Design of a floating point Arithmetical-Logic unit

Multiplication & Division

Ceuca Laura

3rd year Computer Science

Group 30433

Structure of Computer Systems project

Table of contains:

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2. Project Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

3. Analysis and Theoretical Foundation. . . . . . . . . . . . . . . . . . . . . . . . . . 6

4. Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

5. Testing and Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

6. Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

7. Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

Chapter 1: Introduction

Floating-point arithmetic: overview

Floating-point arithmetic uses the representation of real numbers in a formulaic manner as an approximation to support a trade-off between range and precision. Floating-point computation is often found in systems that use very small or very large real value, and require fast processing times. A number is, in general, represented approximately to a fixed number of significant digits and scaled using an exponent in some fixed base. A number that can be represented exactly is of the following form:

Number = significant x baseexponent

Floating-point representation offers the advantage of being able to represent in the same system different orders of magnitude, with a fixed number of digits. This dynamic ranged is owed to the fact that the numbers that can be represented are not uniformly spaced, since the difference between two consecutive representable numbers grows with the chosen scale.

For the past years, floating-point representation and arithmetic have been extensively used in computers, represented in varying formats, but since the 1990s, the most commonly encountered representations are those defined by the IEEE standard.

The speed of floating-point operations, commonly measured in terms of FLOPS, is an important characteristic of a computer system, especially for applications that involve intensive mathematical calculations. A floating-point unit (FPU, colloquially a math coprocessor) is a part of a computer system specially designed to carry out operations on floating-point numbers.

Floating-point representation

Floating-point representation is similar in concept to scientific notation. Logically, a floating-point number consists of:

* A signed digit string of a given length in a given base. This digit string is referred to as the significand, mantissa, or coefficient. The length of the significand determines the precision to which numbers can be represented. The radix point position is usually set just after the most significant digit.
* A signed integer exponent (also referred to as the characteristic, or scale), which modifies the magnitude of the number.

In this project, the IEEE standard representation on 32 bits will be used as follows:

* The base of the exponent will be 2
* The first bit will represent the sign bit
* The exponent will be represented on 8 bits
* The mantissa will be represented in the remaining 23 bits
* The exponent will have an offset of 127 to permit the representation of both negative and positive exponent values

Conversion example:

Let’s take for this example the real number 12.5 and represent it in the IEEE standard 32 bit representation:

The number can be decomposed in base to in: 1.5625 x 23

The exponent is a positive number, so the sign bit will have the value 0.

The exponent is 3, to which we add 127 and get 130, which will be represented on 8 bits as 10000010.

The mantissa is represented by the number 0.5625, represented on 23 bits as 10010000000000000000000.

Putting all of them togheter, we obtain that the final 32 bit representation of 12.5 is:

0 10000010 10010000000000000000000.

This is the representation we are going to use for all operands in this project to implement the multiplication and division of floating-point numbers.

VHDL overview:

VHDL is the abbreviation of VHSIC HDL, which stands for Very High Speed Integrated Circuit Hardware Description Language. It is a language design for the description oh a logic hardware model, i.e. a sequential or combinatoric logic function. It is the most used language for integrated circuit design.

This project will use VHDL for implementing the algorithms that realize the floating point division and multiplication of 32 bit operands.

Also, the operations will be designed and simulated using Xilinx’s Vivado software as IDE.

Chapter 2: Project Objectives

The aim of this project is the implementation of multiplication and division operations between floating-point operands.

The final project will take two 32-bit operands represented in the floating-point IEEE standard (1 bit for sign, 8 bits for exponent, 23 bits for mantissa) and proceed to compute the result of the multiplication and division operations, implementing some algorithms in VHDL. The results will be simulated in Xilinx’s Vivado IDE.

The board used for this project will be Nexys 4 DDR.

To input the numbers we will use as operand for the multiplication and division, we will use the 16 switches that the board possesses. Since the numbers are represented on 32 bits, the input operation will be parted in two. First we input first 16 bits of the number, press a button for confirmation, then enter the last 16 bits of the number.

After both operand have been successfully introduced, a button on the board will be pressed to select the desired operation (multiplication or division).

The result of the operation will be displayed using the 7 segment display of the Nexys board.

The goal is to implement the 32-bit floating point multiplication and division operation and display the result on the board.

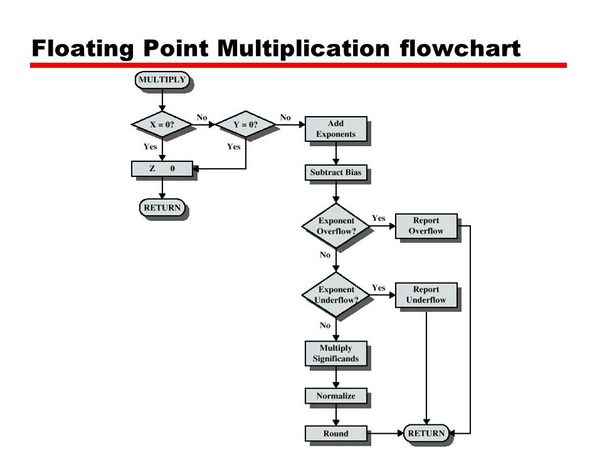
Chapter 3: Analysis and Theoretical Foundation

The numbers will be introduced on the board in the following manner: the first 16 bits introduced (the higher bits of each operands) contain the sign bit (first bit), the 8 bits of the exponent (bits 2 to 9) and the first 7 bits of the mantissa (bits 10 to 16). The next 16 bits (the lower bits of each operand) will represent the remaining 16 bits of the mantissa.

The numbers are introduced in the IEEE standard representation in order to be used in the two floating point arithmetic operations. The output will also be displayed in the floating point representation, using the 7 segment display of the board.

Multiplication Algorithm:

To perform the multiplication of two floating-point operands, we have to divide the numbers in three individual parts: sign, exponent and mantissa and proceed to apply specific computation on each of the parts to obtain the parts that will compose the result of the operation.



[2]

The first step will be to check if any of the operand is zero, in which case no computation is necessary, the result being automatically zero. If none of the operands is zero, then we will compute the result, by performing operations on every part of the operands individually as follows:

Sign bit:

The sign bit of the result is obtained by applying the XOR operation on the sign bits of the operands:

|  |  |  |
| --- | --- | --- |
| First operand’s sign | Second operand’s sign | Result’s sign |
| 1 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |

Exponent:

To compute the exponent resulted from the floating point multiplication operation, we use the pattern of multiplication of numbers with exponents: the exponent of the result is obtained by summing the exponents of the operands.

(a \* bc ) \* (d \* be ) = a \* d \* b(c + e)

Where b represents the base, which will be 2 for the both operands in our case.

Since both exponents are obtained by adding a bias of 127, when summing the exponents, we have to subtract 127 from the sum in order to get an accurate exponent of the result.

Mantissa:

In the case of the mantissa, the result will be obtained by applying a 23 bit integer multiplication algorithm, having as operands the mantissas of the numbers we want to multiply.

To implement the multiplication operations on the physical boards, it has to be performed by means of an algorithm based on addition operations.

