TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Tymofii Chashurin IASM 165505

GENERIC SYNTHESIZABLE FLOATING-POINT UNIT

Master's thesis

Supervisor: Peeter Ellervee
Professor

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Tymofii Chashurin IASM 165505

ÜLDISTATUD SÜNTEESITAV UJUKOMAÜKSUS

Lõputöö liik: magistritöö

Juhendaja: Peeter Ellervee
Professor

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Tymofii Chashurin

15.01.2019

Abstract

A floating-point unit is the essential part of modern computer systems where the carrying out operations with floating-point numbers is involved. Analysis of the existing floating-point unit solutions has shown that many of them are functionally complex and large in terms of the occupied area after synthesis. In addition, the designs that have been discovered have fixed operands' size and, therefore, the number of bits dedicated to the fractional and exponential parts of the floating-point number cannot be changed. Owing to these problems, a decision has been made to develop a floating-point unit in VHDL, which can carry out the simplest mathematical operations while having the possibility of creating a parameterisable design. The operations performed by the floating-point unit are addition, subtraction, multiplication, division and normalization. The parameterisable floating-point unit can be simply reused as a part of other designs, which require it as a separate block.

This thesis is written in English and is 100 pages long, including 5 chapters, 41 figures and 25 tables.

List of abbreviations and terms

FPGA	*Field of Programmable Gate Arrays*
FP	*Floating-Point*
FPU	*Floating-Point Unit*
FPN	*Floating-Point Number*
VHDL	*Very High-Speed Integrated Circuit Hardware Description Language*
FSM	*Finite State Machine*
Generic	*Parameterisable*
IEEE	*Institute of Electrical and Electronics Engineers*
DSP	*Digital Signal Processing*

Table of Contents

List of Figures

List of Tables

1 Introduction

Floating-point arithmetic is an arithmetic that uses formulaic representation of real numbers as an approximation so as to support a trade-off between range and precision [1]. For this reason, floating-point computations are often found in systems, which include quite small and quite large real numbers, which require fast processing times [2] [3]. Carrying out the mathematical operations using the floating-point numbers format provides a wider possible range of results comparing to the fixed point. The limitation of the fixed point format is due to its fixed number of bits, which represent integer and fractional parts of a number. This means that fixed point format is limited in its scope, which restricts the precision and accuracy of the calculation result. A floating-point number is a fractional part of a real number multiplied with its exponent. Using the floating-point arithmetic in calculations provides a wider possible range for the number and, therefore, more precise result, because the precision is defined with the larger number of bits for the mantissa as well as for the exponential part [1]. Using the floating-point format increases the accuracy of the calculations, especially if the operands have approximate values. A great many scientific calculations would be impossible to implement if the floating-point format did not exist. There are several floating-point unit formats: single precision, double precision and extended precision formats. Each of them can provide different precision and accuracy. The more detailed overview of the floating-point formats is given in section 2.3 of this thesis. The explanation about difference between accuracy and precision is provided in part 1.1 along with the problem formulation.

Floating-point calculations can be implemented either in software or hardware. Owing to the resource intensive denormalizing and post-normalizing procedures, the software implementation of the floating-point has certain restrictions. This means if the precision of the floating-point numbers is increased it entails limitations in the processing speed. This is strongly pronounced when the width of operands are larger

then a CPU width. At the same time, a hardware dedicated floating-point unit can bypass this limitation.

A floating-point unit (in further FPU) is the mathematical co-processor of the modern microprocessor devices, either a desktop computer or an embedded system. Modern general purpose computer processors may integrate the FPU within the central processing unit. The FPU can be implemented as a part of a microprocessor or as a separate block. Processors for embedded systems do not always have a dedicated mathematical coprocessor and this depends on their architectures. Typical operations, performed by a mathematical co-processor, are addition, subtraction, multiplication, division, square root and bit shifting. Some previous computer architectures could also execute various transcendental mathematical functions such as exponential or trigonometric calculations. However, in modern processors these are performed with software library routines. This is implemented in order to reduce their power consumption and hardware complexity. Where floating-point calculations have not been provided in hardware, they are performed in software. This technique consumes more processing time but reduces the cost of extra hardware. Co-processors can accelerate the system performance by offloading the processor-intensive tasks from the main processor.

This thesis addresses the generic FPU hardware design, its simulation and testing. In particular, the VHDL generic features for the parameterisable FPU have been considered, because this allows to change the number of bits, which are dedicated for storing the fractional and exponential parts in the FPU operands.

1.1 Problem Formulation and Motivation

The previous section has presented the general overview and aim of using the floating-point numbers format in calculations. As mentioned previously, due to the restrictions of the fixed point format, the usage of floating-point format is highly attractive, because it becomes possible to represent both relatively small as well as relatively large real numbers.

Before the problem formulation such terms as ‘precision’ and ‘accuracy’ need to be defined. Precision specifies the exactness of the real value or, in other words, how many decimal or binary digits have been used to represent this value. Different floating-point formats, which are described in section 2.3, have various fractional and exponential widths. This in turn influences the represented floating-point number’s precision. Accuracy defines how close the real value is to what it is meant to be. That means that if the result of calculations is about its real value, then the result is accurate. Although the two definitions are similar in their meanings and affect each other, they must be distinguished as well. However, since the precision and accuracy often overlap in computer engineering they can have a similar meaning sometimes.

The aim of this thesis is twofold. Firstly, after deep analysis of the existing FPU solutions it has been discovered that there exist a lot of FPU designs that are complex enough and large after being implemented. These designs also occupy a lot of space inside the FPGA chip due to the complexity of functions that consume the hardware resources and affect the synthesis result, although they are fast. Secondly, the discovered FPU designs are not able to be synthesized with the different custom FP format. Having their size as fixed leads to the impossibility of them being parameterisable for the specific synthesis needs. As also has been discovered in [4] [5], sometimes the FPU designs with non-standard 8 bit formats are required. The exponential and fractional widths are closely tied to the different floating-point formats and remained constant. This imposes certain limitations when the FPU needs to be reconfigured and reused as, for example, a part of another VHDL design. Both these statements described above formulate the problem and that is why they have been taken as a motivation for writing this thesis.

1.2 Technical Task

The master’s thesis task consists of several requirements, which must be fulfilled. These are listed below.

- The FPU must be designed in VHDL.

- The FPU must support addition, subtraction, multiplication, division and the input operands' normalization. This is a set of commands, which allows the FPU to be used as a basic mathematical co-processor. The FPU design must perform post-normalization for the result and denormalization for addition/subtraction operations.

- The FPU design must be generic. This means that width of the fractional and exponential parts in operands must be parameterisable. Their width can be defined in a VHDL package, which affects the whole FPU design format.

- The designed FPU must be compliant with the IEEE-754 floating-point standard [6] as much as possible. The specific FPU exceptions such as overflow, underflow or division by zero have to be signalled through the independent one bit outputs. Other exceptions have not be taken into account, because this increases the complexity of the FPU design. Five rounding modes, which are described in the IEEE-754 standard [6] must be implemented. Rounding must take place after post-normalization.

- A generic barrel shifter must be used in the FPU design in order to shift fractions with minimal time delays, avoiding a shift register. The internal structure of the barrel shifter must use as less multiplexers as possible. In order to achieve this, the barrel shifter example from [7] can be used. Although a usage of the shift register for shifting the binary numbers to the left and right can save some hardware resources, the implementing of the parameterisable barrel shifter is still the number one priority for the FPU design, because this minimizes timing delays.

- A compromise between the operation speed and the resulting design size has to be found. The priority is small resulting design size, limiting the FPU maximum operating clock speed.

- The internal units of the FPU have to be implemented as VHDL components for the simplification of the design process and its simulation. Each component must be implemented in a separate VHDL file. The instantiating of the components can take place in the main high-hierarchical design file. Signals are used as a connection between the different components.

Considering all the described above, it is expected to create a parameterisable FPU design, which adequately meets these requirements.

1.3 Thesis Overview

This thesis is structured as follows. The first part of the thesis concisely introduces the introduced problem of the existing FPU designs and the explanation why the parameterisable design of such a kind is needed. It also describes a technical task and motivation for writing the thesis.

The second part is dedicated to the different FP formats and basic operations, which can be performed between the FP numbers. As was mentioned in part 1.2, these operations are addition, subtraction, multiplication and division. Since the hardware multipliers are used intensively for both multiplication and division operations, the estimation of the multipliers' usage is also presented in the second part of this thesis.

The third part is the main part of the thesis. It describes the design methodology in detail as well as means and methods that have been used for the FPU design verification and testing. The design flow is shown for every internal FPU block. Each block is verified separately before being integrated into the main design. A parameterisable testbench has been written in order to simulate and verify the correctness of the FPU behaviour, showing that the parameterisable FPU design in VHDL is a feasible task. The possible design improvements have been described in the end of part 3 of this thesis.

The fourth part describes the FPU testing flow, discussing its performance. The usage of clock cycles has been measured for the different FPU commands according to the simulation results. Due to the multiplexer simplicity, the internal FPU multiplexer has not been tested separately. It has been tested along with others internal FPU blocks.

2 Floating-Point Numbers Representation

Representation of numbers in computers requires a fixed number of binary digits, which are either stored in registers or in data memory. These numbers are both positive and negative, and, depending on the computer system architecture, they can have a wide range of values on a number scale. The representation of numbers in a binary form is a most convenient way to store an integer number inside the register in computer systems [8].

Numbers with a decimal point can be stored in the same registers. However, some bits in registers will contain the whole part of a number and the second part will contain all the digits after the decimal point (the fractional part). Each part stores some number of bits, for example half of the total. As can be seen, the point is fixed and never shifts either left or right, and, based on this, such a format is called a fixed-point. Thus, such a data storing format is inefficient, because whole and fractional parts of large and small numbers can use only a half of the dedicated bits, and, as a result it significantly reduces the accuracy of the represented numbers. The next chapter 2.1 explains the difference between fixed and floating-point formats.

