancing technological innovations and excellence.

It was founded in 1884 as the AIEE (*American Institute of Electrical Engineers*). The IEEE was formed in 1963 when AIEE merged with IRE (*Institute of Radio Engineers*).

One of its many functions is leading standards development organization for the development of industrial standards in a broad range of disciplines as telecommunications, consumer electronics or nanotechnology.

IEEE 754 is one of these standards.

### *1.2.1. Overview*

Standard IEEE 754 specifies formats and methods in order to operate with floating point arithmetic.

These methods for computational with floating point numbers will yield the same result regardless the processing is done in hardware, software or a combination for the two or the implementation.

The standard specifies:

- Formats for binary and decimal floating point data for computation and data interchange
- Different operations as addition, subtraction, multiplication and other operations
- Conversion between integer-floating point formats and the other way around
- Different properties to be satisfied when rounding numbers during arithmetic and conversions
- Floating point exceptions and their handling (NaN, ±∞ or zero)

IEEE 754 specifies four different formats to representing the floating point values:

- Simple Precision (32 bits)
- Double precision (64 bits)
- Simple Extended Precision (≥43 bits but not too used)
- Double Extended Precision (≥79 bits, usually represented by 80)

### *1.2.2. Binary Interchange Format Encodings*

Representations of floating point data in the binary interchange formats are encoded in *k* bits in the following three fields ordered as shown in Figure 1:

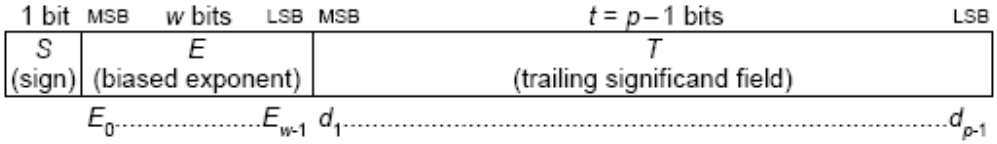



***Figure 1.*** *Floating Point format*

If a Simple Precision format is used the bits will be divided in that way:

- The first bit (31st bit) is set the sign (*S*) of the number (0 positive and 1 negative)
- Next *w* bits (from 30th to 23rd bit) represents the exponent (*E*)
- The rest of the string, *t*, (from 22nd to 0) is reserved to save the mantissa

The range of the enconding biased exponent is divided in three sections:

- Every integer between 1 and $2^w$-2 (being $w$=8 → $254_{(10)}$) in order to encode the normal numbers
- The value 0 which encodes subnormal numbers and the zero value
- The reserved value $2^w$-1 (being $w$=8 → $255_{(10)}$) to encode some special cases as NaN or ±∞

The exponent value has a bias of 127. It means the exponent value will be between -126 ($00000000_{(2)}$) and +127 ($11111110_{(2)}$) being zero at the value ($01111111_{(2)}$).

Exponent and mantissa values determine the different number *r* cases that it can be had.

- If $E = 2^w - 1$ and $T \neq 0$, then *r* is NaN regardless of *S*
- If $E = 2^w - 1$ and $T = 0$, then *r* is ±infinity according with the sign bit *S*
- If $1 \leq E \leq 2^w - 2$, then *r* is a normal number
- If $E = 0$ and $T \neq 0$, then *r* is a subnormal number
- If $E = 0$ and $T = 0$, then *r* is ±zero according with *S*

The mantissa value is 23 bits long but it contains an implicit bit depending on the type of data (1 for normal numbers and 0 for subnormal).

A number can be represented by different ways. As an example, the number $0.11 \cdot 2^5$ can be described as $110 \cdot 2^2$ or $0.011 \cdot 2^6$.

It is desirable to require unique representations. In order to reach this goal the finite non-zero floating point numbers may be normalized by choosing the representation for which the exponent is minimum.

To cope with this problem the standard provides a solution. The numbers will be standardized in two ways:

- Subnormal numbers will start with a zero an it has a form like $\pm 0.XX \cdot 2^0$
- Normal numbers MSB will be high ($\pm 1.XX \cdot 2^E$) where *0<E<255*

Both normal and subnormal numbers MSB will be implied but taken into account in order to get the proper value in decimal.

To calculate the value of the binary bit sequence in decimal this formula will be used:

$$M = \sum_{k=0}^{22} m_{22-k} \cdot 2^{-(1+k)} \tag{1}$$

Finally the different format parameters for simple and double precision are shown in *table 1*:

***Table 1.*** *Binary interchange format parameters*

| Parameter | binary32 | binary64 |
|---|---|---|
| *k*, storage width in bits | 32 | 64 |
| *p*, precision in bits | 24 | 53 |
| *emax*, maximum exponent *e* | 127 | 1023 |
| *Encoding parameters* | | |
| *bias*, $E-e$ | 127 | 1023 |
| sign bit | 1 | 1 |
| *w*, exponent field width in bits | 8 | 11 |
| *t*, trailing significand field width in bits | 23 | 52 |
| *k*, storage width in bits | 32 | 64 |

### *1.2.3. Precision and Rounding*

The number of values which can be represented by floating point arithmetic is finite because it has a finite number of bits.

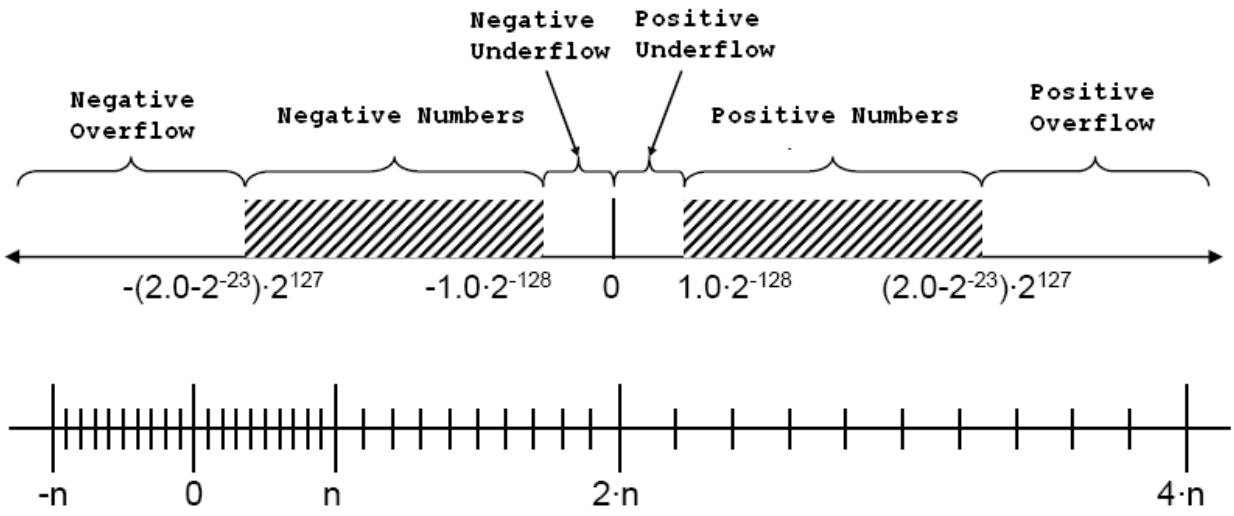



***Figure 2.*** *Floating Point values range*

As it can be seen in the *figure 2*, the standardized numbers range is described as the values between the higher exponent and mantissa value and the lower ones. The subnormal numbers are between zero and the lowest number (positive or negative) which could be represented by normal numbers. However, these ranges are discontinuous because between two numbers there are also infinite real ones. The quantities of numbers, which can be represented, are the same than in fixed point but at the expense of increasing the distance between numbers a higher range is achieved.

The standard IEEE 754 requires that the operation result must be the same which would obtain if a calculation with absolute precision and rounded had been done.

Four types of rounding are described by the standard:

- Rounding to the nearest (to even number in case of tie) is the floating point number that is the closest to *x*.
- Rounding to +∞ is the smallest floating point number (possibly +∞) greater than or equal to *x*.
- Rounding to -∞ is the largest floating point number (possibly -∞) less than or equal to *x*.
- Rounding to zero is the closest floating point number to *x* that is no greater in magnitude than *x* (it is equal to rounding to -∞ if $x \geq 0$ and to +∞ if $x \leq 0$

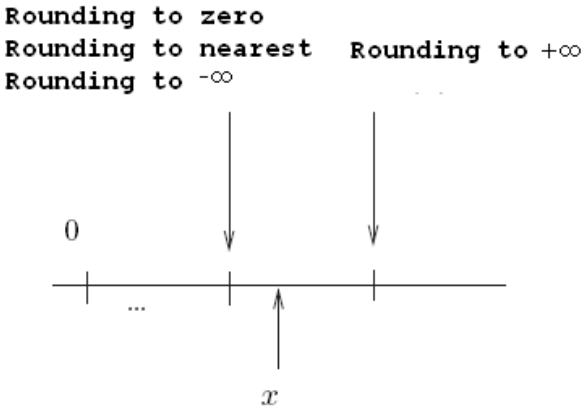



***Figure 3.*** *Rounding Modes*

The finite number of representing values and the rounding cause the appearance of errors in the result. This topic should be discussed when the results will be analyzed.

# CHAPTER 2:

# CODE DEVELOPMENT

Once the standard IEEE 754 has been explained it is time to start with the implementation of the code. First of all thinking about the different steps we should do to perform the operation required is compulsory. It is because of this that this section will talk about the procedure in addition/subtraction operations and a first look at the code design in block diagram way.

A complete description will be done first and the subblocks will be explained immediately afterwards at successive subsections.

## 2.1. 32-bits Floating Point Adder Design

The main goal of this project is the implementation of a 32-bit Floating Point Adder with VHDL code. The format and the main features of the standard have been described before but nothing about the steps to achieve the target has been said.

The first logical step is trying to specify what operations should be done to obtain a proper addition or subtraction. Once the idea will be clear the block diagram of the entire code will be designed.

### *2.1.1. Addition/Subtraction Steps*

Following the established plan, the way to do the operations (addition/subtraction) will be set.

This point will be also used to try to explain why these steps are necessary in order to make clearer and easier the explanation of the code in the next section.

The different steps are as follows:

1. Extracting signs, exponents and mantissas of both A and B numbers. As it has been said, the numbers format is as follows:

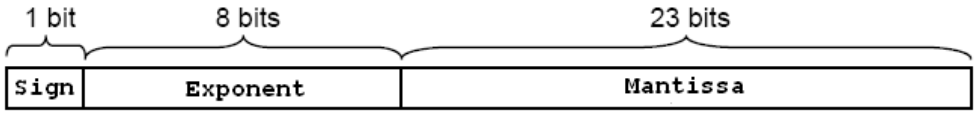



***Figure 4.*** *Floating Point Number format*

Then the first step is finding these values.

2. Treating the special cases:
   - Operations with A or B equal to zero
   - Operations with ±∞
   - Operations with NaN
3. Finding out what type of numbers are given:
   - Normal
   - Subnormal
   - Mixed
4. Shifting the lower exponent number mantissa to the right $[Exp1 - Exp2]$ bits. Setting the output exponent as the highest exponent.

   A's Exponent → 3  B's Exponent → -1  Difference (A-B) → 4

   Number B:

   1 1 0 1 0 0 1 → **0 0 0 0** 1 1 0 1 0 0 1
5. Working with the operation symbol and both signs to calculate the output sign and determine the operation to do.

***Table 1.*** *Sign Operation*

| A's Sign | Symbol | B's Sign | Operation |
|---|---|---|---|
| + | + | + | + |
| + | + | - | - |
| + | - | + | - |
| + | - | - | + |
| - | + | + | - |
| - | + | - | + |
| - | - | + | + |
| - | - | - | - |

6. Addition/Subtraction of the numbers and detection of mantissa overflow (carry bit)

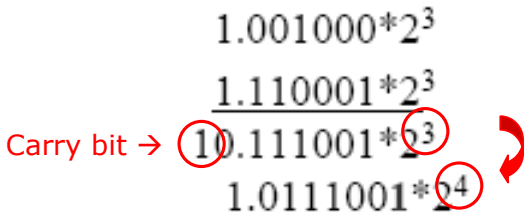



***Figure 5.*** *Example*

7. Standardizing mantissa shifting it to the left up the first one will be at the first position and updating the value of the exponent according with the carry bit and the shifting over the mantissa.

$$0.1010101 \cdot 2^3 \quad \leftarrow \quad 1.010101 \cdot 2^2$$

8. Detecting exponent overflow or underflow (result NaN or ±∞)

This is the way forward to proper operation. Obviously there are some parts which have to be discussed because there will be more aspects to be taken into account but this will happen in next sections where the code will be explained.

### *2.1.2. Block Diagram*

The main idea has been described before. Once the different steps to follow have been explained it is time to start to think in the code implementation.

In this subsection a first block diagram –as a draft- will be made. It still does not go into the most difficult points because in the next section, once a division of the project in three parts will be done, a complete description of each step will be performed.

These three parts are as follows:

- Pre-Adder Block
- Adder Block
- Standardizing Block

They make reference to the three main processes of the project. First the numbers should be treated (*pre-adder*) in order to perform the operation properly (*adder*) and finally, standardizing the result according with the standard IEEE 754 (*standardizing*).

In *figure 6*, a first approximation of the design has been done:

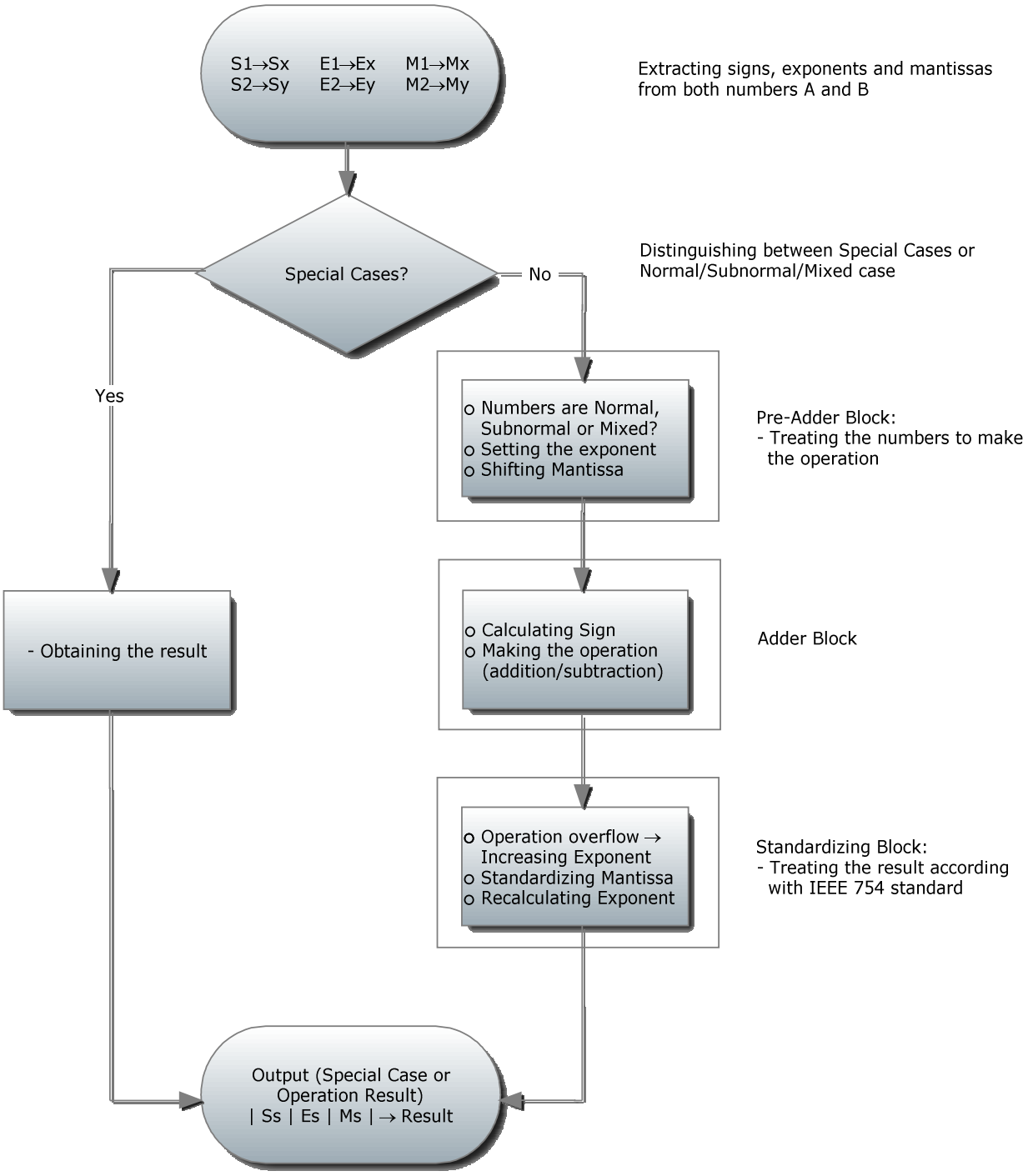



***Figure 6.*** *Block Diagram Code*

## 2.2. Blocks Design

In this section the main blocks described in the previous block diagram will be explained.

The diagram has two branches:

- The special cases one is quiet simple because only the combination of the different exceptions are taken into account. This will be explained in the next chapter over the code directly
- The second one is more interesting. It includes the main operation of the adder. The different operations that should be done are divided in three big blocks: pre-adder, adder and normalizing block.

During the next subsections a first description of each block will be done. A block diagram will be designed to support the explanation and facilitate the comprehension. Moreover it will be used to design the different blocks in VHDL which form the 32-bit Floating Point Adder.

### *2.2.1. Pre-Adder Design*

The first subblock is the Pre-Adder. The goals are:

1. Distinguishing between normal, subnormal or mixed (normal-subnormal combination) numbers.
2. Treating the numbers in order to be added (or subtracted) in the adder block.
   - Setting the Output's exponent
   - Shifting the mantissa
   - Standardizing the subnormal number in mixed numbers case to be treated as a normal case

The block diagram which display this behaviour is shown (*figure 7*) in the next page.

### *2.2.2. Adder Design*

Adder is the easiest part of the blocks. This block only implements the operation (addition or subtraction). It can be said the adder block is the ALU (*Arithmetic Logic Unit*) of the project because it is in charge of the arithmetic operations.

Two functions are implemented in this part of the code:

1. Obtaining the output's sign
2. Implementing the desired operation

In this block two related problems should be taken into account. Firstly, the calculation symbol (+ or -) depends on itself and the A and B's signs. Secondly, positive or negative numbers addition gives the same result. The problem will appear when the signs are different. In these cases the positive number will be kept in the first operand and the negative one in the second operand. All these problems will be explained in detail in next sections.

As it is normal the easiest block has the easiest block diagram (*figure 8*).

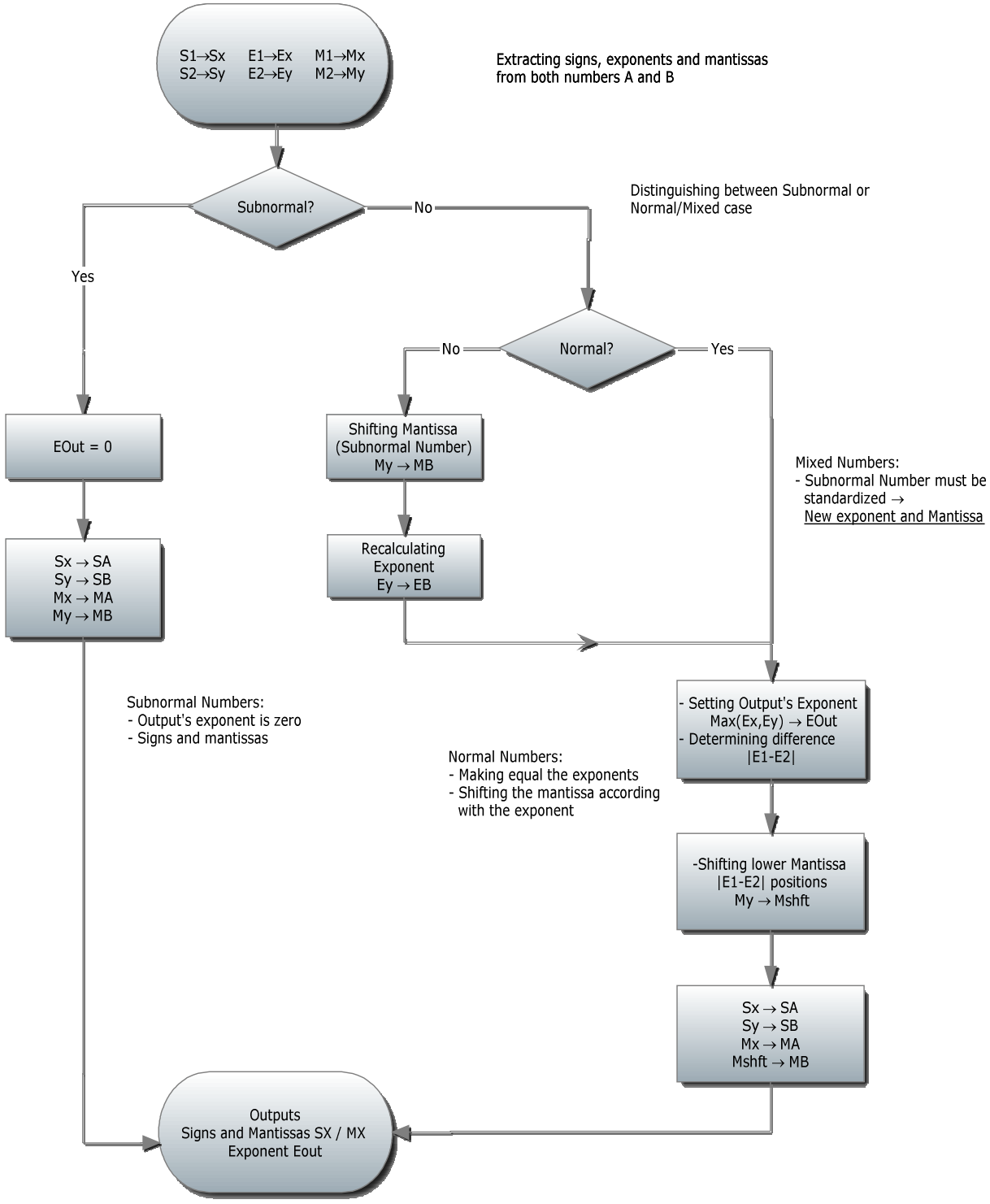



***Figure 7.*** *Pre-Adder Block Diagram*

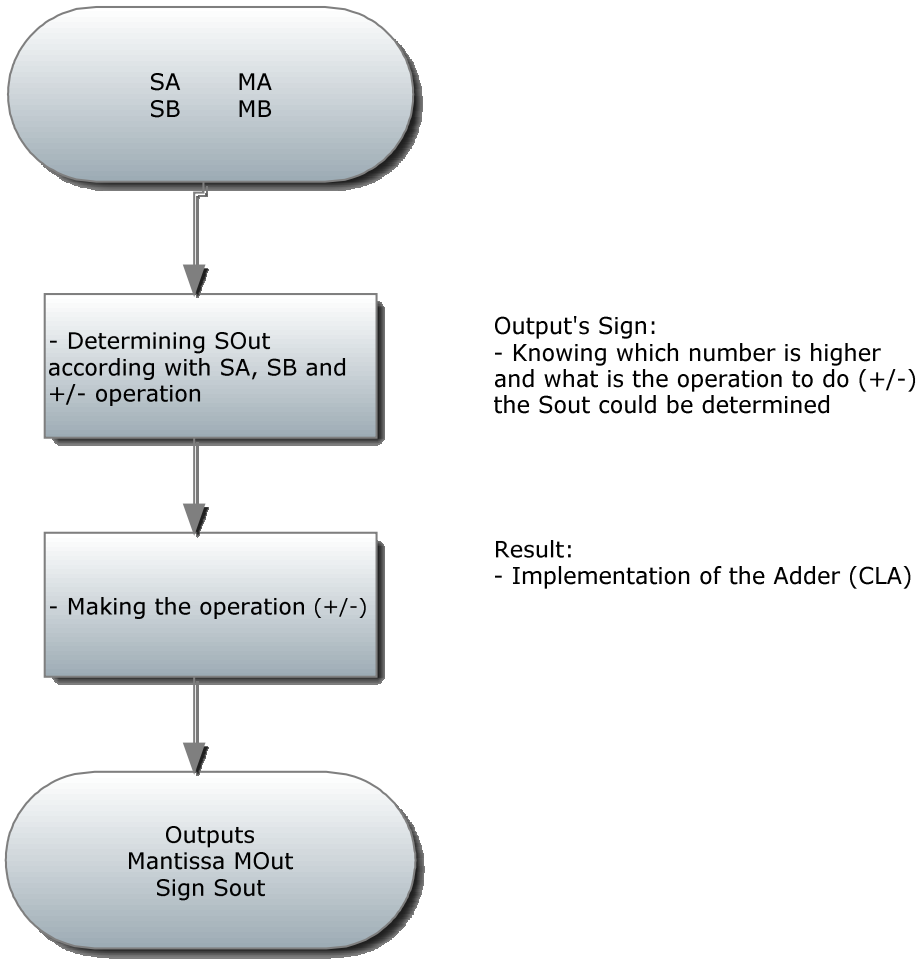



***Figure 8.*** *Adder Block Diagram*

### *2.2.3. Standardizing Design*

Finally the Standardizing Block takes the result of the addition/subtraction and gives it an IEEE 754 format.

The procedure is as follows:

1. Shifting the mantissa to standardize the result
2. Calculating the new exponent according with the addition/subtraction overflow (carry out bit) and the displacement of the mantissa.

The exponent value must be controlled when these steps are going to be made because it could be the number of positions the mantissa must be shifted are higher than the exponent value. In this case the result becomes subnormal. Another exception is when exponent and number of displacements are equal: mantissa will be shifted and exponent will be one.

As the previous subsections a block diagram with this description has been made. It can be seen in the *figure 8* where the different steps to standardize the value are shown.

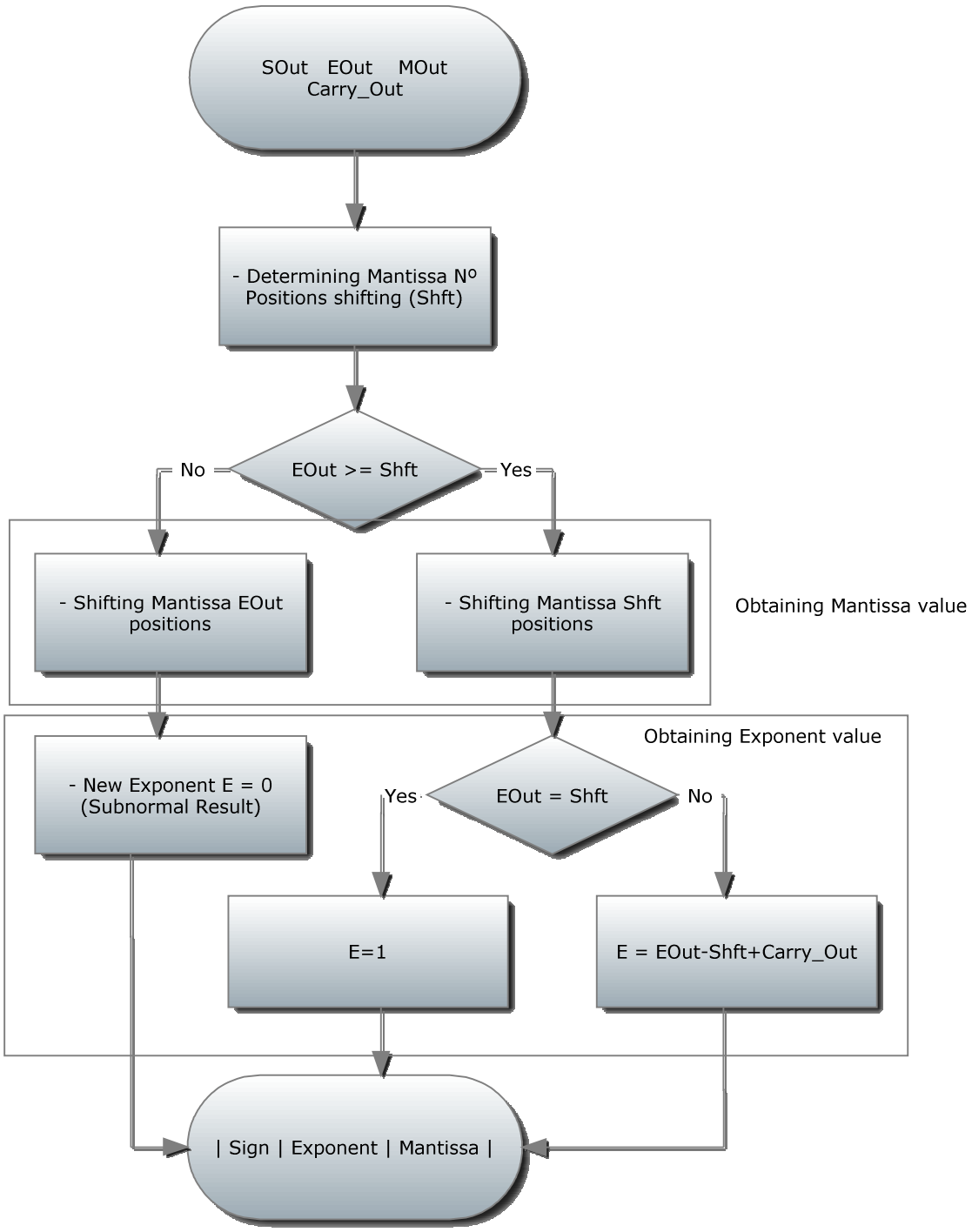



***Figure 9.*** *Standardizing Block Diagram*

# CHAPTER 3: PRE-ADDER

The first block is the Pre-adder. It is in the charge of distinguishing the type of numbers which are introduced as an input.

Four different cases are possible:

1. One of the different combinations which have been explained and labeled as special cases: NaN-Infinity, Infinity-Normal, Zero-Subnormal, etc.
2. A two subnormal numbers introduction.
3. A mixed option between normal and subnormal numbers.
4. A two normal numbers introduction

All this cases must be treated separately because of the process to achieve a successful operation must be different.

## 3.1. Special Cases

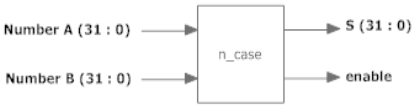
The adder is not always necessary to operate the numbers: there are some special cases which can be solved without it.

As it has been said, in addition to normal and subnormal numbers, infinity, NaN and zero are represented in IEEE 754 standard. Some possible combinations have a direct result, for example, if a zero and a normal number are introduced the output will be the normal number directly. Time and resources are saved implementing this block. The *n_case* block has been designed to run this behaviour.

### *3.1.1. n_case Block*

Both number A and number B are introduced as inputs. Vector *S* is one of the outputs and it contains the result when there is a special case, otherwise

undefined. Finally, *enable* signal enables or disables the *adder* block if it is needed or not.




```
entity n_case is
    Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
              NumberB : in std_logic_vector(31 downto 0);   -- Number B
              enable  : out std_logic;                      -- Enable Adder
              S       : out std_logic_vector(31 downto 0)); -- Output
end n_case;
```

Firstly the possible number values are coded (zero, infinity, NaN, normal and subnormal numbers) in two signals *outA* and *outB* according to the mantissa and exponent value as it can be seen in table 2.

```
outA  <= "000" when EA = X"00" and MA = 0 else                    -- Zero
         "001" when EA = X"00" and MA > 0 else                    -- Subnormal
         "011" when (EA > X"00" and EA < X"FF") and MA > 0 else    -- Normal
         "100" when EA = X"FF" and MA = 0 else                    -- Infinity
         "110" when EA = X"FF" and MA > 0 else                    -- NaN
         "000";

outB  <= "000" when EB = X"00" and MB = 0 else                    -- Zero
         "001" when EB = X"00" and MB > 0 else                    -- Subnormal
         "011" when (EB > X"00" and EB < X"FF") and MB > 0 else    -- Normal
         "100" when EB = X"FF" and MB = 0 else                    -- Infinity
         "110" when EB = X"FF" and MB > 0 else                    -- NaN
         "000";
```

***Table 2.*** *Data coded*

| Exponent | Mantissa | Output | Output Coded |
|---|---|---|---|
| = 0 | = 0 | Zero | 000 |
| = 0 | > 0 | Subnormal | 001 |
| 0<E<255 | > 0 | Normal | 011 |
| = 255 | = 0 | Infinity | 100 |
| = 255 | > 0 | NaN | 110 |

Once both A and B numbers have been coded the different signals combinations are taken into account.

Sign, mantissa and exponent are calculated depending on *outA* and *outB* values. For example, if *outA* is a zero and *outB* is a normal number, the result is the normal number coded in *outB*.

All the possible values are shown in table 3 and also in the VHDL code added.

```
process (SA, SB, outA, outB)
  begin
    -------------------------- Zero
    if (outA = "000") then                              -- Zero +/- Number B
      SS <= SB;
      ES <= EB;
      MS <= MB;
    elsif (outB = "000") then                           -- Number A +/- Zero
      SS <= SA;
      ES <= EA;
      MS <= MA;
    end if;
    -------------------------- Infinite
    if (outA(0) = '1' and outB = "100") then            -- Normal or Subnormal +/- Infinity
      SS <= SB;
      ES <= EB;
      MS <= MB;
    elsif (outB(0) = '1' and outA = "100") then         -- Infinity +/- Normal or Subnormal
      SS <= SA;
      ES <= EA;
      MS <= MA;
    end if;

    if ((outA and outB) = "100" and SA = SB) then       -- +/- Infinity +/- Infinity
      SS <= SA;
      ES <= EA;
      MS <= MA;
    -------------------------- NaN
    elsif ((outA and outB) = "100" and SA /= SB) then   -- + Infinity - Infinity
      SS <= '1';
      ES <= X"FF";
      MS <= "00000000000000000000001";
    if (outA = "110" or outB = "110") then
      SS <= '1';
      ES <= X"FF";
      MS <= "00000000000000000000001";
    end if;
    -------------------------- Normal / Subnormal
    if((outA(0) and outB(0)) = '1') then
      SS <= '-';
      ES <= "--------";
      MS <= "-----------------------";
    end if;
end process;
```

***Table 3.*** *Output coded*

| Sign | Out A | Out B | Sign Output | Output |
|---|---|---|---|---|
| X | Zero | Number B | SB | Number B |
| X | Number A | Zero | SA | Number A |
| X | Normal / Subnormal | Infinity | SB | Infinity |
| X | Infinity | Normal / Subnormal | SA | Infinity |
| SA=SB | Infinity | Infinity | SX | Infinity |
| SA≠SB | Infinity | Infinity | 1 | NaN |
| X | NaN | Number B | 1 | NaN |
| X | Number A | NaN | 1 | NaN |

**X:** do not care **SA:** Number A's sign **SB:** Number B's sign **SX:** Sign A or B (it is the same)

Finally an enable signal has been made. If any normal or subnormal combination is had the enable signal is high, otherwise low.

```
-- If A and B are normal or subnormal numbers, enable = 1
-- If not, enable = 0
enable <= '1' when ((outA(0) and outB(0)) = '1') else '0';
```


***Figure 10.*** *n_case Simulation*

# 3.2. Subnormal Numbers

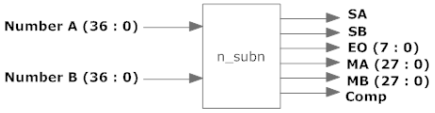
The operation using subnormal numbers is the easiest one.