Overflow:

We encounter an overflow in case the number is too large to be represented in the chosen format. The overflow is encountered in the case of the exponent, if this is larger than 127.

We can detect an exponent overflow by verifying the 8th bit of the exponent to be 1, while the 7th bit is zero. This means the result of the multiplication is overflowing and it represents + (or -, according to the sign bit) infinity.

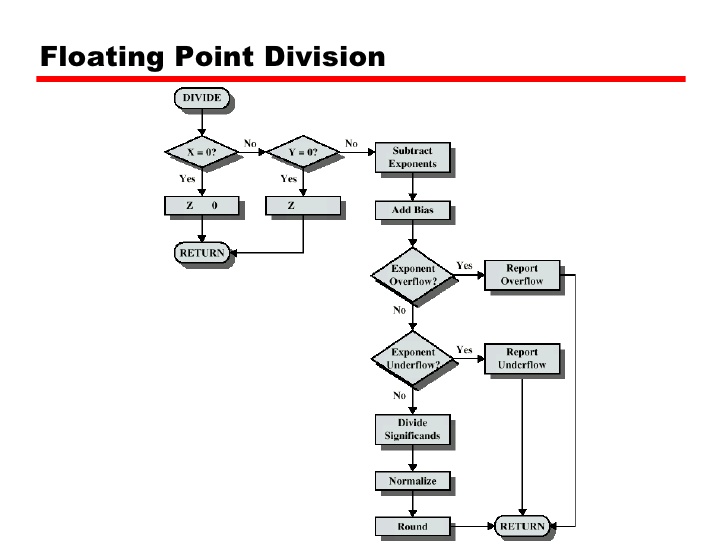
Underflow:

We encounter an overflow in case the number is too small to be represented in the chosen format. The underflow is encountered in the case of the exponent, if this is smaller than -128.

We can detect an exponent underflow by verifying the 8th bit of the exponent to be 1, while the 7th bit is also 1. This means the result of the multiplication is overflowing and it represents + (or -, according to the sign bit) zero.

Division Algorithm:

To perform the division of two floating-point operands, we have to divide the numbers in three individual parts: sign, exponent and mantissa and proceed to apply specific computation on each of the parts to obtain the parts that will compose the result of the operation.

[1]

Sign bit:

The sign bit of the result is obtained by applying the XOR operation on the sign bits of the operands:

|  |  |  |
| --- | --- | --- |
| First operand’s sign | Second operand’s sign | Result’s sign |
| 1 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |

Exponent:

To compute the exponent resulted from the floating point division operation, we use the pattern of division of numbers with exponents: the exponent of the result is obtained by subtracting the exponents of the operands.

(a \* bc ) / (d \* be ) = a \* d \* b(c - e)

Where b represents the base, which will be 2 for the both operands in our case.

Since both operands are obtained by summing a bias of 127, when we perform exponent subtraction, we have to also add 127 to the result of the subtraction in order to get an accurate value of the result exponent.

Mantissa:

In the case of the mantissa, the result will be obtained by applying a 23 bit integer division algorithm, having as operands the mantissas of the numbers we want to divide.

To implement the division operations on the physical boards, it has to be performed by means of a division algorithm.

Overflow:

We encounter an overflow in case the number is too large to be represented in the chosen format. The overflow is encountered in the case of the exponent, if this is larger than 127.

We can detect an exponent overflow by verifying the 8th bit of the exponent to be 1, while the 7th bit is zero. This means the result of the multiplication is overflowing and it represents + (or -, according to the sign bit) infinity.

Underflow:

We encounter an overflow in case the number is too small to be represented in the chosen format. The underflow is encountered in the case of the exponent, if this is smaller than -128.

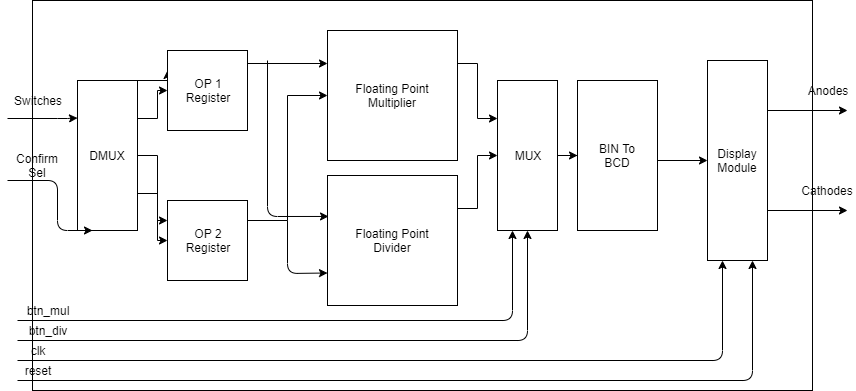
We can detect an exponent underflow by verifying the 8th bit of the exponent to be 1, while the 7th bit is also 1. This means the result of the multiplication is overflowing and it represents + (or -, according to the sign bit) zero.

Chapter 4: Implementation

The implementation of this project was done in VHDL language. The modules that compose the project are the following:

* ALU – main module of the project, contains operand selection logic and result selection logic
* MULTIPLIER – receives 2 32-bit floating point operands and performs the floating point multiplication algorithm on the operands
* DIVIDER – receives 2 32-bit floating point operands and performs the floating point division algorithm on the operands
* BIN\_TO\_BCD – receives the chosen result to be displayed on the BCD 7 Segment. Performs the conversion of the exponent and the mantissa from binary to BCD to be sent to the Display unit
* DISPLAY MODULE – contains the logic to display the BCD result on the 7 segment display, setting the anode and cathode configuration accordingly.

The project structure and interconnection of the modules can be seen in the following illustration:

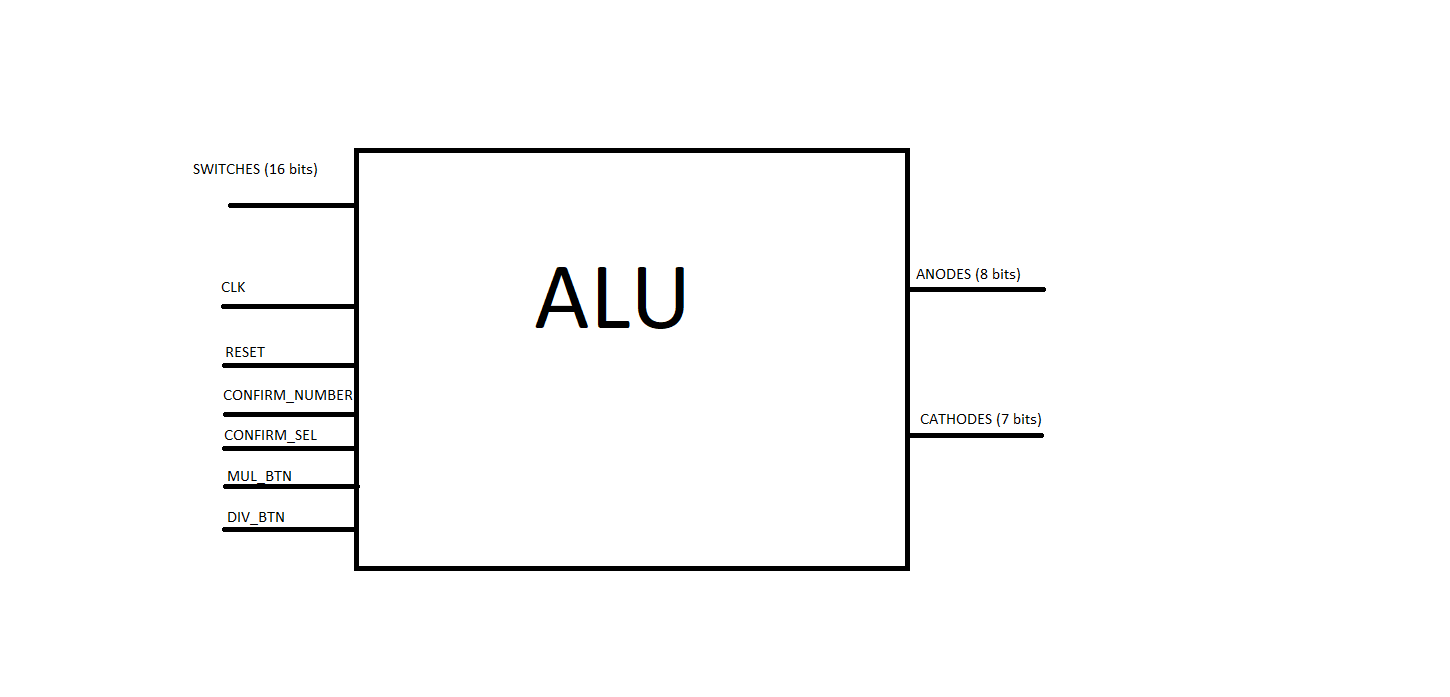


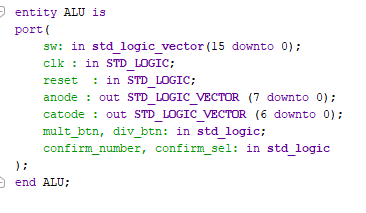
The flow of the project is the following:

1. A 16 bit number is set on the switches of the board
2. The confirm\_button is used so store the number on the switches
3. The selection is set on the switches(1 downto 0)
4. The confirm\_sel button is used to store the chosen selection and to send the saved number in its corresponding place in the operands in a demultiplexer manner
5. After the operands are loaded, the buttons mul\_btn and div\_btn select the desired operation to be performed
6. The result is displayed on the 7 segmen display of the board

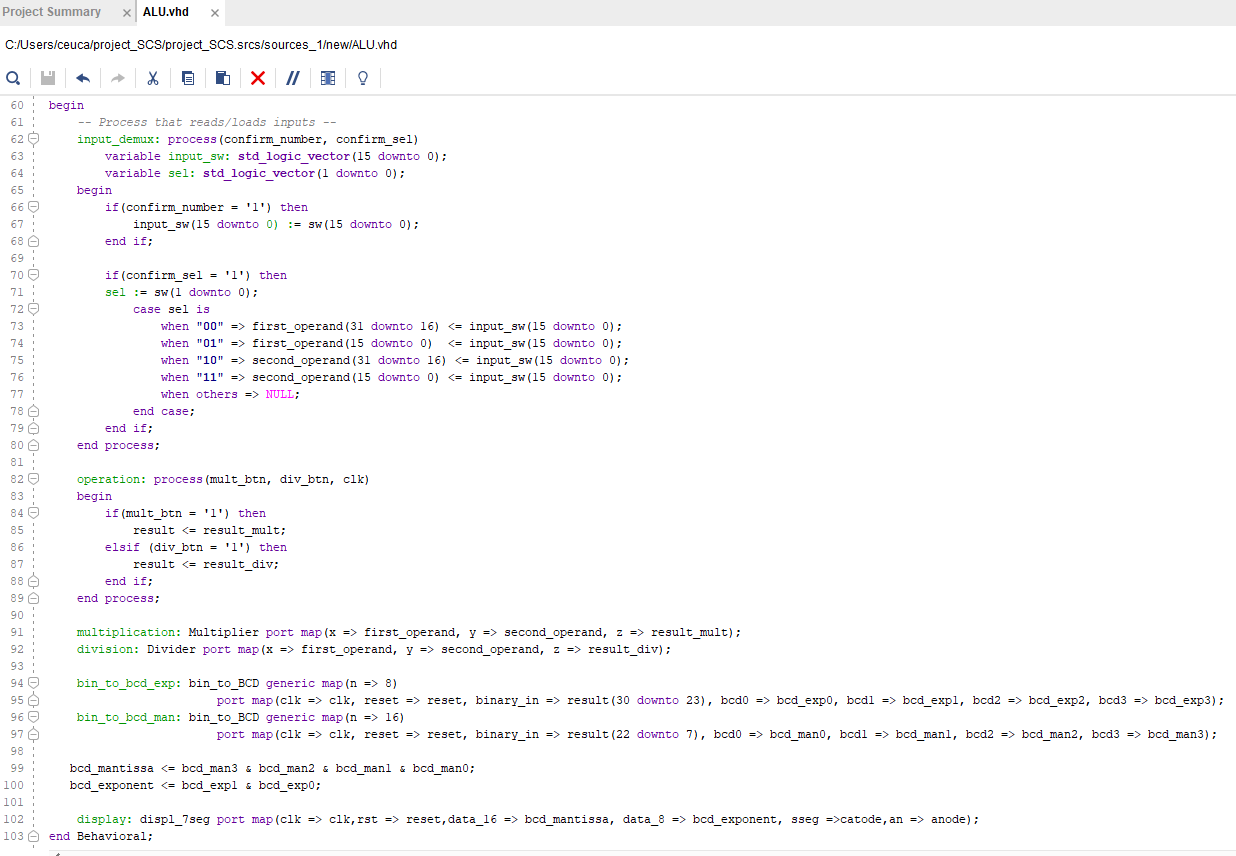
Top Module (ALU)

The ports of the ALU module are as follows, as declared in VHDL:





The Architecture of the ALU module is presented and Explained below:



The first process defines the input demultiplexer, storing the numbers introduced on the switches and, using the selection, placing the number from auxiliary variable in the operands, as follows:

Selection = “00” => number saved in the upper half of the first operand

Selection = “01” =>number saved in the lower half of the first operand

Selection = “10” =>number saved in the upper half of the second operand

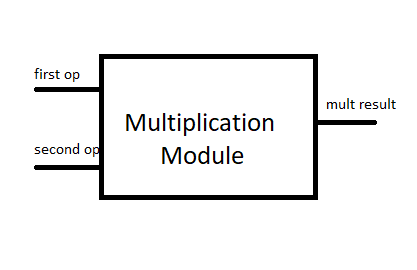
Selection = ”11” =>number saved in the lower half of the second operand

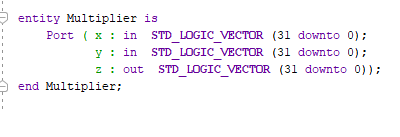
The second process defines the multiplexer that selects the source of the result, either from the multiplication module, or the division module, as selected by the user using the two dedicated buttons for the operations.