2.1 Difference Between Fixed and Floating-Point Formats

As was mentioned before, when using a fixed point format the amount of bits dedicated for fractional and exponential parts representation remains the same. However, due to this the result's precision and accuracy can be lost, because the operand and result have the same number of bits. Figure 1 shows an example of a 16 bit fixed point number in which the radix point separates eight bits of the fraction from eight bits of the integer part. The restriction of this example is that the maximum stored value in the integer part is 2^8 for unsigned numbers. For signed numbers the range of values, which can be stored, is from -2^7 to 2^7, because one bit is reserved for the

sign encoding. However, this limitation can be overpassed using the floating-point format.

Figure 1. Example of the fixed point number format

Floating-point is a technique that allows the radix point to be shifted left and right, allowing the possibility to express relatively large or small numbers, without giving either the fractional or the whole a set number of digits. To express a large number, the floating-point should be moved all the way to the right, and the opposite to express a small number. Usage of floating-point numbers increases the accuracy of the data representation [8]. Mainly the accuracy of the floating-point result depends on the precision of the fractional part. The range of the result is defined by the exponent's precision. Due to the finite number of bits in registers the precision of both fractional and exponential parts is limited by the number of dedicated bits. Thus, the floating-point numbers represent real numbers providing a trade-off between range and precision. The standard floating-point formats are described in the next chapter as well as a general overview of the IEEE-754 standard [6] being given. This standard is also briefly described in [8].

2.2 IEEE-754 Standard Overview

The IEEE Standard for Floating-Point Arithmetic (IEEE-754) [6] is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably. Many hardware floating-point units now use the IEEE-754 standard [6]. This standard has been used in order to design the generic FPU, which has been described in this thesis.

2.2.1 Single precision floating-point format

According to the IEEE-754 standard [6], single precision floating-point numbers are represented as a single 32 bit binary number, consisting of three fields:

- sign (1 bit)
- exponent (8 bit)
- fraction (mantissa) (23 bits)

Assuming this, the values about 10^{-38} and 10^{38} are the minimal and maximum numbers respectively, which can be represented in a single precision floating-point format. The location of bits is shown in figure 2.

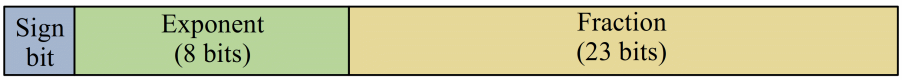


Figure 2. Single precision floating-point format (32 bits)

2.2.2 Double precision floating-point format

According to the IEEE-754 standard [6], a double-precision floating-point format occupies 64 bits in the computer memory. It represents a wide dynamic range of numeric values by using a floating radix point. The double precision floating-point number format includes three fields:

- sign (1 bit)
- exponent (11 bit)
- fraction (mantissa) (52 bits)

The location of bits is shown in figure 3.

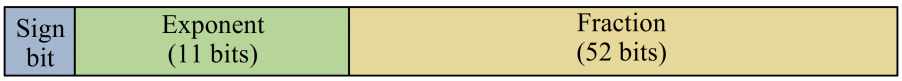


Figure 3. Double precision floating-point number format (64 bits)

The exponent field can be interpreted as either an 11 bit signed integer from −1024 to 1023 in the 2nd's complement format. The fraction field is represented with 52 bits. Therefore, the values of about 10^{-308} and 10^{308} are the minimal and maximum numbers, which can be represented in a double precision number format.

2.2.3 Extended precision floating-point format

Extended precision floating-point format provides greater precision than the basic floating-point formats [6]. Due to a larger number of bits in fractional and exponen-

tial parts, this floating-point unit standard minimizes overflow, underflow and rounding errors and increases the result's accuracy by using a greater precision. The 80 bit floating-point format has a range (including subnormals) from approximately 3.65×10^{-4951} to 1.18×10^{-4932}. The extended precision floating-point number format is shown in figure 4.

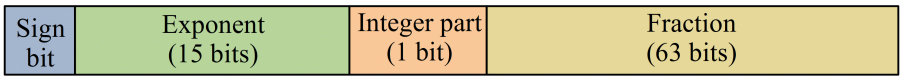


Figure 4. Double precision floating-point number format (80 bits)

The extended precision floating-point number format includes four fields:

- sign (1 bit)
- exponent (15 bit)
- integer part (1 bit)
- fraction (mantissa) (63 bits)

2.2.4 Conversion from a decimal to a binary form

A fractional number in a decimal form can be converted to a binary form by using a simple algorithm. The whole part of the number should be represented as usual with some number of bits, however the number of bits for a fractional part depends on the required precision of the number representation. If higher precision is required, the fractional part must contain more bits. For the fractional part, the bit magnitude increases with the bit index. The example below shows how to convert a decimal fractional number 19.490234375 into a binary form.

19.490234375	19 is 10011 in the binary form	
0.490234375	x2 = **0**.98046875	**0**
0.98046875	x2 = **1**.9609375	**1**
0. 9609375	x2 = **1**.921875	**1**
0.921875	x2 = **1**.84375	**1**
0.84375	x2 = **1**.6875	**1**
0.6875	x2 = **1**.375	**1**

0.375	x2 = **0**.75	**0**
0.75	x2 = **1**.5	**1**
0.5	x2 = **1**.0	**1**

The binary value is **10011.011111011.**

Considering the example described above, the fractional part of the number must be multiplied by two every time until the result reaches the value of one. The number of the conversion steps will be increased in cases when the fractional part is rounded inaccurately. In other words, such a conversion represents an iterative process and should be restricted with an available number of fractional bits. A conversion algorithm is presented below in table 1. This table shows the bit magnitude for the conversion example, which has been shown above.

Table 1. Bit's magnitude of the fractional part

Bit number	Bit value	Bit magnitude
8	0	1/2
7	1	1/4
6	1	1/8
5	1	1/16
4	1	1/32
3	1	1/64
2	0	1/128
1	1	1/256
0	1	1/512

2.2.5 Conversion from a binary to a decimal form

A fractional number in a binary form can be converted to a fractional number in a decimal form. Every bit of the fractional number must be multiplied with its bit magnitude with further addition of the gotten results. For a given fractional part in the previous chapter the conversion can be done as follows:

$$\frac{1}{2}\times 0+\frac{1}{4}\times 1+\frac{1}{8}\times 1+\frac{1}{16}\times 1+\frac{1}{32}\times 1+\frac{1}{64}\times 1+\frac{1}{128}\times 0+\frac{1}{256}\times 1+\frac{1}{512}\times 1=0.190234375$$

This example shows that the result of the conversion from a binary to a decimal form is valid.

2.3 Operations Between Floating-Point Numbers

Basic mathematical operations between the FP numbers are addition, subtraction, multiplication and division. Depending on the performed operation, denormalization of the input operands and post-normalisation of the result can be involved in order to perform calculations and present the result in a normalized form. Before performing the calculations, FP operands need to be extended with additional bits from the left and right sides of the number. Extending the widths of the operands is an important step, which increases the accuracy of calculations and preserves the result from overflow and underflow. After calculations the obtained result must be rounded and truncated in order to fit into the FP format's width. Furthermore, the FP result requires the performing of a post-normalizing procedure in cases when the fractional part of the FP result n does not satisfy a condition when $0.5 \leqslant n < 1$.

2.3.1 Addition and subtraction

In order to perform the addition or subtraction between two FP numbers, their exponents must be equal. The result of the addition or subtraction operation requires can invoke overflow and underflow conditions respectively, which can occur. Owing to the FP format, the fractional part (mantissa) is always positive. In addition, due to the interchangeability of the addition and subtraction operations it becomes possible to operate with positive numbers instead of the negatives. The sign of the result can be calculated independently considering the required operation (addition or subtraction) and initial values of operands. This situation has been described in detail in section 3.1.1. The example of the FP addition is shown below:

$$A=m_A\times bs^n \quad , \quad B=m_B\times bs^{n+m} \quad , \quad A+B=m_A\times bs^n+m_B\times bs^n\times bs^m$$

$$1.2359\times 10^9+0.792\times 10^5=12359\times 10^5+0.792\times 10^5=12359.792\times 10^5$$

$$12359.792\times 10^5=0.12359792\times 10^{5+5}=0.12359792\times 10^{10}$$

The FP number subtraction is performed as follows:

$$0.2359\times10^{7}-0.735\times10^{5}=23.59\times10^{5}-0.735\times10^{5}=23.59\times10^{5}$$
$$23.59\times10^{5}=0.2359\times10^{2+5}=0.2359\times10^{7}$$

2.3.2 Multiplication

FP multiplication does not require denormalization. However, to achieve better accuracy, the FP operands must be normalized. In order to multiply two FP numbers, their fractional parts must be multiplied, whilst the exponents must be added. The result might require post-normalization. The general formula, which describes the FP multiplication and an example are listed below.

$$A=m_A\times2^{e_A}\ ,\ B=m_B\times2^{e_B}\ ,\ A\times B=m_A\times m_B\times2^{e_A+e_B}$$
$$0.35\times10^{3}\times0.51\times10^{2}=0.35\times0.51\times10^{3+2}=0.1785\times10^{5}$$

2.3.3 Division

FP division is performed by dividing the operands' fractions and subtracting their exponential parts. The general formula, which describes the FP division is showed below:

$$A=m_A\times2^{e_A}\ ,\ B=m_B\times2^{e_B}\ ,\ A/B=\frac{m_A}{m_B}\times2^{e_A-e_B}$$

There exist different methods of division. Bit by bit division and division through the subtraction are not considered in this thesis, because the speed of these division algorithms is extremely small. Bit by bit division takes the number of cycles, which is equal to the dividend width in bits. Division through the subtraction depends on the value of operands. Furthermore, both of these division methods require dividends to be greater than divisors.