It is designed in just one block and the procedure is as follows:

1. Obtaining the two sign bits and both mantissas
2. Making a comparison between both A and B numbers in order to acquire the largest number
3. Fixing the result exponent in zero

## *3.2.1. n_subn Block*

Obviously Number A and B are the entries. The outputs are six. *SA*-*MA* and *SB*-*MB* contain the sign and mantissa of A and B respectively. *Comp* signal is referred to the result comparison and *EO* is the result exponent.




```
entity n_subn is
    Port (    NumberA : in std_logic_vector(36 downto 0);    -- Number A
              NumberB : in std_logic_vector(36 downto 0);    -- Number B
              Comp    : out std_logic;                       -- Comparison A & B
              SA      : out std_logic;                       -- Sign A
              SB      : out std_logic;                       -- Sign B
              EO      : out std_logic_vector(7 downto 0);    -- Exponent Output
              MA      : out std_logic_vector(27 downto 0);   -- Mantissa A
              MB      : out std_logic_vector(27 downto 0));  -- Mantissa B
end n_subn;
```

The code is so simply. Sign and mantissa of both numbers are obtained directly from the entries *NumberA* and *NumberB*. The outputs exponent *EO* is always zero because the input exponents are zero as well and *Comp* signal is high when A is bigger than B and low in the opposite case.

The comparison operation does not take into account the sign of the numbers. If the result is negative or positive it will be calculated in the Adder block using *SA*, *SB* and *Comp* signals.

```
  SA <= NumberA(36);                    -- Sign A & B
  SB <= NumberB(36);

  MAa <= NumberA(27 downto 0);          -- Mantissa A & B
  MBb <= NumberB(27 downto 0);

------------------------------------------------ Number Comparison
   C    <= '1' when MAa >= MBb else     -- A > B
           '0' when MBb > MAa else      -- B > A
           '-';

  Comp <= C;

------------------------------------------------ Output's exponent
   EO <= NumberA(35 downto 28);

------------------------------------------------ Mantissa
  MB <= MBb when C = '1' else
        MAa when C = '0' else
        "----------------------------";
  MA <= MAa when C = '1' else
        MBb when C = '0' else
        "----------------------------";
```

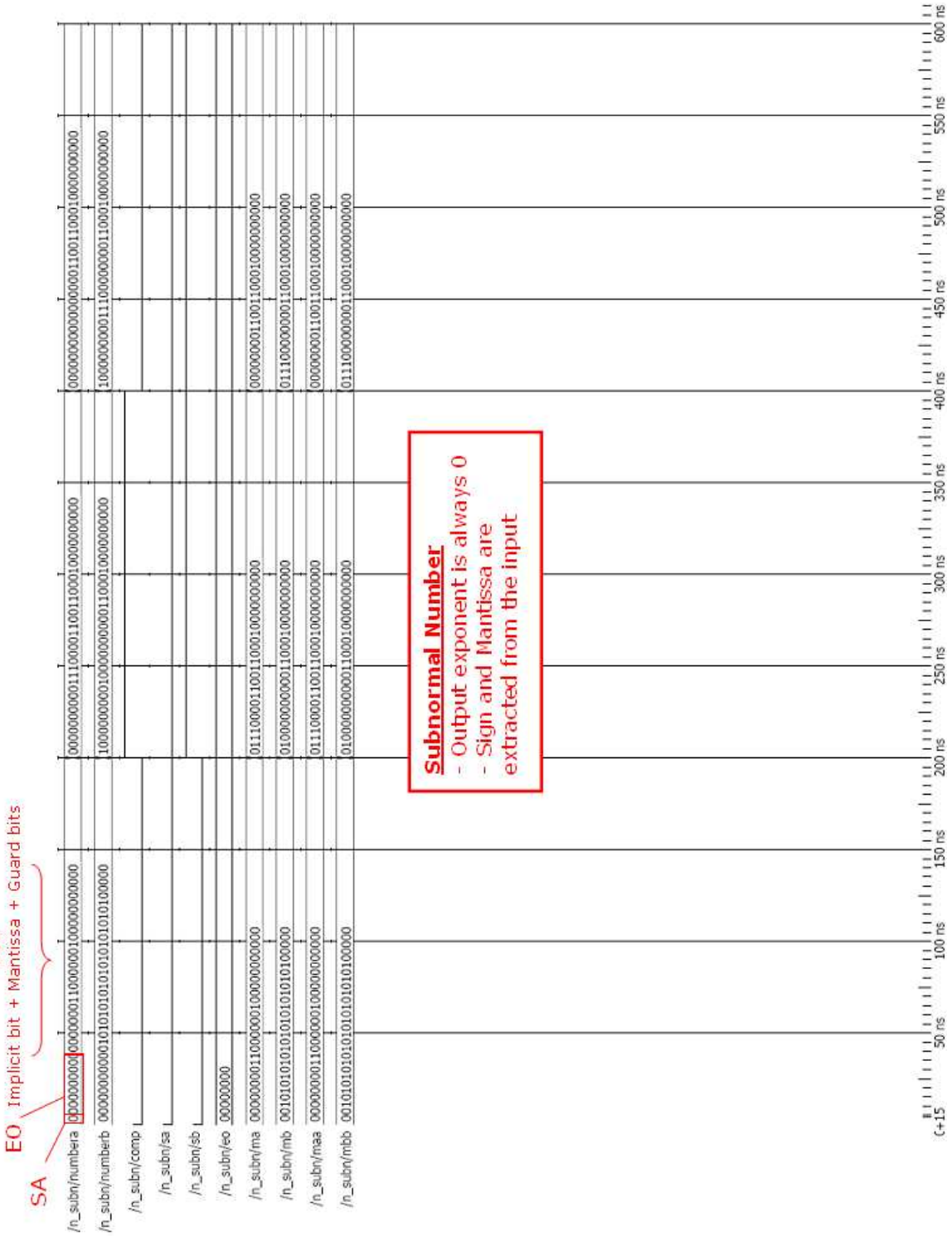



***Figure 11.*** *n_subn Entity*

## 3.3. Mixed Numbers

When there is a mixed combination of numbers (one subnormal and other normal) the subnormal one must have a special treatment in order to be added or subtracted to the normal one.

The subnormal number treatment is going to be discuss in this block because once both numbers will be standardized the next block (normal numbers block) will be in charge of the operation between normal ones.

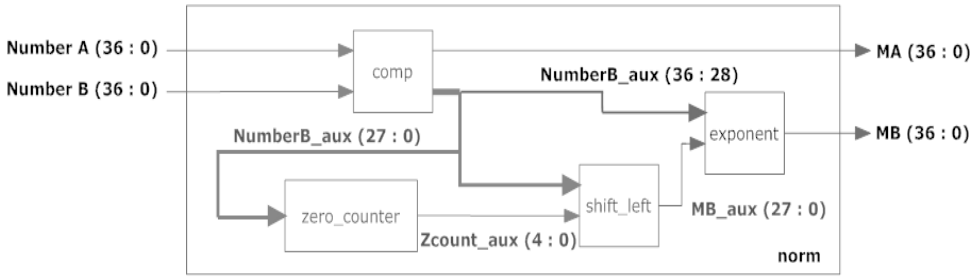



***Figure 12.*** *Mixed numbers block diagram*

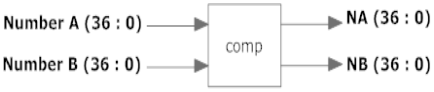
The work operation can be summarized in the following points:

1. Finding out what the subnormal number is
2. Counting the number of zeros the subnormal number has on the beginning
3. Shifting the vector and calculating the new exponent

This block is formed by three entities and each one is responsible for one of the points described.

### *3.3.1. comp Block*

First block is *comp* entity. The block entries are both numbers and the outputs are the same numbers ordered as normal *NA* and subnormal *NB*.




```
entity comp is
    Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
              NumberB : in std_logic_vector(36 downto 0);   -- Number B
              NA      : out std_logic_vector(36 downto 0);  -- Normal number
              NB      : out std_logic_vector(36 downto 0)); -- Subnormal number
end comp;
```

The code is not very extensive. A and B Mantissas are ordered according to the exponent: null exponent indicates what the subnormal number is and then this number is fixed in *NB*, leaving the normal one in *NA*.

```
EA <= NumberA(35 downto 28);          --  Exponent & Mantissa
EB <= NumberB(35 downto 28);

process (NumberA, NumberB, EA, EB)
  begin

    if EA = X"00" then              -- If Number A is subnormal...
      NB <= NumberA;
      NA <= NumberB;
    elsif EB = X"00" then           -- If Number B is subnormal...
      NB <= NumberB;
      NA <= NumberA;
    else
      NA <= "-----------------------------------";
      NB <= "-----------------------------------";
    end if;

  end process;
```

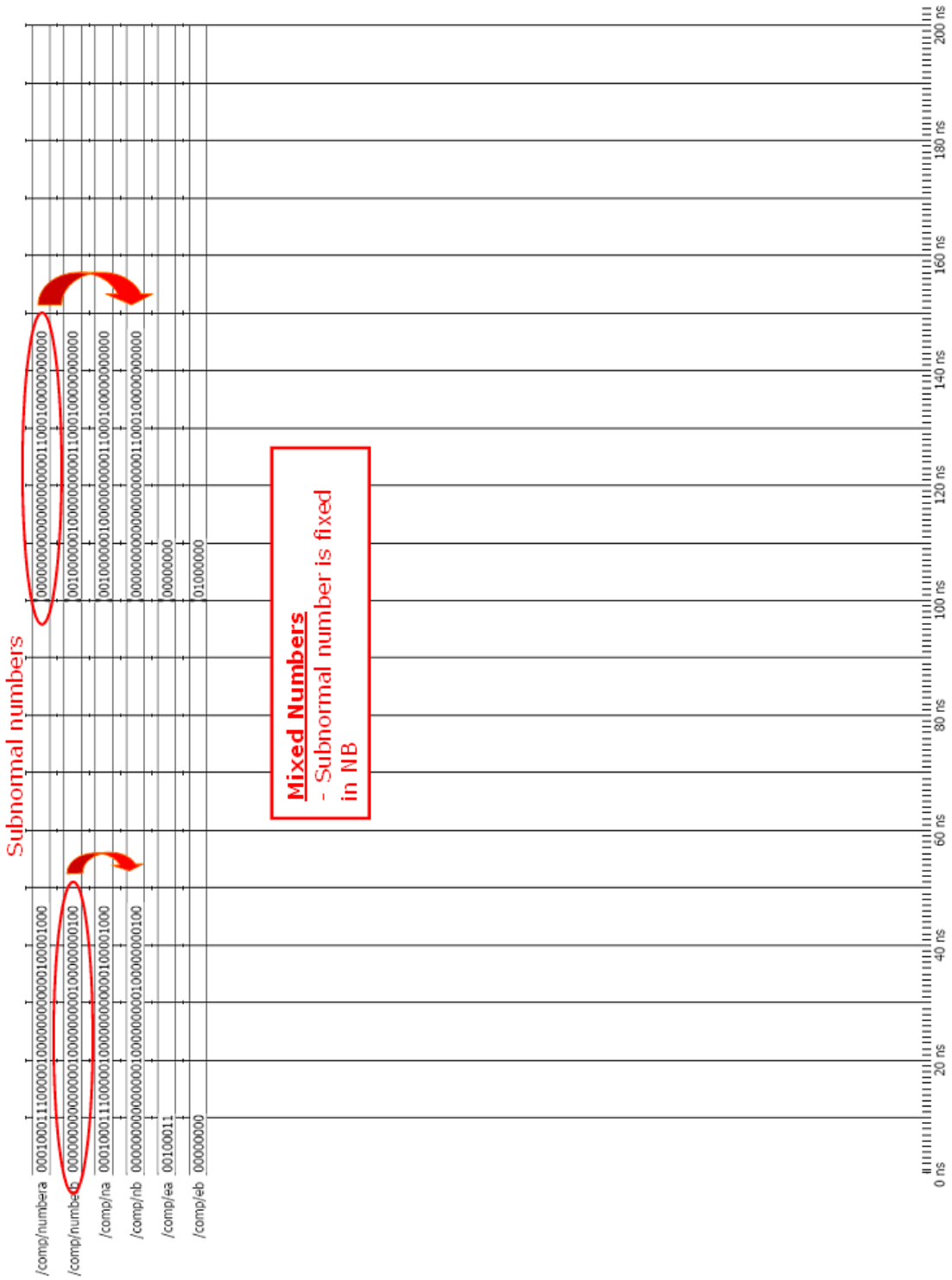
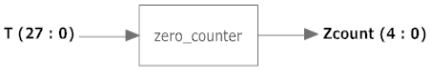



***Figure 13.*** *comp Simulation*

### 3.3.2. *zero Block*

Counting zeros is the *zero* block target. The mantissa which is needed to shift is introduced as an entry in *T* vector and the output *Zcount* contains the number of zeros the mantissa has on the beginning which corresponds with the number of positions the vector must be shifted.




```
entity zero is
    Port (    T      : in std_logic_vector(27 downto 0);    -- Significand
              Zcount : out std_logic_vector(4 downto 0));   -- Number of Zeros
end zero;
```

A zero vector is created (*Zero_vector*) and compared with the *T* vector. The *Zcount* value depends on the number of matches.

```
aux    <= "--------" when T(27 downto 27) = "-" else
          X"1C" when T(27 downto 0) = Zero_vector(27 downto 0) else
          X"1B" when T(27 downto 1) = Zero_vector(27 downto 1) else
          X"1A" when T(27 downto 2) = Zero_vector(27 downto 2) else
          X"19" when T(27 downto 3) = Zero_vector(27 downto 3) else
          X"18" when T(27 downto 4) = Zero_vector(27 downto 4) else
          X"17" when T(27 downto 5) = Zero_vector(27 downto 5) else
          X"16" when T(27 downto 6) = Zero_vector(27 downto 6) else
          X"15" when T(27 downto 7) = Zero_vector(27 downto 7) else
          X"14" when T(27 downto 8) = Zero_vector(27 downto 8) else
          X"13" when T(27 downto 9) = Zero_vector(27 downto 9) else
          X"12" when T(27 downto 10) = Zero_vector(27 downto 10) else
          X"11" when T(27 downto 11) = Zero_vector(27 downto 11) else
          X"10" when T(27 downto 12) = Zero_vector(27 downto 12) else
          X"0F" when T(27 downto 13) = Zero_vector(27 downto 13) else
          X"0E" when T(27 downto 14) = Zero_vector(27 downto 14) else
          X"0D" when T(27 downto 15) = Zero_vector(27 downto 15) else
          X"0C" when T(27 downto 16) = Zero_vector(27 downto 16) else
          X"0B" when T(27 downto 17) = Zero_vector(27 downto 17) else
          X"0A" when T(27 downto 18) = Zero_vector(27 downto 18) else
          X"09" when T(27 downto 19) = Zero_vector(27 downto 19) else
          X"08" when T(27 downto 20) = Zero_vector(27 downto 20) else
          X"07" when T(27 downto 21) = Zero_vector(27 downto 21) else
          X"06" when T(27 downto 22) = Zero_vector(27 downto 22) else
          X"05" when T(27 downto 23) = Zero_vector(27 downto 23) else
          X"04" when T(27 downto 24) = Zero_vector(27 downto 24) else
          X"03" when T(27 downto 25) = Zero_vector(27 downto 25) else
          X"02" when T(27 downto 26) = Zero_vector(27 downto 26) else
          X"01" when T(27 downto 27) = Zero_vector(27 downto 27) else
          X"00";

Zcount <= aux(4 downto 0);
```

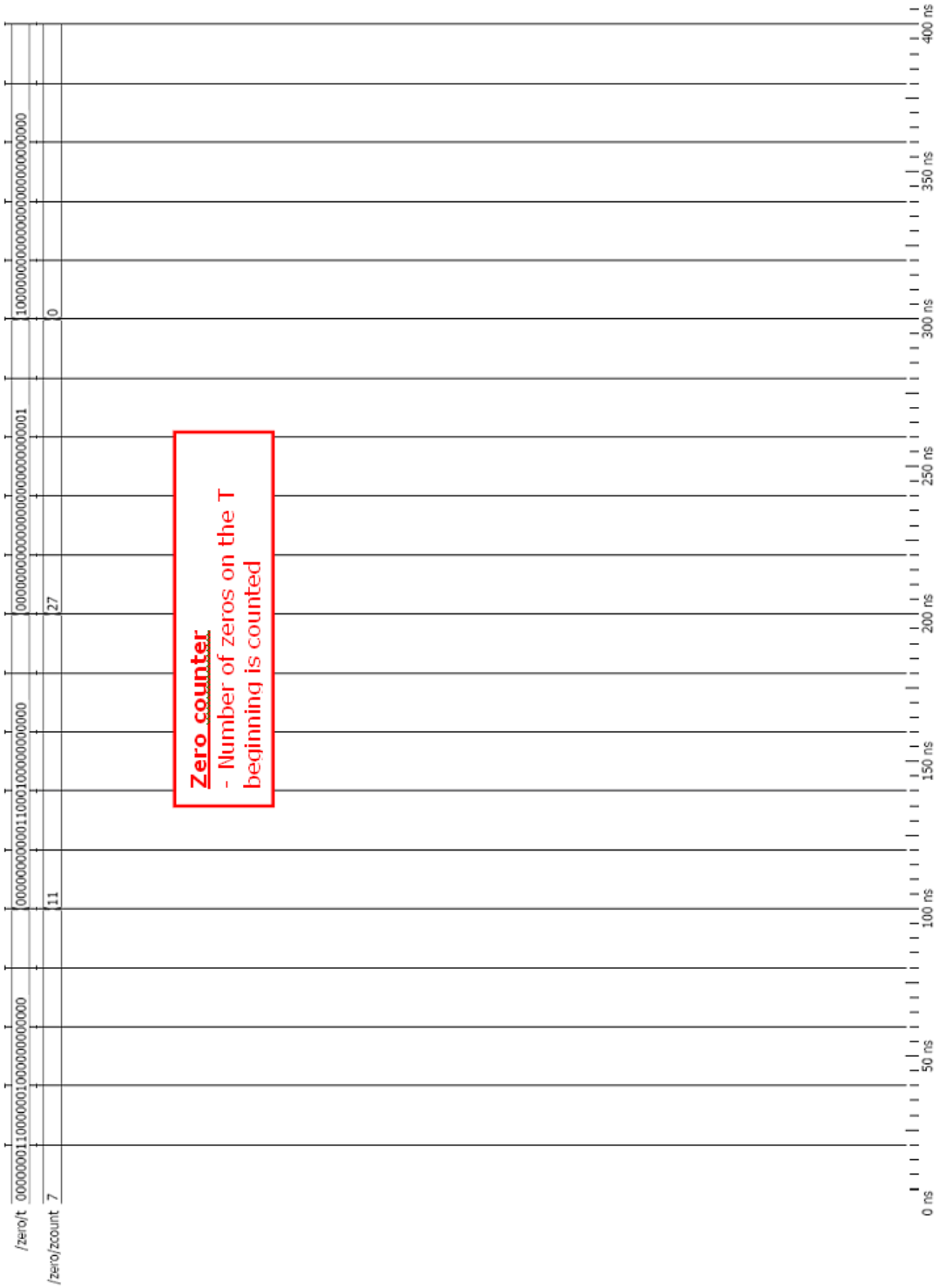



***Figure 14.*** *zero Simulation*

### 3.3.3. shift_left/shift Block

Shifting is required to match the normal and mixed mantissas to perform the addition/subtraction properly.

A logarithmic shift schematic as the figure 15 is used but with some differences.

28 bits (1 implicit bit + 23 mantissa's bits + 4 guard bits) is had in the Floating point Adder design then up to 28 positions must be able to shift. Because of the fact that this shifter consists of 5 stages: the first stage shift one position, the second stage 2, the third one 4, the fourth one 8 and the last one 16. Using any combination 32 positions are able to shift which is big enough to the design purpose.

Both shifting left and shifting right are used in the Floating Point Adder implementation. In this chapter, the first one is explained but the code is quite similar to the second one. There is only a difference: the *T* vector order. If the bits order is changed from 0-27 to 27-0 a shifting right is achieved.

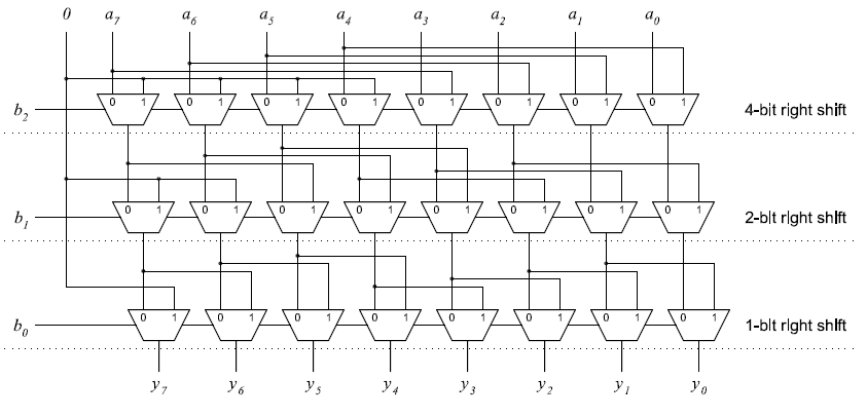



***Figure 15.*** *Logarithmic shift*

The *T* vector and the number of positions to shift (*Shft*) are the entries of the *shift* entity. The shifted signal is set in *S*.


```
entity shift is
    Port (    T     : in std_logic_vector(27 downto 0);    -- Significand to shift
              Shft  : in std_logic_vector(4 downto 0);     -- Exponent's subtraction
              S     : out std_logic_vector(27 downto 0));  -- Output
end shift;
```

The code is implemented as follows. A multiplexor has been designed and exported to this block. Afterwards a loop *for* has been used to generate the different 5 stages. Following the cascade design which has been shown before a 32 positions logarithmic shifter is implemented.

```
signal Z1, Z2, Z3, Z4, Z5  : std_logic_vector(27 downto 0);

begin

-- Components generation

Comp1: for i in 0 to 27 generate

  shifter0_0: if (i=0) generate
              shifter0_0comp: MUX port map (A => '0', B => T(0), Sel => Shft(0), Z => Z1(i));
              end generate;
  shifter0_i: if ((i>0) and (i<28)) generate
              shifter0_icomp: MUX port map (A => T((i-1)), B => T(i), Sel => Shft(0), Z => Z1(i));
              end generate;

  shifter1_0: if ((i>=0) and (i<2)) generate
              shifter1_0comp: MUX port map (A => '0', B => Z1(i), Sel => Shft(1), Z => Z2(i));
              end generate;
  shifter1_i: if ((i>1) and (i<28)) generate
              shifter1_icomp: MUX port map (A => Z1((i-2)), B => Z1(i), Sel => Shft(1), Z => Z2(i));
              end generate;

  shifter2_0: if ((i>=0) and (i<4)) generate
              shifter2_0comp: MUX port map (A => '0', B => Z2(i), Sel => Shft(2), Z => Z3(i));
              end generate;
  shifter2_i: if ((i>3) and (i<28)) generate
              shifter2_icomp: MUX port map (A => Z2((i-4)), B => Z2(i), Sel => Shft(2), Z => Z3(i));
              end generate;

  shifter3_0: if ((i>=0) and (i<8)) generate
              shifter3_0comp: MUX port map (A => '0', B => Z3(i), Sel => Shft(3), Z => Z4(i));
              end generate;
  shifter3_i: if ((i>7) and (i<28)) generate
              shifter3_icomp: MUX port map (A => Z3((i-8)), B => Z3(i), Sel => Shft(3), Z => Z4(i));
              end generate;


  shifter4_0: if ((i>=0) and (i<16)) generate
              shifter4_0comp: MUX port map (A => '0', B => Z4(i), Sel => Shft(4), Z => Z5(i));
              end generate;
  shifter4_i: if ((i>15) and (i<28)) generate
              shifter4_icomp: MUX port map (A => Z4((i-16)), B => Z4(i), Sel => Shft(4), Z => Z5(i));
              end generate;

end generate;

S <= Z5;
```

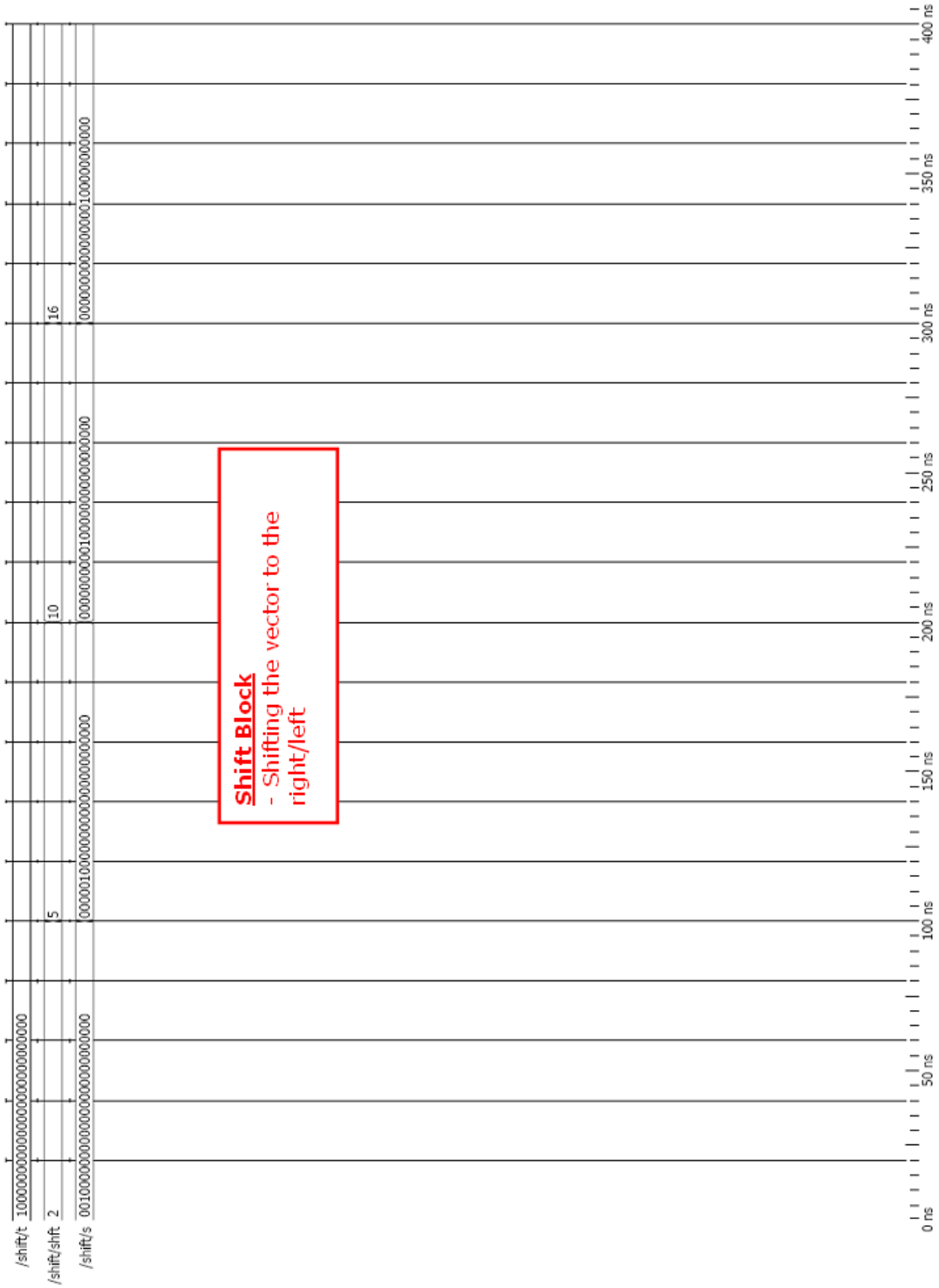
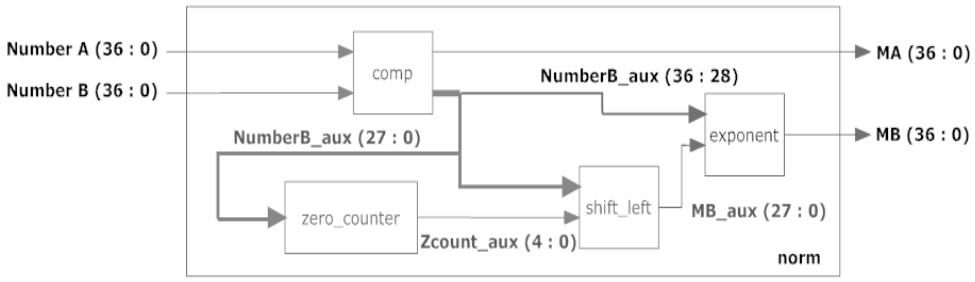



***Figure 16.*** *shift_left/shift Simulation*

### 3.3.4. norm Block

Finally, the rest of the entities are all included in the *norm* block. It also performs the output exponent treatment.

The inputs are the numbers A and B. Once the subnormal one has been shifted it is fixed in *MB*. The normal number is set in *MA*.




```
entity norm is
    Port (    NumberA : in std_logic_vector(36 downto 0);    -- NumberA
              NumberB : in std_logic_vector(36 downto 0);    -- NumberB
              MA      : out std_logic_vector(36 downto 0);   -- Normalized NumberA
              MB      : out std_logic_vector(36 downto 0));  -- Normalized NumberB
end norm;
```

The code could be divided in two parts. The first one implements the connection between the different blocks which the mixed numbers entity works with. The block diagram is coherent with the VHDL code.

```
------------------------------------------------ Components declaration
comp0 : zero
    port map (T => NumberB_aux(27 downto 0), Zcount => Zcount_aux);

comp1 : shift_left
    port map (T => NumberB_aux(27 downto 0), shft => Zcount_aux, S => MB_aux);

comp2 : comp
    port map (NumberA => NumberA, NumberB => NumberB, NA => MA, NB => NumberB_aux);
```

The second one is pretty interesting. As it has been explained before, negative prebiased exponents are not considered by the standard IEEE 754 but there is a possibility a normal and subnormal number may be operated. The number of positions the vector *MB* is shifted could be saved as a positive exponent but introducing a mark in the last guard bit which indicates the positive exponent is actually "negative".

So if a normal number with a quite small exponent is had it is possible that normal and subnormal numbers are able to be operated.

```
------------------------------------------------ New Exponent

process(Zcount_aux, NumberB_aux, EB, MB_aux)
  begin
    if Zcount_aux /= "-----" then
      EB <= "000" & Zcount_aux;                      -- Number shifted
      MB(27 downto 0) <= MB_aux(27 downto 1) & '1'; -- Bit 0 --> Mark
    else
      EB <= "--------";
      MB(27 downto 0) <= MB_aux;
    end if;
      MB(35 downto 28) <= EB;
      MB(36) <= NumberB_aux(36);
end process;
```

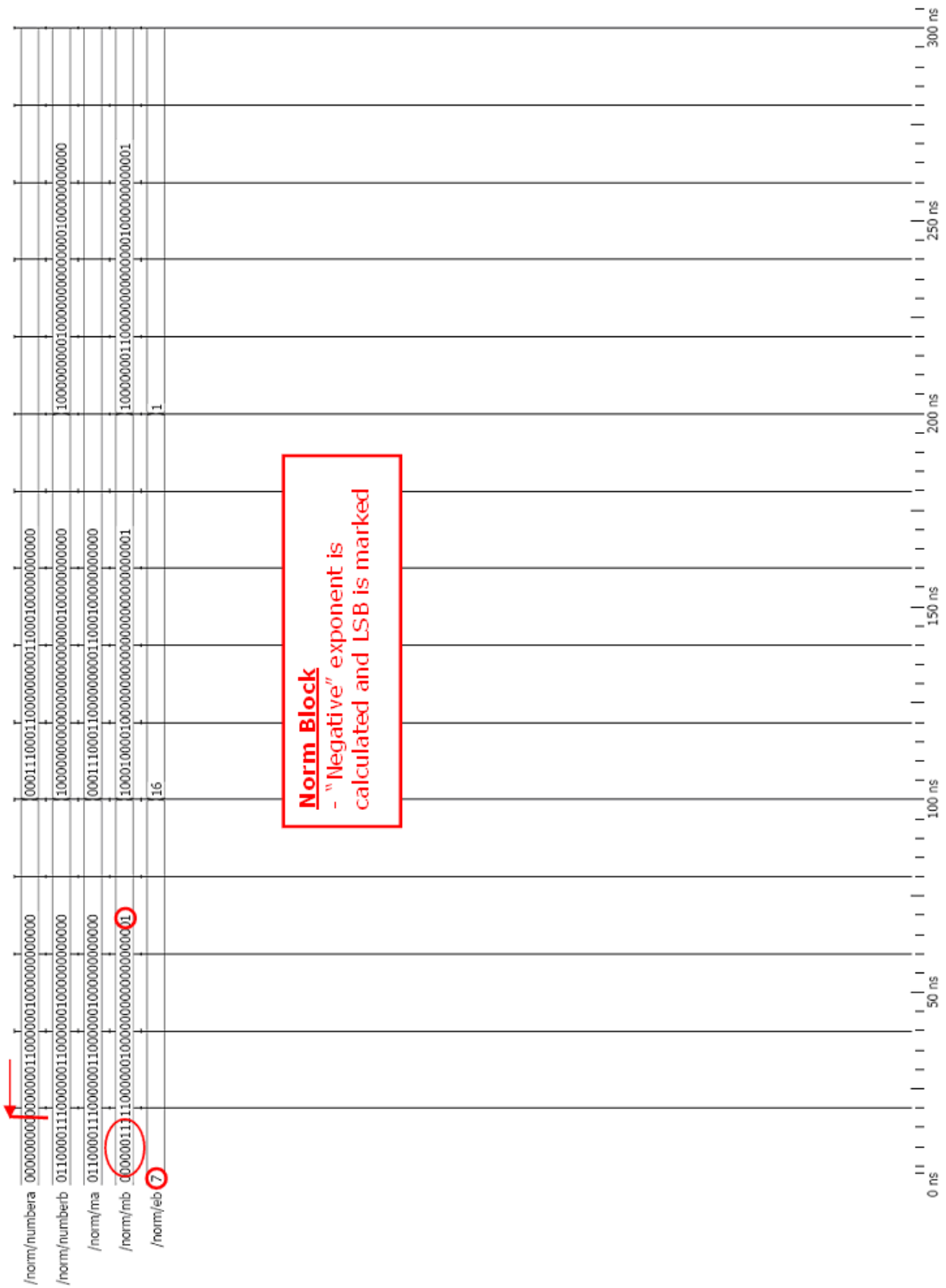



***Figure 17.*** *norm Simulation*

# 3.4. Normal Numbers

Two normal numbers are the most common operation mode because it represents the main operation without any exception.