The following part of the code is the mapping of the other components of the project according to the scheme presented above. The mantissa and the exponent of the result are constructed from the BCD digits return by the converters and passed to the display unit.

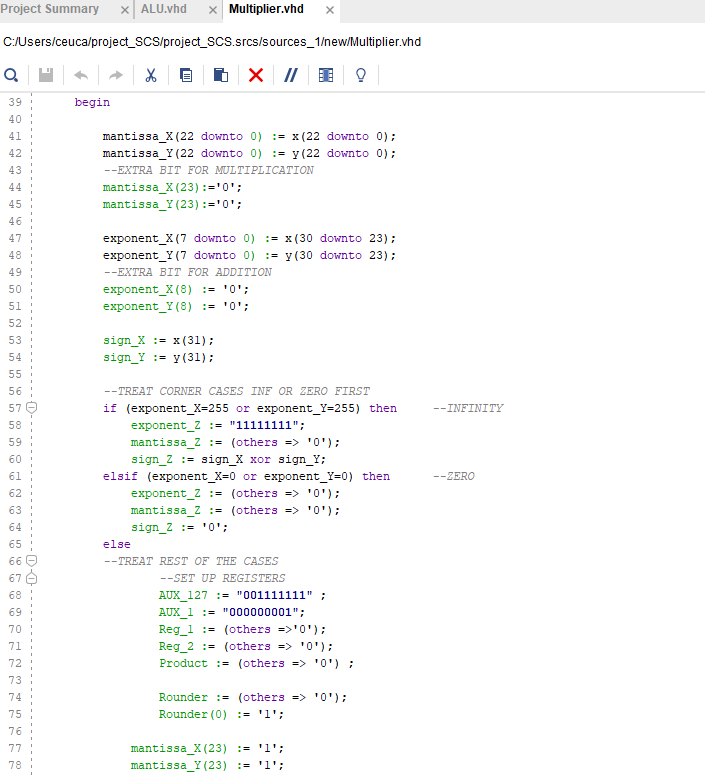
Multiplication Module (Multiplier)

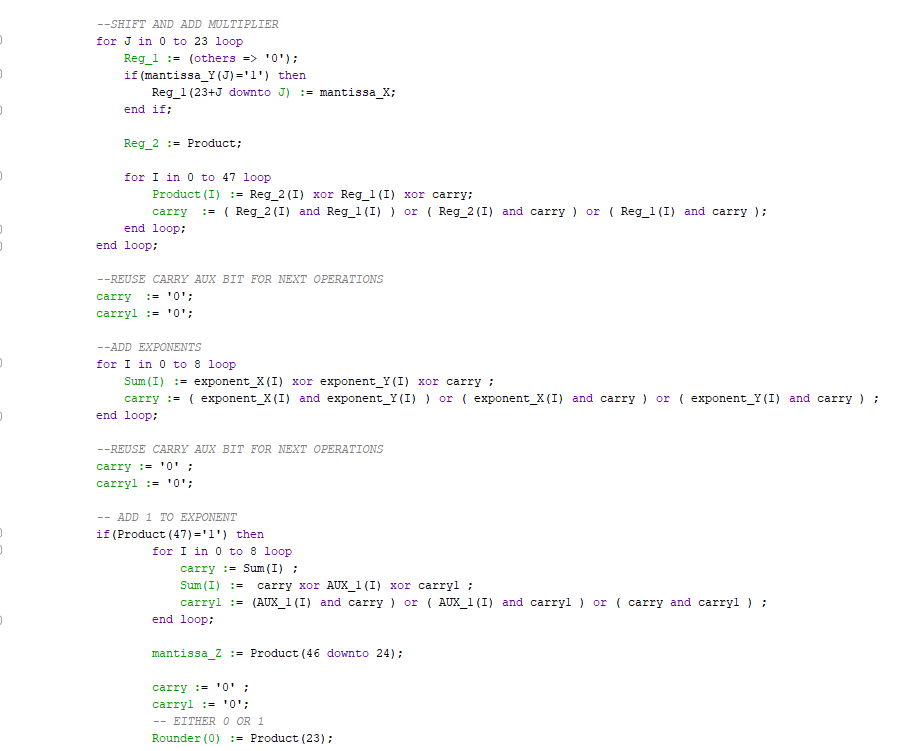
The multiplication module receives two 32 bit operands in the floating point representation as inputs, and outputs a 32 bit floating point number representing the result of the multiplication:

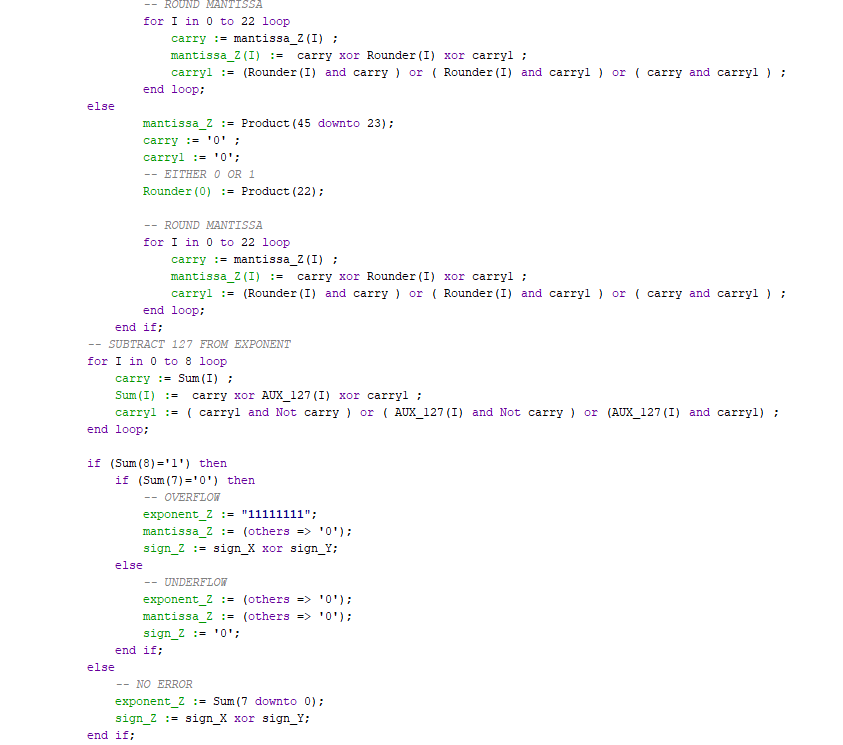




The Architecture of the multiplication module is presented and Explained below:







Firstly, we have to check the exponents of the 2 operands for the following corner cases:

1. If the first or the second operand’s exponent is equal to “11111111” (255) it means we are multiplying with infinity in witch case the result is directly assigned the same value (infinite) and the rest of the algorithm is not performed
2. If the first or the second operand’s exponent is equal to “00000000” (0) it means we are multiplying with zero, in witch case the result is directly assigned the same value (zero) and the rest of the algorithm is not performed.