However, these limitations, which are imposed by these division methods, can be overpassed using the Newton-Raphson division algorithm. The only requirement for the FP operands is that they need to be normalized in order to avoid the possible overflow situation. This algorithm can divide the fractional part through the multiplication, significantly increasing the division speed. The number of cycles required for the Newton-Raphson division is acceptable and usually it is less than the number of bits in an operand. Due to the Newton-Raphson algorithm's iterative nature, each next intermediate result becomes closer to the real value of the result. This division method is described more in detail in section 3.1.3 of this thesis.

2.4 DSP Slices Usage Estimation

Due to the intensive usage of internal FPGA multipliers in the FPU design for both multiplication and division operations, the using of multipliers needs to be estimated. The multiplication time delay have to be considered as well. There are various synthesis tools, which can help to perform this estimation [9]. The tool, which has been chosen for this experiment and for the FPU synthesis is ISE Project Navigator. The hardware multiplication has been performed by the dedicated internal DSP slices, which Spartan-6 and other FPGA families contain. In Spartan-6, for example, the multiplier has a width of 18 bits and it is a part of an internal DSP slice. Thus, if the width of the multiplier is not enough for performing the multiplication, the resulting multiplier width will be increased by using additional 18 bit multipliers.

In order to estimate the DSP slices usage, the series of binary numbers multiplication in VHDL have to be performed. The first FP unit format, which has been used to carry out the experiment is 16 bit and the largest is 80 bits. The relation between the FP format, unsigned fractional and unsigned exponential parts of the FP number are shown in table 3. The table shows that usage of DSP slices rises almost exponentially with the increasing number of bits in the fractional part of the FP operands. Figure 5 illustrates the dependency of the DSP slices usage from the FP number fractional part width.

Time delay in the cascaded multiplier is another factor, which must be considered, because this restricts the maximum operating FPU design frequency. The delay in multiplier increases gradually depending on the operands' widths. The relation between time delay and the fraction's width is shown in figure 6.

From these two graphs it can be discovered that the combinational multiplier uses more DSP slices in order to maintain the time delay as small as possible with increasing number of bits in the fractional part of the FP number. This is confirmed by the numbers, which have been obtained after the series of the FP multiplications have been performed.

Table 2. Usage of DSPs by different FP formats

Example	FP format, bits	Fraction, bits	Exponent, bits	Usage of DSPs
1	16	8	8	1
		10	6	
2	18	10	8	1
		12	6	
3	20	10	10	1
		14	6	1
4	24	12	12	2
		16	8	
5	28	14	14	4
		18	10	
6	32	16	16	
		24	8	
7	40	30	10	8
8	48	35	13	11
9	64	50	14	12
10	80	60	20	23

According to the graphs, starting from 16 bit and finishing with 80 bit FP format, the DSP slices usage increased from 1 to 23. However, the time delay rose only from 5 ns to 15 ns, which is about three times more. Although the time delay grows with increasing number of bits in FP operands, the synthesizer attempts to eliminate the delay by the extensive usage of hardware.

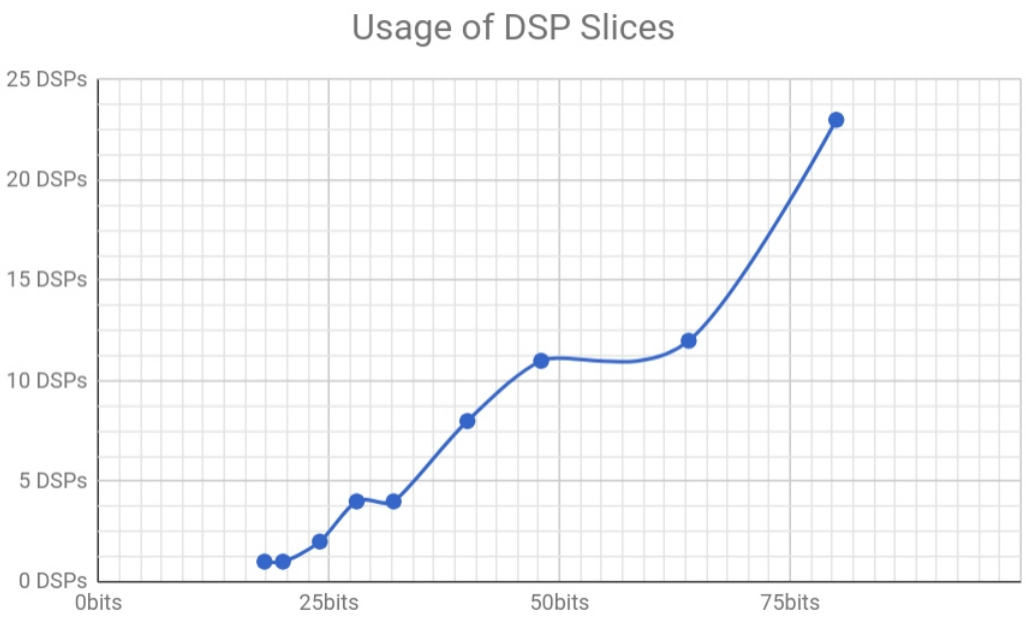


Figure 5. Usage of DSP slices with different fractions

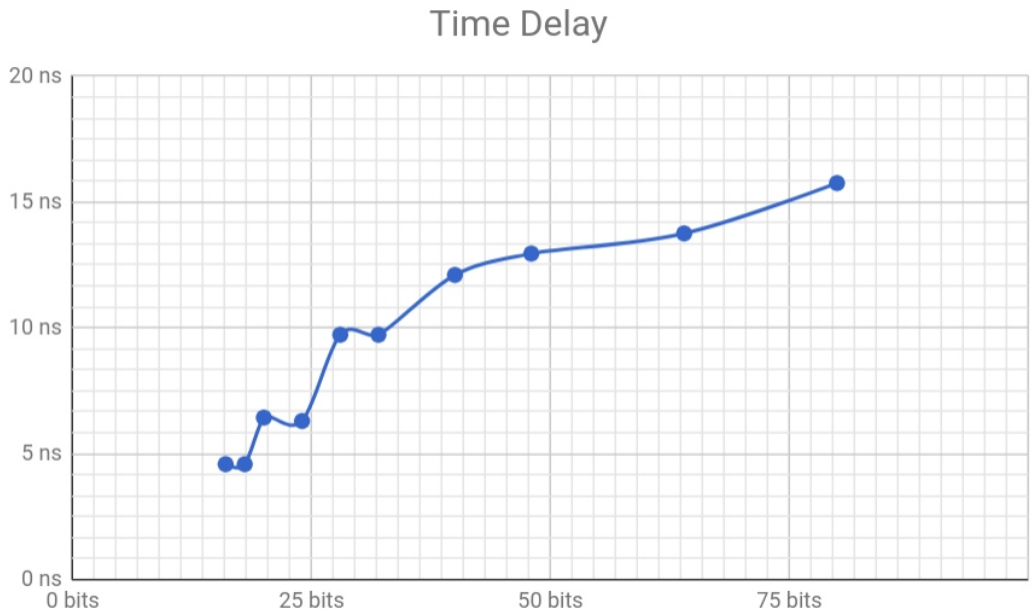


Figure 6. Multiplication time delay for different fraction widths

Conclusions

This part of the thesis has introduced an overview of the different FP standards and basic operations between the FP numbers. The IEEE-754 standard [6] has been briefly highlighted and the DSP slices usage has been estimated. The DSP slices contain the multipliers, which are essential for the FPU multiplication and the iterative division through the multiplication. The multipliers usage grows exponentially with the increasing number of bits in the fractional and exponential part of the FP numbers. However, the time delay is increased more gradually compared with the number of the multipliers, which are used. This concludes to an assumption that the ISE Project Navigator synthesis tool creates the combinational multipliers, inferring the additional internal DSP slices in order to decrease signal propagation delays inside the FPGA.

3 Generic Floating-Point Unit Design

This chapter describes a boundary cases analysis, design and testing methodologies, which have been used for the FPU design and testing flow. In order to clarify the FPU design process, the cases when the FP operands and/or the result acquire their minimum, maximum or inappropriate values have to be considered. This explains how the result's value must be truncated and rounded and how the overflow, underflow and division by zero exceptions must be handled. The boundary cases analysis is required in order to investigate the FP numbers' values, which cannot be handled by a given FPU format due to its limitations. The high-level synthesis principles have been described in [10] [11] [12]. This chapter also explains in more detail the FPU internal blocks' structure and their input and output signals. Each internal FPU block has its specific functionality and is connected with other necessary internal structural units in order to provide the proper FPU operation. From the design point of view, all the internal blocks are implemented as the VHDL components, which are instantiated in the FPU top hierarchical VHDL module. The idea of VHDL components has been described in [13] [14].

3.1 Boundary Cases Analysis

Performing operations between the FP numbers can lead to unexpected results when operands have values, which are close to the bounds of their range or even exceed the maximum or minimum allowed values. The boundary cases analysis helps to define the FPU design paths, which impact on being able to handle the FP result by the FPU. For the overflow, underflow and division by zero exceptions the result of calculation cannot be stored as it is due to the physical limitations of the FPU hardware. This is due to the finite number of the registers bits storing the binary values.

In order to increase the accuracy of the calculation result, the precision of the FP operands is extended with additional bits as was mentioned in part 2.3 of this thesis. According to the IEEE-754 standard [6], a rounding takes a number regarded as

infinitely precise in respect to the obtained result and, if necessary, modifies it to fit in the destination's format, while signalling the underflow, overflow or division by zero exceptions when appropriate. Each FP operation has to be performed as if it is firstly produced as an intermediate result correct to infinite precision and with unbounded range, and then that result is rounded according to one of the rounding attributes. Inexact numeric FP results always have the same sign as the result, which is not rounded.