The procedure is as follows:

1. Making a comparison between both A and B numbers and obtaining the largest number
2. Obtaining the output exponent (the largest one)
3. Shifting the smallest mantissa to equal both exponents

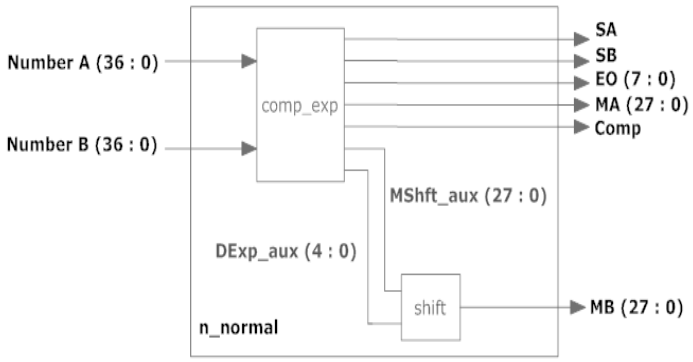
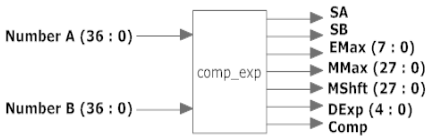



***Figure 18.*** *Normal numbers block diagram*

### *3.4.1. comp_exp Block*

The *comp_exp* entries are the two introduced numbers again. There are several outputs: *SA* and *SB* are the sign of A and B respectively, *EMax* is the output exponent, *MMax* the largest mantissa, *Mshft* the mantissa to shift, *Dexp* the number of positions *Mshft* must be shifted and *Comp* indicates what number is the largest one.




```
entity comp_exp is
    Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
              NumberB : in std_logic_vector(36 downto 0);   -- Number B
              SA      : out std_logic;                      -- Sign A
              SB      : out std_logic;                      -- Sign B
              Emax    : out std_logic_vector(7 downto 0);   -- Output exponent
              Mmax    : out std_logic_vector(27 downto 0);  -- Largest Mantissa
              Mshft   : out std_logic_vector(27 downto 0);  -- Mantissa to shift
              Dexp    : out std_logic_vector(4 downto 0);   -- Subtraction of exponents
              Comp    : out std_logic);                     -- Determine largest number
end comp_exp;
```

Exponents and signs are obtained from the introduced numbers directly. Once the exponents are fixed in *EA* and *EB* signals, these values are used to determine the largest number: if A is larger than B or number B's LSB (negative exponent mark) is high, *Comp* will be '1', otherwise '0'.

Using this signal the output exponent could be determined.

```
SA <= NumberA(36);                    -- Sign A & B
SB <= NumberB(36);

EA <= NumberA(35 downto 28);          --  Exponent & Mantissa
EB <= NumberB(35 downto 28);
MA <= NumberA(27 downto 0);
MB <= NumberB(27 downto 0);

------------------------------ Exponent Comparison

C <= '1' when (EA > EB) or (MB(0) = '1') else     -- Exponent A > Exponent B
     '0' when EA < EB else                        -- Exponent B > Exponent A
     '1' when MA >= MB else                       -- EA = EB --> A > B
     '0' when MA < MB else                        -- EA = EB --> B > A
     '-';

Comp <= C;

------------------------------ Largest exponent
Emax <= EA when C = '1' else
        EB when C = '0' else
        "--------";
```

Next step is determining the difference between both exponents. Once more time *comp* signal fixes the largest exponent and determines the subtraction order.

If B's LSB is high a negative exponent is had. In this case *EA* and *EB* are added.

```
------------------------------ Difference between exponents
dif   <= EA-EB when (C = '1') and (MB(0) = '0') else
         EB-EA when C = '0' else
         EA+EB when (C = '1') and (MB(0) = '1') else
         "--------";

process (dif)
  begin

    if dif <= X"1B" then            -- If the difference is less than or equal to 27...
      Dexp <= dif(4 downto 0);      -- Use directly the subtraction between exponents
    elsif dif > X"1B" then          -- If the difference is greater...
      Dexp <= "11100";              -- The difference is 28
    else
      Dexp <= "-----";
    end if;

  end process;

------------------------------ Mantissa

Mshft <= MB when C = '1' else
         MA when C = '0' else
         "----------------------------";
Mmax  <= MA when C = '1' else
         MB when C = '0' else
         "----------------------------";
```

The mantissa to shift corresponds with the smallest number (using *comp* again).

Finally a maximum value is set if the difference between exponents is greater than 28 which is the maximum number of bits that the mantissa has.

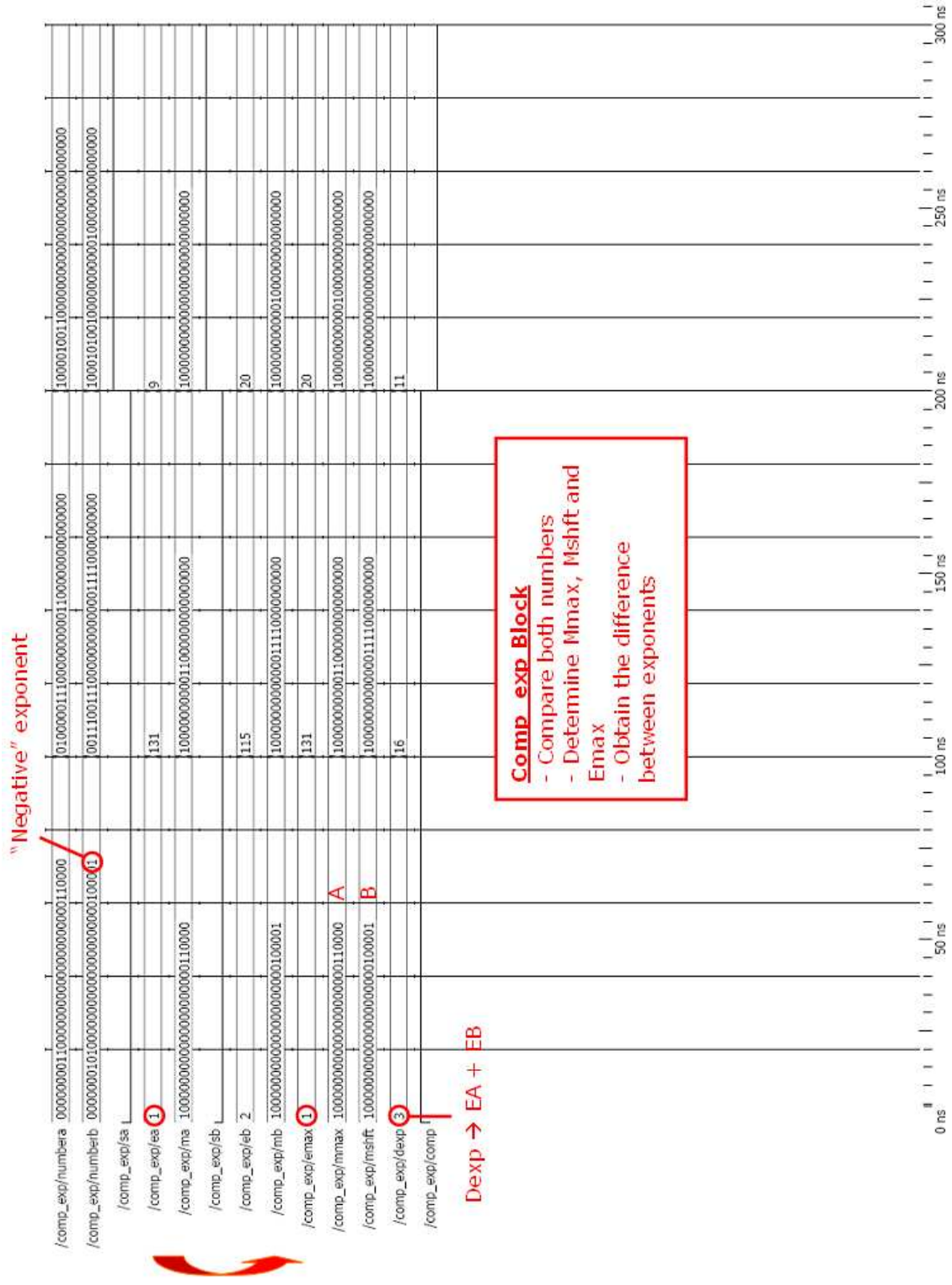



***Figure 19.*** *comp_exp Simulation*

### 3.4.2. shift Block

A shifter is needed to match the exponents. The entity is the same than in the mixed case. The vector *T* is the input which contains the mantissa to shift, *shft* fix the number of positions to move and *S* is the output with the result of the operation.


```
entity shift is
    Port (    T     : in std_logic_vector(27 downto 0);    -- Significand to shift
              Shft  : in std_logic_vector(4 downto 0);     -- Exponent's subtraction
              S     : out std_logic_vector(27 downto 0));  -- Output
end shift;
```

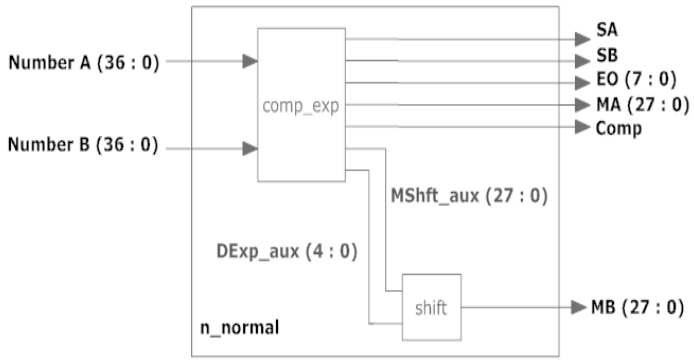
The code is quite similar. Only a part is added because it is enough to see its operation.

```
-- Components generation

Comp1: for i in 0 to 27 generate

  shifter0_0: if (i=0) generate
              shifter0_0comp: MUX port map (A => '0', B => T(27), Sel => Shft(0), Z => Z1(27-i));
              end generate;
  shifter0_i: if ((i>0) and (i<28)) generate
              shifter0_icomp: MUX port map (A => T(27-(i-1)), B => T(27-i), Sel => Shft(0), Z => Z1(27-i));
              end generate;

  shifter1_0: if ((i>=0) and (i<2)) generate
              shifter1_0comp: MUX port map (A => '0', B => Z1(27-i), Sel => Shft(1), Z => Z2(27-i));
              end generate;
  shifter1_i: if ((i>1) and (i<28)) generate
              shifter1_icomp: MUX port map (A => Z1(27-(i-2)), B => Z1(27-i), Sel => Shft(1), Z => Z2(27-i));
              end generate;
```

Changing the order of the vector, a displacement in the other direction is achieved. The simulation is not required because the result is the same but on the right.

### 3.4.3. n_normal Block

The *n_normal* block includes the two blocks which have been explained above. The entries are *NumberA* and *NumberB* and the outputs are both sign A (*SA*) and sign B (*SB*), the result exponent (*EO*), the *Comp* signal and the two mantissas (*MA* and *MB*).



```
entity n_normal is
    Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
              NumberB : in std_logic_vector(36 downto 0);   -- Number B
              Comp    : out std_logic;                      -- A & B Comparison
              SA      : out std_logic;                      -- Sign A
              SB      : out std_logic;                      -- Sign B
              EO      : out std_logic_vector(7 downto 0);   -- Exponent Output
              MA      : out std_logic_vector(27 downto 0);  -- Greatest Mantissa
              MB      : out std_logic_vector(27 downto 0)); -- Shifted Mantissa
end n_normal;
```

The VHDL code implements just the interconnection between the different blocks.

*Comp_exp* fix the mantissa which has to been shifted and the number of positions it must be displaced.

*Shift* block collects these two signals and gives the mantissa in order to be operated in the next block: the *Adder* block.

```
component comp_exp port (NumberA, NumberB : in std_logic_vector(36 downto 0);
                         SA, SB           : out std_logic;
                         Emax             : out std_logic_vector(7 downto 0);
                         Mmax, Mshft      : out std_logic_vector(27 downto 0);
                         Dexp             : out std_logic_vector(4 downto 0);
                         Comp             : out std_logic);
end component;

component shift port (T    : in std_logic_vector(27 downto 0);
                      shft : in std_logic_vector(4 downto 0);
                      S    : out std_logic_vector(27 downto 0));
end component;

signal Mshft_aux      : std_logic_vector(27 downto 0);
signal Dexp_aux       : std_logic_vector(4 downto 0);

begin
comp0 : comp_exp
    port map (NumberA => NumberA, NumberB => NumberB,
              SA => SA, SB => SB, Emax => EO, Mmax => MA, Mshft => Mshft_aux, Dexp => Dexp_aux, Comp => Comp);

comp1 : shift
    port map (T => Mshft_aux, shft => Dexp_aux,
              S => MB);
```

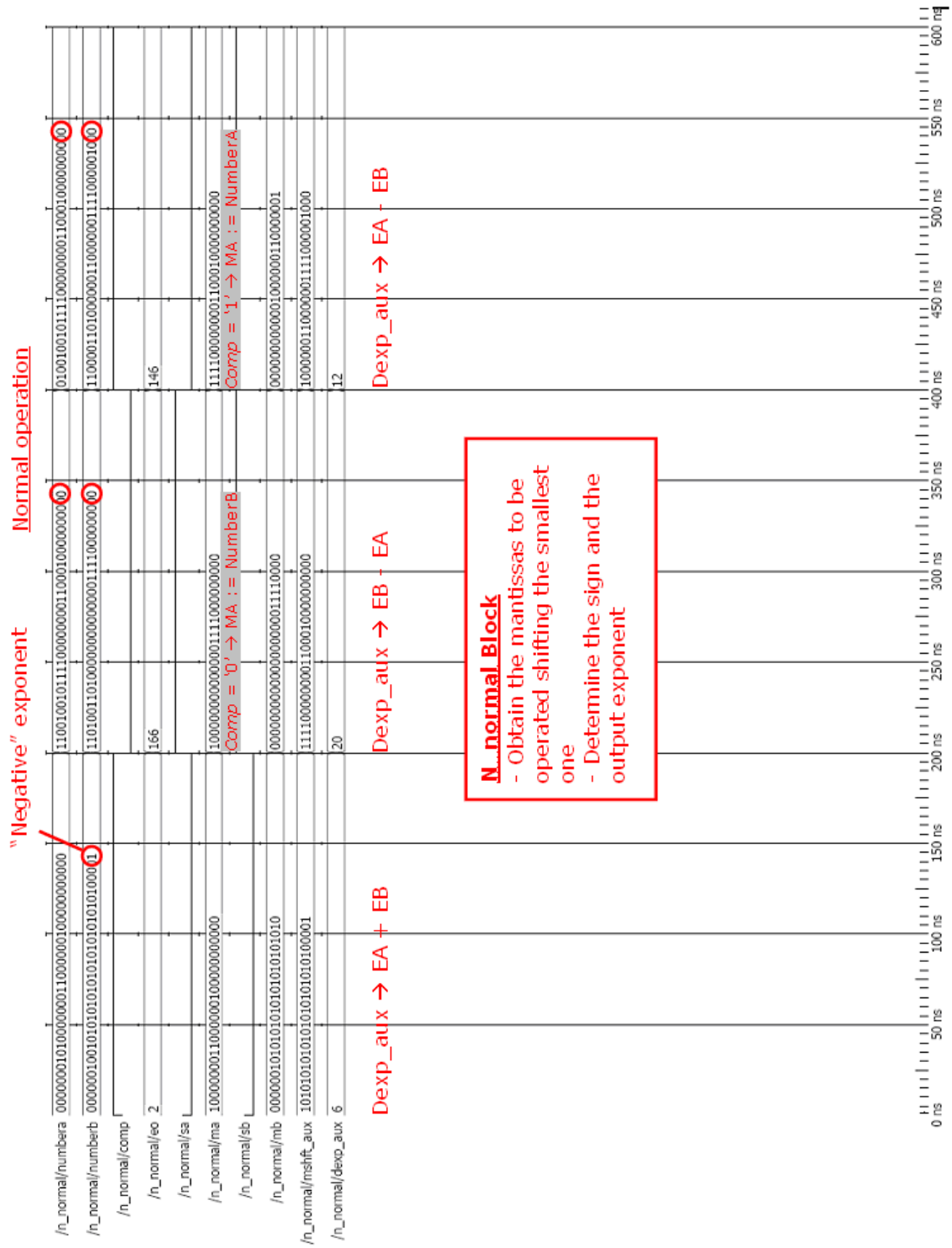



***Figure 20.*** *n_normal Simulation*

# 3.5. Pre-Adder

Finally a complete Pre-Adder block diagram will be shown and explained. As it could be seen there are some blocks which are not discussed. These blocks are 4:

1. The first one is so important: *Selector* block
2. A demultiplexor (*demux*) to route the signal in the correspondent block
3. A multiplexor (*mux_ns*) to choose between the mixed numbers or the normal ones
4. A multiplexor (mux_adder) to choose between the normal or subnormal numbers

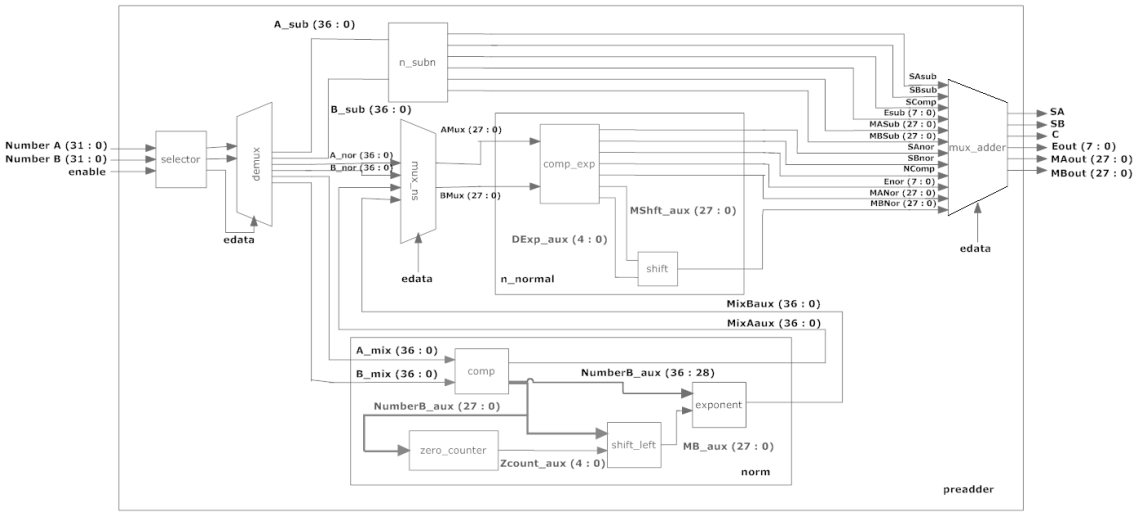



***Figure 21.*** *preadder block diagram*

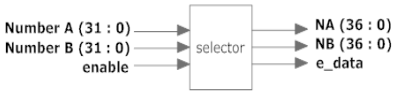
These blocks are going to be grouped in two different chapters. First one includes only the *selector* block which is more important and has more complexity.

The second group contains the different multiplexors and demultiplexors. They are going to be treated all together because of code's simplicity.

### *3.5.1. selector Block*

*Selector* block prepares the numbers: the entries are shorter than outputs because the implicit bit (high if the number is normal and low in subnormal's case) and the guard bits are added in this block. *Enable* signal enables this block (and therefore the entire *preadder* block) when we do not have a special case.

The outputs are the two numbers with the added bits and the *e_data* signal which distinguish between normal, subnormal and mixed numbers.




```
entity selector is
    Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
              NumberB : in std_logic_vector(31 downto 0);   -- Number B
              enable  : in std_logic;                       -- Enable
              e_data  : out std_logic_vector(1 downto 0);   -- Enable type data
              NA      : out std_logic_vector(36 downto 0);  -- Number A'
              NB      : out std_logic_vector(36 downto 0)); -- Number B'
end selector;
```

If *enable* signal is high it means we do not have a special case. Then the outputs signals *NA* and *NB* are made: first the sign and exponent bits are placed in its positions.

Next step, the implicit bit is fixed according with the exponent value. The mantissa and the guard bits are added too.

```
SA <= NumberA(31);
SB <= NumberB(31);
EA <= NumberA(30 downto 23);
EB <= NumberB(30 downto 23);
MA <= NumberA(22 downto 0);
MB <= NumberB(22 downto 0);

process (SA, SB, EA, EB, MA, MB, enable)
begin
  if enable = '1' then
    NA(36) <= SA;                -- Exponent & sign A
    NA(35 downto 28) <= EA;
    NB(36) <= SB;                -- Exponent & sign B
    NB(35 downto 28) <= EB;
    ------------------------------------ Mantissa A
    if (EA > X"00") then
      NA(27) <= '1';             -- Implicit bit
      NA(26 downto 4) <= MA;     -- Mantissa
      NA(3 downto 0) <= X"0";    -- Guard bits
    elsif EA = X"00" then
      NA(27) <= '0';             -- Implicit bit
      NA(26 downto 4) <= MA;     -- Mantissa
      NA(3 downto 0) <= X"0";    -- Guard bits
    else
      NA <= "-------------------------------------";
    end if;

   ------------------------------------ Mantissa B
   if (EB > X"00") then
     NB(27) <= '1';              -- Implicit bit
     NB(26 downto 4) <= MB;      -- Mantissa
     NB(3 downto 0) <= X"0";     -- Guard bits
   elsif EB = X"00" then
     NB(27) <= '0';              -- Implicit bit
     NB(26 downto 4) <= MB;      -- Mantissa
     NB(3 downto 0) <= X"0";     -- Guard bits
   else
     NB <= "-------------------------------------";
   end if;
 else
   NA <= "-------------------------------------";
   NB <= "-------------------------------------";
 end if;
end process;
```

If the exponent is bigger than 0 → Normal Number

Implicit bit → '1'

If the exponent is 0 → Subnormal Number

Implicit bit → '0'

Finally the *e_data* signal is fixed as follows:

1. Subnormal numbers → e_data := "00"
2. Normal numbers → e_data := "01"
3. Mixed numbers → e_data := "10"

```
e_data <= "00" when EA = X"00" and EB = X"00" and enable = '1' else          -- Subnormals
          "01" when EA > X"00" and EB > X"00" and enable = '1' else          -- Normals
          "10" when (EA = X"00" or EB = X"00") and enable = '1' else        -- Combination
          "--";
```

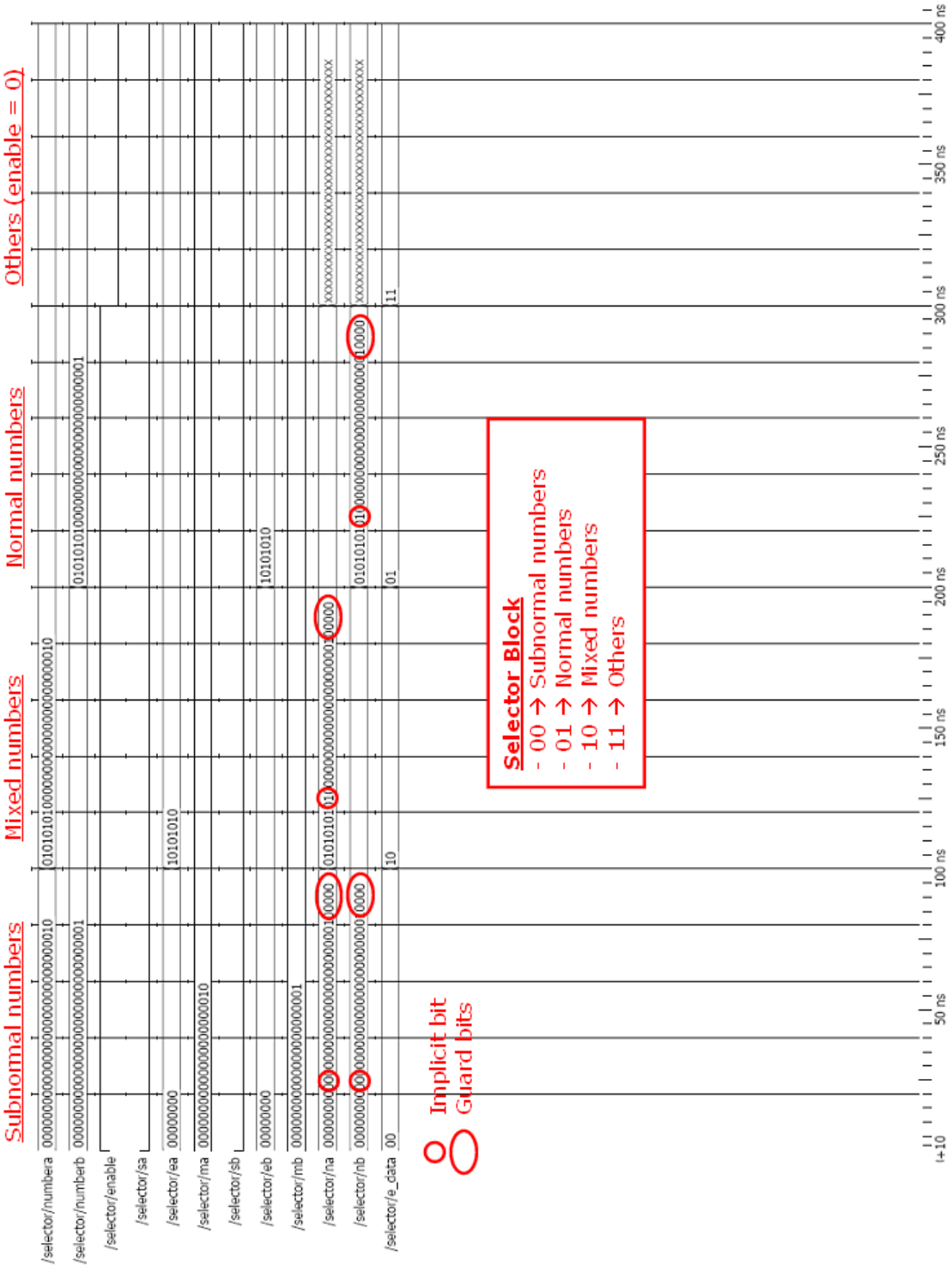
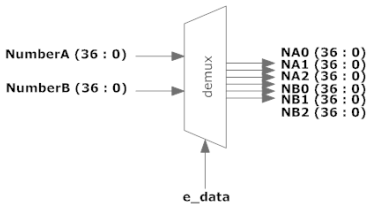



***Figure 22.*** *selector Simulation*

### *3.5.2. MUX/DEMUX Blocks*

The operation of the *demux* demultiplexor is routing the A and B numbers to the subnormal, normal or mixed block according with the *e_data* value.

*NumberA*, *NumberB* and the enable signal *e_data* are the entries and the outputs are 3 pairs of signals but only one pair is activated in each time. The typical demultiplexor's behaviour.

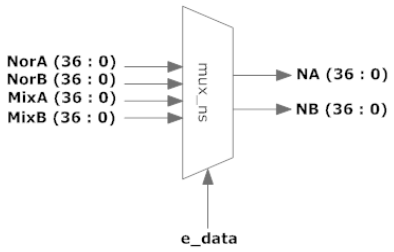



```
entity demux is
    Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
              NumberB : in std_logic_vector(36 downto 0);   -- Number B
              e_data  : in std_logic_vector(1 downto 0);    -- Enable type data
              NA0     : out std_logic_vector(36 downto 0);  -- Number A1
              NB0     : out std_logic_vector(36 downto 0);  -- Number B1
              NA1     : out std_logic_vector(36 downto 0);  -- Number A2
              NB1     : out std_logic_vector(36 downto 0);  -- Number B2
              NA2     : out std_logic_vector(36 downto 0);  -- Number A3
              NB2     : out std_logic_vector(36 downto 0)); -- Number B3
end demux;
```

```
process (NumberA, NumberB, e_data)
begin

  case e_data is
    ------------------------------------------- Subnormals
    when "00"    => NA0 <= NumberA;
                    NB0 <= NumberB;
                    NA1 <= "-----------------------------------";
                    NB1 <= "-----------------------------------";
                    NA2 <= "-----------------------------------";
                    NB2 <= "-----------------------------------";
    ------------------------------------------- Normals
    when "01"    => NA0 <= "-----------------------------------";
                    NB0 <= "-----------------------------------";
                    NA1 <= NumberA;
                    NB1 <= NumberB;
                    NA2 <= "-----------------------------------";
                    NB2 <= "-----------------------------------";
    ------------------------------------------- MiX
    when "10"    => NA0 <= "-----------------------------------";
                    NB0 <= "-----------------------------------";
                    NA1 <= "-----------------------------------";
                    NB1 <= "-----------------------------------";
                    NA2 <= NumberA;
                    NB2 <= NumberB;
    when others => NA0 <= "-----------------------------------";
                    NB0 <= "-----------------------------------";
                    NA1 <= "-----------------------------------";
                    NB1 <= "-----------------------------------";
                    NA2 <= "-----------------------------------";
                    NB2 <= "-----------------------------------";
  end case;
```

The *mux_ns* multiplexor's target is selecting which signal must be introduced in the normal numbers block: normal numbers or a standardized numbers from the mixed numbers block.

The entries are the two pairs of numbers and *e_data* signal and the outputs are the A and B numbers according with *e_data* value.

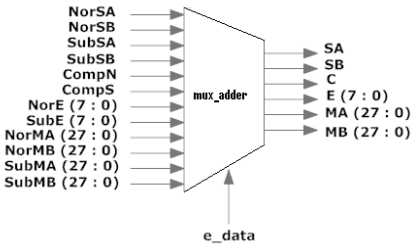



```
entity mux_ns is
    Port (    NorA    : in std_logic_vector(36 downto 0);   -- Normal A
              NorB    : in std_logic_vector(36 downto 0);   -- Normal B
              MixA    : in std_logic_vector(36 downto 0);   -- Mixed A
              MixB    : in std_logic_vector(36 downto 0);   -- Mixed B
              e_data  : in std_logic_vector(1 downto 0);    -- Enable type data
              NA      : out std_logic_vector(36 downto 0);  -- Number A'
              NB      : out std_logic_vector(36 downto 0)); -- Number B'
end mux_ns;
```

```
NA <= NorA when e_data = "01" else                  -- Normal numbers
      MixA when e_data = "10" else                  -- Mixed numbers
      "-----------------------------------";

NB <= NorB when e_data = "01" else                  -- Normal numbers
      MixB when e_data = "10" else                  -- Mixed numbers
      "-----------------------------------";
```

Finally *mux_adder* multiplexor is in charge of selecting which data are going to be introduced in the adder.

The entries are the *comp* signal, the two mantissas, signs and exponents and all of them multiplied by two: one for the subnormal numbers and another for the normal/mixed numbers. The output is one of the pair's members according with *e_data*.



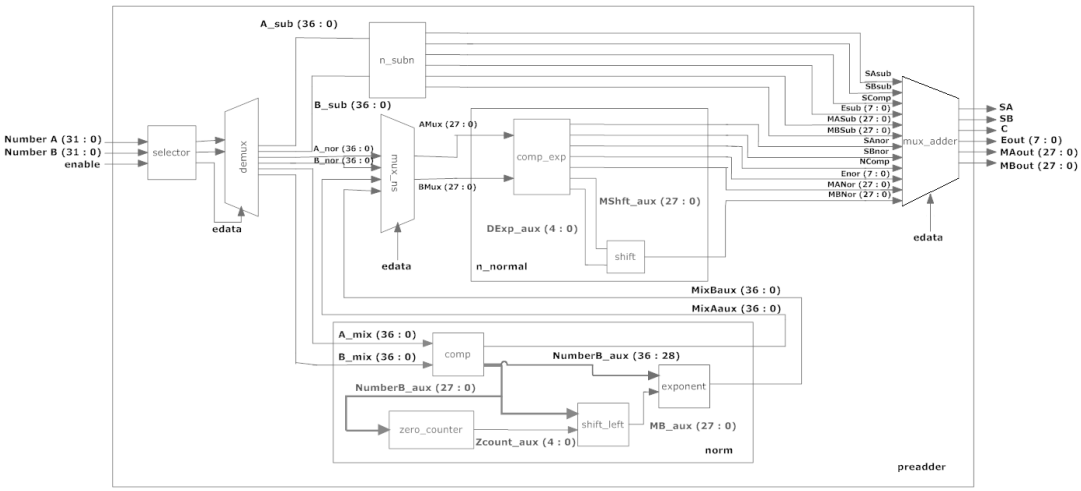

```
entity mux_adder is
    Port (    NorSA   : in std_logic;                       -- Sign A normal
              NorSB   : in std_logic;                       -- Sign B normal
              SubSA   : in std_logic;                       -- Sign A subnormal
              SubSB   : in std_logic;                       -- Sign B subnormal
              CompN   : in std_logic;                       -- Comparison Normal numbers
              CompS   : in std_logic;                       -- Compaison Sub numbers
              NorE    : in std_logic_vector(7 downto 0);    -- Exponent Output normal
              SubE    : in std_logic_vector(7 downto 0);    -- Exponent Output subnormal
              NorMA   : in std_logic_vector(27 downto 0);   -- Mantissa normal A
              NorMB   : in std_logic_vector(27 downto 0);   -- Mantissa normal B
              SubMA   : in std_logic_vector(27 downto 0);   -- Mantissa subnormal A
              SubMB   : in std_logic_vector(27 downto 0);   -- Mantissa subnormal B
              e_data  : in std_logic_vector(1 downto 0);    -- Enable type data
              SA      : out std_logic;                      -- Sign A
              SB      : out std_logic;                      -- Sign B
              C       : out std_logic;                      -- Comparison
              E       : out std_logic_vector(7 downto 0);   -- Output Exponent
              A       : out std_logic_vector(27 downto 0);  -- Mantissa A
              B       : out std_logic_vector(27 downto 0)); -- Mantissa B
end mux_adder;
```

```
A  <= NorMA when e_data = "01" or e_data = "10" else          -- Normal/Mix numbers
      SubMA when e_data = "00" else                           -- Subnormal numbers
      "----------------------------";

B  <= NorMB when e_data = "01" or e_data = "10" else          -- Normal/Mix numbers
      SubMB when e_data = "00" else                           -- Subnormal numbers
      "----------------------------";

C  <= CompN when e_data = "01" or e_data = "10" else          -- Normal/Mix numbers
      CompS when e_data = "00" else                           -- Subnormal numbers
      '-';

SA <= NorSA when e_data = "01" or e_data = "10" else          -- Normal/Mix sign A
      SubSA when e_data = "00" else                           -- Subnormal sign A
      '-';

SB <= NorSB when e_data = "01" or e_data = "10" else          -- Normal / Mix sign B
      SubSB when e_data = "00" else                           -- Subnormal sign B
      '-';

E  <= NorE when e_data = "01" or e_data = "10" else           -- Normal / Mix exponent
      SubE when e_data = "00" else                            -- Subnormal exponent
      "--------";
```

### *3.5.3. preadder Block*

Finally the *preadder* block is going to be explained. The special cases block is not considered in this block diagram because it will be added next to *adder* block in a complete block diagram.