If no corner case is found, then we prepare the register for performing the algorithm:

The significants (mantissa) of the two operands are multiplied by means of a shift and add multiplier: each bit of an operand is multiplied to the bits of the other operand. The results obtained are all shifted according to their position and everything is added by means of a 1 bit full adder.

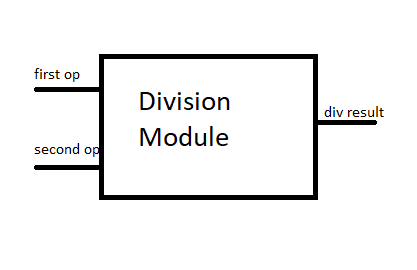
Next, the exponents of the operands are added by means of 1 bit full adders. Since both exponents have an offset of 127 added to them, it has to be subtracted from the result’s exponent in order to get an accurate value.

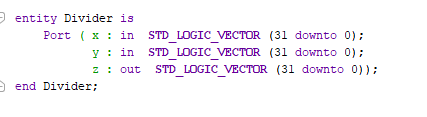
The mantissa of the result is rounded, also making use (behaviorally ) of 1 bit full adders.

Finally, before we pass the computed components of the result to the output of the module, we have to check we are not, in fact, in one of the overflow or underflow cases. To do that, we have to verify the first to bits of the computed exponent sum. If the first bit of the sum is one, then we are in an error case and have to check the second bit to determine the case:

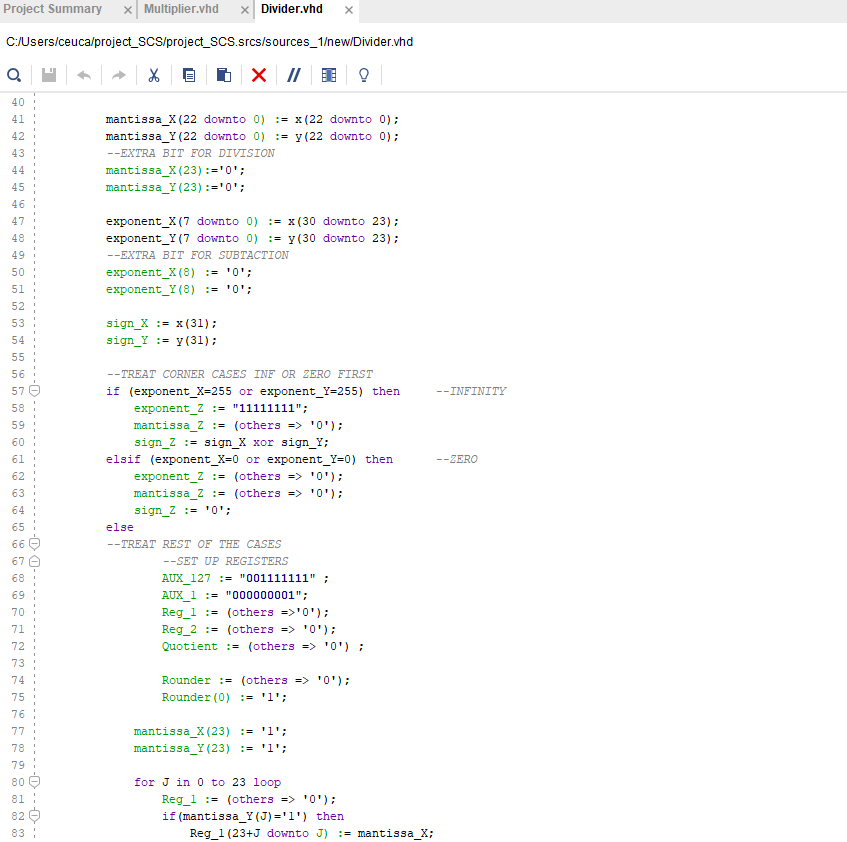
1. If the second bit is 0, we are in the overflow case, that of the result being too large to be represented on the 32 bit floating point representation. The result will be assigned the infinity value, that is the exponent will have the value “11111111”.
2. If the second bit is 1, we are in the underflow case, that of the result being too small to be represented on the 32 bit floating point representation. The result will be assigned the zero value, that is the exponent will have the value “00000000”.

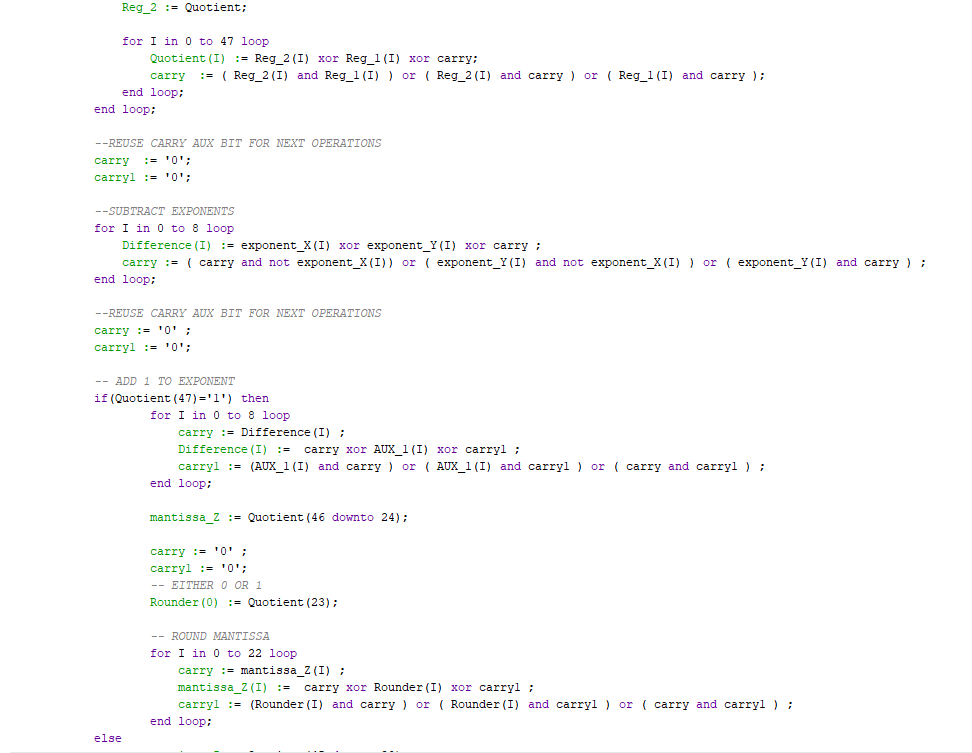
Division Module (Divider)