The IEEE-754 standard [6] defines various rounding types and calculation exceptions. The two most important rounding modes are rounding to the nearest and directed rounding. Each of two rounding types has its own rounding attributes. According to the standard, the FP numbers in a binary form must be rounded to the nearest value. However, the generic FPU, which has been designed, supports five rounding modes. Due to the simplification of the FPU design, some of the IEEE-754 requirements [6], such as inexact exception have not been considered. The implemented rounding modes and signalled exceptions are described in detail in part 3.2.6 of this thesis.

3.1.1 Addition and subtraction

Addition and subtraction of the FP operands can be performed when their exponential parts are equal. In order to satisfy this requirement, a special procedure must be applied, which is called denormalization. The denormalization procedure have to be performed on one of the input FP operands. This requires the extending of both operands' precision by adding additional least significant bits to them. The result of the FP operation needs post-normalization in order to be outputted as a normalized FP value.

Denormalization of a FP operand in a way when its exponent is decreased, requires shifting of its fractional part to the right. Therefore, in order to preserve the precision and accuracy, the two extra bits must be added to the operand after the right least significant bit. Due to the practical implementation approach, considering the overflow and underflow exceptions described in the IEEE-754 standard [6], in cases when these exceptions arise, the maximum and minimum number value must be outputted for overflow and underflow situation respectively.

Both addition and subtraction operations are interchangeable. The subtraction of a negative number can be replaced with the addition with an opposite sign. All possible addition and subtraction scenarios are shown in table 3, which illustrates their interchangeability depending on the FP operands' signs. This in turn helps to define the design path for the adder/subtractor, which is a part of the FPU.

Table 3. Interchangeability of addition and subtraction operations

	A > 0; B > 0	**A > 0; B < 0**	**A < 0; B > 0**	**A < 0; B < 0**
Addition	A + B	A + (-B)	(-A) + B	(-A) + (-B)
Subtraction	A + (-B)	A + B	(-A) + (-B)	(-A) + B

Based on the data in the table above, in order to implement addition and subtraction two ways can be used. One of these requires the usage of a number's two's complement form. This means that the half of a number's possible range is used for representing the positive numbers and the other half is used for negatives. This entails confusion, because numbers in a two's complement's form are not straightforward to deal with. Another way of the adder/subtractor implementation is to operate only with positive operands, removing the sign which can be calculated independently. The required addition or subtraction operation is calculated based on operands signs and their values. The second method of the adder//subtractor implementation has been used in this thesis. This method has been described in part 3.2.3.

The denormalization of the FP operands have to be performed with the higher precision in order to avoid a loss of accuracy in the result. In fact, when the FP operands' exponents differ from each other not more than by the fraction's width, the accuracy can be lost to a certain extent. In addition, the most accurate result of the addition and subtraction can be achieved when the FP operands have the same value in the exponential part. The example below demonstrates the addition of two FP numbers with eight digits after the radix point and without the extra bits:

$$0.11111111_2 \times 2^3 + 0.11111111_2 \times 2^0 = 0.11111111_2 \times 2^3 + 0.00011111 \times 2^3 =$$

$$= 1.00011110_2 \times 2^3 = 1.1171875_{10} \times 2^3$$

The example above demonstrates the exponents, which have become equal and the second operand's fraction, which has been shifted to the right by three positions.

Because of this the accuracy can be lost to some extent, depending on the least significant bits in the fractions.

In order to prevent this situation with accuracy loss to some extent, the precision of the fractions in the FP operands must be extended before performing a calculation. In fact, the precision of operands' fractions cannot be extended infinitely, because the number of bits in registers is always limited. However, according to the IEEE-754 standard [6], the precision has to be extended by the two least significant bits. The example below demonstrates obtaining of a more accurate addition result, using the operands' precision extended by two additional least significant bits:

$$0.1111111100_2 \times 2^3 + 0.1111111100_2 \times 2^0 = 0.1111111100_2 \times 2^3 + 0.0001111111_2 \times 2^3 =$$

$$= 1.0001111100_2 \times 2^3 = 1.120117188_{10} \times 2^3$$

Therefore, two examples described above show the difference in accuracy between the resulted mantissas (fractions) using the different precision for the FP numbers representation. This difference is:

$$m_{diff} = 1.120117188_{10} - 1.1171875_{10} = 0.002929688_{10}$$

The expressions below show an example how the extra precision bits affect the result, when adding a number, which is almost equal to 1:

$$0.11111111_2 + 0.00000001_2 = 1.00000000_2 = 1_{10}$$

$$0.11111111_2 + 0.0000000111_2 = 1.0000000011_2 = 1.0029296875_{10}$$

This analysis shows the importance of the FP operands' precision extention in order to obtain more accurate results. However, after rounding the FP mantissa must be truncated to the number of bits, which are defined by a chosen FP format. For any FP widths the precision of the fractional part must always be extended by the two least significant bits.

3.1.2 Multiplication

FP multiplication does not require denormalization. The fractional parts of the FP numbers must be multiplied and their exponents must be added. The multiplication result requires performing post-normalization.

If both input FP operands have been normalized before the multiplication (when condition $0.5 \leqslant fraction < 1$ is satisfied), then the overflow situation will not take place. The example below shows such a feature:

$$0.1111111111_2 \times 0.1111111111_2 = 0.11111111100000000001_2 \approx 0.1111111110_2$$

$$0.9990234375_{10} \times 0.9990234375_{10} \approx 0.998046875_{10}$$

The fractional result, which is almost equal to 1, can be rounded by truncating it within the equal amount of bits as the input operands have in their fractional part. This happens because the least significant bit of the result's fractional part is likely to always have a zero value in the cases when input operands fractions approximately equal to 1.

The overflow and underflow conditions never take place in fraction multiplication. In some cases the result can be rounded up to 1 and in other cases down to 0. The sign of the multiplication result is calculated as an exclusive OR operation between the signs of the input operands.

3.1.3 Division

Division of two FP numbers is performed by division of their fractional parts, whilst subtracting their exponents. The general formula, which describes the FP division is as follows:

$$A = m_A \times 2^{e_A} \quad , \quad B = m_B \times 2^{e_B} \quad , \quad m_A \geq m_B \quad , \quad A/B = \frac{m_A}{M_B} \times 2^{e_A - e_B}$$

Different dividend and divisor values have been used in order to investigate the boundary cases for division. These values are shown in table 4.

Table 4. Results of division caused by different dividend and divisor values

N	Dividend	Divisor	Result of the division
1	0	Any number	0
2	0.00000001	0.00000001	1
3	0.00000001	0.11111111	0
4	0.11111111	0.00000001	Overflow
5	0.11111111	0.11111111	1
6	Any number	0	Division by zero exception

The underflow exception cannot occur for the division. However, the overflow might take place if the divisor is much smaller than the dividend. The division by zero exception is a mathematical exception, which happens if the divisor is equal to zero.

Various division algorithms have been described in [15]. In order to increase the speed of division and eliminate dependency on a dividend and divisor values, the Newton-Raphson division algorithm will be used in the practical part of this thesis. The idea of this algorithm is to obtain a division result through the series of multiplications. The algorithm finds the opposite value of a divisor D and multiplies that reciprocal value by dividend N to find the final quotient R. To demonstrate how the algorithm works, the simple example in a decimal form has been shown below. The dividend and divisor values are: $N=0.40625$, $D=0.75$. The result is calculated as follows:

$$R_0=2-D=2-0.75=1.25$$

$$N_1=N\times R_0=0.40625\times 1.25=0.5078125$$

$$D_1=D\times R_0=0.75\times 1.25=0.9375$$

$$R_1=2-D_1=2-0.9375=1.0625$$

$$N_2=N_1\times R_1=0.5078125\times 1.0625=0.539550781$$

$$D_2=D_1\times R_1=0.9375\times 1.0625=0.99609$$

$$R_2=2-D_2=2-0.99609=1.00391$$

$$N_3=N_2\times R_2=0.539550781\times 1.00391=0.5416600425$$

$$D_3=D_2\times R_2=0.99609\times 1.00391=0.999984712$$

The result of division, which has been obtained after a few iterations is 0.5416600425. The series of calculations must be performed with the increasing index of each of the operands R_x, N_x and D_x until the D approaches to the value of 1. In the example described above the maximum value of D is 0.999984712 and takes place on the third iteration.

Below is shown the same example, but in a binary form with a 16 bit fractional part after the radix point.

$$N=0.01101_2 \quad , \quad D=0.11_2$$

$$R_0=10.0_2-0.11_2=1.01$$

$$N_1=0.01101_2\times 01.01_2=0.1000001_2$$

$$D_1=0.11_2\times 01.01_2=0.1111_2$$

$$R_1=10.0_2-0.1111_2=1.0001_2$$

$$N_2=0.1000001_2\times 1.0001_2=0.10001010001_2$$

$$D_2=0.1111_2\times 1.0001_2=0.11111111_2$$

$$R_2=10.0_2-0.11111111_2=1.00000001_2$$

$N_3 = 0.10001010001_2 \times 1.00000001_2 = 0.1000101010101010001$

$D_3 = 0.11111111_2 \times 1.00000001_2 = 0.1111111111111111_2$

The calculations were performed until D acquired the maximum value - 0.11111111, which is equal to the value of 0.99609375 in a decimal form. However, the factions, consisting from the less amount of bits, can produce an inaccurate results because of the smaller precision.

3.2 Floating-Point Unit Hardware Description

Different parts of the FPU have been described in this chapter. The algorithm of each FPU internal block has been presented in more detail in the form of a diagram. Each diagram has been explained, covering the most important steps in the algorithms.