*NumberA, NumberB* and *enable* are the inputs. A and B sign (*SA* and *SB*), the *C* signal, output's exponent *Eout*, and both *MAout* and *MBout* mantissas are the outputs of the design.




```
entity preadder is
    Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
              NumberB : in std_logic_vector(31 downto 0);   -- Number B
              enable  : in std_logic;                       -- Enable
              SA      : out std_logic;                      -- Sign A
              SB      : out std_logic;                      -- Sign B
              C       : out std_logic;                      -- Comparison
              EOut    : out std_logic_vector(7 downto 0);   -- Exponent Output
              MAOut   : out std_logic_vector(27 downto 0);  -- Greatest Mantissa
              MBOut   : out std_logic_vector(27 downto 0)); -- Shifted Mantissa
end preadder;
```

The components description is shown in this part of the code. Normal numbers block (*n_normal*), subnormal numbers block (*n_subn*), mixed numbers block (*norm*), the multiplexor and demultiplexors (*mux_ns, mux_adder* and *demux*) and the *selector* entity are added there.

```
-------------------------------------------------------------------------------- Normal Numbers
component n_normal port (NumberA, NumberB : in std_logic_vector(36 downto 0);
                         Comp             : out std_logic;
                         SA               : out std_logic;
                         SB               : out std_logic;
                         EO               : out std_logic_vector(7 downto 0);
                         MA, MB           : out std_logic_vector(27 downto 0));
end component;

-------------------------------------------------------------------------------- MUX/DEMUX
component mux_ns port (NorA, NorB, MixA, MixB : in std_logic_vector(36 downto 0);
                       e_data                 : in std_logic_vector(1 downto 0);
                       NA, NB                 : out std_logic_vector(36 downto 0));
end component;

component demux port (NumberA, NumberB             : in std_logic_vector(36 downto 0);
                      e_data                        : in std_logic_vector(1 downto 0);
                      NA0, NB0, NA1, NB1, NA2, NB2 : out std_logic_vector(36 downto 0));
end component;

component mux_adder port (NorSA, NorSB, SubSA, SubSB : in std_logic;
                          CompN, CompS               : in std_logic;
                          NorE, SubE                 : in std_logic_vector(7 downto 0);
                          NorMA, NorMB, SubMA, SubMB : in std_logic_vector(27 downto 0);
                          e_data                     : in std_logic_vector(1 downto 0);
                          SA, SB, C                  : out std_logic;
                          E                          : out std_logic_vector(7 downto 0);
                          A, B                       : out std_logic_vector(27 downto 0));
end component;

-------------------------------------------------------------------------------- Mixed Numbers
component norm port (NumberA, NumberB : in std_logic_vector(36 downto 0);
                     MA, MB           : out std_logic_vector(36 downto 0));
end component;

-------------------------------------------------------------------------------- Subnormal Numbers
component n_subn port (NumberA, NumberB : in std_logic_vector(36 downto 0);
                       Comp             : out std_logic;
                       SA               : out std_logic;
                       SB               : out std_logic;
                       EO               : out std_logic_vector(7 downto 0);
                       MA, MB           : out std_logic_vector(27 downto 0));
end component;

-------------------------------------------------------------------------------- Selector
component selector port (NumberA, NumberB : in std_logic_vector(31 downto 0);
                         enable           : in std_logic;
                         e_data           : out std_logic_vector(1 downto 0);
                         NA, NB           : out std_logic_vector(36 downto 0));
end component;
```

Finally the connection between the different components is described in the second part of the code.

```
comp0 : n_normal
      port map (NumberA => Amux, NumberB => Bmux,
                Comp => NComp, SA => SAnor, SB => SBnor, EO => Enor, MA => MAnor, MB => MBnor);

comp1 : n_subn
      port map (NumberA => A_sub, NumberB => B_sub,
                Comp => SComp, SA => SAsub, SB => SBsub, EO => Esub, MA => MAsub, MB => MBsub);

comp2 : norm
      port map (NumberA => A_mix, NumberB => B_mix,
                MA => MixAaux, MB => MixBaux);

comp3 : demux
      port map (NumberA => NA_out_select, NumberB => NB_out_select, e_data => edata,
                NA0 => A_sub, NB0 => B_sub, NA1 => A_nor, NB1 => B_nor, NA2 => A_mix, NB2 => B_mix);

comp4 : mux_ns
      port map (NorA => A_nor, NorB => B_nor, MixA => MixAaux, MixB => MixBaux, e_data => edata,
                NA => Amux, NB => Bmux);

comp5 : selector
      port map (NumberA => NumberA, NumberB => NumberB, enable => enable,
                e_data => edata, NA => NA_out_select, NB => NB_out_select);

comp6 : mux_adder
      port map (NorSA => SAnor, NorSB => SBnor, SubSA => SAsub, SubSB => SBsub, CompN => NComp, CompS => SComp,
                NorE => Enor, SubE => Esub, NorMA => MAnor, NorMB => MBnor, SubMA => MAsub, SubMB => MBsub,
                e_data => edata, SA => SA, SB => SB, C => C, E => Eout, A => MAout, B => MBout);
```

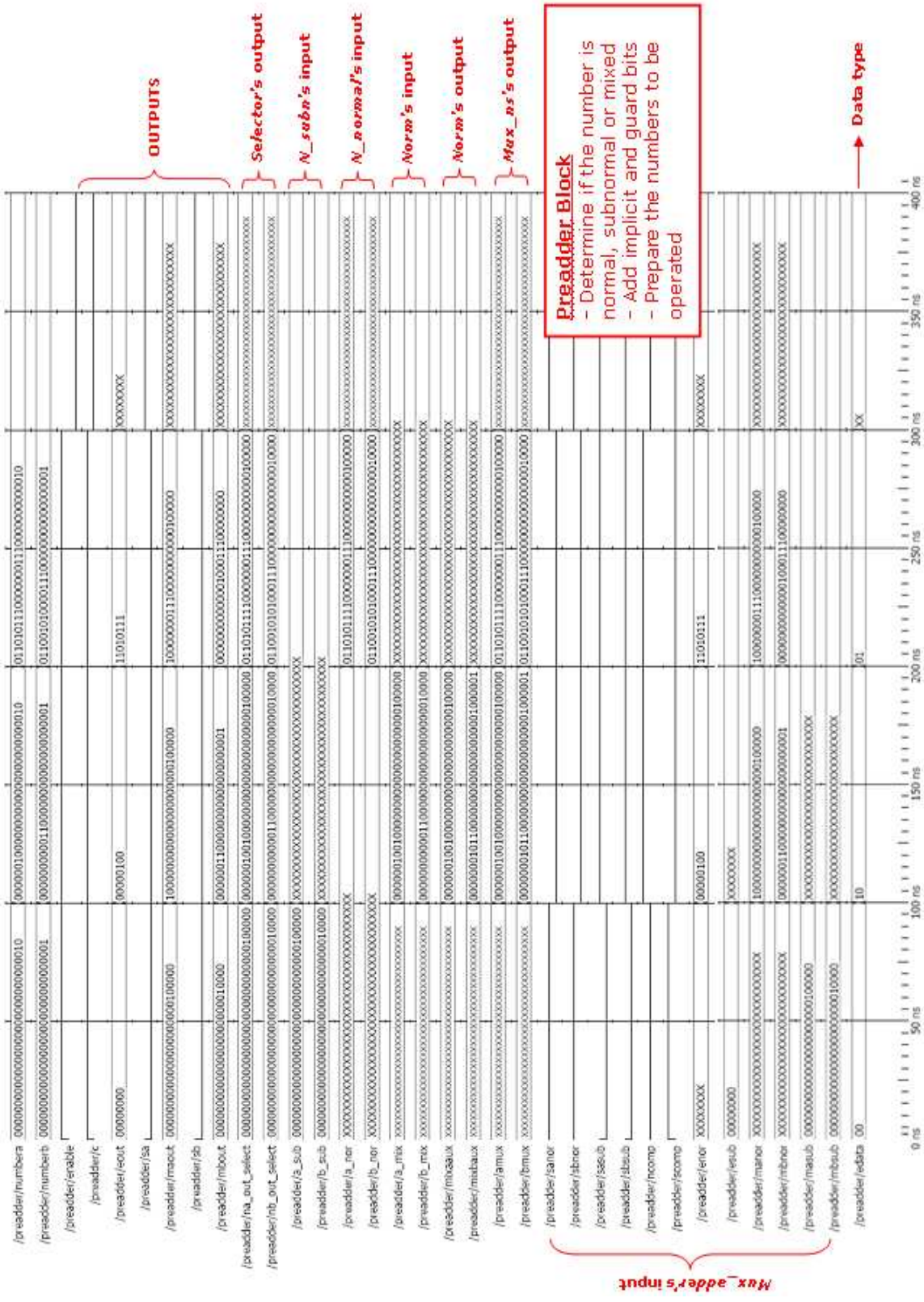



***Figure 23.*** *preadder Simulation*

# CHAPTER 4: ADDER

This chapter will deal with the adder and the standardizing block. The first one is in charge of operating the numbers which have been prepared in the Pre-adder block. The second one will standardize the result according with standard IEEE 754.

The block procedure is as follows:

1. Calculating the output's sign according to the sign numbers and the operation symbol
2. Addition/Subtraction of the both A and B numbers
3. Standardizing the result as IEEE 754 standard says
4. Grouping sign, exponent and mantissa in a single vector

The result is reached and the last step is multiplexing this value with the other one obtained as a special cases event explained in previous chapters.

## 4.1. Adder

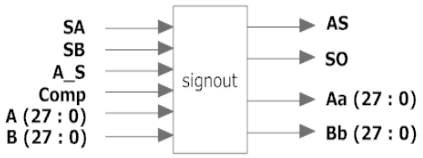
The adder is a fundamental piece of the design because it implements the addition/subtraction operation, main purpose of the 32 bit Floating Point Adder.

The Adder block is composed by two entities: *signout* and *adder*. *Signout* is responsible for the sign operation and the *adder* is the adder strictly speaking.

### *4.1.1. Signout Block*

*Signout* entity has six inputs: numbers *A* and B, both signs *SA* and *SB,* signal *A_S* which indicates if we add or subtract and the bit *Comp* (high if A is greater than B otherwise low).

The outputs are the two numbers *Aa* and *Bb*, the outputs sign *SO* and the signal *AS* that has the same function than *A_S*: determine the sign of the operation.




```
entity signout is
    Port (    SA     : in std_logic;                          -- Sign A
              SB     : in std_logic;                          -- Sign B
              A      : in std_logic_vector(27 downto 0);      -- Number A
              B      : in std_logic_vector(27 downto 0);      -- Number B
              A_S    : in std_logic;                          -- Add (0) or Sub (1)
              Comp   : in std_logic;                          -- Determine largest number
              Aa     : out std_logic_vector(27 downto 0);     -- Number A'
              Bb     : out std_logic_vector(27 downto 0);     -- Number B'
              AS     : out std_logic;                         -- A_S'
              SO     : out std_logic);                        -- Determine Output's sign
end signout;
```

Three different parts are visible in the code.

Firstly the outputs sign will be determined using the bits *A_S*, *Comp*, *SA* and *SB*, A's sign and B respectively.

```
SB_aux <= SB xor A_S;                    -- Sign B because of the operation

SO <= SA when Comp = '1' else            -- A > B --> Sign A
      SB_aux when Comp = '0' else        -- B > A --> Sign B
      '-';
```

An exclusive OR operand performs the function that is shown in table **4**.

***Table 4.*** *SB xor A_S*

| A_S | SB | SB_aux |
|---|---|---|
| + | + | + |
| + | - | - |
| - | + | - |
| - | - | + |

Basically, it does the mathematical combination between the operation's symbol and the B number sign. Once the "new" B's sign is found out, the outputs sign *SO* is determined with the aid of *SA* and the bit *Comp*.

***Table 5.*** *SO determined*

| SA | SB_aux | Comp | SO |
|---|---|---|---|
| + | + | 0 | + |
| + | + | 1 | + |
| + | - | 0 | - |
| + | - | 1 | + |
| - | + | 0 | + |
| - | + | 1 | - |
| - | - | 0 | - |
| - | - | 1 | - |

First of all, the two vectors A and B are reordered according with their original value (remember the numbers have been exchanged –or not- in *preadder* block when the exponents have been made equal)

When A is greater than B, the outputs sign *SO* will be equal to A's sign, *SA*. Otherwise, if B is greater than A, the output will keep the sign of the number B (the "new B's sign" one, *SB_aux*).

Secondly, if both *SA* and *SB* signs are equal is realized that both number A and number B will be added with the only difference of the sign. On the other hand, if *SA* and *SB* are different, what number is the negative one will be determined in order to simplify the adder implementation.

```
Aaux <= A when Comp = '1' else
        B when Comp = '0' else
        "----------------------------";

Baux <= B when Comp = '1' else
        A when Comp = '0' else
        "----------------------------";

process (SA, SB_aux, A, B)
  begin
  ------------------------ if Sign A is equal to Sign B
    if (SA xor SB_aux) = '0' then
      Aa <= A;                          -- Nothing changes
      Bb <= B;
  ------------------------ if Sign A is 1 and Sign B is 0
    elsif SA = '1' and SB_aux = '0' then
      Aa <= B;                          -- A is changed by B
      Bb <= A;

  ------------------------ if Sign A is 0 and Sign B is 1
    elsif SA = '0' and SB_aux = '1' then
      Aa <= A;                          -- Nothing changes
      Bb <= B;
    else
      Aa <= "----------------------------";
      Bb <= "----------------------------";
    end if;
end process;
```

As it is seen in the code, the negative number when we have two different signs always will be in the vector B called *Bb* setting the positive one in vector *Aa*. In another way it does not care: *Aa* will be *A* and *Bb* will be *B*.

Finally a bit indicating when a subtraction is produced is needed in order to achieve a properly operation in the adder. If A and B signs are equal that means an addition will be calculated (*AS* low). In the other hand, if A and B are different, the number A *Aa* and the negative number B, which had being moved to the vector *Bb,* are going to be subtracted (*AS* high).

```
AS <= '1' when SA /= SB_aux else        -- Complement to 1 is needed when
      '0';                              -- the signs are different
```

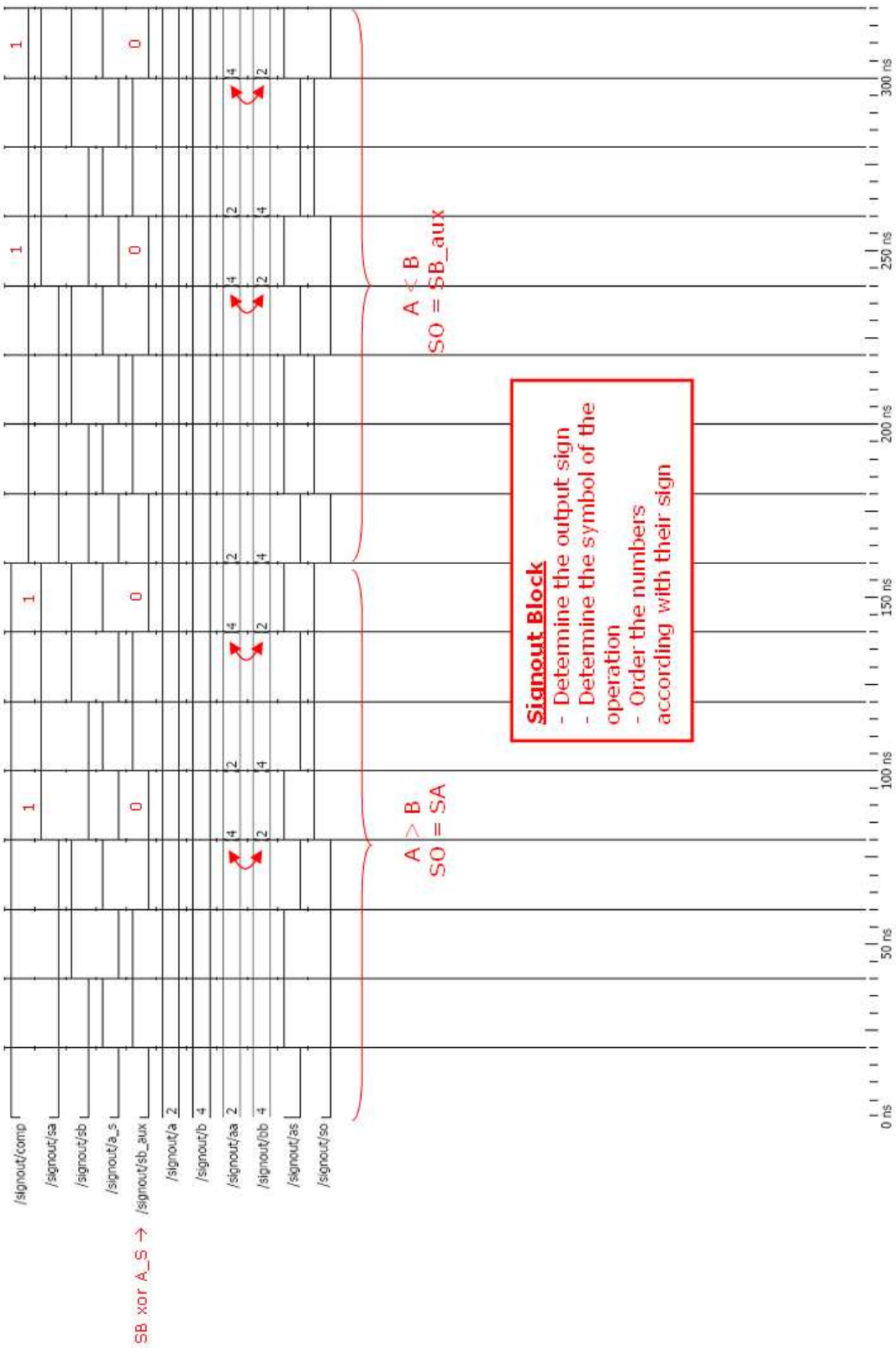
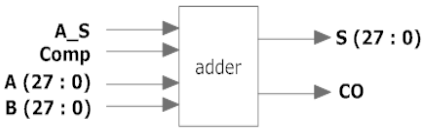



***Figure 24.*** *signout Simulation*

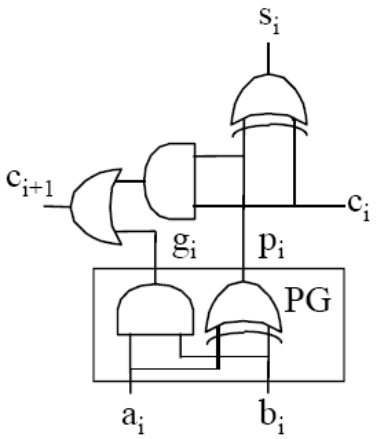
### 4.1.2. Adder Block

The *Adder* block is in charge of the addition/subtraction operation. The two numbers A and B and the bit *A_S* which indicates the operation's symbol are the entries. The outputs are the result vector *S* and the Carry bit *Co* which shows if there was an overflow.




```
entity adder is
    Port (    A     : in std_logic_vector(27 downto 0);     -- Number A
              B     : in std_logic_vector(27 downto 0);     -- Number B
              A_S   : in std_logic;                         -- Add(0) / Sub (1)
              S     : out std_logic_vector(27 downto 0);    -- Output
              Co    : out std_logic);                       -- Carry out
end adder;
```

First of all the type of adder will be explained. A Carry Look Ahead structure has been implemented. This structure allows a faster addition than other structures. It improves by reducing the time required to determine carry bits. This is achieved calculating the carry bits before the sum which reduces the wait time to calculate the result of the large value bits.




```
c_g <= A and B;                 -- Carry generation
c_p <= A xor B;                 -- Carry propagation

Cout <= c_g or (c_p and Cin);   -- Carry out
S <= c_p xor Cin;               -- Bit's sum
```

***Figure 25.*** *1-bit Carry Look Ahead Structure*

The implementation of the Carry Look Ahead structure is shown at the figure above. The idea is to obtain the carry generation and the carry propagation independently of each bit in order to obtain last carry faster.

The code has been designed implementing a 1-bit CLA structure and generating the other components up to 28 (the number of bits of the adder) by the function *generate*. Before that the *A_S* signal is used to determine if the second operand should be in complement to 1 (subtraction) or not (addition) using an exclusive or gate.

```
-- Components generation

Comp1: for i in 0 to 27 generate

  B1(i) <= B(i) xor A_S;

  sumador_0: if (i=0) generate
             sumador_0comp: CLA port map (A => A(i), B => B1(i), Cin => A_S, S => S(i), Cout => aux(i));
             end generate;
  sumador_i: if ((i>0) and (i<28)) generate
             sumador_icomp: CLA port map (A => A(i), B => B1(i), Cin => aux(i-1), S => S(i), Cout => aux(i));
             end generate;

end generate;

Co <= aux(27);
```

Finally the *Co* bit is fixed by the carry of the last component.

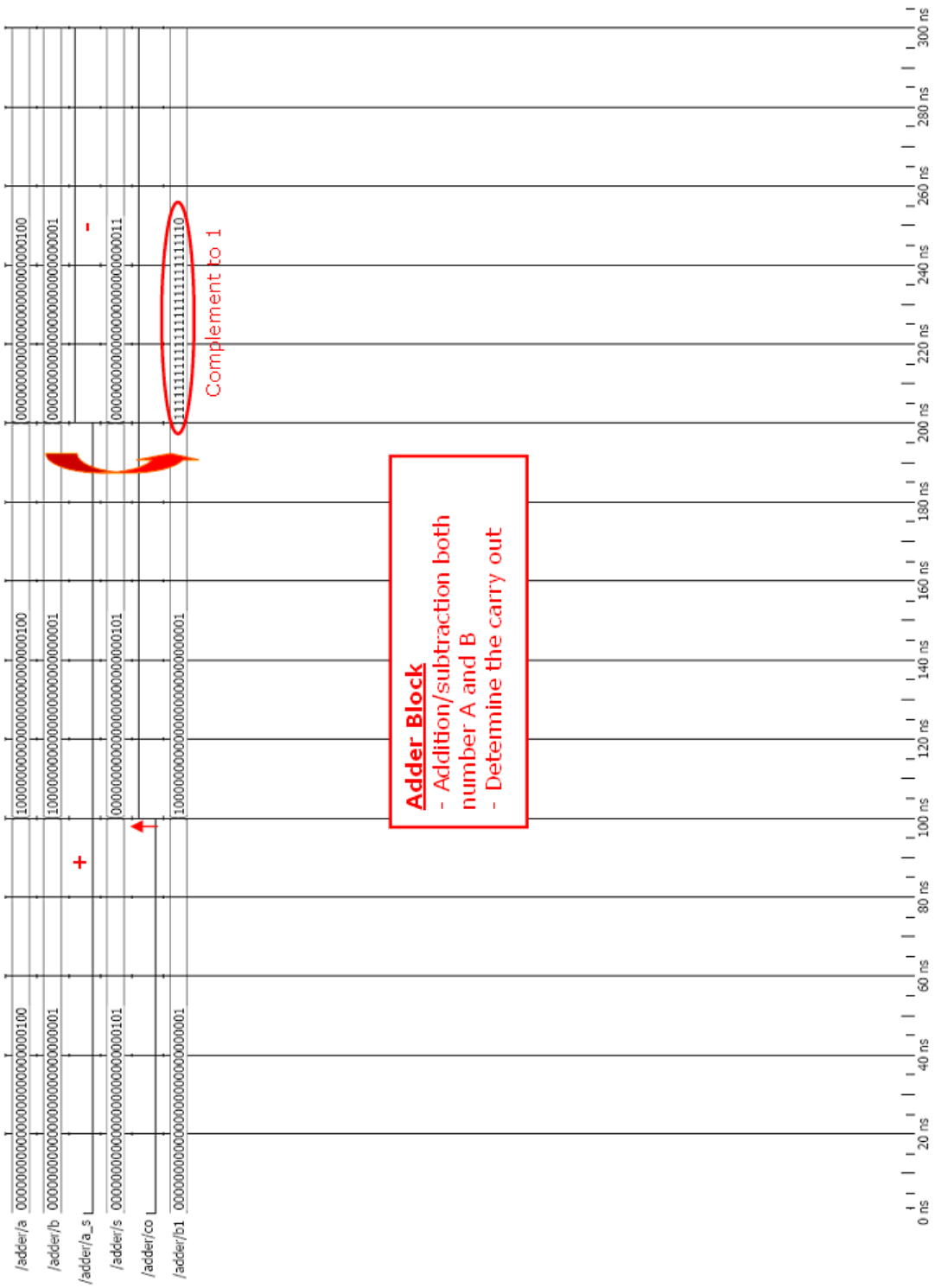
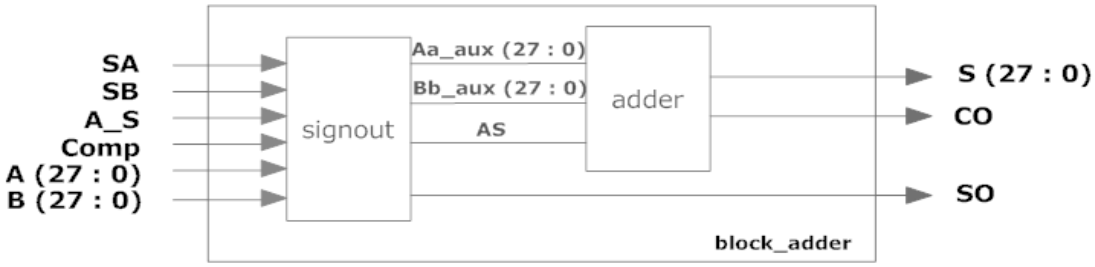



***Figure 26.*** *adder Simulation*

### 4.1.3. Block_Adder Block

Finally the *Block_Adder* block joins both *signout* and *adder* entities to implement the complete adder. The inputs are the signs and the value of A and B (*SA-SB* and *A-B* respectively), the bit *Comp* and *A_S*. Moreover the outputs are the result *S*, the carry *CO* and the outputs sign *SO.*




```
entity block_adder is
    Port (    SA    : in std_logic;                          -- Sign A
              SB    : in std_logic;                          -- Sign B
              A     : in std_logic_vector(27 downto 0);      -- Number A
              B     : in std_logic_vector(27 downto 0);      -- Number B
              A_S   : in std_logic;                          -- Add(0) / Sub (1)
              Comp  : in std_logic;                          -- Comparison
              S     : out std_logic_vector(27 downto 0);     -- Output
              SO    : out std_logic;                         -- Output's sign
              Co    : out std_logic);                        -- Carry out
end block_adder;
```

The code is quite brief. Basically the connection of the different blocks is done in this block.

```
component adder port (A, B : in std_logic_vector(27 downto 0);
                      A_S  : in std_logic;
                      S    : out std_logic_vector(27 downto 0);
                      Co   : out std_logic);
end component;


component signout port (SA, SB    : in std_logic;
                        A, B      : in std_logic_vector(27 downto 0);
                        A_S, Comp : in std_logic;
                        Aa, Bb    : out std_logic_vector(27 downto 0);
                        AS, SO    : out std_logic);
end component;

signal Aa_aux, Bb_aux, S_aux  : std_logic_vector(27 downto 0);
signal AS_aux, SO_aux, Co_aux : std_logic;

begin

component00: signout port map (SA => SA, SB => SB, A => A, B => B, A_S => A_S, Comp => Comp,
                               Aa => Aa_aux, Bb => Bb_aux, AS => AS_aux, SO => SO_aux);

component01: adder port map (A => Aa_aux, B => Bb_aux, A_S => AS_aux, S => S_aux, Co => Co_aux);

--------- If a complement to 1 is used and Output's sign is 1 a C2 is needed
S  <= (S_aux xor X"FFFFFFF")+'1' when ((AS_aux and SO_aux) = '1') else
      S_aux;

Co <= '0' when ((SB xor A_S) /= SA) else
      Co_aux;

SO <= SO_aux;
```

The most interesting part is on the bottom: if a subtraction operation is done and the outputs sign is set (that means negative number is greater than the positive) that means a complement to 2 is needed over the result because as it has been explained the negative number always is moved to the vector B and the result is "negative" (C to 2) when truly it is not. An example is shown:

***Table 6.*** *Example correct operation*

| SA | A | SB | B | A_s | SO | Result |
|---|---|---|---|---|---|---|
| - | 101 | + | 100 | + | - | 001 |

At the table **6** it is shown the correct and theoretical operation of the adder. The signs are not taken into account because they have their own bits. The result is 1-decimal positive with *SO* negative.

***Table 7.*** *Example wrong operation*

| Aa | Bb | AS | SO' | Result |
|---|---|---|---|---|
| 100 | 101 | - | - | 111 |

At table 7 the operation of the adder is shown without the complement to 2 part. The negative vector is move to Bb then a negative binary result (-1) is obtained not being correct according to the IEEE 754 standard. *AS* is recalculated in *signout* according to the sign values. Then if *AS* (subtraction) and *SO* (negative) are set a complement to 2 is necessary to reach a correct result.

Note the complement to 2 is not necessary when we have two negative numbers because it has been considered like an addition of two positive numbers.

Finally the carry value is also corrected in the same circumstances: when a subtraction is operated and we have a negative number at the output it will always have a carry out high. If the complement to 2 is needed that implies a carry low to obtain a proper result.

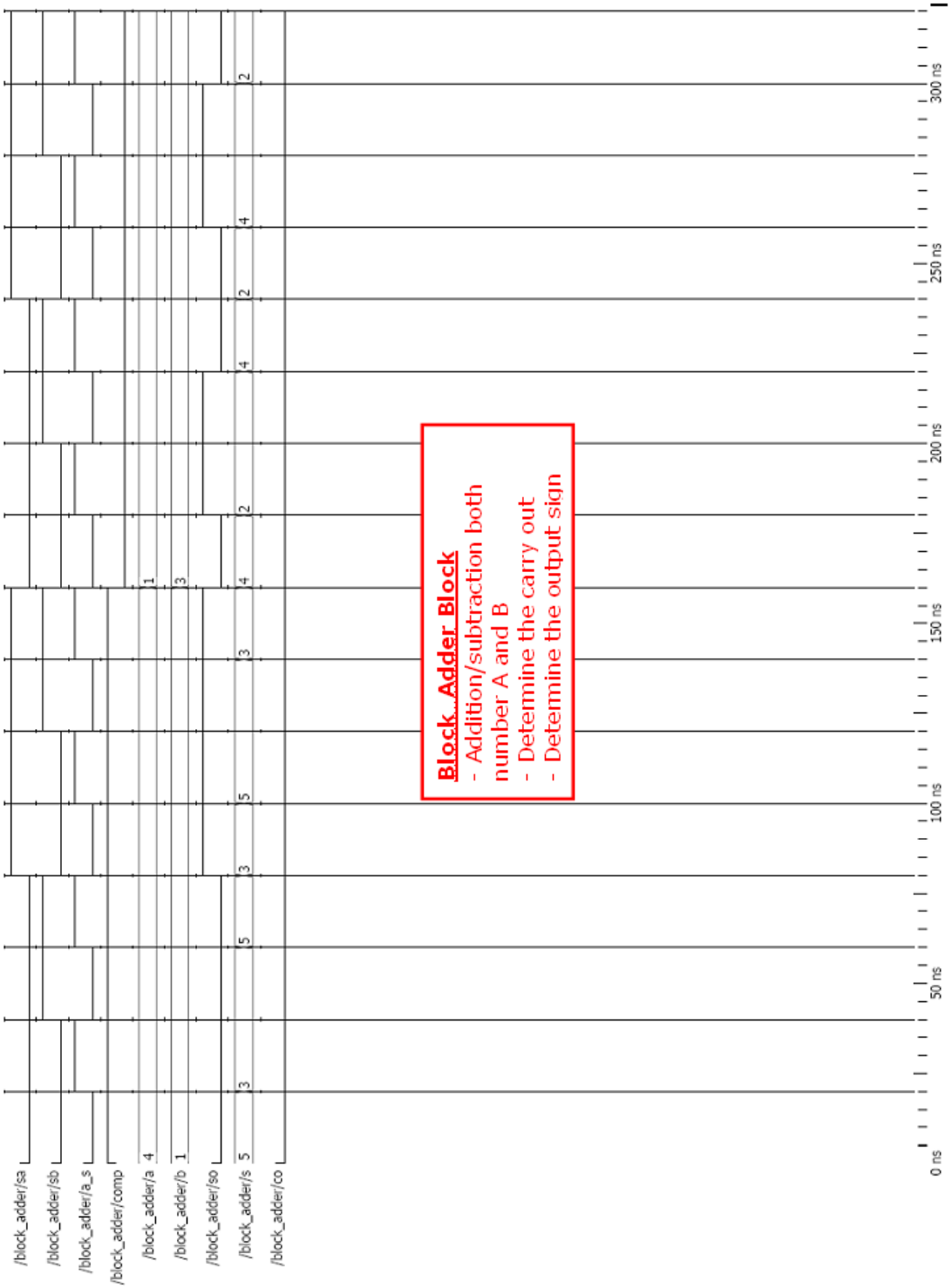



***Figure 27.*** *Block_Adder Simulation*

# 4.2. Standardizing Block

The Standardizing block, as its name suggest, is responsible for displaying the addition/subtraction operation value according to the IEEE 754 standard.

This block is composed of four entities. *Shift_left* and *zero* blocks have been explained in the previous chapter. *Round* and *vector* are the two new ones. Basically they are in charge of dealing with the result obtained from the adder and showing it in the same format as the numbers had been introduced

## *4.2.1. round Block*

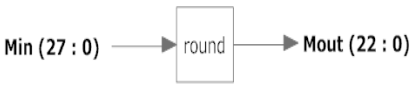
*Round* block provides more accuracy to the design. Four bits at the end of the vector had been added in the Pre-Adder block. Now it is time to use these bits in order to round the result.

$$1\,0\,0\,1\,0\,0\,0 \cdot 2^{4}$$
$$1\,1\,0\,1\,0\,0\,1 \cdot 2^{1}$$

$$1\,0\,0\,1\,0\,0\,0 \cdot 2^{5}$$
$$0\,0\,0\,0\,1\,1\,0\,1\,0\,0\,1 \cdot 2^{5}$$

Guard Bits

This block has only one input and one output. The input is the vector *Min* and the output *Mout*. Note *Min* is larger than *Mout* (27 bits against 22). The reason is *Min* contains the implicit and round bits that will be treated during the *round* code execution.




```
entity round is
    Port (    Min    : in std_logic_vector(27 downto 0);    -- Input's mantissa
              Mout   : out std_logic_vector(22 downto 0));  -- Output's mantissa
end round;
```

The process to round is chosen arbitrarily: if the round bits are greater than the value "1000" the value of the mantissa will be incremented by one. Otherwise the value keeps the same value.

```
process(Min)
  begin

    if Min(3 downto 0) = "----" then
      M_aux <= "-----------------------";
    elsif Min(3 downto 0) >= "1000" then     -- Round Mantissa
      M_aux <= Min(26 downto 4) + '1';
    else
      M_aux <= Min(26 downto 4);
    end if;
  end process;

  Mout <= M_aux;
```

## *4.2.2. shift_left/zero Block*

Both *shift_left* and *zero* blocks are completely reused from the mixed number block. It has been explained in the last chapter and it is not going to be commented again.