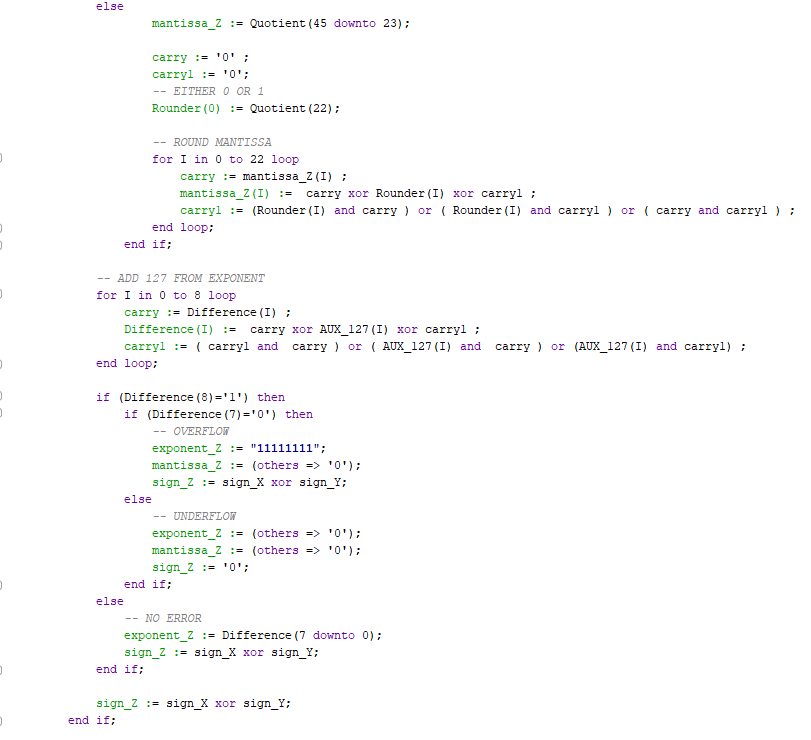
The division module receives two 32 bit operands in the floating point representation as inputs, and outputs a 32 bit floating point number representing the result of the division



The Architecture of the division module is presented and Explained below:







Firstly, we have to check the exponents of the 2 operands for the following corner cases:

1. If the first operand’s exponent is equal to “00000000” (zero) it means we are dividing zero with a number in witch case the result is directly assigned the value “00000000”(zero) and the rest of the algorithm is not performed
2. If the second operand’s exponent is equal to “00000000” (0) it means we are dividing with zero, in witch case the result is directly assigned the value (zero) and we find ourselves in a case of error, so the rest of the algorithm is not performed.

If no corner case is found, then we prepare the register for performing the algorithm:

The significants (mantissa) of the two operands are divided by means of repeated subtraction divider: the second operand is repeatedly subtracted from the first one and a counter is incremented, that will give the quotient of the division.

Next, the exponents of the operands are subtracted by means of 1 bit subtractors. Since both exponents have an offset of 127 added to them, that will be subtracted completely, it has to be added to the result’s exponent in order to get an accurate value.

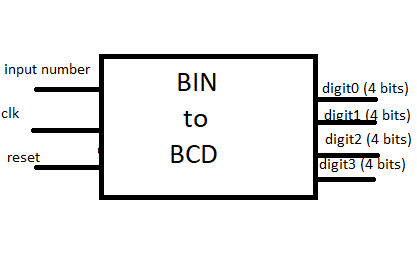
The mantissa of the result is rounded, also making use (behaviorally ) of 1 bit full adders.

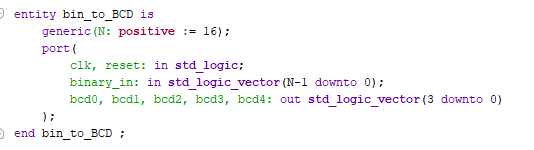
Finally, before we pass the computed components of the result to the output of the module, we have to check we are not, in fact, in one of the overflow or underflow cases. To do that, we have to verify the first to bits of the computed exponent sum. If the first bit of the sum is one, then we are in an error case and have to check the second bit to determine the case:

1. If the second bit is 0, we are in the overflow case, that of the result being too large to be represented on the 32 bit floating point representation. The result will be assigned the infinity value, that is the exponent will have the value “11111111”.
2. If the second bit is 1, we are in the underflow case, that of the result being too small to be represented on the 32 bit floating point representation. The result will be assigned the zero value, that is the exponent will have the value “00000000”.

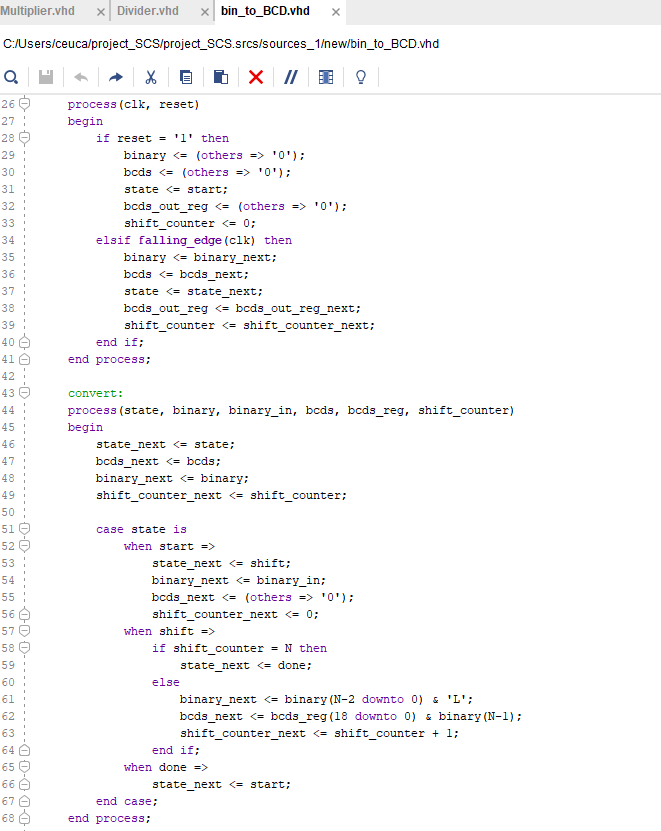
Conversion Module (Bin to BCD)

The conversion module receives as input a number on n bits (generic number of bits, since we need an 8 bit and a 16 bit converter) and outputs the number in a binary coded decimal representation, that is easily displayed in the 7 segment display of the board.





The Architecture of the conversion module is presented and Explained below:



The conversion module is structured as a finite state automata, having 2 processes:

The first one handles the reset of the result and the state in case a reset signal is received, as well as the switching of the states in order on each falling edge of the clock, also received as an input.

The second process handles the conversion, with the actions performed in each state of the automata:

In the state labeled start, the input is read and the shift register is prepared for the algorithm.

In the shift state, the actual conversion is performed, using the shifting algorithm to convert from binary to binary coded decimal. The steps described in the code are repeated until we reach the nth iteration of shifting.

The result is then decomposed into the different digits obtained and passed further to the output.