Referring back to the technical task described in chapter 1.2, the FPU must have an adder/subtractor, a multiplier and a divider. Therefore, the FPU have to include the functional hardware blocks, which perform the required mathematical operations. In addition, the denormalization unit is essential for addition and subtraction, whilst the post-normalization unit is important for every operation. A dedicated finite state machine is needed in order to synchronize different parts of the FPU. This state machine has to produce the output ready signal, which indicates that the calculated result is valid and can be read from the FPU output. The post-normalizing circuitry is synchronized by the same state machine and used in order to post-normalize the result of the operation when it is needed. The four operations required from the technical task have been implemented by using three hardware units: an adder/subtractor, a multiplier and a divider. For addition and subtraction the input FP operands pass to the denormalizing unit first in order to have the exponents equal. The multiplication and division are performed without the need for input operands to be normalized. However, in order to avoid the possible overflow situation for division, the normalized input operands need to be applied, considering that fact that the FPU has an ability to normalize the input operands.

3.2.1 FPU top-hierarchical block description

The top-hierarchical FPU block is shown in figure 7. According to this figure, the FPU has inputs and outputs, which are both single wires and buses. The input FP

operands, output result, and inputs for selecting of rounding modes and required operations are the data buses. Other signals are single inputs and outputs. The symbol's *n* value is equal to the *OpWidth*, which is why the width of the operands is written as *[OpWidth-1..0]*. The explanation about the all FPU signals is presented in table 5. The table consists of the names of signals, their types, width and the functional purpose. The FPU inputs and outputs are the top-level signals in the FPU hierarchical structure. They descend inside the FPU to its internal blocks, which all together perform the necessary computational and internal data flow operations.

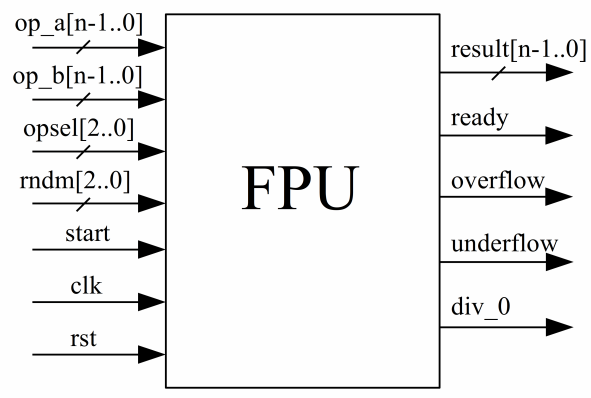


Figure 7. FPU top hierarchical block

The FPU can perform four operations such as the addition, subtraction, multiplication, division and normalization of the operands A and B. The list of the operations and flags affected by each operation are shown in table 6. The *opsel[2..0]* signal's value is latched at the rising edge of the signal *start.* Any manipulations with this signal during the calculation do not affect the FPU internal state after the signal value has been latched. This feature allows the FPU to avoid the unnecessary influence of the *opsel[2..0]* input on the FPU result.

The FPU supports five rounding modes that can be chosen by *rndm[2..0]* inputs. The list of the rounding algorithms is presented in table 7. Rounding takes place after the post-normalization procedure and it is described in detail in part 4.1.5 of this thesis.

The FPU exceptions are signalled through the overflow, underflow and division by zero outputs. The input c*lk* is required in order that the FPU be synchronized by the external oscillator. The input *rst* resets the FPU to its initial state.

Table 5. FPU input and output signals

Signal	**Type**	**Description**
op_a[n-1..0]	Input, bus	Operand A input data bus
op_b[n-1..0]	Input, bus	Operand B input data bus
opsel[2..0]	Input, bus	Three wire input control bus is used for selection of the operation, which have to be performed. These inputs are edge-sensitive and latch their values at the rising edge of the input signal *start*
rndm[2..0]	Input, bus	Three wire input control bus is used for selecting the rounding mode. These inputs' values are latched at the rising edge of the input signal *start*
start	Input, single	Rising edge-triggered control input. This input activates the performing of the selected FPU operation and the result's rounding
clk	Input, single	FPU clock synchronization input (it is recommended to have the duty cycle of 0.5 in order to guarantee the proper operation of the FPU)
rst	Input, single	FPU reset input. A high level is active
result[n-1..0]	Output, bus	The output result data bus. Its width is equal to the input operands' width
ready	Output, single	This output signal provides a high pulse when the FPU result has been obtained
overflow	Output, single	Overflow exception flag. Arises to a high level when the overflow takes place
underflow	Output, single	Underflow exception flag. Arises to a high level when the underflow takes place
div_0	Output, single	Division by zero flag. This output acquires a high level if the operand B has the zero value and the division operation has been attempted to execute

The next chapter describes the FPU internal block diagram and contains the explanation of the general FPU operation algorithm. The chapters after the next chapter describe each of the FPU internal blocks in more detail.

Table 6. FPU operations

opsel[2..0] signal's value	**Executed operation**	**Affected exception flags**
000	Addition	Overflow, underflow
001	Subtraction	Overflow, underflow
010	Multiplication	Overflow
011	Division	Overflow, division by zero
100	Reserved	-
101	Reserved	-
110	Normalization of the operand A	Underflow
111	Normalization of the operand B	Underflow

Table 7. FPU rounding modes

rndm[2..0] signal's value	**Selected rounding mode**
000	TowardsZero
001	to nearest/TiedToEven
010	to nearest/TiedAwayFromZero
011	TowardPlusInfinity
100	TowardMinusInfinity
101	Reserved
110	Reserved
111	Reserved

3.2.2 FPU internal block diagram description

The FPU consists of the several independent units, which are internally connected through the signal buses and separate signals. Before synthesis the FPU can be configured to be synthesized with a different FP format. The sign bit always occupies only one bit in any FP format, including customer formats. The FPU format is defined in the configuration package by three constants. The FPU internal block diagram is

shown in figure 8. It consists of an adder/subtractor, which is a single unit, a multiplier, a divider, a denormalization unit, a post-normalization unit, an exponent and fraction multiplexer, a barrel shifter, and a control FSM. The multiplier, the divider and the post-normalizer unit accept the operands directly form the FPU inputs, while the adder/subtractor receives them through the denormalizer, which modifies one of the input operands in a way to obtain both exponents equal before the fractions can be added or subtracted. This modification includes shifting the smaller operand's bits to the right by the number, which is a difference between exponential parts' values of both operands. The adder/subtractor, multiplier and the divider (in further computational units) produce the intermediate results and their fractional parts are passed to the post-normalization unit directly. However, the exponential parts of each result are connected to the post-normalization unit through the multiplexer, which commutates the different exponential results, depending on the operation being selected. This multiplexer is used in order to commutate the barrel shifter's fractional and shifting control signals between the denormalization and post-normalization units. Shifting control signals derive from the multiplexer and they specify the number of shifts and the shifting direction (left of right). Shifting control signals are connected directly to the barrel shifter. The multiplexer is controlled by an internal FSM, which synchronizes all the internal FPU units.

The internal FSM controls the internal FPU operation. Depending on the operation between the input operands, the FSM connects different fractions to the barrel shifter at the specific clock cycle during the ongoing calculation. Furthermore, the FSM generates the internal *ready* signal, which indicates that current operation has been completed. This signal is a single pulse, whose rising edge in turn triggers the post-normalization unit to output the FP result to the FPU *result[n-1..0]* output.

A post-normalizer is a block, which takes the intermediate operation result, normalizes, rounds and truncates it in order to fit it into the specific FP format. The post-normalizer receives the intermediate resulting exponents from the multiplexer. This has been implemented in order to reduce the number of inputs in the post-normalization unit and simplify the FPU design flow.