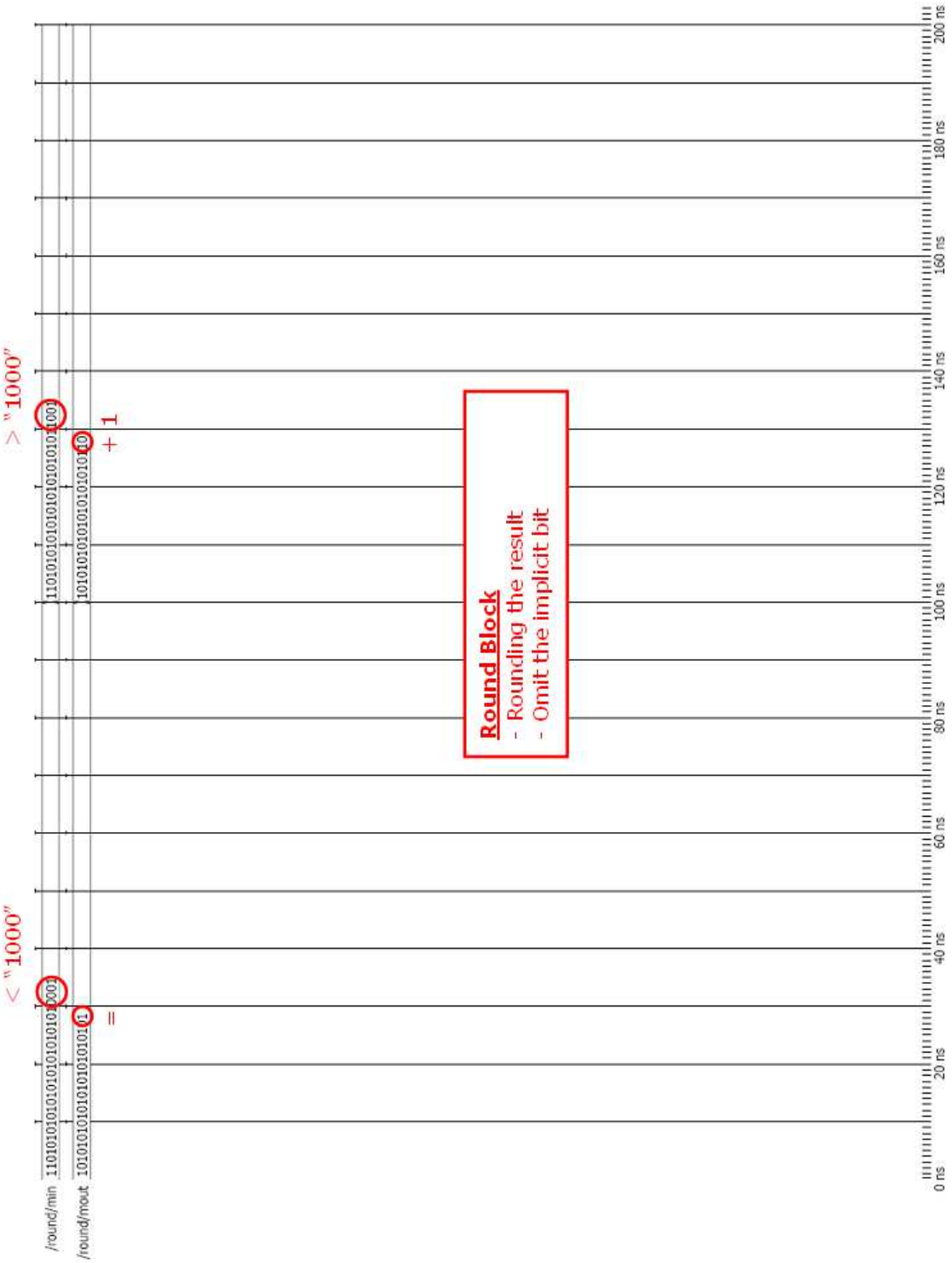
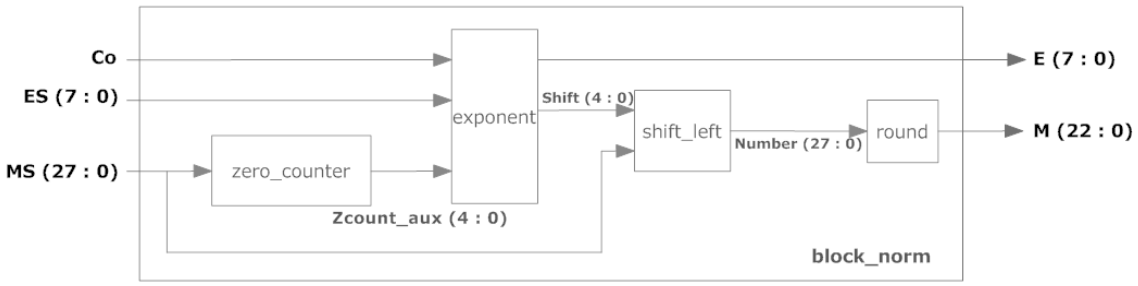



***Figure 28.*** *Round Simulation*

### 4.2.3. block_norm Block

The block_norm implement the standardizing function. It is composed by the entities *signout*, *shift_left* and *zero_counter*. The entries are the result's mantissa (*MS*) and exponent (*ES*) and the add's carry *Co*. The outputs will be the standardized result (its mantissa (*M*) and exponent (*E*)).




```
entity block_norm is
    Port (    MS : in std_logic_vector(27 downto 0);     -- Number S
              ES : in std_logic_vector(7 downto 0);      -- Exponent S
              Co : in std_logic;                         -- Carry out
              M  : out std_logic_vector(22 downto 0);    -- Output's Mantissa
              E  : out std_logic_vector(7 downto 0));    -- Output's Exponent
end block_norm;
```

The first part of the code implements the connection between the different components which compose the block. *Zero, shift_left* and *round* are connected as shown in the code and the block diagram.

```
------------------------------------------------ Components declaration
comp0 : zero
    port map (T => MS, Zcount => Zcount_aux);

comp1 : shift_left
    port map (T => MS, shft => Shift, S => Number);

comp2 : round
    port map (Min => Number, Mout => M);
```

The second part of the code refers to the exponent treatment. Three different cases have been taken into account:

1. If the exponent is larger than the number of zeros (number of positions the vector should be shifted) it means the number is normal and the standardized exponent will be the exponent minus the positions shifted plus the carry.
2. If the exponent is shorter than the number of zeros it means the output will be subnormal, only the value which marks the exponent could be shifted and the final exponent will be zero.
3. Last case referred when the exponent is equal to the number of zeros. On this occasion the vector will be shifted the number of positions the exponent marks (or the signal *zero_counter*) and the result will be normal with exponent one.

```
------------------------------------------------ Normal or Subnormal Number
                                                --      & New Wxponent
process(MS, ES, Shift, Zcount_aux, Co)
  begin

    if Zcount_aux = "-----" then
      Shift <= "-----";
      E <= "--------";
    elsif ES > Zcount_aux then          -- If the number is normal...
      Shift <= Zcount_aux;               -- ... the number is shifted --> Output normal
      E <= ES - Shift + Co;
    elsif ES < Zcount_aux then           -- If the number is normal...
      Shift <= ES(4 downto 0);           -- ... the number is shifted --> Output subnormal
      E <= X"00";
    elsif ES = Zcount_aux then           -- If N° Zeros = Exponent...
      Shift <= Zcount_aux;               -- ... the mantissa is shifted and EO = 1
      E <= X"01";
    end if;

  end process;
```

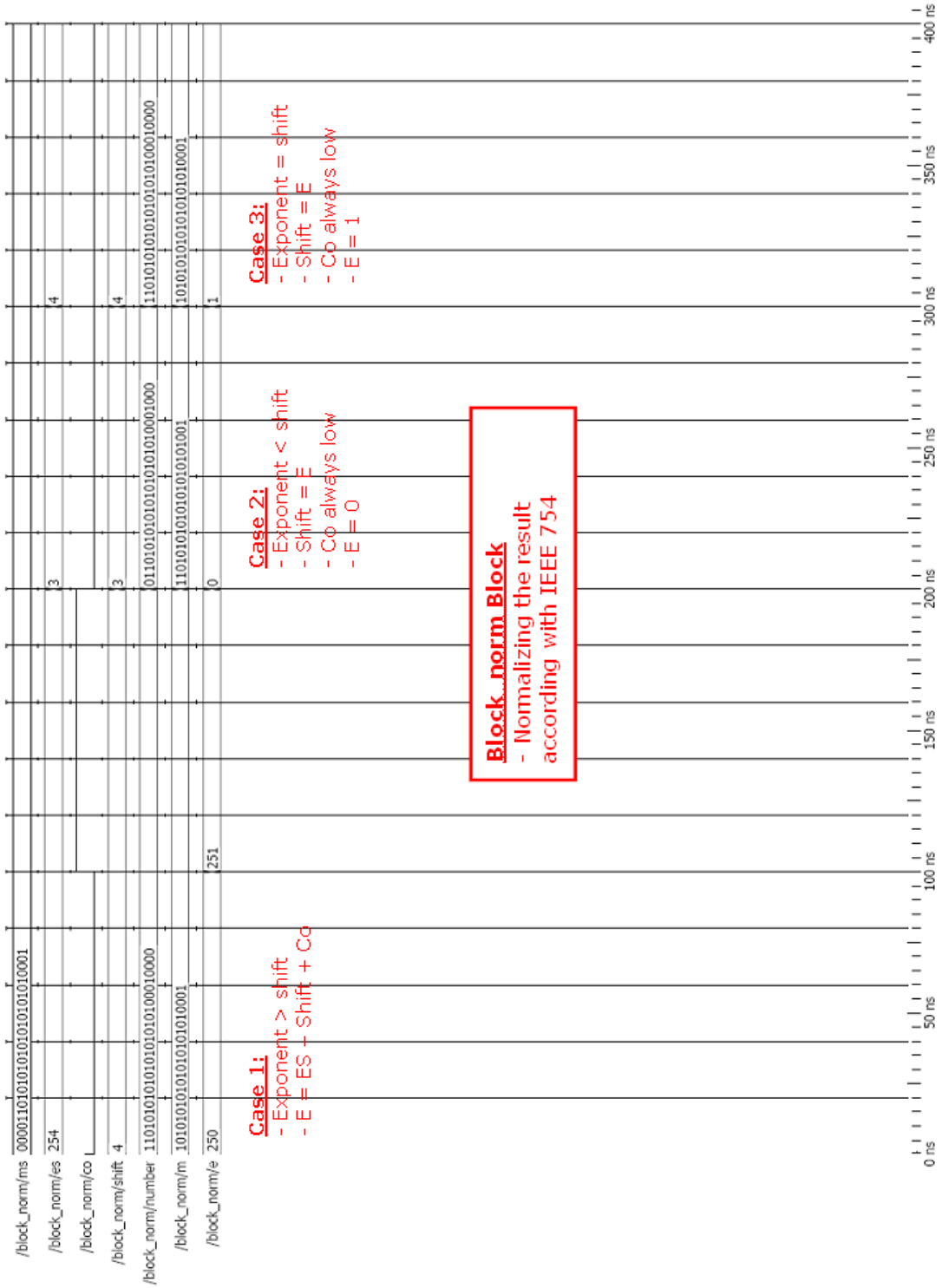
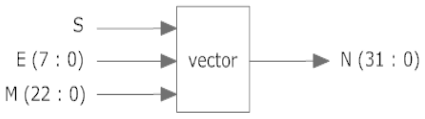



***Figure 29.*** *norm_block Simulation*

### 4.2.4. *vector Block*

This block has an easy function: regrouping the sign, exponent and mantissa in a single vector to be consistent with the format adopted for data entry.

It has three inputs: sign *S* which it comes from the adder block, mantissa *M* and exponent *E*. The output is the vector *N* which keeps the format of numbers A and B (the main entries of the system).




```
entity vector is
    Port (    S : in std_logic;                          -- Sign
              E : in std_logic_vector(7 downto 0);       -- Exponent
              M : in std_logic_vector(22 downto 0);      -- Mantissa
              N : out std_logic_vector(31 downto 0));    -- Vector
end vector;
```

The code is so simple. Sign, mantissa and exponent are set in the proper position as follows:

```
N(31) <= S;
N(30 downto 23) <= E;
N(22 downto 0) <= M;
```

***Table 8.*** *Bit positions*

| 31 | 30..23 | 22..0 |
|---|---|---|
| Sign | Exponent | Mantissa |

Note this entity will be out of the standardizing block because it uses signals from two different blocks. However it is part of the standardizing process and it is clearer to be explained in this chapter.

Two simulations are added: the first one test the entity operation and the second one perform the behaviour when it is joined to the *block_norm* entity.

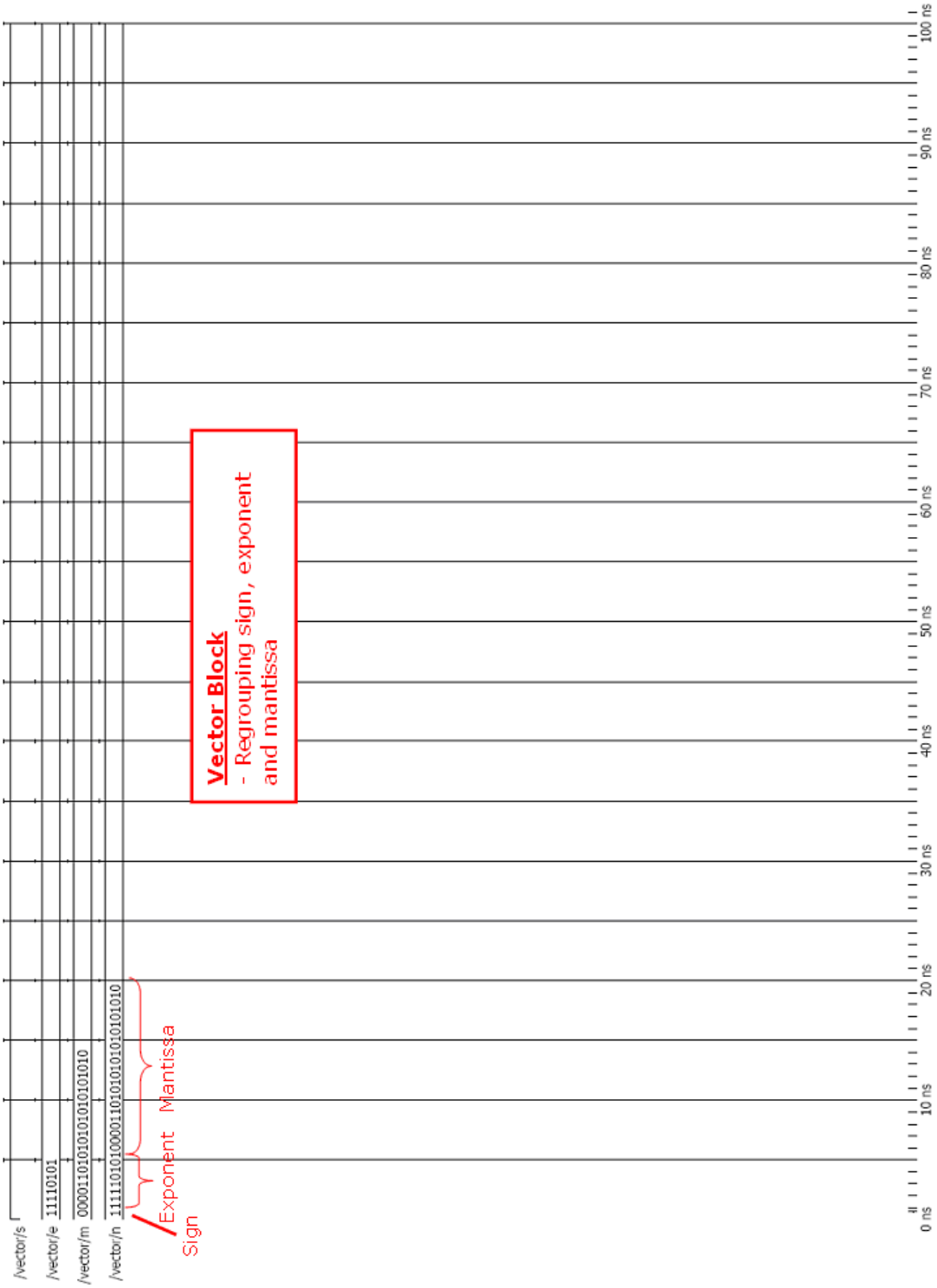



***Figure 30.*** *vector Simulation*

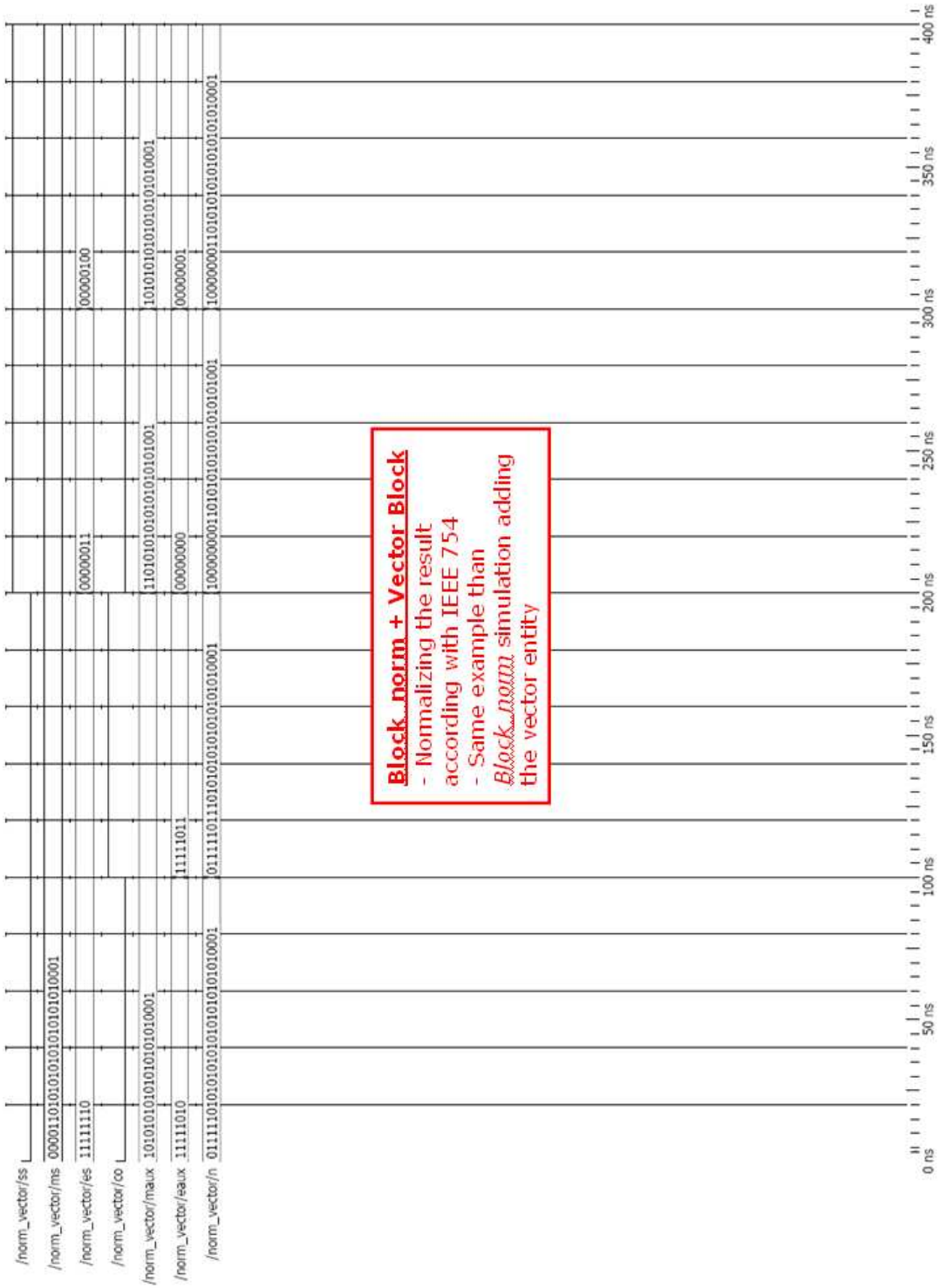



***Figure 31.*** *block_norm + vector Simulation*

# CHAPTER 5: 32-BITS FLOATING POINT ADDER

Finally all the different entities and sub-blocks has been described and explained. In this chapter the blocks will be joined in order to test totally the Floating Point Adder. The procedure will be as follows:

1. *n_case* Entity sort the data type according with the standard IEEE 754 and enable the adder if it is needed to operate the numbers. Otherwise (special cases) the result is done by it.
2. If the numbers are normal, subnormal or a mix, the Pre-Adder sub-block deals with the treatment of the numbers in order to be added / subtracted.
3. The Adder Block adds or subtracts the two numbers given
4. It is time to standardize the result according with the standard: shifting the mantissa and recalculating the new exponent.
5. Finally the result will be choose between the special case or the operated one depending on the input values

Two more entities will be explained in this section: the multiplexor which takes care of the last step of the list and the *fpadder* grouping all this points and making them work together.

On this occasion, the simulations will not be added to the code. Being the complete Floating Point Adder it is considered make another point in this chapter to demonstrate the proper functioning.

Moreover a table will be used to collect the different binary values, convert to decimal and as similar or different the results are.

# 5.1. Floating Point Adder

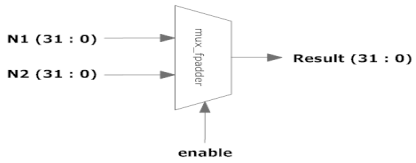
As it has been said, this first section will contain two entities: the multiplexor that is in charge of setting the correct result in the output (special cases or operated result) and the entity which groups all the blocks.

Continuing with the format used before, the ports and the block diagram are explained at the beginning and immediately afterwards the behaviour.

## *5.1.1. Mux_fpadder Block*

The multiplexor is not so complicated. It has three inputs: *N1*, *N2* and *enable*. The first signal refers to the *n_case* result and the second one to the vector obtained in the adder block. *Enable* decides which one will be at the output.

Finally, the output is *Result* which contains the 32 bits (Sign, exponent and mantissa) result.




```
entity mux_fpadder is
    Port (    N1     : in std_logic_vector(31 downto 0);   -- N_case number
              N2     : in std_logic_vector(31 downto 0);   -- Adder number
              enable : in std_logic;                       -- enable
              Result : out std_logic_vector(31 downto 0)); -- Result
end mux_fpadder;
```

The code is pretty simple. If *enable* is high it means the numbers were normal, subnormal or mixed and then the vector which comes from the adder is the correct result. Otherwise, if *enable* is low, a special case combination is had and the block *n_case* is who has the proper value.

```
    Result <= N1 when enable = '0' else                    -- N_case number
              N2 when enable = '1' else                    -- Adder number
          "--------------------------------";
```

## *5.1.2. fpadder Block*

Finally, the entire Floating Point Adder is designed. The last entity is *fpadder* which joins all the different blocks previously described.

The inputs are both *NumberA* and *NumberB* numbers and the operand *A_S*. Obviously, the output is the final result of the operation according with the standard.

```
entity fpadder is
    Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
              NumberB : in std_logic_vector(31 downto 0);   -- Number B
              A_S     : in std_logic;                       -- Add / Sub
              Result  : out std_logic_vector(31 downto 0)); -- Result
end fpadder;
```

On the next page a complete block diagram is shown:

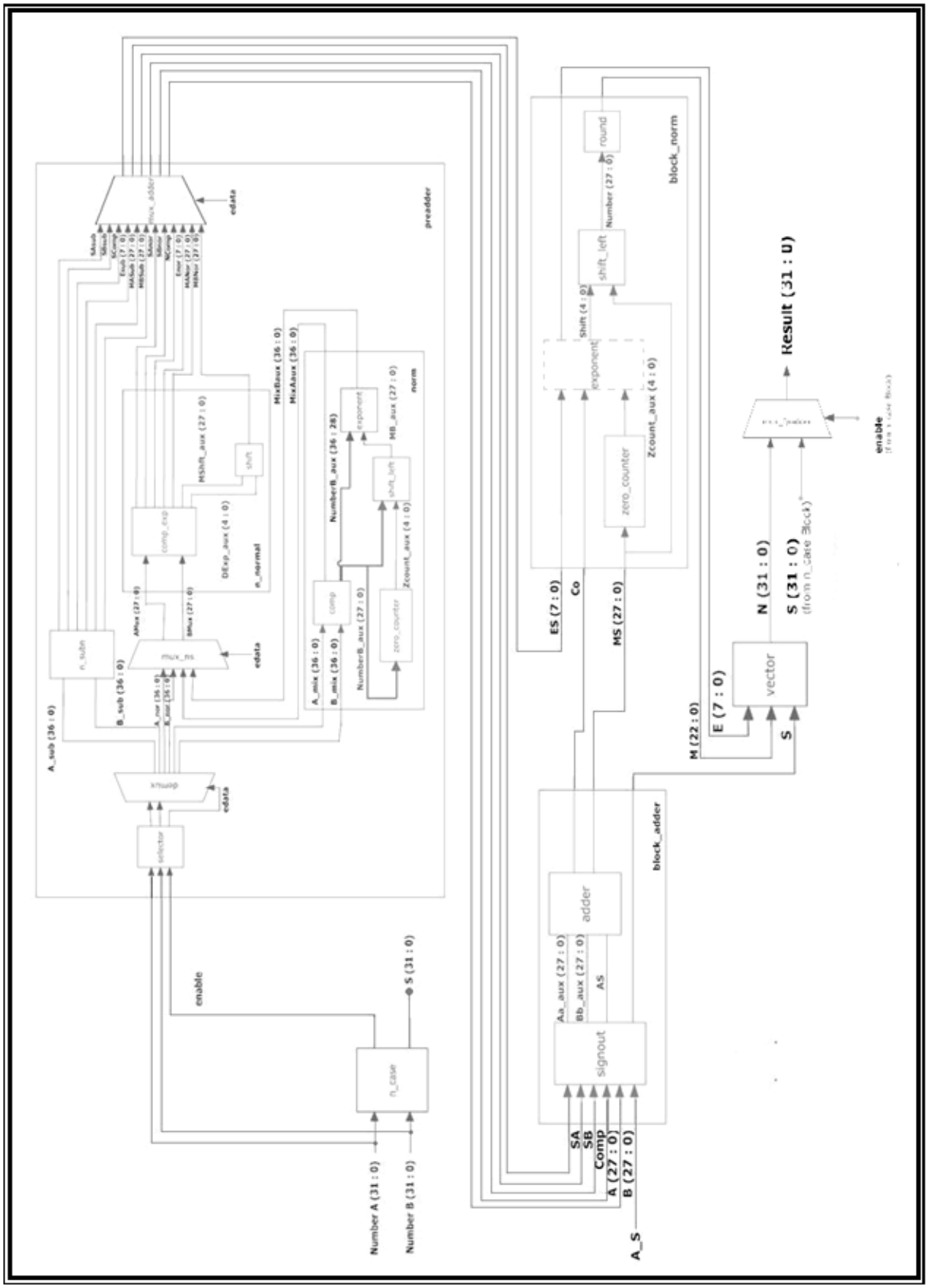



***Figure 32.*** *Block Diagram*

The first part of the code includes all the component declarations. As it can be seen all the main blocks are here: *n_case*, *preadder*, *block_adder*, *norm_vector* (*norm* + *vector* blocks) and *mux_fpadder*.

```
-------------------------------------------------------------------------------- n_case

component n_case port (NumberA, NumberB : in std_logic_vector(31 downto 0);
                       enable           : out std_logic;
                       S                : out std_logic_vector(31 downto 0));
end component;

-------------------------------------------------------------------------------- Pre-Adder

component preadder port (NumberA, NumberB : in std_logic_vector(31 downto 0);
                         enable           : in std_logic;
                         SA, SB           : out std_logic;
                         C                : out std_logic;
                         EOut             : out std_logic_vector(7 downto 0);
                         MAOut, MBOut     : out std_logic_vector(27 downto 0));
end component;

-------------------------------------------------------------------------------- Adder

component block_adder port (SA, SB : in std_logic;
                            A, B   : in std_logic_vector(27 downto 0);
                            A_S    : in std_logic;
                            Comp   : in std_logic;
                            S      : out std_logic_vector(27 downto 0);
                            SO     : out std_logic;
                            Co     : out std_logic);
end component;

component norm_vector port (SS : in std_logic;
                            MS : in std_logic_vector(27 downto 0);
                            ES : in std_logic_vector(7 downto 0);
                            Co : in std_logic;
                            N  : out std_logic_vector(31 downto 0));
end component;

-------------------------------------------------------------------------------- Mux_fpadder

component mux_fpadder port (N1, N2  : in std_logic_vector(31 downto 0);
                            enable  : in std_logic;
                            Result : out std_logic_vector(31 downto 0));
end component;
```

The second part of the code is responsible for connecting the different blocks properly as it is represented in the block diagram.

```
comp0 : preadder
      port map (NumberA => NumberA, NumberB => NumberB, enable => enable_aux,
                SA => SA_aux, SB => SB_aux, C => Comp_aux, EOut => EOut_aux, MAOut => MA_aux, MBOut => MB_aux);

comp1 : block_adder
      port map (SA => SA_aux, SB => SB_aux, A => MA_aux, B => MB_aux, A_S => A_S, Comp => Comp_aux,
                S => MOut_aux, SO => S_aux, Co => Carry);

comp2 : norm_vector
      port map (SS => S_aux, MS => MOut_aux, ES => EOut_aux, Co => Carry,
                N => Nadder);

comp3 : mux_fpadder
      port map (N1 => Ncase, N2 => Nadder, enable => enable_aux,
                Result => Result);

comp4 : n_case
      port map (NumberA => NumberA, NumberB => NumberB,
                enable => enable_aux, S => Ncase);
```

## 5.2. Simulations

At this point the simulations to test the operation will be comment. As it has been done before four different cases could happen: special case, normal, subnormal or mixed numbers.

All the different possibilities must be tested and this is the reason why the different data types will be treated separately.

The procedure will be as follows:

1. Enough different cases for each data type to demonstrate the correct working will be taken into account. The binary values of the entries and the output will be grouped in a table.
2. Using the simulation the result will be obtained and added to the table.
3. Decimal value of the numbers and the result will be calculated with the formula which had been explained at the standard IEEE 754 chapter.
4. Simulation value will be compared with the arithmetic value in order to see as similar or different the numbers will be.

The discussion about the accuracy of the 32-bit Floating Point Adder and the general standard IEEE 754 will be carried out in the next chapter.

### *5.2.1. Special Cases*

Recovering the table added in the second chapter, 8 different cases are possible.

*Zero-NumberX*, *NaN-NumberX* and *Normal/Subnormal-Infinity* cases can be taken into account only one time (*Zero-NumberX* or *NumberX-Zero* tests the same result). Then the simulation will contain 5 combinations.

***Table 9.*** *Special Cases combination*

| Sign | Out A | Out B | Sign Output | Output |
|---|---|---|---|---|
| X | Zero | Number B | SB | Number B |
| X | Number A | Zero | SA | Number A |
| X | Normal / Subnormal | Infinity | SB | Infinity |
| X | Infinity | Normal / Subnormal | SA | Infinity |
| SA=SB | Infinity | Infinity | SX | Infinity |
| SA≠SB | Infinity | Infinity | 1 | NaN |
| X | NaN | Number B | 1 | NaN |
| X | Number A | NaN | 1 | NaN |

**X:** do not care **SA:** Number A's sign **SB:** Number B's sign **SX:** Sign A or B (it is the same)

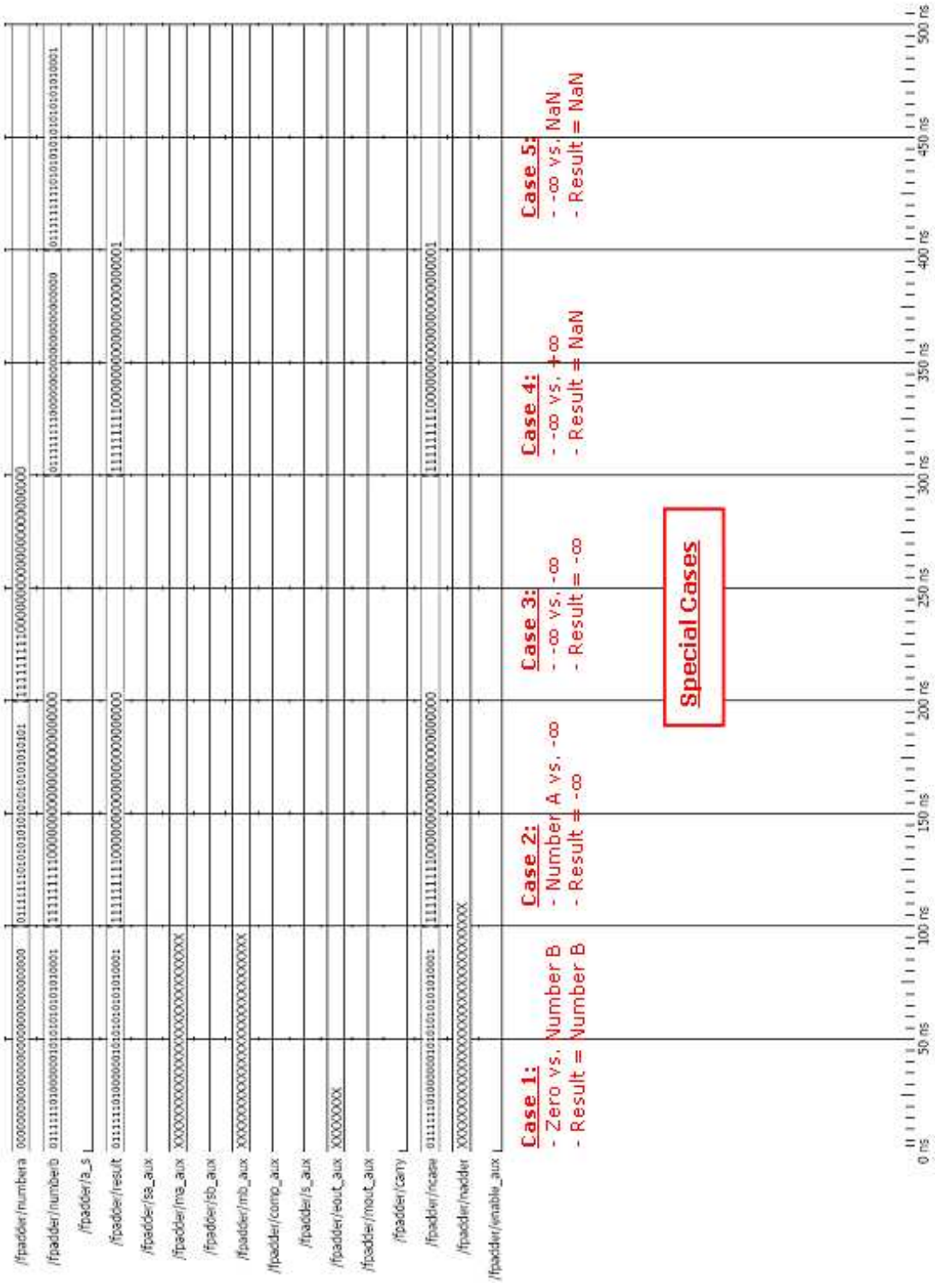



***Figure 33.*** *Special Cases Simulation*

The results are collected in the next table:

***Table 10.*** *Special Cases Results*

| SA | EA | MA | $A_{(10}$ | SB | EB | MB | $B_{(10}$ |
|---|---|---|---|---|---|---|---|
| 0 | 00000000 | 00000000000000000000000 | 0 | 0 | 11111101 | 10101010101010101010101 | +1.417843161699894e+038 |
| 0 | 11111110 | 10101010101010101010101 | +1.417843161699894e+038 | 1 | 11111111 | 00000000000000000000000 | -∞ |
| 1 | 11111111 | 00000000000000000000000 | -∞ | 1 | 11111111 | 00000000000000000000000 | -∞ |
| 1 | 11111111 | 00000000000000000000000 | -∞ | 0 | 11111111 | 00000000000000000000000 | +∞ |
| 1 | 11111111 | 00000000000000000000000 | -∞ | 0 | 11111111 | 10101010101010101010001 | NaN |

| SS | ES | MS | $S_{(10}$ | $A_{(10} \pm B_{(10}$ |
|---|---|---|---|---|
| 0 | 11111101 | 10101010101010101010101 | +1.417843161699894e+038 | --- |
| 1 | 11111111 | 00000000000000000000000 | -∞ | --- |
| 1 | 11111111 | 00000000000000000000000 | -∞ | --- |
| 1 | 11111111 | 00000000000000000000001 | NaN | --- |
| 1 | 11111111 | 00000000000000000000001 | NaN | --- |

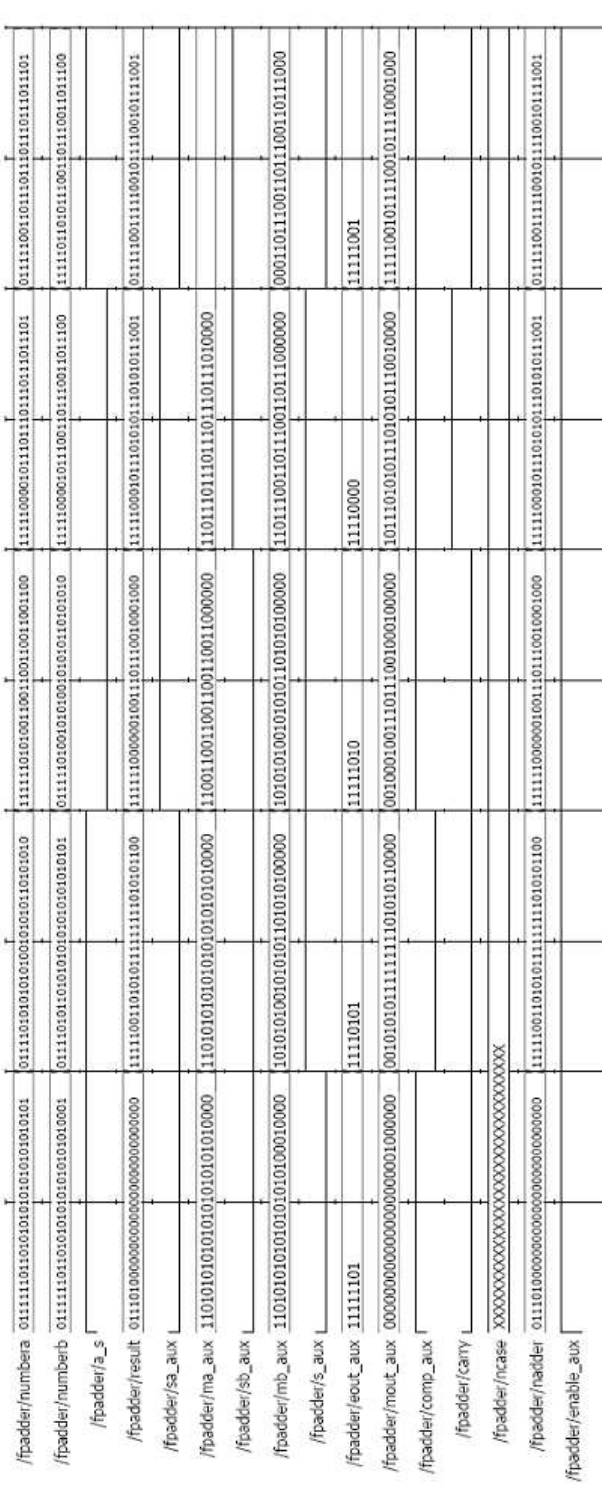
**SX:** Sign Number X **EX:** Exponent Number x **MX:** Mantissa Number X **XS:** Result **X(10:** Base-10 Number

As it can be seen the results are consistent with the theoretical explanation.