Display Module (display 7 segment)

The display module receives as input two BCD numbers, one on 8 bits representing the exponent of the floating point number, and the second one on 16 bits representing the mantissa of the floating point number. The number will be displayed on the 7 segment displays of the board in the form:

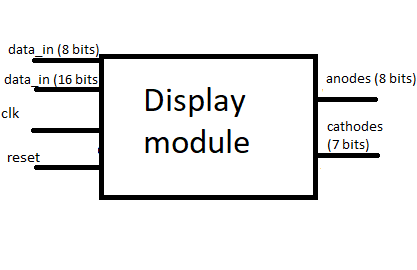
M M M M E X X

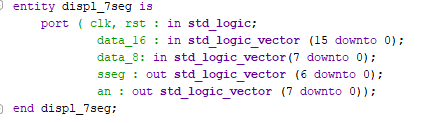
Where:

* The M’s represent the BCD digits of the mantissa;
* E is hardcoded to represent the scientific notation
* The X’s represent the 2 BCD digits of the exponent

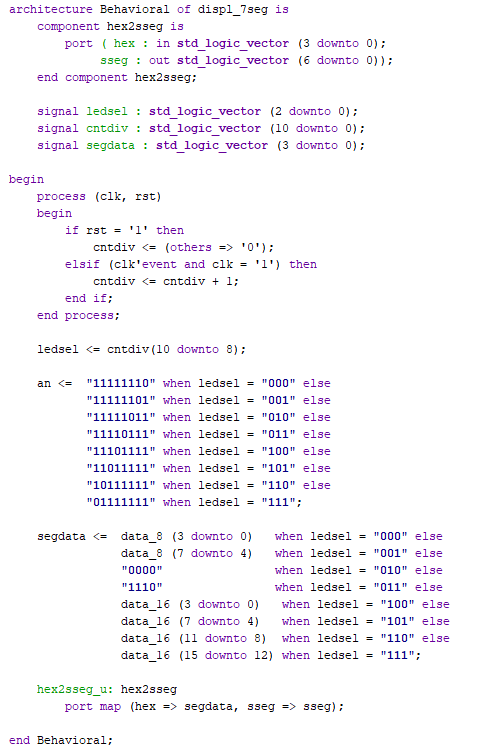
Also, as inputs it receives a clock signal coming from the board, and a reset signal.

The output of the module is represented by the anode and cathode configuration that maps into the board to display the input on the 7 segment leds.





The architecture of the display module is presented below:



This module uses one process for the counter, that we need for switching the anodes of the board fast enough to give the impression that they are all lit at the same time.

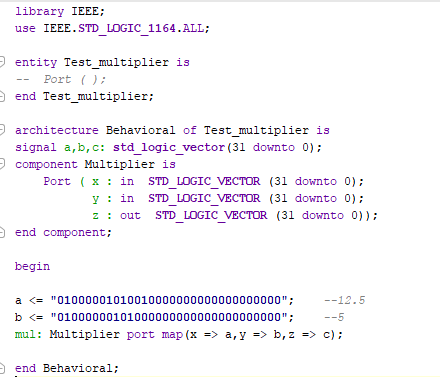
Using the counter, this module maps the anode configuration, and for each anode, the cathode configuration needed to display the data received as input.

Chapter 5: Testing and Results

Testing was carried out both using simulation for each particular operation (modules and sub-modules), as well as on the physical board. The steps and results obtain from simulating the modules is presented below:

Multiplication module

A test bench has been developed to simulate the result of the multiplication. The test bench feeds 2 input values into the module that is instantiated in the test. The result of the module is mapped in the test bench to be available on the simulation diagram. The test bench for multiplication is described below:

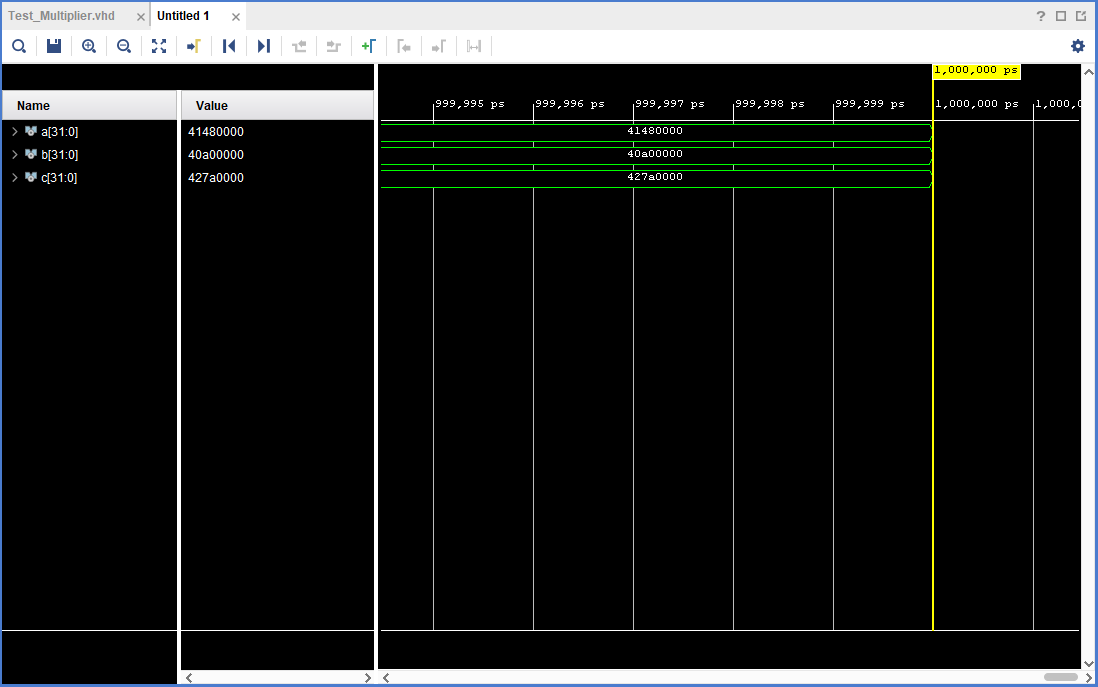


As we can see, the input variable used were:

12.5 represented in the floating point notation as 0 10000010 10010000000000000000000

5 represented in the floating point notation as 0 10000001 01000000000000000000000

The simulation diagram obtained is the following:

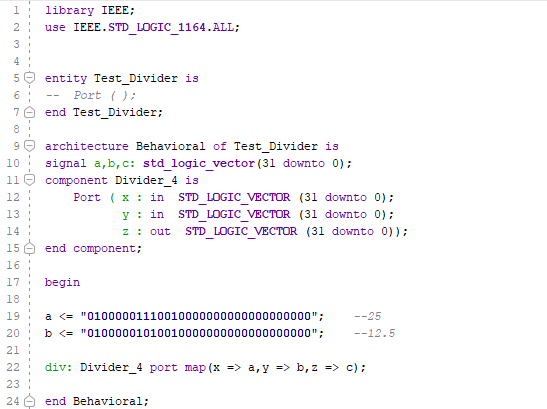


The resulted value of the multiplication module is:

427A0000 HEX = 0 10000100 11110100000000000000000 = 62.5 = 12.5 \* 5

Division module

A test bench has been developed to simulate the result of the division. The test bench feeds 2 input values into the module that is instantiated in the test. The result of the module is mapped in the test bench to be available on the simulation diagram. The test bench for division is described below:

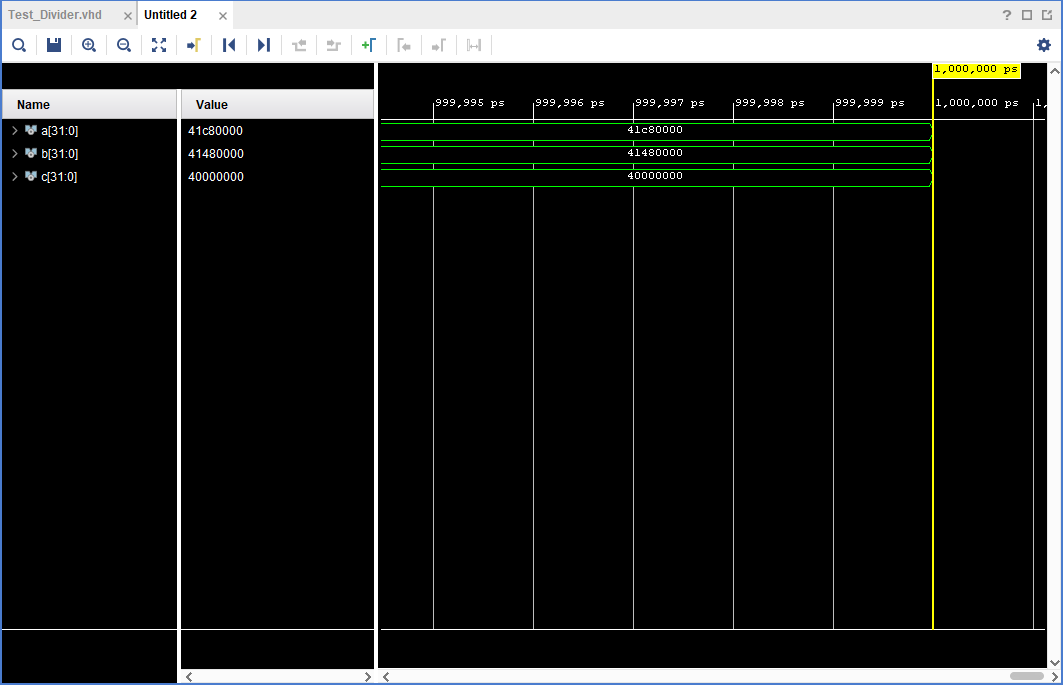


As we can see, the input variable used were:

25 represented in the floating point notation as 0 10000011 10010000000000000000000

12.5 represented in the floating point notation as 0 10000010 10010000000000000000000

The simulation diagram obtained is the following:



The resulted value of the division module is:

40000000 HEX = 0 10000000 00000000000000000000000= 2 = 25 / 12.5

Corner cases

The corner cases treated individually by the algorithm are:

Multiplication with infinity

Multiplication with zero

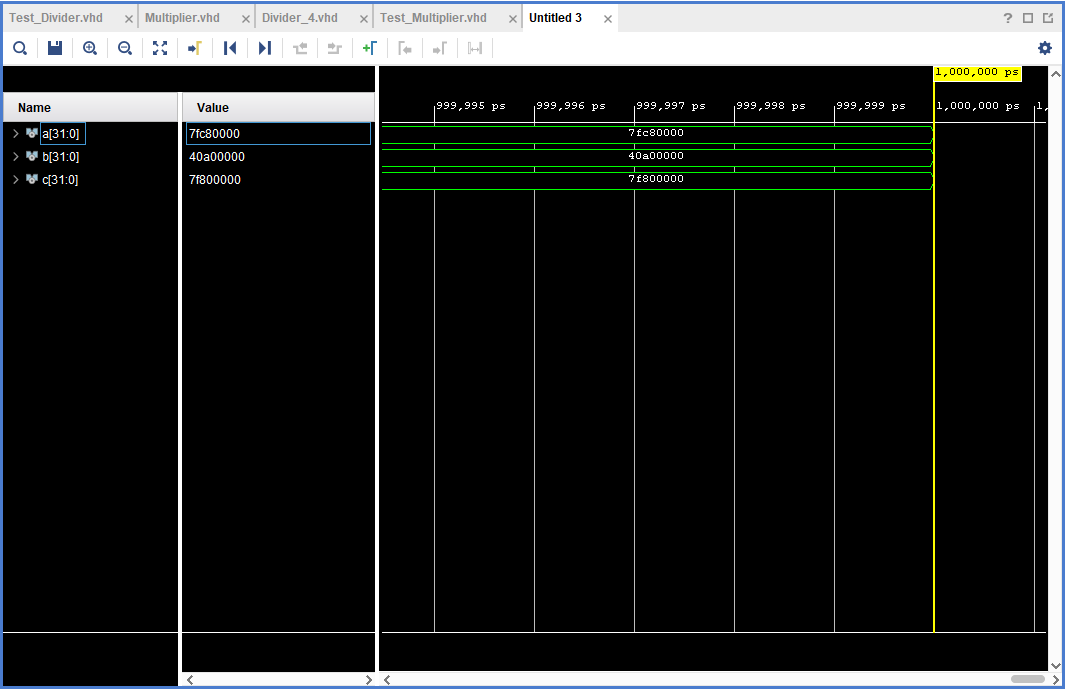
Division of or with infinity

Division of or with zero

Overflow

Underflow

An example of simulating a corner case, namely multiplication with infinity is presented below:



The result is assigned the value of infinity (exponent = 255) and the mantissa is not computed (assigned zero).

Visualizing the project on the FPGA board

The final test was running the project on the Xilinx’s Nexys 4DDR board. The result of the selected operation can be visualized on the 7 segment displays of the board. It needs to be mentioned that only 16 out of the 23 bits of the mantissa could be represented, so I displayed the 16 most significant bits. The exponent is represented biased, on 3 7 segment displays after the one that displays the letter “E”.



Chapter 6: Conclusions

In conclusion, the project was successfully implemented in VHDL, simulated in Vivado Design Suite and ran on the Nexys 4DDR board.

The project consisted in implementing an Arithmetical Logical Unit capable of handling single precision floating point representation, more specifically multiplication and division operations.

A Multiplication and a Division module were implemented that encapsulate the algorithms used to perform the arithmetic operations and that return a result in the same single precision floating point representation as the operands.

The input and output modules were adapted to the limited resources of the board used : only 16 input switches, 5 input buttons, and 8 seven segment displays to output the result.

The final project was tested using simulation for each module performing an operation and the results obtained were verified using a floating point converter. Also, the project was programmed on a Nexys 4 board, in order to be manually tested.

Chapter 7: Bibliography

1. Pictures [1], [2] William Stallings – Computer Organization and Architecture

2. allaboutfpga.com – Binary to BCD coverter

3. SCS laboratory guide – Display module

4. Karen Smoler Miller (University of Wisconsin) – Floating Point Arithmetics