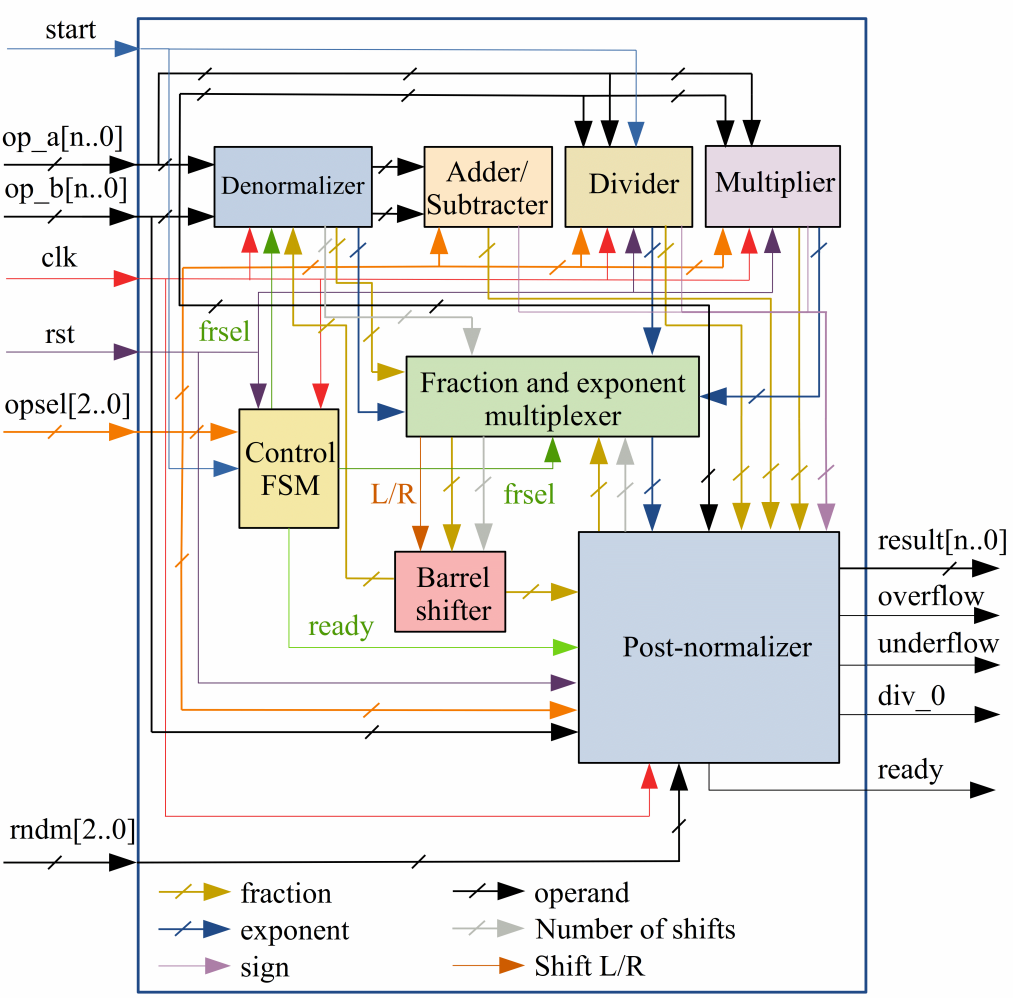


Figure 8. FPU internal block diagram

The majority of the internal FPU blocks (but not all of them) have clock *clk* and reset *rst* input signals, which are used for the FPU synchronization and reset. The multiplier, the divider, the post-normalizer and control FSM have both of these inputs, whilst the denormalization unit has only the *clk* input. Due to the combinational nature of the adder/subtractor, multiplexer and barrel shifter, these internal units do not use the *clk* and *rst* signals. A *start* input initiates the FSM operation. The division unit is also triggered, if the division operation has been selected. The *rndm[2..0]* input selects the rounding mode, which will be performed on the FP result after its normalization. This input is connected directly to the post-normalization unit as is shown in figure 8. The auxiliary extra digital logic is used among the FPU internal blocks in order to avoid the *output* signal's jitter during the calculations. The inputs and outputs of the FPU internal blocks are described in the next chapters in more detail.

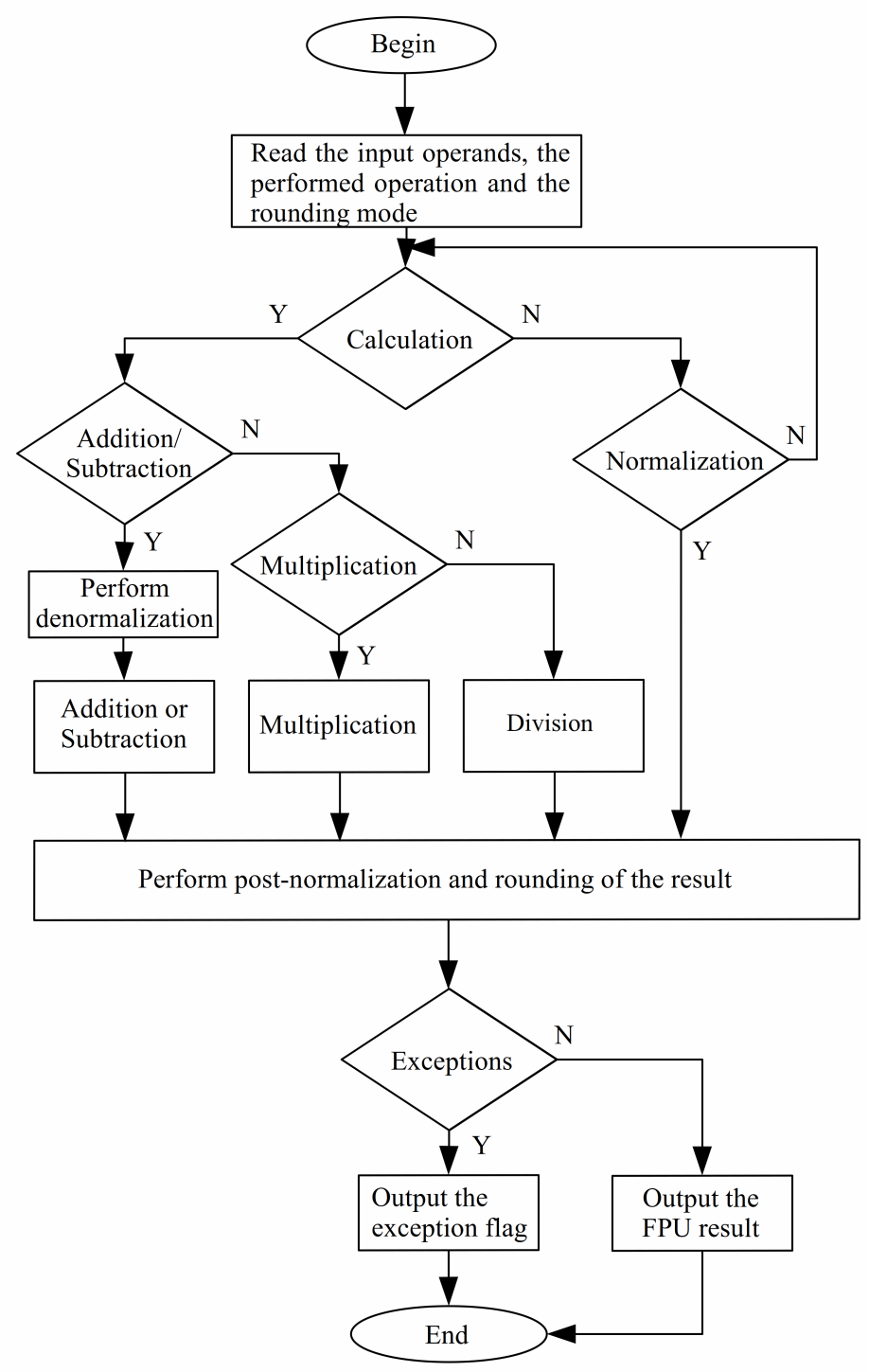


Figure 9. FPU general algorithm

The general algorithm of the FPU operation is illustrated in figure 9. At first, the FPU reads the input operands' values, the operation, which is need to be executed and the rounding mode. Each operation is triggered by a rising edge of the *start* input signal. At this rising edge the *opsel[2..0]* and *rndm[2..0]* inputs are latched and store their value until the arrival of the next rising edge of the *start* signal. It is recommended not

to change *opsel[2..0]* and *rndm[2..0]* input signals' values, applying the additional pulses at the *start* input during the ongoing calculation. This prevention measure guarantees the correctness of the FP result.

The FPU algorithm depends on the selected operation. In cases, when the normalization of the input operands has been selected, the post-normalizer starts to normalize them immediately. In other cases one of the calculations (addition, subtraction, multiplication or division) will be triggered. Computational operations require additional extra clock cycles. The exact number of clock cycles required for each operation has been described further in part 4 of the thesis.

The FPU design has been implemented in such a way that the width of the fractional and exponential parts of the FPU are defined in the VHDL package, which is a part of the FPU design. All the internal FPU buses, which connect internal FPU blocks between themselves, change their widths according to the constants, which are defined in this package.

3.2.3 Denormalization unit description

A denormalization unit makes the exponents of two input operands equal by shifting one of the fractions to the right. It calculates the difference between exponents of two input operands and shifts to the right the operand's fraction with the smaller exponent by a numbers of bits, which is equal to this difference. The denormalization unit provides the modified input operands' fractions to the adder/subtractor.

The denormalizer is connected through the internal multiplexer to the barrel shifter, which performs logical shifting. The logical shift left is not used for denormalization purposes. Denormalization requires two clock cycles in cases, when the operands' exponents are not equal. If the exponents are equal, then denormalization is not performed, taking one clock cycle and passing the fractions from the denormalization unit to the adder/subtractor directly. The common exponent after the denormalization procedure is passed to the internal FPU multiplexer. The top hierarchical block of the denormalization unit is shown in figure 10.

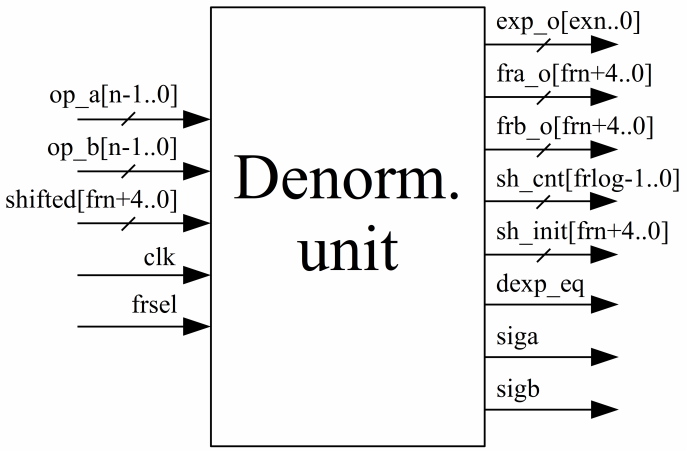


Figure 10. Denormalizer's top hierarchical block

The description of all the inputs and outputs of the denormalizer is shown in table 8. The number *frn* is the input operands fractions' width. The number *exn* is equal to the number of bits in one of the input operands' exponent. The constant *frlog* is a calculated logarithm based on the fractions' width. This calculation formula has been described in more detail in part 3.2.6 of this thesis.

The algorithm, which describes the operation of the denormalization unit, is shown in figure 12. According to the diagram, when the input FP operands A and B are read, their exponents are compared first. If the exponential part of operand A is less than the exponential part of operand B or vice versa, then the difference between their exponential parts is the number of the required shifts to the right for the smaller operand's fraction.

If the difference between two exponents is larger than the value of *[FractionalWidth-1]*, then the lowest fraction is assigned with all zeros, because its shifted value is too small and can be neglected. The larger operand is outputted as it is. However, both operands are extended with two least significant bits from the right side and three most significant bits from the left side in order to keep the accuracy in the addition/subtraction unit and deal with overflow situations. If the difference between operands' exponents is zero, then their fractional parts remain unchanged. Both fractions are always extended with the extra bits in order to preserve the calculation's precision. If the difference between the exponential parts is less than the value of the *[FractionalWidth-1]*, then, depending on their comparison result, the fractional part of the FP number with the larger exponent value remains unchanged.