### 5.2.2. Normal Numbers

*n_case* block is only a combination between the entries and no more blocks are involved in this operation. Normal numbers have more complexity.

In this section the blocks which are responsible for normal numbers are tested.



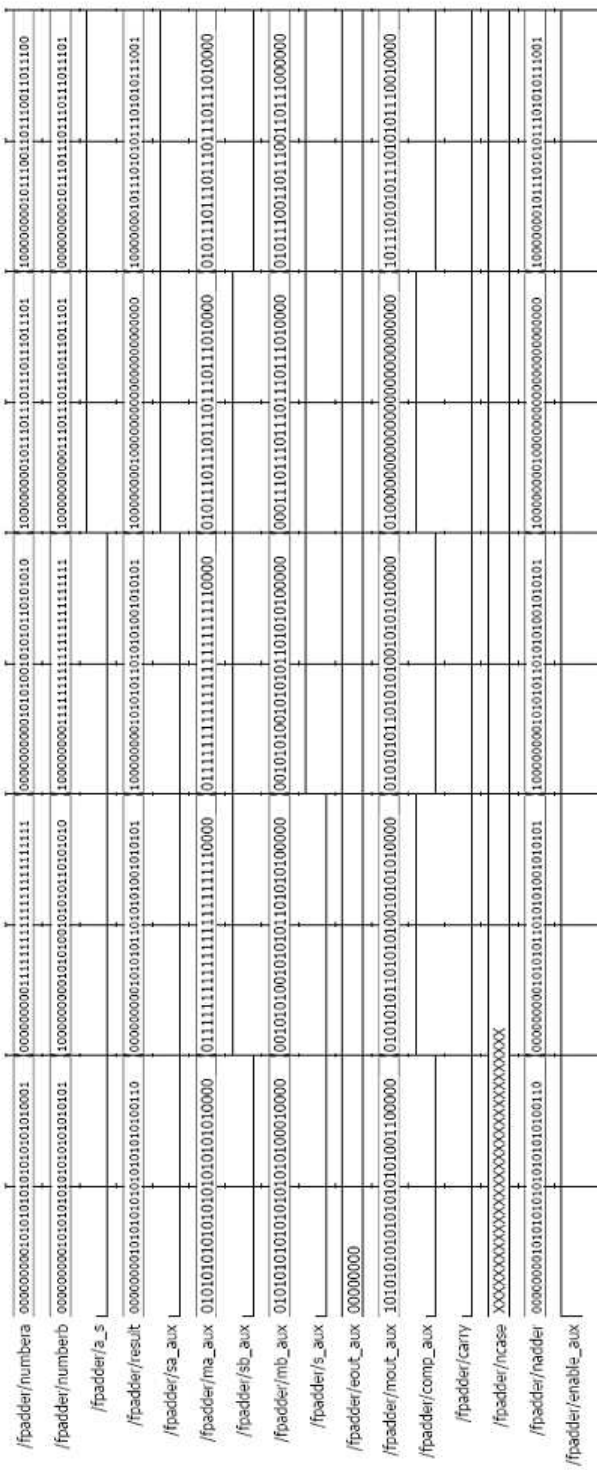

| SA | EA | MA | $A_{(10}$ | AS | SB | EB | MB | $B_{(10}$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 11111101 | 10100101010101010010101 | +1.417843161699894e+038 | - | 0 | 11111101 | 10100101010101010010101 | +1.417842756051702e+038 |
| 0 | 11110101 | 01010100101010110101010 | +4.422139697774293e+035 | - | 0 | 11110101 | 10100101010101010010101 | +5.538449850390212e+035 |
| 1 | 11111010 | 10011001100110011001100 | -1.701411733192644e+037 | + | 0 | 11111010 | 01010100101010110101010 | +1.415084703287774e+037 |
| 1 | 11110000 | 10111011101110111011101 | -1.799996137003937e+034 | + | 1 | 11110000 | 10111001101110011011100 | -1.791851358103467e+034 |
| 0 | 11111001 | 10110111011101110111001 | +9.215980221460157e+036 | - | 1 | 11110111 | 10111001101110011011100 | -2.293569738372437e+036 |

| SS | ES | MS | $S_{(10}$ | $A_{(10} \pm B_{(10}$ |
|---|---|---|---|---|
| 0 | 11101000 | 00000000000000000000000 | +4.056481920730334e+031 | +4.056481920730334e+031 |
| 1 | 11110011 | 01010111111111010101100 | -1.116310152615919e+035 | -1.116310152615919e+035 |
| 1 | 11111000 | 00010011101110010001000 | -2.863270299048707e+036 | -2.863270299048707e+036 |
| 1 | 11110001 | 01110101011101010111001 | -3.029857503386945e+034 | -3.591847495107403e+034 |
| 0 | 11111001 | 11110010111100101111001 | +1.036276540755903e+037 | +1.150954995983259e+037 |

***Figure 34.*** *Normal Numbers Simulation*

### 5.2.3. Subnormal Numbers

Turn to the subnormal numbers. Different possibilities with the sign of the numbers and the operation symbol will be treated in order to test more combinations.



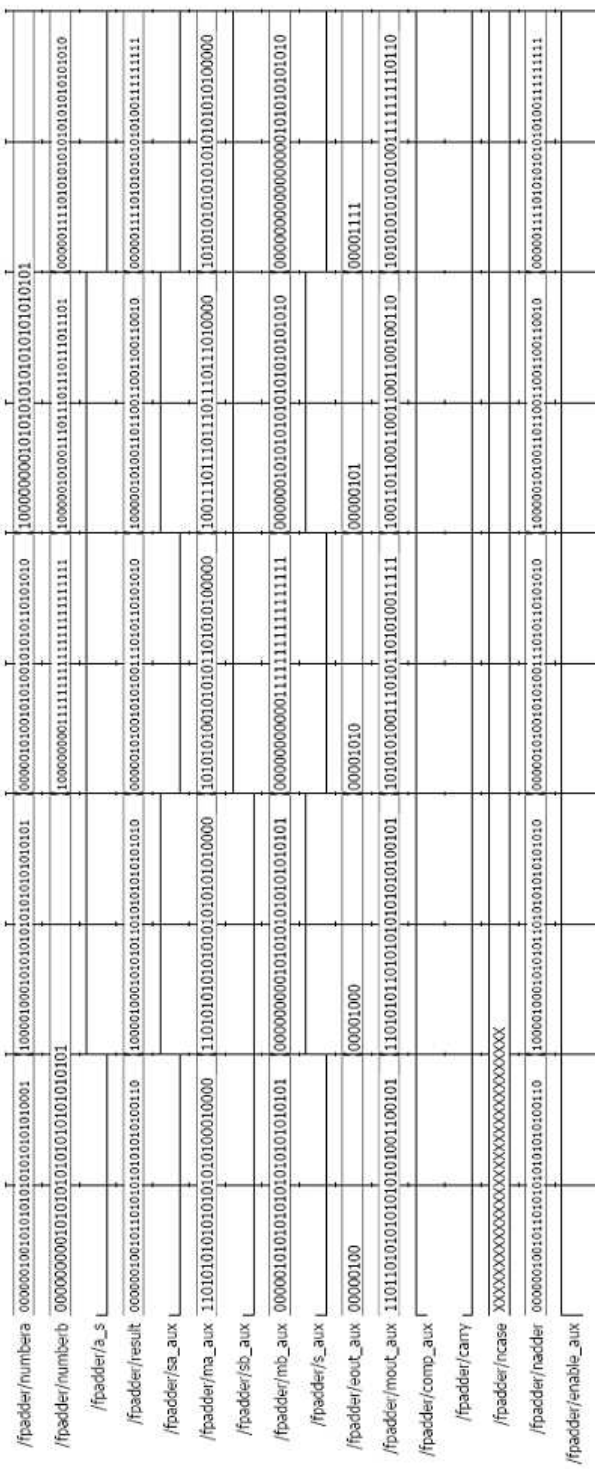

| SA | EA | MA | $A_{(10}$ | AS | SB | EB | MB | $B_{(10}$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 00000000 | 10101010101010101010001 | +7.836622933188571e-039 | + | 0 | 00000000 | 10101010101010101010101 | +7.836628538382429e-039 |
| 0 | 00000000 | 11111111111111111111111 | +1.175494210692441e-038 | + | 1 | 00000000 | 01010100010101011010010 | -3.887821313958274e-039 |
| 0 | 00000000 | 01010100101010110101010 | 3.887821313958274e-039 | + | 1 | 00000000 | 11111111111111111111111 | -1.175494210692441e-038 |
| 1 | 00000000 | 10111011101110110111101 | -8.620290691571439e-039 | - | 1 | 00000000 | 00111011101110110111101 | -2.742818937460002e-039 |
| 1 | 00000000 | 10111001101110011011100 | -8.528095061708117e-039 | - | 0 | 00000000 | 10111011101110111011101 | 8.620290691571439e-039 |

| SS | ES | MS | $S_{(10}$ | $A_{(10} \pm B_{(10}$ |
|---|---|---|---|---|
| 0 | 00000001 | 01010101010101010100110 | +1.567325147157100e-038 | +1.567325147157100e-038 |
| 0 | 00000000 | 10101011101010100010101 | +7.867120792966137e-039 | +7.867120792966137e-039 |
| 1 | 00000000 | 10101011101010100010101 | -7.867120792966137e-039 | -7.867120792966137e-039 |
| 1 | 00000000 | 10000000000000000000000 | -5.877471754111438e-039 | -5.877471754111438e-039 |
| 1 | 00000001 | 01111010101110101010111 | -1.714838575327956e-038 | -1.714838575327956e-038 |

***Figure 35.*** *Subnormal Numbers Simulation*

### *5.2.4. Mixed Numbers*

Finally, the mixed numbers. The other combinations will be tried.




| SA | EA | MA | $A_{(10}$ | AS | SB | EB | MB | $B_{(10}$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 00000100 | 10101010101010101010100001 | +1.56732531531291 6e-037 | + | 0 | 00000000 | 10101010101010101010101 | +7.83662853838242 9e-039 |
| 1 | 00001000 | 10101010101010101010101 | -2.50772122196547 9e-036 | - | 0 | 00000000 | 10101010101010101010101 | +7.83662853838242 9e-039 |
| 0 | 00001010 | 01010100101010110101010 | +8.00909558899567 48e-036 | - | 1 | 00000000 | 11111111111111111111111 | -1.17549421069244 1e-038 |
| 1 | 00000000 | 10101010101010101010101 | -7.83662853838242 9e-039 | - | 1 | 00000101 | 00111011101110111011101 | -2.31964199130926 0e-037 |
| 1 | 00000000 | 10101010101010101010101 | -7.83662853838242 9e-039 | + | 0 | 00001111 | 01010101010101010101010 | +2.56790643945715 4e-034 |

| SS | ES | MS | $S_{(10}$ | $A_{(10} \pm B_{(10}$ |
|---|---|---|---|---|
| 0 | 00000100 | 10110101010101010100110 | +1.60650842297 2366e-037 | +1.64569160069674 0e-037 |
| 1 | 00001000 | 10101011101010110101010 | -2.51163947667 9485e-036 | -2.51555785050386 1e-036 |
| 0 | 00001010 | 01010100111010110101010 | +8.01497306071 0860e-036 | +7.99734064684982 4e-036 |
| 1 | 00000101 | 00110110011001100110010 | -2.28045877154 5933e-037 | -2.39800827669308 5e-037 |
| 0 | 00001111 | 01010101010100111111111 | +2.56786717978 2547e-034 | +2.56782807317 1770e-034 |

***Figure 36.*** *Mixed Numbers Simulation*

# CHAPTER 6: RESULTS

Finally in the last chapter the results, they have been obtained before, will be evaluated.

Firstly a theorical and brief introduction about floating point errors is compulsory because this information is important to understand the behaviour of the results achieved.

At last, the report will finished with a conclusion where the main goals of the adder will be discussed.

## 6.1. Errors

There is a lot of literature which speaks about errors in a floating point system. The most of these errors are produced in the conversion between the internal binary format and the external decimal one or conversely.

Usually the computers use a fixed quantity of memory to represent each sort of number. This representation makes the electronic design easier but it involves rounding and it can lead to erroneous values.

This project focuses on the design of the binary floating point adder. Hence this type of errors will not be taken into account unless when a decimal representation with MATLAB is used (we will see it later).

The floating point format is discontinuous. It means not all real numbers have representation and this is another error source especially important with high numbers where the gap between them is largest.

### *6.1.1. Gap between Numbers*

Once again, the real numbers could have an infinite number of digits and the floating point format is used to represent it with a computer.

The accuracy of the number is represented by the number of digits of the mantissa. A 24bits mantissa could be represented by 7 decimal digits.

In numerical analysis, errors are very often expressed in terms of relative errors. And yet, when errors of "nearly atomic" function are expressed, it is more adequate and accurate to express errors in terms of the last bit of the significant: the last significant weight give us the precision of the system. Let us define that notion more precisely. William Krahan coined the term *ulp* (unit in the last place) in 1960 and its definition was as follows:

*Ulp(x) is the gap between the two floating point numbers nearest to x, even if x is one of them.*

Mathematically the *ulp* could be defined as follows:

$$ulp = \beta^{-p+e} \tag{2}$$

The value in our system will be (when $e=e_{min}$) $ulp = 2^{-24+1} = 1.1921 \cdot 10^{-7}$

As it has been said, the floating point format is discontinuous that means not all the real numbers have a representation in this format. The *ulp* represents the step between two consecutive numbers. Using *MATLAB* with $p$=6, β=2 and $0<e<3$ (simplifying results) a representation of this discontinuous format has been obtained:

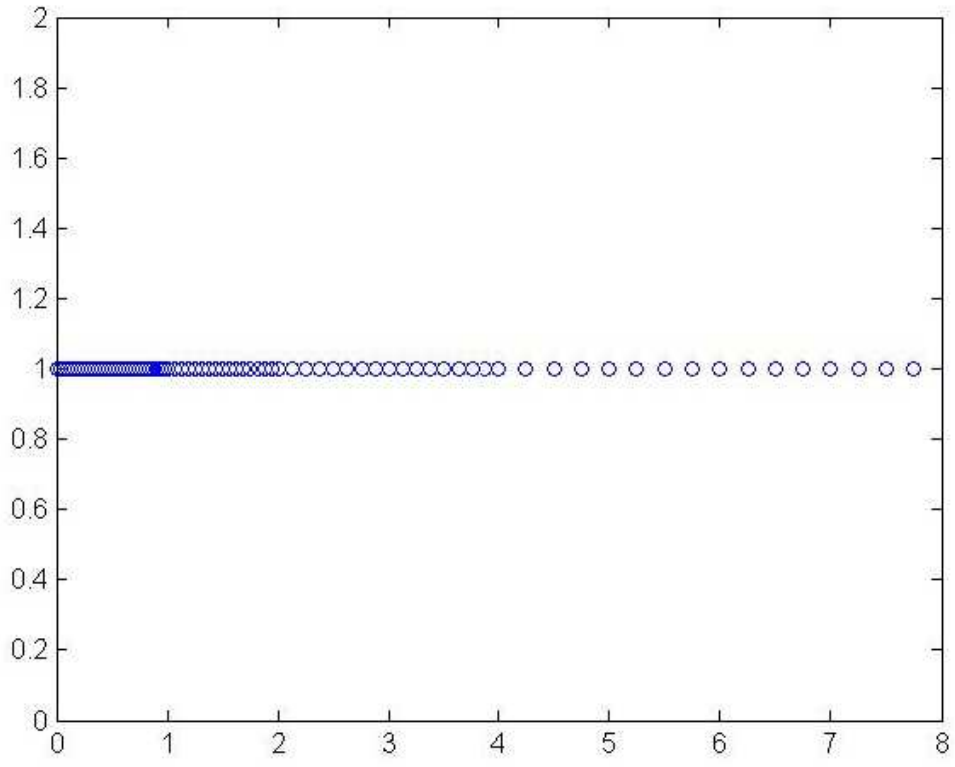



***Figure 37.*** *ulp representation*

The *ulp* is doubled as the exponent increases by one because of the base (power of two).

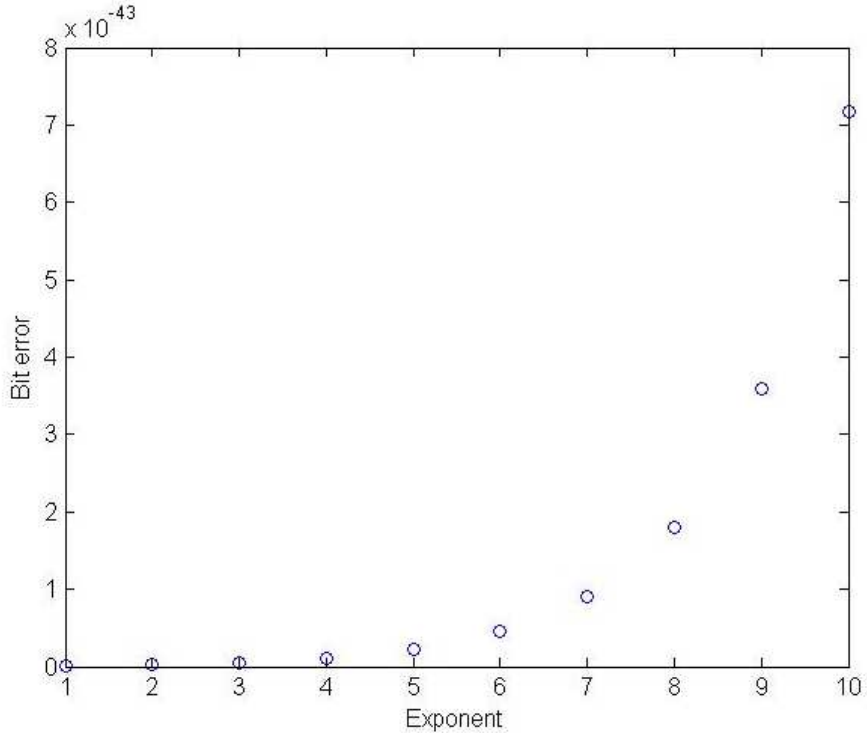



***Figure 38.*** *ulp increase*

### *6.1.2. Rounding or Truncation*

To represent a real number in a computer an adequate floating point number must be chosen. At first glance it seems the nearest one will be the best choice but sometimes this will not be an option unless all the number digits are known.

We need a procedure which requires only one digit more to represent the number. Using this condition rounding and truncation are our two possibilities.

Truncation consists of chosen the *m* more significant bits/digits of a number *x*. On the other hand, rounding needs to know the next bit/digit too and according with it add one to the LSB/last digit (5, 6, 7, 8 and 9 cases) or not (1, 2, 3, 4).

We use an example to find out what method is better and why.

Rounding (2/3) = (0.666….) → 0.667

Truncation 2/3) = (0.666….) → 0.666

If a mantissa of 3 bits is considered we have a maximum truncation error when the number ends with a periodic 9 as 0.666**99999**. The error is $0.999 \cdot 10^{-4} \approx 10^{-4}$ which matches up with the bit error (or the *ulp*).

The maximum rounding error is produced when the numbers have a decimal part ended in 4 plus infinity 9 digits as 0.666**49999.** The error is calculated rounding the maximum error $4.9999 \cdot 10^{-4} \approx 5 \cdot 10^{-4}$. Then the relative error is equal to half the last digit (½*ulp*).

In conclusion we have demonstrated the relative error caused by truncation is equal to the *ulp* and the rounding one ½*ulp*.

As it has said the error increases according with the exponent of the number.

### 6.1.3. Floating Point Addition

Finally a brief comment about the floating point addition/subtraction will be done.

The basic arithmetic operations have their equivalent in floating point format. The goal is noting these operations always have errors.

Let see an example:

Being x=1867=0.1867·$10^4$ and y=0.32=0.3200·$10^0$, then:

$$fl(x+y) = fl(0.1867 \cdot 10^4 + 0.000032 \cdot 10^4) = 0.1867 \cdot 10^4 \neq 1867.32 = x + y$$

The floating point representation is correct but anyway we have an error. A mantissa shifting is necessary to add two numbers with different exponents and we can lose some "information" during the procedure.

## 6.2. Results analysis

Once a small introduction to the errors is done we are going to use the values from last section tables.

To show the result we have implemented a simple program using MATLAB. First, inside the *for* loop we calculate the mantissa value and set its value in *Aux*. Then according with the data type (normal or subnormal) we use the appropriate formula for each case.

```
for i=1:N
    Aux=A(i)*2^-(1+(i-1));
    M=M+Aux;
end
  if NS==0
    Number=((-1)^S)*(NS+M)*2^(-126);
  else
    Number=((-1)^S)*(NS+M)*2^(E-127);
  end
end
```

Mantissa size

Mantissa value

Normal or Subnormal?

Subnormal Number

Normal Number

### 6.2.1. Subnormal Numbers

In the table 14 we have grouped all the results obtained during a new simulation with subnormal numbers.

As it can be seen the relative error is always zero. The error for a subnormal bits is $2^{(-23-126)} = 1.4012 \cdot 10^{-45}$. The number is small enough. In addition there is not any shifting and the double precision format of MATLAB provides this great accuracy.

***Table 11.*** *Subnormal Numbers Results*

| MA/MB | $A_{(10}$/ $B_{(10}$ | AS | $A_{(10} \pm B_{(10}$ | S | $S_{(10}$ | $E_r$ |
|---|---|---|---|---|---|---|
| 1\|00000000\|00011001100110011001100 | -7.052964983894954e-039 | - | -1.097127855243694e-038 | 1\|00000000\|11101110111011101110110 | -1.097127855243694e-038 | 0 |
| 1\|00000000\|01010101010101010101010 | -3.918313568541982e-039 | | | | | |
| 0\|00000000\|00000000000000000000101 | 7.006492321624085e-045 | - | 8.407790785948902e-045 | 0\|00000000\|00000000000000000000110 | 8.407790785948902e-045 | 0 |
| 1\|00000000\|00000000000000000000001 | -1.401298464324817e-045 | | | | | |
| 1\|00000000\|10111001101110011011100 | -8.528095061708117e-039 | - | -1.714838575327956e-038 | 1\|00000001\|01110101011101010111001 | -1.714838575327956e-038 | 0 |
| 0\|00000000\|10111011101110111011101 | 8.620290691571439e-039 | | | | | |
| 1\|00000000\|10111011101110111011101 | -8.620290691571439e-039 | - | -5.877471754111438e-039 | 1\|00000000\|10000000000000000000000 | -5.877471754111438e-039 | 0 |
| 1\|00000000\|00111011101110111011101 | -2.742818937460002e-039 | | | | | |
| 0\|00000000\|01010100101010101010010 | 3.887821313958274e-039 | + | -7.867120792966137e-039 | 1\|00000000\|10101011010101001010101 | -7.867120792966137e-039 | 0 |
| 1\|00000000\|11111111111111111111111 | -1.175494210692441e-038 | | | | | |
| 0\|00000000\|11111111111111111111111 | 1.175494210692441e-038 | + | 7.867120792966137e-039 | 0\|00000000\|10101011010101001010101 | 7.867120792966137e-039 | 0 |
| 1\|00000000\|01010100101010101010010 | -3.887821313958274e-039 | | | | | |
| 0\|00000000\|10101010101010101010001 | 7.836622933188571e-039 | + | 1.567325147157100e-038 | 0\|00000001\|01010101010101010100110 | 1.567325147157100e-038 | 0 |
| 0\|00000000\|10101010101010101010101 | 7.836628538382429e-039 | | | | | |
| 1\|00000000\|00000000000000010101010 | -4.778427763347626e-043 | + | -7.160635152699815e-043 | 1\|00000000\|00000000000001111111111 | -7.160635152699815e-043 | 0 |
| 1\|00000000\|00000000000000001010101 | -2.382207389352189e-043 | | | | | |

## *6.2.2. Mixed Numbers*

***Table 12.*** *Mixed Numbers Results*

| MA/MB | $A_{(10}/B_{(10}$ | AS | $A_{(10} \pm B_{(10}$ | S | $S_{(10}$ | $E_r$ |
|---|---|---|---|---|---|---|
| 1\|00000000\|10101010101010101010101 | -7.836628538382429e-39 | + | -9.860768564144011e-33 | 1\|00010100\|10011001100110011001101010001 | -9.860764400935319e-33 | -4.221992094207698e-7 |
| 1\|00010100\|10011001100110011001100 | -9.860760727515472e-33 | | | | | |
| | | | | | | |
| 0\|00000001\|10101010101010101010101 | +1.959157204660530e-38 | - | +1.253860706271035e-38 | 0\|00000001\|01011101110111011101111 | +1.606508955465783e-38 | +0.2812499406281 89 |
| 0\|00000000\|10011001100110011001100 | +7.052964983894954e-39 | | | | | |
| | | | | | | |
| 1\|00001000\|10101010101010101010101 | -2.507721221965479e-36 | - | -2.515557850503861e-36 | 1\|00001000\|10101011010101010101010 | -2.511639476679485e-36 | -0.0015576560179 65 |
| 0\|00000000\|10101010101010101010101 | +7.836628538382429e-39 | | | | | |
| | | | | | | |
| 0\|00001010\|01010100101010110101010 | +8.009095588956748e-36 | - | +8.020850531063673e-36 | 0\|00001010\|01010100111010101010101 | +8.014973060710860e-36 | -7.327739533420206e-4 |
| 1\|00000000\|11111111111111111111111 | -1.175494210692441e-38 | | | | | |
| | | | | | | |
| 1\|00000000\|10101010101010101010101 | -7.836628538382429e-39 | - | +2.241275705925436e-37 | 0\|00000101\|00110110011001100110010 | -2.280458771545933e-37 | +0.0174824835324 39 |
| 1\|00000101\|00111011101110111011101 | -2.319641991309260e-37 | | | | | |
| | | | | | | |
| 1\|00000000\|10101010101010101010101 | -7.836628538382429e-39 | + | +2.567828073171770e-34 | 0\|00001111\|01010101010100111111111 | +2.567867179782547e-34 | +1.522945059478952e-5 |
| 0\|00001111\|01010101010101010101010 | +2.567906439457154e-34 | | | | | |
| | | | | | | |
| 1\|00000000\|10010010010010010010010 | -6.717110175756367e-39 | + | +7.769457604780681e-37 | 0\|00000111\|00001001100001100001100 | +7.803043281776324e-37 | +0.0043227827094 36 |
| 0\|00000111\|00001010101010101010101 | +7.836628706538244e-37 | | | | | |
| | | | | | | |
| 0\|00010010\|10101010101010101010101 | +2.567906531292650e-33 | + | +2.567918286231955e-33 | 0\|00010010\|10101010101010101110101 | +2.567912408764405e-33 | -2.288806299533920e-6 |
| 0\|00000000\|11111111111111111111101 | +1.175493930432748e-38 | | | | | |

The smaller the number, the greater the relative errors we have because each time, we are nearer to the bit error value.

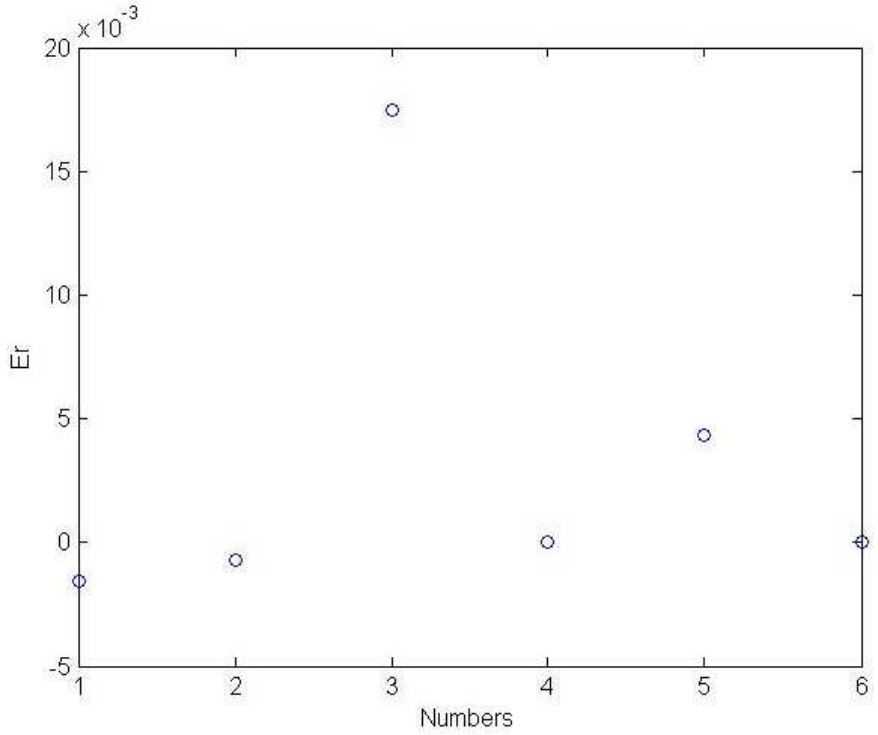



***Figure 39.*** *Relative error*

The reason to get this error is that using mixed numbers we have to shift the subnormal number, first to standardize it and later to make equal the exponents causing a big displacement because the other number is normal and its exponent will be greater.

### *6.2.3. Normal Numbers*

Finally the normal numbers turn. The results have been grouped in the table 13.

The relative error is not so big in most cases but there are two values where it increase so much. Two possible causes: the first one, as we have said, the greater the number, the greater the error we can have. The second one, as in the mixed numbers, we must shift one of the numbers losing some bits in the operation and increasing the error if the exponents are very different.

***Table 13.*** *Normal Numbers Results*

| MA/MB | $A_{(10}$ / $B_{(10}$ | AS | $A_{(10}$ ±$B_{(10}$ | S | $S_{(10}$ | $E_r$ |
|---|---|---|---|---|---|---|
| 0\|11111101\|10100101010101010101010101 | +1.417843161699894e+38 | - | +4.056481920730334e+31 | 0\|11101000\|00000000000000000000000000 | +4.056481920730334e+31 | 0 |
| 0\|11111101\|10100101010101010101010001 | +1.417842756051702e+38 | | | | | |
| | | | | | | |
| 0\|11110101\|01010100010101011010101010 | +4.422139697774293e+35 | - | -1.116310152615919e+35 | 1\|11110011\|01010111111111010101100 | -1.116310152615919e+35 | 0 |
| 0\|11110101\|10100101010101010101010101 | +5.538449850390212e+35 | | | | | |
| | | | | | | |
| 1\|11111010\|10011001100110011001100100 | -1.701411733192644e+37 | + | -2.863270299048707e+36 | 1\|11111000\|00010011101110010001000 | -2.863270299048707e+36 | 0 |
| 0\|11111010\|01010100101010101101010010 | +1.415084703287774e+37 | | | | | |
| | | | | | | |
| 0\|11110000\|10111011100111011101110101 | +1.799996137003937e+34 | + | +8.144778900470302e+31 | 0\|11101001\|00000000100000001000000000 | +8.144778900470302e+31 | 0 |
| 1\|11110000\|10111001101110011011100100 | -1.791851358103467e+34 | | | | | |
| | | | | | | |
| 0\|11111001\|10111011101110111011101101 | +9.215980221460157e+36 | - | +1.036276509064638e+37 | 0\|11111001\|11110010111100101111001 | +1.036276540755903e+37 | +3.058186181824276e-008 |
| 1\|11111010\|10111001101110011011100100 | -1.146784869186219e+36 | | | | | |
| | | | | | | |
| 0\|11111101\|10100101010101010101010101 | +1.417843161699894e+38 | - | +2.835685917751596e+38 | 0\|11111110\|01010101010101010100110 | +2.268548166293808e+38 | -0.2000002002716400 |
| 1\|11111101\|10100101010101010101010001 | -1.417842756051702e+38 | | | | | |
| | | | | | | |
| 0\|10011001\|01110111011101110111011010 | +98426320 | + | -13421752 | 1\|10010110\|10011001100110010111000 | -13421752 | 0 |
| 1\|10011001\|10101010101010101010010001 | -111848072 | | | | | |
| | | | | | | |
| 0\|10110110\|10101010101010101010010001 | +5.284222818320384e+16 | + | +6.034822523571405e+16 | 0\|10110110\|11011111111111111111111011 | +6.755397293572096e+16 | +0.119402810469769 |
| 0\|10110011\|10101010101010101010010001 | +7.505997052510208e+15 | | | | | |

## 6.3. Conclusions

The importance and usefulness of floating point format nowadays does not allow any discussion. Any computer or electronic device which operates with real numbers implements this type of representation and operation.

During this report I have tried to explain the operation and benefits of using this notation against the fixed point. The main feature is that it can represent a very large or small numbers with a relatively small and finite quantity of memory.

The clear utility of the floating point format was the main reason why I decide to do this work. The other reason was the possibility of implementing it in VHDL. I could work on something that I like as programming and design something which has a current use. What is the result?

The reached goal is the implementation of a 32bits adder/subtractor based on floating point arithmetic according with the IEEE 754 standard.

This design works with all the numbers defined by the standard: normal and subnormal. Furthermore, all the exceptions are taken into account as NaN, zero or infinity.

The VHDL code has been implemented so that all the operations are carried out with combinational logic which reaches a faster response because there are not any sequential devices as flip-flops which delays the execution time.

If I have to defend this project I will appoint two features.

The first one deals with type of architecture used. For example, the adder or the shifter is implemented with a known structure. Predetermined operations as addition (+) or shifting (SLL or SLR) are allow but I decided using a *generate* function and designing my own device which improves the time response.

Finally the mixed numbers option. IEEE 754 does not say anything about the operations between subnormal and normal numbers. I have designed a trick which allows the operation. I standardize the subnormal number and set a "false positive subnormal exponent" (truly it is negative but IEEE 754 does not allow negative exponent prebiased). Adding the normal exponent and the false subnormal one I know the number of positions I must shift the mantissa and then the operation is done properly.

Obviously the design is not perfect. The accuracy is not optimal. In the future using a double precision format would be an improvement. The execution time would increase but the accuracy also would be better.

Maybe a complete FPU design would be another good improvement. Multiplication and division have an easier implementation than the addition/subtraction and it would do the project more complete.

Finally using the code over a FPGA and testing it physically over a board would be the last aim which would leave the design completely finished.

# CHAPTER 7: BIBLIOGRAPHY


Muller, Jean-Michel; Brisebarre, Nicolas; De Dinechin, Florent; Jeannerod, Claude-Pierre; Lefèvre, Vincent; Melquiond, Guillaume; Revol, Nathalie; Stehlé, Damien and Serge Torres. "Handbook of Floating-Point Arithmetic" Birkhäuser Boston, 2010.

Lu, Mi. Texas A&M University "Arithmetic and Logic in Computer Systems" Hoboken, New Jersey, Wiley-Interscience, 2004.

Floating Point book. Wikibooks, 2012. en.wikibooks.org/wiki/Floating_Point

Overton, Michael L. "Numerical Computing with IEEE Floating Point Arithmetic". Cambridge University Press, 2001.

Mozos, Daniel; Sánchez-Elez, Marcos and José Luis Risco. "Aritmética en coma flotante: Ampliación de estructura de computadoras". Valencia, Spain: Facultad de Informática, 2008.

Floating Point book. Wikibooks, 2012. en.wikibooks.org/wiki/Floating_Point

Overton, Michael L. "Numerical Computing with IEEE Floating Point Arithmetic". Cambridge University Press, 2001.

Altera. "Floating-Point Adder/Subtractor". San José, CA, 1996

Carrera, Manuel; López, Efrén O. and Juan Naranjo. "ALU Logarítmica". Pontificia Universidad Javeriana, Facultad de ingeniería electrónica, Bogotá DC, 2005.

Pardo, Fernando, and José A. Boluda. "VHDL. Lenguaje para síntesis y modelado de circuitos" Ra-ma Ed.

Universidad Tecnológica de la Mixteca. "Lenguaje VHDL: Código para representar sistemas digitales en VHDL", 2009.

Madrenas, Jordi. "Disseny de subsistemes". Disseny Microelectrònic II, Universitat Politècnica de Catalunya, Barcelona, Spain, 2010.

# ANNEX:

# VHDL CODE

## PRE-ADDER BLOCK: *n_case* Block

```
1  ----------------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------------
4  -- Block 01 --> n_case
5  -- Description: Identify the types of data between:
6  --    -NaN       --> E=255 & T>0
7  --    -Infinity  --> E=255 & T=0
8  --    -Normal    --> 0<E<255 & T>0
9  --    -Subnormal --> E=0 & T>0
10 --    -Zero      --> E=0 & T=0
11 -- and solve the operations which we can made without the adder
12 ----------------------------------------------------------------------------------------------
13
14 library IEEE;
15 use IEEE.STD_LOGIC_1164.ALL;
16 use IEEE.STD_LOGIC_ARITH.ALL;
17 use IEEE.STD_LOGIC_UNSIGNED.ALL;
18
19 ----------------------------------------------------------------------------------------------
20 -- Entity declaration
21 ----------------------------------------------------------------------------------------------
22
23 entity n_case is
24     Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
25               NumberB : in std_logic_vector(31 downto 0);   -- Number B
26               enable  : out std_logic;                      -- Enable Adder
27               S       : out std_logic_vector(31 downto 0)); -- Output
28 end n_case;
29
30 ----------------------------------------------------------------------------------------------
31 -- Architecture description
32 ----------------------------------------------------------------------------------------------
33
34 architecture behavioral of n_case is
35
36 -- Signals declaration
37 ----------------------------------------------------------------------------------------------
38
39   signal outA, outB   : std_logic_vector(2 downto 0);
40
41   signal EA, EB       : std_logic_vector(7 downto 0);
42   signal MA, MB       : std_logic_vector(22 downto 0);
43   signal SA, SB       : std_logic;
44
45   signal SS           : std_logic;
46   signal ES           : std_logic_vector(7 downto 0);
47   signal MS           : std_logic_vector(22 downto 0);
48
49   begin
50
51     SA <= NumberA(31);
52     SB <= NumberB(31);
53     EA <= NumberA(30 downto 23);
54     EB <= NumberB(30 downto 23);
55     MA <= NumberA(22 downto 0);
56     MB <= NumberB(22 downto 0);
57
58     outA  <= "000" when EA = X"00" and MA = 0 else                    -- Zero
59              "001" when EA = X"00" and MA > 0 else                    -- Subnormal
60              "011" when (EA > X"00" and EA < X"FF") and MA > 0 else    -- Normal
61              "100" when EA = X"FF" and MA = 0 else                    -- Infinity
62              "110" when EA = X"FF" and MA > 0 else                    -- NaN
63              "000";
64
65     outB  <= "000" when EB = X"00" and MB = 0 else                    -- Zero
66              "001" when EB = X"00" and MB > 0 else                    -- Subnormal
67              "011" when (EB > X"00" and EB < X"FF") and MB > 0 else    -- Normal
68              "100" when EB = X"FF" and MB = 0 else                    -- Infinity
69              "110" when EB = X"FF" and MB > 0 else                    -- NaN
70              "000";
```

```
71
72      -- If A and B are normal or subnormal numbers, enable = 1
73      -- If not, enable = 0
74      enable <= '1' when ((outA(0) and outB(0)) = '1') else '0';
75
76      process (SA, SB, outA, outB)
77        begin
78          -------------------------- Zero
79          if (outA = "000") then                              -- Zero +/- Number B
80            SS <= SB;
81            ES <= EB;
82            MS <= MB;
83          elsif (outB = "000") then                           -- Number A +/- Zero
84            SS <= SA;
85            ES <= EA;
86            MS <= MA;
87          end if;
88          -------------------------- Infinite
89          if (outA(0) = '1' and outB = "100") then            -- Normal or Subnormal +/- Infinity
90            SS <= SB;
91            ES <= EB;
92            MS <= MB;
93          elsif (outB(0) = '1' and outA = "100") then         -- Infinity +/- Normal or Subnormal
94            SS <= SA;
95            ES <= EA;
96            MS <= MA;
97          end if;
98
99          if ((outA and outB) = "100" and SA = SB) then       -- +/- Infinity +/- Infinity
100           SS <= SA;
101           ES <= EA;
102           MS <= MA;
103         -------------------------- NaN
104         elsif ((outA and outB) = "100" and SA /= SB) then   -- + Infinity - Infinity
105           SS <= '1';
106           ES <= X"FF";
107           MS <= "00000000000000000000001";
108         end if;
109         if (outA = "110" or outB = "110") then
110           SS <= '1';
111           ES <= X"FF";
112           MS <= "00000000000000000000001";
113         end if;
114         -------------------------- Normal / Subnormal
115         if((outA(0) and outB(0)) = '1') then
116           SS <= '-';
117           ES <= "--------";
118           MS <= "-----------------------";
119         end if;
120     end process;
121
122     S(31) <= SS;
123     S(30 downto 23) <= ES;
124     S(22 downto 0) <= MS;
125
126 end behavioral;
```

## PRE-ADDER BLOCK: *Select* Block

```
1  ----------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 1 --> Selector
6  -- Description: Identify the type of the numbers between:
7  --      - Normal & Subnormal (or Subnormal & Normal)
8  --      - Normal & Normal
9  --      - Subnormal & Subnormal
10 -- and activate the correspondent block.
11 -- Moreover, I add the implicit and the guard bits
12 ----------------------------------------------------------------------------------------
13
14 library IEEE;
15 use IEEE.STD_LOGIC_1164.ALL;
16 use IEEE.STD_LOGIC_ARITH.ALL;
17 use IEEE.STD_LOGIC_UNSIGNED.ALL;
18
19 ----------------------------------------------------------------------------------------
20 -- Entity declaration
21 ----------------------------------------------------------------------------------------
22
23 entity selector is
24     Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
25               NumberB : in std_logic_vector(31 downto 0);   -- Number B
26               enable  : in std_logic;                       -- Enable
27               e_data  : out std_logic_vector(1 downto 0);   -- Enable type data
28               NA      : out std_logic_vector(36 downto 0);  -- Number A'
29               NB      : out std_logic_vector(36 downto 0)); -- Number B'
30 end selector;
31
32 ----------------------------------------------------------------------------------------
33 -- Architecture description
34 ----------------------------------------------------------------------------------------
35
36 architecture behavioral of selector is
37
38 -- Signals declaration
39 ----------------------------------------------------------------------------------------
40   signal EA, EB          : std_logic_vector(7 downto 0);
41   signal MA, MB          : std_logic_vector(22 downto 0);
42   signal SA, SB          : std_logic;
43
44    begin
45
46      SA <= NumberA(31);
47      SB <= NumberB(31);
48      EA <= NumberA(30 downto 23);
49      EB <= NumberB(30 downto 23);
50      MA <= NumberA(22 downto 0);
51      MB <= NumberB(22 downto 0);
52
53      process (SA, SB, EA, EB, MA, MB, enable)
54      begin
55        if enable = '1' then
56          NA(36) <= SA;                -- Exponent & sign A
57          NA(35 downto 28) <= EA;
58          NB(36) <= SB;                -- Exponent & sign B
59          NB(35 downto 28) <= EB;
60          ------------------------------------ Mantissa A
61          if (EA > X"00") then
62            NA(27) <= '1';             -- Implicit bit
63            NA(26 downto 4) <= MA;     -- Mantissa
64            NA(3 downto 0) <= X"0";    -- Guard bits
65          elsif EA = X"00" then
66            NA(27) <= '0';             -- Implicit bit
67            NA(26 downto 4) <= MA;     -- Mantissa
68            NA(3 downto 0) <= X"0";    -- Guard bits
69          else
70            NA <= "-----------------------------------";
71          end if;
72          ------------------------------------ Mantissa B
```

```
74            NB(27) <= '1';              -- Implicit bit
75            NB(26 downto 4) <= MB;      -- Mantissa
76            NB(3 downto 0) <= X"0";     -- Guard bits
77          elsif EB = X"00" then
78            NB(27) <= '0';              -- Implicit bit
79            NB(26 downto 4) <= MB;      -- Mantissa
80            NB(3 downto 0) <= X"0";     -- Guard bits
81          else
82            NB <= "-----------------------------------";
83          end if;
84        else
85          NA <= "-----------------------------------";
86          NB <= "-----------------------------------";
87        end if;
88      end process;
89
90      e_data <= "00" when EA = X"00" and EB = X"00" and enable = '1' else         -- Subnormals
91                "01" when EA > X"00" and EB > X"00" and enable = '1' else         -- Normals
92                "10" when (EA = X"00" or EB = X"00") and enable = '1' else       -- Combination
93                "--";
94
95    end behavioral;
```

## PRE-ADDER BLOCK: Normal Numbers: *Comp_Exp* Block

```
1  ----------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 2 --> Normal Numbers
6  -- Description: Prepare normal numbers for addition or subtraction operation
7  --
8  -- Comp_exp: Calculate the difference between exponents and the largest one
9  --           Determine the largest number (to calculate the output's sign)
10 --           Determine the shortest mantissa to shift in the next block
11 ----------------------------------------------------------------------------------------
12
13 library IEEE;
14 use IEEE.STD_LOGIC_1164.ALL;
15 use IEEE.STD_LOGIC_ARITH.ALL;
16 use IEEE.STD_LOGIC_UNSIGNED.ALL;
17
18 ----------------------------------------------------------------------------------------
19 -- Entity declaration
20 ----------------------------------------------------------------------------------------
21
22 entity comp_exp is
23     Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
24               NumberB : in std_logic_vector(36 downto 0);   -- Number B
25               SA      : out std_logic;                      -- Sign A
26               SB      : out std_logic;                      -- Sign B
27               Emax    : out std_logic_vector(7 downto 0);   -- Output exponent
28               Mmax    : out std_logic_vector(27 downto 0);  -- Largest Mantissa
29               Mshft   : out std_logic_vector(27 downto 0);  -- Mantissa to shift
30               Dexp    : out std_logic_vector(4 downto 0);   -- Subtraction of exponents
31               Comp    : out std_logic);                     -- Determine largest number
32 end comp_exp;
33
34 ----------------------------------------------------------------------------------------
35 -- Architecture description
36 ----------------------------------------------------------------------------------------
37
38 architecture behavioral of comp_exp is
39
40 -- Signals declaration
41 ----------------------------------------------------------------------------------------
42   signal EA, EB    : std_logic_vector(7 downto 0);
43   signal MA, MB    : std_logic_vector(27 downto 0);
44
45   signal dif : std_logic_vector(7 downto 0);
46   signal C   : std_logic;
47
48   begin
49
50     SA <= NumberA(36);                  -- Sign A & B
51     SB <= NumberB(36);
52
53     EA <= NumberA(35 downto 28);        --  Exponent & Mantissa
54     EB <= NumberB(35 downto 28);
55     MA <= NumberA(27 downto 0);
56     MB <= NumberB(27 downto 0);
57
58     ----------------------------- Exponent Comparison
59
60     C <= '1' when (EA > EB) or (MB(0) = '1') else     -- Exponent A > Exponent B
61          '0' when EA < EB else                        -- Exponent B > Exponent A
62          '1' when MA >= MB else                       -- EA = EB --> A > B
63          '0' when MA < MB else                        -- EA = EB --> B > A
64          '-';
65
66     Comp <= C;
67
68     ----------------------------- Largest exponent
69     Emax <= EA when C = '1' else
70             EB when C = '0' else
71             "--------";
72
```

```
73      ----------------------------- Difference between exponents
74      dif   <= EA-EB when (C = '1') and (MB(0) = '0') else
75               EB-EA when C = '0' else
76               EA+EB when (C = '1') and (MB(0) = '1') else
77               "--------";
78
79      process (dif)
80        begin
81
82          if dif <= X"1B" then            -- If the difference is less than or equal to 27...
83            Dexp <= dif(4 downto 0);      -- Use directly the subtraction between exponents
84          elsif dif > X"1B" then          -- If the difference is greater...
85            Dexp <= "11100";              -- The difference is 28
86          else
87            Dexp <= "-----";
88          end if;
89
90        end process;
91
92      ----------------------------- Mantissa
93
94      Mshft <= MB when C = '1' else
95               MA when C = '0' else
96               "----------------------------";
97      Mmax  <= MA when C = '1' else
98               MB when C = '0' else
99               "----------------------------";
100
101   end behavioral;
```

## PRE-ADDER BLOCK: Normal Numbers: *Shift* Block: *MUX* Entity

```
1   ----------------------------------------------------------------------------------------
2   -- Logarithmic Shifter
3   ----------------------------------------------------------------------------------------
4   library IEEE;
5   use IEEE.STD_LOGIC_1164.ALL;
6   use IEEE.STD_LOGIC_ARITH.ALL;
7   use IEEE.STD_LOGIC_UNSIGNED.ALL;
8
9   ----------------------------------------------------------------------------------------
10  -- Entity declaration
11  ----------------------------------------------------------------------------------------
12  entity MUX is port (
13      A, B, Sel    : in std_logic;
14      Z            : out std_logic);
15  end MUX;
16
17  ----------------------------------------------------------------------------------------
18  -- Architecture description
19  ----------------------------------------------------------------------------------------
20  architecture behavioral of MUX is
21
22  begin
23
24    Z <= A when Sel = '1' else
25         B;
26
27  end behavioral;
```

## PRE-ADDER BLOCK: Normal Numbers: *Shift* Block

```
1  ----------------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 2 --> Normal Numbers
6  -- Description: Prepare normal numbers for addition or subtraction operation
7  --
8  -- Shift: Shift the shortest significand to do the addition/subtraction
9  ----------------------------------------------------------------------------------------------
10
11 library IEEE;
12 use IEEE.STD_LOGIC_1164.ALL;
13 use IEEE.STD_LOGIC_ARITH.ALL;
14 use IEEE.STD_LOGIC_UNSIGNED.ALL;
15
16 ----------------------------------------------------------------------------------------------
17 -- Entity declaration
18 ----------------------------------------------------------------------------------------------
19
20 entity shift is
21     Port (    T     : in std_logic_vector(27 downto 0);     -- Significand to shift
22               Shft  : in std_logic_vector(4 downto 0);      -- Exponent's subtraction
23               S     : out std_logic_vector(27 downto 0));   -- Output
24 end shift;
25
26 ----------------------------------------------------------------------------------------------
27 -- Architecture description
28 ----------------------------------------------------------------------------------------------
29
30 architecture behavioral of shift is
31
32 -- Signals and components declaration
33 ----------------------------------------------------------------------------------------------
34
35   component MUX port (A, B, Sel : in std_logic; Z : out std_logic); end component;
36
37   signal Z1, Z2, Z3, Z4, Z5  : std_logic_vector(27 downto 0);
38
39   begin
40
41  -- Components generation
42
43  Comp1: for i in 0 to 27 generate
44
45    shifter0_0: if (i=0) generate
46                shifter0_0comp: MUX port map (A => '0', B => T(27), Sel => Shft(0), Z => Z1(27-i));
47                end generate;
48    shifter0_i: if ((i>0) and (i<28)) generate
49                shifter0_icomp: MUX port map (A => T(27-(i-1)), B => T(27-i), Sel => Shft(0), Z => Z1(27-i));
50                end generate;
51
52    shifter1_0: if ((i>=0) and (i<2)) generate
53                shifter1_0comp: MUX port map (A => '0', B => Z1(27-i), Sel => Shft(1), Z => Z2(27-i));
54                end generate;
55    shifter1_i: if ((i>1) and (i<28)) generate
56                shifter1_icomp: MUX port map (A => Z1(27-(i-2)), B => Z1(27-i), Sel => Shft(1), Z => Z2(27-i));
57                end generate;
58
59    shifter2_0: if ((i>=0) and (i<4)) generate
60                shifter2_0comp: MUX port map (A => '0', B => Z2(27-i), Sel => Shft(2), Z => Z3(27-i));
61                end generate;
62    shifter2_i: if ((i>3) and (i<28)) generate
63                shifter2_icomp: MUX port map (A => Z2(27-(i-4)), B => Z2(27-i), Sel => Shft(2), Z => Z3(27-i));
64                end generate;
65
66    shifter3_0: if ((i>=0) and (i<8)) generate
67                shifter3_0comp: MUX port map (A => '0', B => Z3(27-i), Sel => Shft(3), Z => Z4(27-i));
68                end generate;
69    shifter3_i: if ((i>7) and (i<28)) generate
70                shifter3_icomp: MUX port map (A => Z3(27-(i-8)), B => Z3(27-i), Sel => Shft(3), Z => Z4(27-i));
71                end generate;
72
73    shifter4_0: if ((i>=0) and (i<16)) generate
74                shifter4_0comp: MUX port map (A => '0', B => Z4(27-i), Sel => Shft(4), Z => Z5(27-i));
75                end generate;
76    shifter4_i: if ((i>15) and (i<28)) generate
77                shifter4_icomp: MUX port map (A => Z4(27-(i-16)), B => Z4(27-i), Sel => Shft(4), Z => Z5(27-i));
78                end generate;
79
80  end generate;
81
82   S <= Z5;
83
84 end behavioral;
```

## PRE-ADDER BLOCK: Normal Numbers: *n_normal* Block

```
1  ----------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 2 --> Normal Numbers
6  -- Description: Prepare normal numbers for addition or subtraction operation
7  ----------------------------------------------------------------------------------
8
9  library IEEE;
10 use IEEE.STD_LOGIC_1164.ALL;
11 use IEEE.STD_LOGIC_ARITH.ALL;
12 use IEEE.STD_LOGIC_UNSIGNED.ALL;
13
14 ----------------------------------------------------------------------------------
15 -- Entity declaration
16 ----------------------------------------------------------------------------------
17
18 entity n_normal is
19     Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
20               NumberB : in std_logic_vector(36 downto 0);   -- Number B
21               Comp    : out std_logic;                      -- A & B Comparison
22               SA      : out std_logic;                      -- Sign A
23               SB      : out std_logic;                      -- Sign B
24               EO      : out std_logic_vector(7 downto 0);   -- Exponent Output
25               MA      : out std_logic_vector(27 downto 0);  -- Greatest Mantissa
26               MB      : out std_logic_vector(27 downto 0)); -- Shifted Mantissa
27 end n_normal;
28
29 ----------------------------------------------------------------------------------
30 -- Architecture description
31 ----------------------------------------------------------------------------------
32
33 architecture behavioral of n_normal is
34
35 -- Signals and components declaration
36 ----------------------------------------------------------------------------------
37
38   component comp_exp port (NumberA, NumberB : in std_logic_vector(36 downto 0);
39                            SA, SB           : out std_logic;
40                            Emax             : out std_logic_vector(7 downto 0);
41                            Mmax, Mshft      : out std_logic_vector(27 downto 0);
42                            Dexp             : out std_logic_vector(4 downto 0);
43                            Comp             : out std_logic);
44   end component;
45
46   component shift port (T    : in std_logic_vector(27 downto 0);
47                         shft : in std_logic_vector(4 downto 0);
48                         S    : out std_logic_vector(27 downto 0));
49   end component;
50
51   signal Mshft_aux      : std_logic_vector(27 downto 0);
52   signal Dexp_aux       : std_logic_vector(4 downto 0);
53
54   begin
55   comp0 : comp_exp
56       port map (NumberA => NumberA, NumberB => NumberB,
57                 SA => SA, SB => SB, Emax => EO, Mmax => MA, Mshft => Mshft_aux, Dexp => Dexp_aux, Comp => Comp);
58
59   comp1 : shift
60       port map (T => Mshft_aux, shft => Dexp_aux,
61                 S => MB);
62
63 end behavioral;
64
```

## PRE-ADDER BLOCK: Subnormal Numbers: *n_subn* Block

```
----------------------------------------------------------------------------------------
-- Floating point adder (32 bits)
----------------------------------------------------------------------------------------
-- Block 02 --> Pre - Adder
--    Sub Block 3 --> SubNormal Numbers
-- Description: Prepare subnormal numbers for addition or subtraction operation
----------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------
-- Entity declaration
----------------------------------------------------------------------------------------

entity n_subn is
    Port (    NumberA : in std_logic_vector(36 downto 0);    -- Number A
              NumberB : in std_logic_vector(36 downto 0);    -- Number B
              Comp    : out std_logic;                       -- Comparison A & B
              SA      : out std_logic;                       -- Sign A
              SB      : out std_logic;                       -- Sign B
              EO      : out std_logic_vector(7 downto 0);    -- Exponent Output
              MA      : out std_logic_vector(27 downto 0);   -- Mantissa A
              MB      : out std_logic_vector(27 downto 0));  -- Mantissa B
end n_subn;

----------------------------------------------------------------------------------------
-- Architecture description
----------------------------------------------------------------------------------------

architecture behavioral of n_subn is

-- Signals declaration
----------------------------------------------------------------------------------------

  signal MAa, MBb    : std_logic_vector(27 downto 0);
  signal C           : std_logic;

  begin

    SA <= NumberA(36);                    -- Sign A & B
    SB <= NumberB(36);

    MAa <= NumberA(27 downto 0);          -- Mantissa A & B
    MBb <= NumberB(27 downto 0);

  ------------------------------------------------ Number Comparison
    C    <= '1' when MAa >= MBb else      -- A > B
            '0' when MBb > MAa else       -- B > A
            '-';

    Comp <= C;

  ------------------------------------------------ Output's exponent
    EO <= NumberA(35 downto 28);

  ------------------------------------------------ Mantissa
    MB <= MBb when C = '1' else
          MAa when C = '0' else
          "----------------------------";
    MA <= MAa when C = '1' else
          MBb when C = '0' else
          "----------------------------";

  end behavioral;
```

## PRE-ADDER BLOCK: Mixed Numbers: *Comp* Block

```
1  ----------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 4 --> Mixed numbers
6  -- Description: Prepare a mix of numbers (normal & subnormal) for addition
7  --              or subtraction operation
8  --
9  -- Comp    : Determine the subnormal number
10 ----------------------------------------------------------------------------------------
11
12 library IEEE;
13 use IEEE.STD_LOGIC_1164.ALL;
14 use IEEE.STD_LOGIC_ARITH.ALL;
15 use IEEE.STD_LOGIC_UNSIGNED.ALL;
16
17 ----------------------------------------------------------------------------------------
18 -- Entity declaration
19 ----------------------------------------------------------------------------------------
20
21 entity comp is
22     Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
23               NumberB : in std_logic_vector(36 downto 0);   -- Number B
24               NA      : out std_logic_vector(36 downto 0);  -- Normal number
25               NB      : out std_logic_vector(36 downto 0)); -- Subnormal number
26 end comp;
27
28 ----------------------------------------------------------------------------------------
29 -- Architecture description
30 ----------------------------------------------------------------------------------------
31
32 architecture behavioral of comp is
33
34 -- Signals declaration
35 ----------------------------------------------------------------------------------------
36   signal EA, EB    : std_logic_vector(7 downto 0);
37
38   begin
39
40     EA <= NumberA(35 downto 28);         --  Exponent & Mantissa
41     EB <= NumberB(35 downto 28);
42
43     process (NumberA, NumberB, EA, EB)
44       begin
45
46         if EA = X"00" then              -- If Number A is subnormal...
47           NB <= NumberA;
48           NA <= NumberB;
49         elsif EB = X"00" then           -- If Number B is subnormal...
50           NB <= NumberB;
51           NA <= NumberA;
52         else
53           NA <= "-----------------------------------";
54           NB <= "-----------------------------------";
55         end if;
56
57       end process;
58
59   end behavioral;
```

## PRE-ADDER BLOCK: Mixed Numbers: *Zero* Block

```
1   ----------------------------------------------------------------------------------------
2   -- Floating point adder (32 bits)
3   ----------------------------------------------------------------------------------------
4   -- Block 02 --> Pre - Adder
5   --    Sub Block 4 --> Mixed numbers
6   -- Description: Prepare subnormal numbers to be operated with the normal ones
7   --
8   -- Zero: Count the number of zeros to shift the subnormal number
9   ----------------------------------------------------------------------------------------
10
11  library IEEE;
12  use IEEE.STD_LOGIC_1164.ALL;
13  use IEEE.STD_LOGIC_ARITH.ALL;
14  use IEEE.STD_LOGIC_UNSIGNED.ALL;
15
16  ----------------------------------------------------------------------------------------
17  -- Entity declaration
18  ----------------------------------------------------------------------------------------
19
20  entity zero is
21      Port (    T      : in std_logic_vector(27 downto 0);    -- Significand
22                Zcount : out std_logic_vector(4 downto 0));   -- Number of Zeros
23  end zero;
24
25  ----------------------------------------------------------------------------------------
26  -- Architecture description
27  ----------------------------------------------------------------------------------------
28
29  architecture behavioral of zero is
30
31  -- Signals and components declaration
32  ----------------------------------------------------------------------------------------
33
34    signal Zero_vector : std_logic_vector(27 downto 0);
35    signal aux         : std_logic_vector(7 downto 0);
36
37    begin
38
39    Zero_vector <= X"0000000";
40
41     aux    <= "--------" when T(27 downto 27) = "-" else
42               X"1C" when T(27 downto 0) = Zero_vector(27 downto 0) else
43               X"1B" when T(27 downto 1) = Zero_vector(27 downto 1) else
44               X"1A" when T(27 downto 2) = Zero_vector(27 downto 2) else
45               X"19" when T(27 downto 3) = Zero_vector(27 downto 3) else
46               X"18" when T(27 downto 4) = Zero_vector(27 downto 4) else
47               X"17" when T(27 downto 5) = Zero_vector(27 downto 5) else
48               X"16" when T(27 downto 6) = Zero_vector(27 downto 6) else
49               X"15" when T(27 downto 7) = Zero_vector(27 downto 7) else
50               X"14" when T(27 downto 8) = Zero_vector(27 downto 8) else
51               X"13" when T(27 downto 9) = Zero_vector(27 downto 9) else
52               X"12" when T(27 downto 10) = Zero_vector(27 downto 10) else
53               X"11" when T(27 downto 11) = Zero_vector(27 downto 11) else
54               X"10" when T(27 downto 12) = Zero_vector(27 downto 12) else
55               X"0F" when T(27 downto 13) = Zero_vector(27 downto 13) else
56               X"0E" when T(27 downto 14) = Zero_vector(27 downto 14) else
57               X"0D" when T(27 downto 15) = Zero_vector(27 downto 15) else
58               X"0C" when T(27 downto 16) = Zero_vector(27 downto 16) else
59               X"0B" when T(27 downto 17) = Zero_vector(27 downto 17) else
60               X"0A" when T(27 downto 18) = Zero_vector(27 downto 18) else
61               X"09" when T(27 downto 19) = Zero_vector(27 downto 19) else
62               X"08" when T(27 downto 20) = Zero_vector(27 downto 20) else
63               X"07" when T(27 downto 21) = Zero_vector(27 downto 21) else
64               X"06" when T(27 downto 22) = Zero_vector(27 downto 22) else
65               X"05" when T(27 downto 23) = Zero_vector(27 downto 23) else
66               X"04" when T(27 downto 24) = Zero_vector(27 downto 24) else
67               X"03" when T(27 downto 25) = Zero_vector(27 downto 25) else
68               X"02" when T(27 downto 26) = Zero_vector(27 downto 26) else
69               X"01" when T(27 downto 27) = Zero_vector(27 downto 27) else
70               X"00";
71
72    Zcount <= aux(4 downto 0);
73
74  end behavioral;
```

## PRE-ADDER BLOCK: Mixed Numbers: *shift_left* Block

```
----------------------------------------------------------------------------------------------
-- Floating point adder (32 bits)
----------------------------------------------------------------------------------------------
-- Block 02 --> Pre - Adder
--    Sub Block 4 --> Mixed Numbers
-- Description: Prepare normal numbers for addition or subtraction operation
--
-- Shift_left: Shift the subnormal significand to get a normal one
----------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------------
-- Entity declaration
----------------------------------------------------------------------------------------------

entity shift_left is
    Port (    T     : in std_logic_vector(27 downto 0);    -- Significand to shift
              Shft  : in std_logic_vector(4 downto 0);     -- Exponent's subtraction
              S     : out std_logic_vector(27 downto 0));  -- Output
end shift_left;

----------------------------------------------------------------------------------------------
-- Architecture description
----------------------------------------------------------------------------------------------

architecture behavioral of shift_left is

-- Signals and components declaration
----------------------------------------------------------------------------------------------

  component MUX port (A, B, Sel : in std_logic; Z : out std_logic); end component;

 signal Z1, Z2, Z3, Z4, Z5  : std_logic_vector(27 downto 0);

 begin

 -- Components generation

 Comp1: for i in 0 to 27 generate

   shifter0_0: if (i=0) generate
               shifter0_0comp: MUX port map (A => '0', B => T(0), Sel => Shft(0), Z => Z1(i));
               end generate;
   shifter0_i: if ((i>0) and (i<28)) generate
               shifter0_icomp: MUX port map (A => T((i-1)), B => T(i), Sel => Shft(0), Z => Z1(i));
               end generate;

   shifter1_0: if ((i>=0) and (i<2)) generate
               shifter1_0comp: MUX port map (A => '0', B => Z1(i), Sel => Shft(1), Z => Z2(i));
               end generate;
   shifter1_i: if ((i>1) and (i<28)) generate
               shifter1_icomp: MUX port map (A => Z1((i-2)), B => Z1(i), Sel => Shft(1), Z => Z2(i));
               end generate;

   shifter2_0: if ((i>=0) and (i<4)) generate
               shifter2_0comp: MUX port map (A => '0', B => Z2(i), Sel => Shft(2), Z => Z3(i));
               end generate;
   shifter2_i: if ((i>3) and (i<28)) generate
               shifter2_icomp: MUX port map (A => Z2((i-4)), B => Z2(i), Sel => Shft(2), Z => Z3(i));
               end generate;

   shifter3_0: if ((i>=0) and (i<8)) generate
               shifter3_0comp: MUX port map (A => '0', B => Z3(i), Sel => Shft(3), Z => Z4(i));
               end generate;
   shifter3_i: if ((i>7) and (i<28)) generate
               shifter3_icomp: MUX port map (A => Z3((i-8)), B => Z3(i), Sel => Shft(3), Z => Z4(i));
               end generate;

   shifter4_0: if ((i>=0) and (i<16)) generate
               shifter4_0comp: MUX port map (A => '0', B => Z4(i), Sel => Shft(4), Z => Z5(i));
               end generate;
   shifter4_i: if ((i>15) and (i<28)) generate
               shifter4_icomp: MUX port map (A => Z4((i-16)), B => Z4(i), Sel => Shft(4), Z => Z5(i));
               end generate;
```

```
79
80      end generate;
81
82      S <= Z5;
83
84  end behavioral;
```

## PRE-ADDER BLOCK: Mixed Numbers: *norm* Block

```
1   ----------------------------------------------------------------------------------------
2   -- Floating point adder (32 bits)
3   ----------------------------------------------------------------------------------------
4   -- Block 02 --> Pre - Adder
5   --    Sub Block 4 --> Mixed numbers
6   -- Description: Prepare the subnormal number for addition or subtraction
7   --              with the normal one
8   --
9   -- Norm: Normalize the subnormal number to operate like a normal number
10  ----------------------------------------------------------------------------------------
11
12  library IEEE;
13  use IEEE.STD_LOGIC_1164.ALL;
14  use IEEE.STD_LOGIC_ARITH.ALL;
15  use IEEE.STD_LOGIC_UNSIGNED.ALL;
16
17  ----------------------------------------------------------------------------------------
18  -- Entity declaration
19  ----------------------------------------------------------------------------------------
20
21  entity norm is
22      Port (    NumberA : in std_logic_vector(36 downto 0);    -- NumberA
23                NumberB : in std_logic_vector(36 downto 0);    -- NumberB
24                MA      : out std_logic_vector(36 downto 0);   -- Normalized NumberA
25                MB      : out std_logic_vector(36 downto 0));  -- Normalized NumberB
26  end norm;
27
28  ----------------------------------------------------------------------------------------
29  -- Architecture description
30  ----------------------------------------------------------------------------------------
31
32  architecture behavioral of norm is
33
34  -- Signals and components declaration
35  ----------------------------------------------------------------------------------------
36
37    component zero port (T : in std_logic_vector(27 downto 0); Zcount : out std_logic_vector(4 downto 0));
38    end component;
39
40
41    component shift_left port (T    : in std_logic_vector(27 downto 0);
42                               Shft : in std_logic_vector(4 downto 0);
43                               S    : out std_logic_vector(27 downto 0));
44    end component;
45
46    component comp port (NumberA : in std_logic_vector(36 downto 0);
47                         NumberB : in std_logic_vector(36 downto 0);
48                         NA      : out std_logic_vector(36 downto 0);
49                         NB      : out std_logic_vector(36 downto 0));
50    end component;
51
52    signal Zcount_aux          : std_logic_vector(4 downto 0);
53
54    signal EB                  : std_logic_vector(7 downto 0);
55    signal NumberB_aux         : std_logic_vector(36 downto 0);
56    signal MB_aux              : std_logic_vector(27 downto 0);
57
58    begin
59
60    ------------------------------------------------ Components declaration
61    comp0 : zero
62        port map (T => NumberB_aux(27 downto 0), Zcount => Zcount_aux);
63
64    comp1 : shift_left
65        port map (T => NumberB_aux(27 downto 0), shft => Zcount_aux, S => MB_aux);
66
67    comp2 : comp
68        port map (NumberA => NumberA, NumberB => NumberB, NA => MA, NB => NumberB_aux);
69
70    ------------------------------------------------ New Exponent
```

```
71
72      process(Zcount_aux, NumberB_aux, EB, MB_aux)
73        begin
74          if Zcount_aux /= "-----" then
75            EB <= "000" & Zcount_aux;                      -- Number shifted
76            MB(27 downto 0) <= MB_aux(27 downto 1) & '1'; -- Bit 0 --> Mark
77          else
78            EB <= "--------";
79            MB(27 downto 0) <= MB_aux;
80          end if;
81            MB(35 downto 28) <= EB;
82            MB(36) <= NumberB_aux(36);
83      end process;
84
85    end behavioral;
```