Table 8. Denormalization unit input and output signals

Signal	**Type**	**Description**
op_a[n-1..0]	Input, bus	Operand A input data bus. Derives directly from the FPU input
op_b[n-1..0]	Input, bus	Operand B input data bus. Derives directly from the FPU input
shifted[frn+4..0]	Input, bus	Shifted fraction from the barrel shifter's output
clk	Input, single	FPU clock synchronization input
frsel	Input, single	Control input for storing of the denormalized operands
exp_o[exn..0]	Output, bus	The common exponent for both operands. It is extended by one most significant bit to prevent the overflow
fra_o[frn+4..0]	Output, bus	Denormalized operand's A fractional part
frb_o[frn+4..0]	Output, bus	Denormalized operand's B fractional part
sh_cnt[frlog-1..0]	Output, bus	Number of the required shifts. It is connected to the FPU internal multiplexer
sh_init[frn+4..0]	Output, bus	The initial fraction, which is needed to be shifted. It is connected to the FPU internal multiplexer
dexp_eq	Output, single	This flag is set, when the exponential parts of both input operands are equal
siga	Output, single	Sign of the operand A
sigb	Output, single	Sign of the operand B

However, other fraction is shifted to the right by a number of bit positions determined by the exponents difference. Three extra bits from the left side for the resulting fractions *ShFrA* and *ShFrB* are not shown in the algorithm for simplification. The resulted fractions' length after denormalization is always equal to *[FractionalWidth+4]*. The testing of the denormalization unit is described in part 4.1.1 of the thesis.

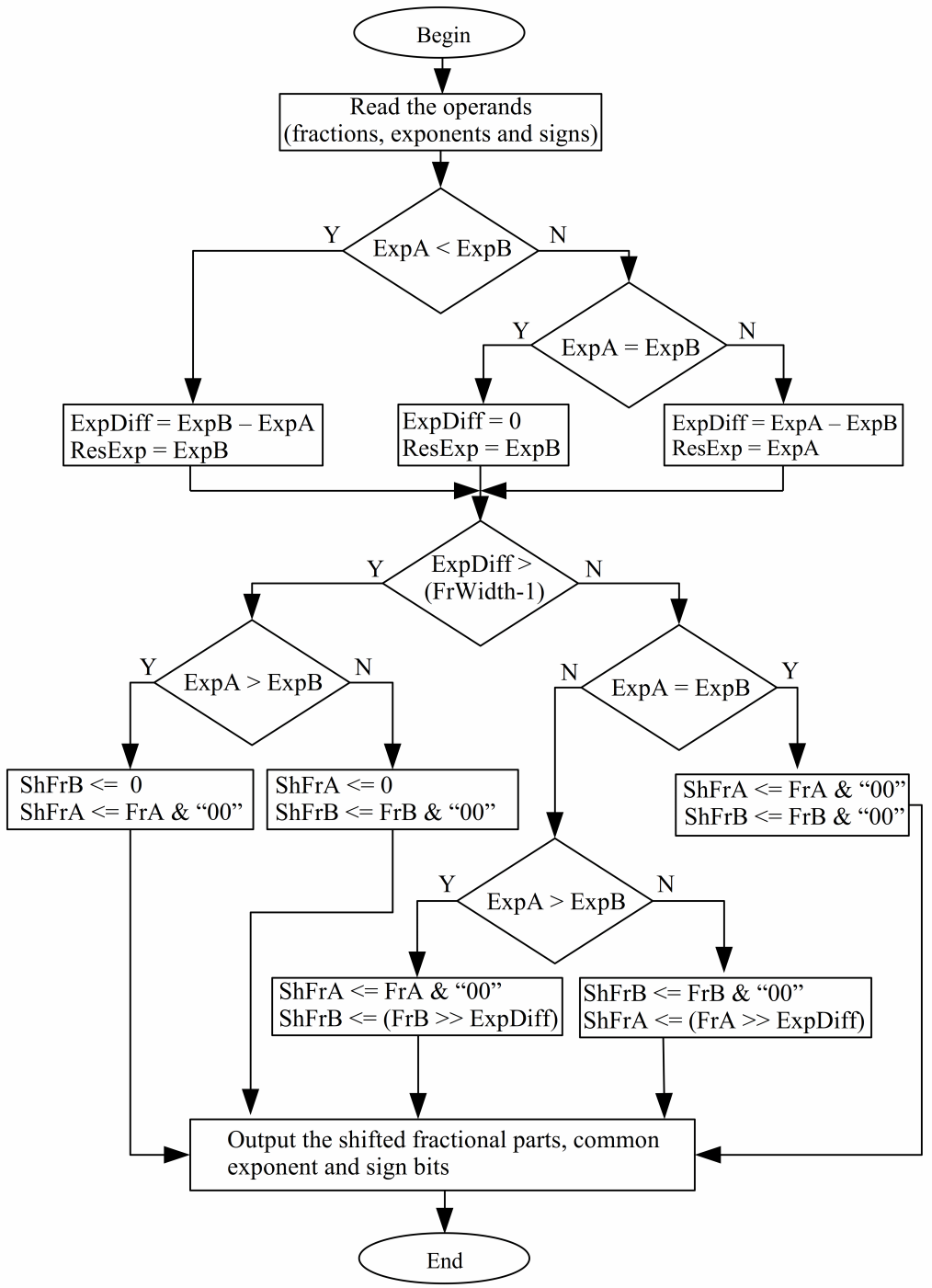


Figure 11. Denormalization unit algorithm

3.2.4 Addition/Subtraction unit description

An addition/subtraction unit is a fully combinational circuit. The width of its data buses is equal to *[FractionalWidth+4],* which is five bits wider than the width of the input operands. The adder/subtractor receives its operands from the denormalization unit, which has been described in the previous section. The top hierarchical block of the adder/subtractor is shown in figure 12.

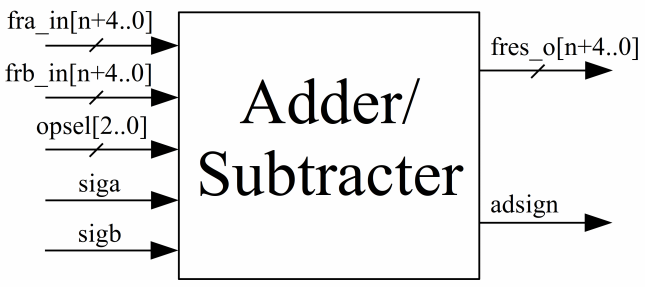


Figure 12. Adder/subtractor top hierarchical block

The inputs *fra_in[n+4..0]* and *frb_in[n+4..0]* are the operands' denormalized fractions, which are derived from the denormalization unit. The inputs *siga* and *sigb* are the operands' signs. The *fres_o[n+4..0]* and *adsign* are the fractional result and sign respectively. Both the input and output fractions are extended with five bits. That is why the width of these signals is *[n+4..0]* instead of *[n-1..0]*. The width extension is essential for increasing of the precision and for dealing with the fraction overflow situation. In the adder/subtractor the selection between addition or subtraction operation is defined by the *opsel[2..0]* signal, which derives from the FPU input. If the signal *opsel[2..0]* has a value of "000", the addition is performed, whilst the subtraction takes place when the *opsel[2..0]* signal's value is "001". The addition/subtraction unit does not check the overflow or underflow conditions, if they arise. Instead of this, the result of the addition or subtraction is passed directly to the post-normalization unit, which checks these conditions by itself.

In more detail, the addition/subtraction unit does not deal with negative numbers. By contrast, depending on the input operands' values and their signs and due to the interchangeability of the addition and subtraction operations, the adder/subtractor replaces the required operation, which is being performed with the opposite one as well as the sign of the result. This situation has been described in section 3.1.1 of this thesis.

Table 9. Addition and subtraction interchangeability

Operands relation	**A sign**	**B sign**	**Required operation**	**Signal fra_lt_frb**	**Signal s_addop**	**Resulted operation**	**Resulted sign**
|A| > |B|	+	+	Addition	1	0	A + B	+
|A| > |B|	+	-	Addition	1	1	A - B	+
|A| > |B|	-	+	Addition	1	1	A - B	-
|A| > |B|	-	-	Addition	1	0	A + B	-
|A| < |B|	+	+	Addition	0	0	A + B	+
|A| < |B|	+	-	Addition	0	1	B - A	-
|A| < |B|	-	+	Addition	0	1	B - A	+
|A| < |B|	-	-	Addition	0	0	A + B	-
|A| > |B|	+	+	Subtraction	1	1	A - B	+
|A| > |B|	+	-	Subtraction	1	0	A + B	+
|A| > |B|	-	+	Subtraction	1	0	A + B	-
|A| > |B|	-	-	Subtraction	1	1	A - B	-
|A| < |B|	+	+	Subtraction	0	0	A + B	-
|A| < |B|	+	-	Subtraction	0	1	A - B	+
|A| < |B|	-	+	Subtraction	0	1	B - A	-
|A| < |B|	-	-	Subtraction	0	0	A + B	+
|A| = |B|	+	+	Addition	X	0	A + B	+
|A| = |B|	+	-	Addition	X	1	A - B	X
|A| = |B|	-	+	Addition	X	1	B - A	X
|A| = |B|	-	-	Addition	X	0	A + B	-
|A| = |B|	+	+	Subtraction	X	1	A - B	X
|A| = |B|	+	-	Subtraction	X	0	A + B	+
|A| = |B|	-	+	Subtraction	X	0	A + B	-
|A| = |B|	-	-	Subtraction	X	1	A - B	X

X means “don’t care” value

The addition/subtraction unit defines the larger fraction between two fractions and then applies this for calculating of the result. Based on the fractions' signs, the actual operation may be different from the operation, which is defined by the *opsel[2..0]* input signal. The actual operation and the sign are calculated by the separate boolean expressions, which create look-up tables in the FPGA after synthesis. If the resulted operation is required to be an addition, then two fractions are simply added.