## PRE-ADDER BLOCK: MUX/DEMUX

```
1  ----------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 5 --> DEMUX
6  -- Description: Demultiplexor
7  ----------------------------------------------------------------------------------
8
9  library IEEE;
10 use IEEE.STD_LOGIC_1164.ALL;
11 use IEEE.STD_LOGIC_ARITH.ALL;
12 use IEEE.STD_LOGIC_UNSIGNED.ALL;
13
14 ----------------------------------------------------------------------------------
15 -- Entity declaration
16 ----------------------------------------------------------------------------------
17
18 entity demux is
19     Port (    NumberA : in std_logic_vector(36 downto 0);   -- Number A
20               NumberB : in std_logic_vector(36 downto 0);   -- Number B
21               e_data  : in std_logic_vector(1 downto 0);    -- Enable type data
22               NA0     : out std_logic_vector(36 downto 0);  -- Number A1
23               NB0     : out std_logic_vector(36 downto 0);  -- Number B1
24               NA1     : out std_logic_vector(36 downto 0);  -- Number A2
25               NB1     : out std_logic_vector(36 downto 0);  -- Number B2
26               NA2     : out std_logic_vector(36 downto 0);  -- Number A3
27               NB2     : out std_logic_vector(36 downto 0)); -- Number B3
28 end demux;
29
30 ----------------------------------------------------------------------------------
31 -- Architecture description
32 ----------------------------------------------------------------------------------
33
34 architecture behavioral of demux is
35
36   begin
37
38     process (NumberA, NumberB, e_data)
39     begin
40
41       case e_data is
42         ------------------------------------------ Subnormals
43         when "00"   => NA0 <= NumberA;
44                        NB0 <= NumberB;
45                        NA1 <= "-----------------------------------";
46                        NB1 <= "-----------------------------------";
47                        NA2 <= "-----------------------------------";
48                        NB2 <= "-----------------------------------";
49         ------------------------------------------ Normals
50         when "01"   => NA0 <= "-----------------------------------";
51                        NB0 <= "-----------------------------------";
52                        NA1 <= NumberA;
53                        NB1 <= NumberB;
54                        NA2 <= "-----------------------------------";
55                        NB2 <= "-----------------------------------";
56         ------------------------------------------ MiX
57         when "10"   => NA0 <= "-----------------------------------";
58                        NB0 <= "-----------------------------------";
59                        NA1 <= "-----------------------------------";
60                        NB1 <= "-----------------------------------";
61                        NA2 <= NumberA;
62                        NB2 <= NumberB;
63         when others => NA0 <= "-----------------------------------";
64                        NB0 <= "-----------------------------------";
65                        NA1 <= "-----------------------------------";
66                        NB1 <= "-----------------------------------";
67                        NA2 <= "-----------------------------------";
68                        NB2 <= "-----------------------------------";
69       end case;
70
71     end process;
72
73   end behavioral;
```

```
1  ----------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 6 --> MUX
6  -- Description: Multiplexor
7  ----------------------------------------------------------------------------------------
8
9  library IEEE;
10 use IEEE.STD_LOGIC_1164.ALL;
11 use IEEE.STD_LOGIC_ARITH.ALL;
12 use IEEE.STD_LOGIC_UNSIGNED.ALL;
13
14 ----------------------------------------------------------------------------------------
15 -- Entity declaration
16 ----------------------------------------------------------------------------------------
17
18 entity mux_ns is
19     Port (    NorA    : in std_logic_vector(36 downto 0);   -- Normal A
20               NorB    : in std_logic_vector(36 downto 0);   -- Normal B
21               MixA    : in std_logic_vector(36 downto 0);   -- Mixed A
22               MixB    : in std_logic_vector(36 downto 0);   -- Mixed B
23               e_data  : in std_logic_vector(1 downto 0);    -- Enable type data
24               NA      : out std_logic_vector(36 downto 0);  -- Number A'
25               NB      : out std_logic_vector(36 downto 0)); -- Number B'
26 end mux_ns;
27
28 ----------------------------------------------------------------------------------------
29 -- Architecture description
30 ----------------------------------------------------------------------------------------
31
32 architecture behavioral of mux_ns is
33
34   begin
35
36     NA <= NorA when e_data = "01" else                  -- Normal numbers
37           MixA when e_data = "10" else                  -- Mixed numbers
38           "-----------------------------------";
39
40     NB <= NorB when e_data = "01" else                   -- Normal numbers
41           MixB when e_data = "10" else                   -- Mixed numbers
42           "-----------------------------------";
43
44   end behavioral;
45
```

```
1  ----------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  --    Sub Block 7 --> MUX
6  -- Description: Multiplexor
7  ----------------------------------------------------------------------------------------
8
9  library IEEE;
10 use IEEE.STD_LOGIC_1164.ALL;
11 use IEEE.STD_LOGIC_ARITH.ALL;
12 use IEEE.STD_LOGIC_UNSIGNED.ALL;
13
14 ----------------------------------------------------------------------------------------
15 -- Entity declaration
16 ----------------------------------------------------------------------------------------
17
18 entity mux_adder is
19     Port (    NorSA   : in std_logic;                       -- Sign A normal
20               NorSB   : in std_logic;                       -- Sign B normal
21               SubSA   : in std_logic;                       -- Sign A subnormal
22               SubSB   : in std_logic;                       -- Sign B subnormal
23               CompN   : in std_logic;                       -- Comparison Normal numbers
24               CompS   : in std_logic;                       -- Compaison Sub numbers
25               NorE    : in std_logic_vector(7 downto 0);    -- Exponent Output normal
26               SubE    : in std_logic_vector(7 downto 0);    -- Exponent Output subnormal
27               NorMA   : in std_logic_vector(27 downto 0);   -- Mantissa normal A
28               NorMB   : in std_logic_vector(27 downto 0);   -- Mantissa normal B
29               SubMA   : in std_logic_vector(27 downto 0);   -- Mantissa subnormal A
30               SubMB   : in std_logic_vector(27 downto 0);   -- Mantissa subnormal B
31               e_data  : in std_logic_vector(1 downto 0);    -- Enable type data
32               SA      : out std_logic;                      -- Sign A
33               SB      : out std_logic;                      -- Sign B
34               C       : out std_logic;                      -- Comparison
35               E       : out std_logic_vector(7 downto 0);   -- Output Exponent
36               A       : out std_logic_vector(27 downto 0);  -- Mantissa A
37               B       : out std_logic_vector(27 downto 0));  -- Mantissa B
38 end mux_adder;
39
40 ----------------------------------------------------------------------------------------
41 -- Architecture description
42 ----------------------------------------------------------------------------------------
43
44 architecture behavioral of mux_adder is
45
46   begin
47
48     A  <= NorMA when e_data = "01" or e_data = "10" else          -- Normal/Mix numbers
49           SubMA when e_data = "00" else                          -- Subnormal numbers
50           "----------------------------";
51
52     B  <= NorMB when e_data = "01" or e_data = "10" else          -- Normal/Mix numbers
53           SubMB when e_data = "00" else                          -- Subnormal numbers
54           "----------------------------";
55
56     C  <= CompN when e_data = "01" or e_data = "10" else          -- Normal/Mix numbers
57           CompS when e_data = "00" else                          -- Subnormal numbers
58           '-';
59
60     SA <= NorSA when e_data = "01" or e_data = "10" else          -- Normal/Mix sign A
61           SubSA when e_data = "00" else                          -- Subnormal sign A
62           '-';
63
64     SB <= NorSB when e_data = "01" or e_data = "10" else          -- Normal / Mix sign B
65           SubSB when e_data = "00" else                          -- Subnormal sign B
66           '-';
67
68     E  <= NorE when e_data = "01" or e_data = "10" else           -- Normal / Mix exponent
69           SubE when e_data = "00" else                           -- Subnormal exponent
70           "--------";
71
72   end behavioral;
```

## PRE-ADDER BLOCK: *preadder* Block

```
1  ----------------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------------
4  -- Block 02 --> Pre - Adder
5  -- Prepare the numbers to be used by the adder
6  ----------------------------------------------------------------------------------------------
7
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13 ----------------------------------------------------------------------------------------------
14 -- Entity declaration
15 ----------------------------------------------------------------------------------------------
16
17 entity preadder is
18     Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
19               NumberB : in std_logic_vector(31 downto 0);   -- Number B
20               enable  : in std_logic;                       -- Enable
21               SA      : out std_logic;                      -- Sign A
22               SB      : out std_logic;                      -- Sign B
23               C       : out std_logic;                      -- Comparison
24               EOut    : out std_logic_vector(7 downto 0);   -- Exponent Output
25               MAOut   : out std_logic_vector(27 downto 0);  -- Greatest Mantissa
26               MBOut   : out std_logic_vector(27 downto 0)); -- Shifted Mantissa
27 end preadder;
28
29 ----------------------------------------------------------------------------------------------
30 -- Architecture description
31 ----------------------------------------------------------------------------------------------
32
33 architecture behavioral of preadder is
34
35 -- Signals and components declaration
36 ----------------------------------------------------------------------------------------------
37
38   ------------------------------------------------------------------------------ Normal Numbers
39   component n_normal port (NumberA, NumberB : in std_logic_vector(36 downto 0);
40                            Comp             : out std_logic;
41                            SA               : out std_logic;
42                            SB               : out std_logic;
43                            EO               : out std_logic_vector(7 downto 0);
44                            MA, MB           : out std_logic_vector(27 downto 0));
45   end component;
46
47   ------------------------------------------------------------------------------ MUX/DEMUX
48   component mux_ns port (NorA, NorB, MixA, MixB : in std_logic_vector(36 downto 0);
49                          e_data                 : in std_logic_vector(1 downto 0);
50                          NA, NB                 : out std_logic_vector(36 downto 0));
51   end component;
52
53   component demux port (NumberA, NumberB             : in std_logic_vector(36 downto 0);
54                         e_data                       : in std_logic_vector(1 downto 0);
55                         NA0, NB0, NA1, NB1, NA2, NB2 : out std_logic_vector(36 downto 0));
56   end component;
57
58   component mux_adder port (NorSA, NorSB, SubSA, SubSB : in std_logic;
59                             CompN, CompS               : in std_logic;
60                             NorE, SubE                 : in std_logic_vector(7 downto 0);
61                             NorMA, NorMB, SubMA, SubMB : in std_logic_vector(27 downto 0);
62                             e_data                     : in std_logic_vector(1 downto 0);
63                             SA, SB, C                  : out std_logic;
64                             E                          : out std_logic_vector(7 downto 0);
65                             A, B                       : out std_logic_vector(27 downto 0));
66   end component;
67
68   ------------------------------------------------------------------------------ Mixed Numbers
69   component norm port (NumberA, NumberB : in std_logic_vector(36 downto 0);
70                        MA, MB           : out std_logic_vector(36 downto 0));
71   end component;
72
73   ------------------------------------------------------------------------------ Subnormal Numbers
74   component n_subn port (NumberA, NumberB : in std_logic_vector(36 downto 0);
75                          Comp             : out std_logic;
76                          SA               : out std_logic;
```

```
77                          SB              : out std_logic;
78                          EO              : out std_logic_vector(7 downto 0);
79                          MA, MB          : out std_logic_vector(27 downto 0));
80      end component;
81
82      ------------------------------------------------------------------------------- Selector
83      component selector port (NumberA, NumberB : in std_logic_vector(31 downto 0);
84                               enable           : in std_logic;
85                               e_data           : out std_logic_vector(1 downto 0);
86                               NA, NB           : out std_logic_vector(36 downto 0));
87      end component;
88
89      signal NA_out_select, NB_out_select : std_logic_vector(36 downto 0);
90      signal A_sub, B_sub : std_logic_vector(36 downto 0);
91      signal A_nor, B_nor : std_logic_vector(36 downto 0);
92      signal A_mix, B_mix : std_logic_vector(36 downto 0);
93      signal MixAaux, MixBaux : std_logic_vector(36 downto 0);
94      signal Amux, Bmux : std_logic_vector(36 downto 0);
95      signal SAnor, SBnor, SAsub, SBsub, NComp, SComp : std_logic;
96      signal Enor, Esub : std_logic_vector(7 downto 0);
97      signal MAnor, MBnor, MAsub, MBsub : std_logic_vector(27 downto 0);
98
99      signal edata : std_logic_vector(1 downto 0);
100
101     begin
102
103     comp0 : n_normal
104           port map (NumberA => Amux, NumberB => Bmux,
105                     Comp => NComp, SA => SAnor, SB => SBnor, EO => Enor, MA => MAnor, MB => MBnor);
106
107     comp1 : n_subn
108           port map (NumberA => A_sub, NumberB => B_sub,
109                     Comp => SComp, SA => SAsub, SB => SBsub, EO => Esub, MA => MAsub, MB => MBsub);
110
111     comp2 : norm
112           port map (NumberA => A_mix, NumberB => B_mix,
113                     MA => MixAaux, MB => MixBaux);
114
115   comp3 : demux
116         port map (NumberA => NA_out_select, NumberB => NB_out_select, e_data => edata,
117                   NA0 => A_sub, NB0 => B_sub, NA1 => A_nor, NB1 => B_nor, NA2 => A_mix, NB2 => B_mix);
118
119   comp4 : mux_ns
120         port map (NorA => A_nor, NorB => B_nor, MixA => MixAaux, MixB => MixBaux, e_data => edata,
121                   NA => Amux, NB => Bmux);
122
123   comp5 : selector
124         port map (NumberA => NumberA, NumberB => NumberB, enable => enable,
125                   e_data => edata, NA => NA_out_select, NB => NB_out_select);
126
127   comp6 : mux_adder
128         port map (NorSA => SAnor, NorSB => SBnor, SubSA => SAsub, SubSB => SBsub, CompN => NComp, CompS => SComp,
129                   NorE => Enor, SubE => Esub, NorMA => MAnor, NorMB => MBnor, SubMA => MAsub, SubMB => MBsub,
130                   e_data => edata, SA => SA, SB => SB, C => C, E => Eout, A => MAout, B => MBout);
131
132 end behavioral;
```

## ADDER BLOCK: *Signout* Block

```
1   ----------------------------------------------------------------------------------------
2   -- Floating point adder (32 bits)
3   ----------------------------------------------------------------------------------------
4   -- Block 03 --> Adder
5   -- Description: Prepare normal numbers for addition or subtraction operation
6   --
7   -- Signout: Calculate the B's and output's sign because of addition or subtraction
8   ----------------------------------------------------------------------------------------
9
10  library IEEE;
11  use IEEE.STD_LOGIC_1164.ALL;
12  use IEEE.STD_LOGIC_ARITH.ALL;
13  use IEEE.STD_LOGIC_UNSIGNED.ALL;
14
15  ----------------------------------------------------------------------------------------
16  -- Entity declaration
17  ----------------------------------------------------------------------------------------
18
19  entity signout is
20      Port (    SA     : in std_logic;                          -- Sign A
21                SB     : in std_logic;                          -- Sign B
22                A      : in std_logic_vector(27 downto 0);      -- Number A
23                B      : in std_logic_vector(27 downto 0);      -- Number B
24                A_S    : in std_logic;                          -- Add (0) or Sub (1)
25                Comp   : in std_logic;                          -- Determine largest number
26                Aa     : out std_logic_vector(27 downto 0);     -- Number A'
27                Bb     : out std_logic_vector(27 downto 0);     -- Number B'
28                AS     : out std_logic;                         -- A_S'
29                SO     : out std_logic);                        -- Determine Output's sign
30  end signout;
31
32  ----------------------------------------------------------------------------------------
33  -- Architecture description
34  ----------------------------------------------------------------------------------------
35
36  architecture behavioral of signout is
37
38  -- Signals declaration
39  ----------------------------------------------------------------------------------------
40
41    signal SB_aux : std_logic;
42    signal Aaux, Baux  : std_logic_vector(27 downto 0);
43
44    begin
45
46      SB_aux <= SB xor A_S;                  -- Sign B because of the operation
47
48      SO <= SA when Comp = '1' else          -- A > B --> Sign A
49            SB_aux when Comp = '0' else      -- B > A --> Sign B
50            '-';
51
52      AS <= '1' when SA /= SB_aux else       -- Complement to 1 is needed when
53            '0';                             -- the signs are different
54
55      Aaux <= A when Comp = '1' else
56              B when Comp = '0' else
57              "----------------------------";
58
59      Baux <= B when Comp = '1' else
60              A when Comp = '0' else
61              "----------------------------";
62
63      process (SA, SB_aux, Aaux, Baux)
64        begin
65        ----------------------- if Sign A is equal to Sign B
66          if (SA xor SB_aux) = '0' then
67            Aa <= Aaux;                     -- Nothing changes
68            Bb <= Baux;
69        ----------------------- if Sign A is 1 and Sign B is 0
70          elsif SA = '1' and SB_aux = '0' then
71            Aa <= Baux;                     -- A is changed by B
72            Bb <= Aaux;
```

```
73
74        ----------------------- if Sign A is 0 and Sign B is 1
75          elsif SA = '0' and SB_aux = '1' then
76            Aa <= Aaux;                       -- Nothing changes
77            Bb <= Baux;
78          else
79            Aa <= "----------------------------";
80            Bb <= "----------------------------";
81          end if;
82      end process;
83
84    end behavioral;
```

## ADDER BLOCK: *Adder* Block: CLA Entity

```
1   ----------------------------------------------------------------------------------------
2   -- Adder Carry LookAhead
3   ----------------------------------------------------------------------------------------
4   library IEEE;
5   use IEEE.STD_LOGIC_1164.ALL;
6   use IEEE.STD_LOGIC_ARITH.ALL;
7   use IEEE.STD_LOGIC_UNSIGNED.ALL;
8
9   ----------------------------------------------------------------------------------------
10  -- Entity declaration
11  ----------------------------------------------------------------------------------------
12  entity CLA is port (
13      A, B, Cin    : in std_logic;
14      S, Cout      : out std_logic);
15  end CLA;
16
17  ----------------------------------------------------------------------------------------
18  -- Architecture description
19  ----------------------------------------------------------------------------------------
20  architecture behavioral of CLA is
21
22  -- Signals declaration
23  ----------------------------------------------------------------------------------------
24    signal c_g, c_p : std_logic;
25
26  begin
27
28    c_g <= A and B;                -- Carry generation
29    c_p <= A xor B;                -- Carry propagation
30
31    Cout <= c_g or (c_p and Cin);  -- Carry out
32    S <= c_p xor Cin;              -- Bit's sum
33
34  end behavioral;
```

## ADDER BLOCK: *Adder* Block

```
1   ----------------------------------------------------------------------------------------
2   -- Floating point adder (32 bits)
3   ----------------------------------------------------------------------------------------
4   -- Block 03 --> Adder
5   -- Description: Implement the addition with a CLA adder
6   ----------------------------------------------------------------------------------------
7
8   library IEEE;
9   use IEEE.STD_LOGIC_1164.ALL;
10  use IEEE.STD_LOGIC_ARITH.ALL;
11  use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13  ----------------------------------------------------------------------------------------
14  -- Entity declaration
15  ----------------------------------------------------------------------------------------
16
17  entity adder is
18      Port (    A     : in std_logic_vector(27 downto 0);     -- Number A
19                B     : in std_logic_vector(27 downto 0);     -- Number B
20                A_S   : in std_logic;                         -- Add(0) / Sub (1)
21                S     : out std_logic_vector(27 downto 0);    -- Output
22                Co    : out std_logic);                       -- Carry out
23  end adder;
24
25  ----------------------------------------------------------------------------------------
26  -- Architecture description
27  ----------------------------------------------------------------------------------------
28
29  architecture behavioral of adder is
30
31  -- Signals and components declaration
32  ----------------------------------------------------------------------------------------
33
34    component CLA port (A, B, Cin : in std_logic; S, Cout : out std_logic); end component;
35
36    signal B1, aux, S_aux   : std_logic_vector(27 downto 0);
37
38  begin
39
40  -- Components generation
41
42  Comp1: for i in 0 to 27 generate
43
44    B1(i) <= B(i) xor A_S;
45
46    sumador_0: if (i=0) generate
47               sumador_0comp: CLA port map (A => A(i), B => B1(i), Cin => A_S, S => S_aux(i), Cout => aux(i));
48               end generate;
49    sumador_i: if ((i>0) and (i<28)) generate
50               sumador_icomp: CLA port map (A => A(i), B => B1(i), Cin => aux(i-1), S => S_aux(i), Cout => aux(i));
51               end generate;
52
53  end generate;
54
55  S <= S_aux;
56
57  Co <= aux(27);
58
59
60  end behavioral;
```

## ADDER BLOCK: *Block_Adder* Block

```
1   ----------------------------------------------------------------------------------------------
2   -- Floating point adder (32 bits)
3   ----------------------------------------------------------------------------------------------
4   -- Block 03 --> Adder
5   -- Description: Implement the addition with a CLA adder
6   ----------------------------------------------------------------------------------------------
7
8   library IEEE;
9   use IEEE.STD_LOGIC_1164.ALL;
10  use IEEE.STD_LOGIC_ARITH.ALL;
11  use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13  ----------------------------------------------------------------------------------------------
14  -- Entity declaration
15  ----------------------------------------------------------------------------------------------
16
17  entity block_adder is
18      Port (    SA    : in std_logic;                          -- Sign A
19                SB    : in std_logic;                          -- Sign B
20                A     : in std_logic_vector(27 downto 0);      -- Number A
21                B     : in std_logic_vector(27 downto 0);      -- Number B
22                A_S   : in std_logic;                          -- Add(0) / Sub (1)
23                Comp  : in std_logic;                          -- Comparison
24                S     : out std_logic_vector(27 downto 0);     -- Output
25                SO    : out std_logic;                         -- Output's sign
26                Co    : out std_logic);                        -- Carry out
27  end block_adder;
28
29  ----------------------------------------------------------------------------------------------
30  -- Architecture description
31  ----------------------------------------------------------------------------------------------
32
33  architecture behavioral of block_adder is
34
35  -- Signals and components declaration
36  ----------------------------------------------------------------------------------------------
37
38    component adder port (A, B : in std_logic_vector(27 downto 0);
39                          A_S  : in std_logic;
40                          S    : out std_logic_vector(27 downto 0);
41                          Co   : out std_logic);
42    end component;
43
44
45    component signout port (SA, SB    : in std_logic;
46                            A, B      : in std_logic_vector(27 downto 0);
47                            A_S, Comp : in std_logic;
48                            Aa, Bb    : out std_logic_vector(27 downto 0);
49                            AS, SO    : out std_logic);
50    end component;
51
52    signal Aa_aux, Bb_aux, S_aux  : std_logic_vector(27 downto 0);
53    signal AS_aux, SO_aux, Co_aux : std_logic;
54
55    begin
56
57    component00: signout port map (SA => SA, SB => SB, A => A, B => B, A_S => A_S, Comp => Comp,
58                                   Aa => Aa_aux, Bb => Bb_aux, AS => AS_aux, SO => SO_aux);
59
60    component01: adder port map (A => Aa_aux, B => Bb_aux, A_S => AS_aux, S => S_aux, Co => Co_aux);
61
62    --------- If a complement to 1 is used and Output's sign is 1 a C2 is needed
63    S  <= (S_aux xor X"FFFFFFF")+'1' when ((AS_aux and SO_aux) = '1') else
64          S_aux;
65
66    Co <= '0' when ((SB xor A_S) /= SA) else
67          Co_aux;
68
69    SO <= SO_aux;
70
71  end behavioral;
```

## STANDARDIZING BLOCK: *Round* Block

```
----------------------------------------------------------------------------------------
-- Floating point adder (32 bits)
----------------------------------------------------------------------------------------
-- Block 04 --> Normalize
-- Description: Normalize the result
--
-- round: Round the result deleting the guard bits
----------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------
-- Entity declaration
----------------------------------------------------------------------------------------

entity round is
    Port (    Min    : in std_logic_vector(27 downto 0);    -- Input's mantissa
              Mout   : out std_logic_vector(22 downto 0));  -- Output's mantissa
end round;

----------------------------------------------------------------------------------------
-- Architecture description
----------------------------------------------------------------------------------------

architecture behavioral of round is

-- Signals and components declaration
----------------------------------------------------------------------------------------

  signal M_aux : std_logic_vector(22 downto 0);

  begin

  process(Min)
    begin

      if Min(3 downto 0) = "----" then
        M_aux <= "-----------------------";
      elsif Min(3 downto 0) >= "1000" then     -- Round Mantissa
        M_aux <= Min(26 downto 4) + '1';
      else
        M_aux <= Min(26 downto 4);
      end if;
    end process;

    Mout <= M_aux;

end behavioral;
```

## STANDARDIZING BLOCK: *Vector* Block

```
----------------------------------------------------------------------------------------
-- Floating point adder (32 bits)
----------------------------------------------------------------------------------------
-- Block 04 --> Normalize
-- Description: Normalize the result
--
-- vector: Regroup sign, exponent and mantissa in a single vector
----------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------
-- Entity declaration
----------------------------------------------------------------------------------------
```

```
18
19  entity vector is
20      Port (    S : in std_logic;                          -- Sign
21                E : in std_logic_vector(7 downto 0);       -- Exponent
22                M : in std_logic_vector(22 downto 0);      -- Mantissa
23                N : out std_logic_vector(31 downto 0));    -- Vector
24  end vector;
25
26  ----------------------------------------------------------------------------------------------
27  -- Architecture description
28  ----------------------------------------------------------------------------------------------
29
30  architecture behavioral of vector is
31
32  -- Signals and components declaration
33  ----------------------------------------------------------------------------------------------
34
35  begin
36
37    N(31) <= S;
38    N(30 downto 23) <= E;
39    N(22 downto 0) <= M;
40
41  end behavioral;
```

## STANDARDIZING BLOCK: *Block_norm* Block

```
1   ----------------------------------------------------------------------------------------------
2   -- Floating point adder (32 bits)
3   ----------------------------------------------------------------------------------------------
4   -- Block 04 --> Normalize
5   --    Sub Block 4 --> Normalize the result
6   ----------------------------------------------------------------------------------------------
7
8   library IEEE;
9   use IEEE.STD_LOGIC_1164.ALL;
10  use IEEE.STD_LOGIC_ARITH.ALL;
11  use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13  ----------------------------------------------------------------------------------------------
14  -- Entity declaration
15  ----------------------------------------------------------------------------------------------
16
17  entity block_norm is
18      Port (    MS : in std_logic_vector(27 downto 0);     -- Number S
19                ES : in std_logic_vector(7 downto 0);      -- Exponent S
20                Co : in std_logic;                         -- Carry out
21                M  : out std_logic_vector(22 downto 0);    -- Output's Mantissa
22                E  : out std_logic_vector(7 downto 0));    -- Output's Exponent
23  end block_norm;
24
25  ----------------------------------------------------------------------------------------------
26  -- Architecture description
27  ----------------------------------------------------------------------------------------------
28
29  architecture behavioral of block_norm is
30
31  -- Signals and components declaration
32  ----------------------------------------------------------------------------------------------
33
34    component zero port (T : in std_logic_vector(27 downto 0); Zcount : out std_logic_vector(4 downto 0));
35    end component;
36
37
38    component shift_left port (T    : in std_logic_vector(27 downto 0);
39                               Shft : in std_logic_vector(4 downto 0);
40                               S    : out std_logic_vector(27 downto 0));
41    end component;
42
43    component round port (Min : in std_logic_vector(27 downto 0);
44                          Mout : out std_logic_vector(22 downto 0));
45    end component;
46
47    signal Zcount_aux, Shift : std_logic_vector(4 downto 0);
48    signal Number            : std_logic_vector(27 downto 0);
```

```
                                    _     _
49
50    begin
51
52    ------------------------------------------------ Components declaration
53    comp0 : zero
54        port map (T => MS, Zcount => Zcount_aux);
55
56    comp1 : shift_left
57        port map (T => MS, shft => Shift, S => Number);
58
59    comp2 : round
60        port map (Min => Number, Mout => M);
61
62    ------------------------------------------------ Normal or Subnormal Number
63                                                  --      & New Wxponent
64    process(MS, ES, Shift, Zcount_aux, Co)
65      begin
66
67        if Zcount_aux = "-----" then
68          Shift <= "-----";
69          E <= "--------";
70        elsif ES > Zcount_aux then         -- If the number is normal...
71          Shift <= Zcount_aux;              -- ... the number is shifted --> Output normal
72          E <= ES - Shift + Co;
73        elsif ES < Zcount_aux then          -- If the number is normal...
74          Shift <= ES(4 downto 0);          -- ... the number is shifted --> Output subnormal
75          E <= X"00";
76        elsif ES = Zcount_aux then          -- If N° Zeros = Exponent...
77          Shift <= Zcount_aux;              -- ... the mantissa is shifted and EO = 1
78          E <= X"01";
79        end if;
80
81      end process;
82
83  end behavioral;
```

## STANDARDIZING BLOCK: *norm+vector* Block

```
1   ----------------------------------------------------------------------------------------
2   -- Floating point adder (32 bits)
3   ----------------------------------------------------------------------------------------
4   -- Block 04 --> Normalize
5   --    Sub Block 4 --> Normalize the result
6   ----------------------------------------------------------------------------------------
7
8   library IEEE;
9   use IEEE.STD_LOGIC_1164.ALL;
10  use IEEE.STD_LOGIC_ARITH.ALL;
11  use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13  ----------------------------------------------------------------------------------------
14  -- Entity declaration
15  ----------------------------------------------------------------------------------------
16
17  entity norm_vector is
18      Port (    SS  : in std_logic;                       -- Sign S
19                MS : in std_logic_vector(27 downto 0);    -- Number S
20                ES : in std_logic_vector(7 downto 0);     -- Exponent S
21                Co : in std_logic;                        -- Carry out
22                N  : out std_logic_vector(31 downto 0));  -- Output Number
23  end norm_vector;
24
25  ----------------------------------------------------------------------------------------
26  -- Architecture description
27  ----------------------------------------------------------------------------------------
28
29  architecture behavioral of norm_vector is
30
31  -- Signals and components declaration
32  ----------------------------------------------------------------------------------------
```

```
33
34    component block_norm port (MS : in std_logic_vector(27 downto 0);
35                               ES : in std_logic_vector(7 downto 0);
36                               Co : in std_logic;
37                               M  : out std_logic_vector(22 downto 0);
38                               E  : out std_logic_vector(7 downto 0));
39    end component;
40
41
42    component vector port (S : in std_logic;
43                           E : in std_logic_vector(7 downto 0);
44                           M : in std_logic_vector(22 downto 0);
45                           N : out std_logic_vector(31 downto 0));
46    end component;
47
48    signal Maux      : std_logic_vector(22 downto 0);
49    signal Eaux      : std_logic_vector(7 downto 0);
50
51    begin
52
53    ------------------------------------------------ Components declaration
54    comp0 : block_norm
55        port map (MS => MS, ES => ES, Co => Co, M => Maux, E => Eaux);
56
57    comp1 : vector
58        port map (S => SS, E => Eaux, M => Maux, N => N);
59
60
61  end behavioral;
```

## 32bit Floating Point Adder BLOCK: MUX

```
1  ----------------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------------
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7  use IEEE.STD_LOGIC_ARITH.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10 ----------------------------------------------------------------------------------------------
11 -- Entity declaration
12 ----------------------------------------------------------------------------------------------
13
14 entity mux_fpadder is
15     Port (    N1     : in std_logic_vector(31 downto 0);   -- N_case number
16               N2     : in std_logic_vector(31 downto 0);   -- Adder number
17               enable : in std_logic;                       -- enable
18               Result : out std_logic_vector(31 downto 0)); -- Result
19 end mux_fpadder;
20
21 ----------------------------------------------------------------------------------------------
22 -- Architecture description
23 ----------------------------------------------------------------------------------------------
24
25 architecture behavioral of mux_fpadder is
26
27   begin
28
29     Result <= N1 when enable = '0' else                  -- N_case number
30               N2 when enable = '1' else                  -- Adder number
31           "--------------------------------";
32
33   end behavioral;
```

## 32bit Floating Point Adder BLOCK:

```
1  ----------------------------------------------------------------------------------------------
2  -- Floating point adder (32 bits)
3  ----------------------------------------------------------------------------------------------
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7  use IEEE.STD_LOGIC_ARITH.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10 ----------------------------------------------------------------------------------------------
11 -- Entity declaration
12 ----------------------------------------------------------------------------------------------
13
14 entity fpadder is
15     Port (    NumberA : in std_logic_vector(31 downto 0);   -- Number A
16               NumberB : in std_logic_vector(31 downto 0);   -- Number B
17               A_S     : in std_logic;                       -- Add / Sub
18               Result  : out std_logic_vector(31 downto 0)); -- Result
19 end fpadder;
20
21 ----------------------------------------------------------------------------------------------
22 -- Architecture description
23 ----------------------------------------------------------------------------------------------
24
25 architecture behavioral of fpadder is
26
27 -- Signals and components declaration
28 ----------------------------------------------------------------------------------------------
29
30   ------------------------------------------------------------------------------- n_case
31
32   component n_case port (NumberA, NumberB : in std_logic_vector(31 downto 0);
33                          enable           : out std_logic;
34                          S                : out std_logic_vector(31 downto 0));
35   end component;
```

```
36
37    ------------------------------------------------------------------------------- Pre-Adder
38
39    component preadder port (NumberA, NumberB : in std_logic_vector(31 downto 0);
40                             enable           : in std_logic;
41                             SA, SB           : out std_logic;
42                             C                : out std_logic;
43                             EOut             : out std_logic_vector(7 downto 0);
44                             MAOut, MBOut     : out std_logic_vector(27 downto 0));
45    end component;
46
47    ------------------------------------------------------------------------------- Adder
48
49    component block_adder port (SA, SB : in std_logic;
50                                A, B   : in std_logic_vector(27 downto 0);
51                                A_S    : in std_logic;
52                                Comp   : in std_logic;
53                                S      : out std_logic_vector(27 downto 0);
54                                SO     : out std_logic;
55                                Co     : out std_logic);
56    end component;
57
58    component norm_vector port (SS : in std_logic;
59                                MS : in std_logic_vector(27 downto 0);
60                                ES : in std_logic_vector(7 downto 0);
61                                Co : in std_logic;
62                                N  : out std_logic_vector(31 downto 0));
63    end component;
64
65     ------------------------------------------------------------------------------- Mux_fpadder
66
67     component mux_fpadder port (N1, N2  : in std_logic_vector(31 downto 0);
68                                 enable  : in std_logic;
69                                 Result : out std_logic_vector(31 downto 0));
70     end component;
71
72   signal MA_aux, MB_aux, MOut_aux : std_logic_vector(27 downto 0);
73   signal EOut_aux : std_logic_vector(7 downto 0);
74   signal Comp_aux, Carry : std_logic;
75   signal SA_aux, SB_aux, S_aux : std_logic;
76   signal enable_aux : std_logic;
77   signal Ncase, Nadder : std_logic_vector(31 downto 0);
78
79   begin
80
81   comp0 : preadder
82         port map (NumberA => NumberA, NumberB => NumberB, enable => enable_aux,
83                   SA => SA_aux, SB => SB_aux, C => Comp_aux, EOut => EOut_aux, MAOut => MA_aux, MBOut => MB_aux);
84
85   comp1 : block_adder
86         port map (SA => SA_aux, SB => SB_aux, A => MA_aux, B => MB_aux, A_S => A_S, Comp => Comp_aux,
87                   S => MOut_aux, SO => S_aux, Co => Carry);
88
89   comp2 : norm_vector
90         port map (SS => S_aux, MS => MOut_aux, ES => EOut_aux, Co => Carry,
91                   N => Nadder);
92
93   comp3 : mux_fpadder
94         port map (N1 => Ncase, N2 => Nadder, enable => enable_aux,
95                   Result => Result);
96
97   comp4 : n_case
98         port map (NumberA => NumberA, NumberB => NumberB,
99                   enable => enable_aux, S => Ncase);
100
101 end behavioral;
```