However, in the case of subtraction, a smaller fraction is always subtracted from a larger. This has been implemented in order to simplify dealing with negative numbers. The relation between the input operands, the resulting operation and its result sign is shown in table 9. According to this table, there are three possible groups of operands' values. These groups are related to such conditions as $|A| > |B|$, $|A| < |B|$ and $|A| = |B|$. The addition/subtraction unit has the internal signal *fra_lt_frb*. This signal acquires a high level when the fraction of the operand A is larger than the fraction of the operand B, whilst becoming zero otherwise. Another signal is called *addsub*, which has the same value as the FPU *opsel[2..0]* input's least significant bit. It defines for the adder/subtractor's internal logic, which operation (addition or subtraction) must be performed. The value of the signal *addsub* is '0' for the addition and '1' for the subtraction. This signal is not shown in table 9. The signal *s_addop* depends on the two previous signals' values and it defines the resulted operation, which is being performed. If this signal is equal to '0', then the addition operation is performed, but for the subtraction the smaller exponent is always subtracted from the larger as was mentioned above. *X* means that the value is "don't care". These three signals *fra_lt_frb*, *addsub* and *s_addop* define the actual operation and the sign of the result. Their VHDL expressions have been written by analysing the data given in table 9. The boolean expressions for these signals, including for the resulted sign's value are shown below:

```
addsub <= '1' when opsel = "001" else '0';
fra_lt_frb <= '1' when (efra > efrb) else '0';
s_addop <= ((not addsub) and (asign xor bsign)) or
           (addsub and (not (asign xor bsign)));
res_sign <= (fra_lt_frb and asign) xor ((not fra_lt_frb) and
           (((not addsub) and bsign) xor (addsub and (not bsign))));
```

Due to the simplification of the FPU internal structure, the denormalization unit output is always connected with the adder/subtractor's input. The adder/subtractor has

been written in a parameterisable form in order to meet the different widths of the internal interconnecting FPU buses. The testing of the addition/subtraction unit has been described in chapter 4.1.2 of this thesis.

3.2.5 Multiplication unit description

A multiplication unit performs multiplication between the FP numbers, producing the fractional and exponential parts, which are passed to the post-normalization unit. In order to preserve precision and accuracy of the multiplication result, the input operands' fractional parts must be normalized. The multiplication unit calculates the result using a fast parallel multiplier, which is synthesized from the built-in FPGA DSP slices, which perform the multiplication. The multiplier's top hierarchical block is shown in figure 13. The input data for the multiplier is obtained directly from the FPU inputs as well as the *opsel[2..0]*, *clk* and *rst* signals.

The fractional part of the result is passed to the multiplier's output port *mfres_o[mf-1..0]*, which is connected with the post-normalization unit. Its width *mf* is two times larger than the width of the input faction of each operand. The exponential part of the result is outputted to the *mexp_o[me+1..0]* port, which is connected with a multiplexer. The exponential width is one bit wider than the width of the input exponents *me*. The sign of the multiplication is the signal *msign*. The multiplication algorithm is shown in figure 14.

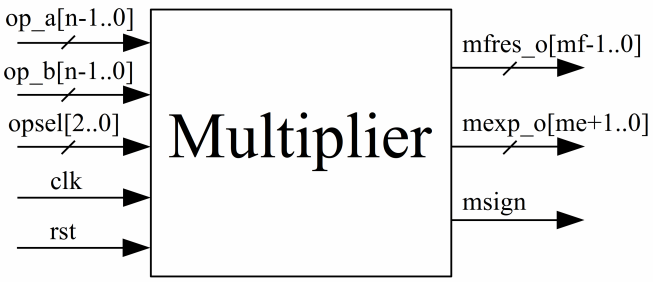


Figure 13. Multiplier's top hierarchical block

According to the above diagram, the multiplication unit reads the values of the fractional parts, exponents and sign bits of the input FP operands from the FPU inputs. If the signal *opsel[2..0]* is equal to "010", then each of the exponents are extended with the additional bit from the left side, producing new signed exponents, whose widths are equal to *[ExpWidth+1]*. The bit extension procedure is important, because it provides the possibility to check the overflow and underflow conditions when they occur.

According to to the FP multiplication algorithm, which is shown in figure 15, the input operands' fractions are multiplied, while their exponents are added. Owing to the nature of the binary division, the resulting fraction is two times wider, than the fraction's width of the input operands.

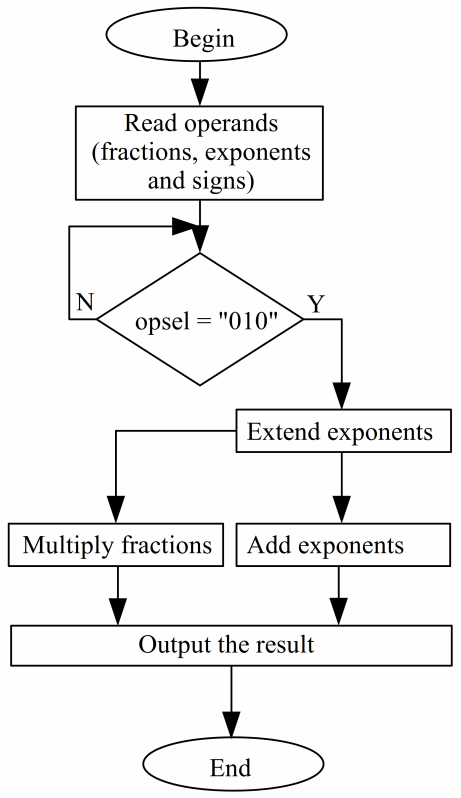


Figure 14. Multiplication unit algorithm

The exponents are added as a signed VHDL data type. Therefore, the sign of the addition operation is taken into account. The resulting exponent's value is passed directly to the post-normalization unit, which defines whether the overflow or underflow occurred or not. The output result's sign is calculated as an exclusive OR operation between signs of the input operands. The multiplier's simulation results and the performance estimation are listed in section 4.1.3 of this thesis.

3.2.6 Division unit description

A divider performs division between the input FP operands. The operands' fractional parts are divided, whilst their exponents are subtracted. Different division algorithms have been described in [15]. However, the Newton-Raphson division

algorithm, which has been described in section 3.1.3, has been used for the FPU implementation due to its simplicity. The speed of the division is the second advantage of this algorithm. Furthermore, the algorithm is not dependent on the operands' values. The only one condition, which must be fulfilled is that fractions have to be normalized. Using not normalized operands entails a possible overflow and wrong division result. The divider's top hierarchical block is shown in figure 15.

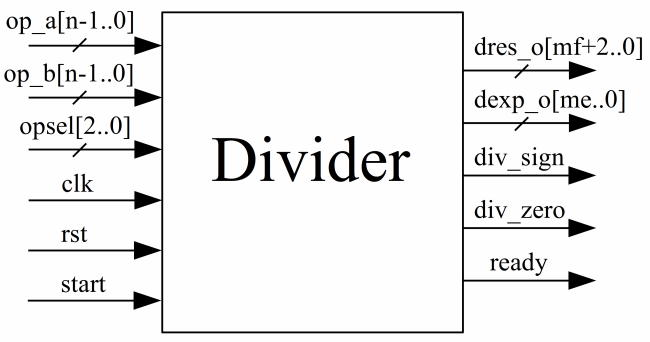


Figure 15. Divider's top hierarchical block

The input operands *op_a[n-1..0]*, *op_b[n-1..0]* and the *opsel[2..0]* signal are directly derived from the FPU inputs. The signals *clk* and *rst* are clock and reset signals respectively, whilst the *start* signal's rising edge initiates the division. The fractional division result and a sign bit are passed to the post-normalization unit, whilst the result's exponential part is connected to the multiplexer. Due to the Newton-Raphson division algorithm's iterative nature, the fractional result's precision has been limited to the input operands' fraction width and it is only extended with three bits from the MSB side in order to avoid the fractional overflow. The *ready* signal produces a single high level pulse when the division unit has finished the calculation. A division by zero exception is generated by the divider itself and propagated to the post-normalization unit.

The Newton-Raphson division algorithm calculates the result through a series of multiplications, which is why the result is always approximate. Both the final fraction and exponent are calculated in parallel in order to maintain the FPU throughput at the top. In order to save the calculation time, the fractional result and exponent are calculated in parallel. The number of the iterations, which are needed to obtain the division result depends on the fractional parts' widths. The division algorithm is shown in figure 16. The division unit starts its operation when a value of the *opsel[2..0]* signal is equal to "011" and a single pulse has been applied to the the FPU input *start*. The division starts if the divisor is not equal to zero. A dividend *N* and a divisor *D* are read

from the FPU input. The dividend is always operand A, whilst the divisor is always operand B. The sign extension procedure is applied to both operands' exponential parts. The resulting division sign is calculated as an exclusive OR operation between the input operands' signs. The division unit does not generate the overflow and underflow exceptions by itself and the post-normalizer performs the normalization of the result instead. By contrast, as was mentioned above the division by zero exception is generated by the division unit itself. According to the diagram, which is described in figure 16, *N* is the dividend (numerator), *D* is the divisor (denominator) and *R* is a reminder. In order to avoid the potential overflow in the result's fractional part, the extra most significant bits from the left are added to the operands' fractions. The whole part in the fraction is needed in cases, when, for example, the divider's fraction is in a range between 0.0625 and 0.5. Therefore, to avoid a possible overflow situation, the divisor should not be less than 0.0625, if the dividend's value is close to one. The fractions' overflow condition is not occur if dividend's and divisor's fractions are normalised. According to the diagram, which is listed in figure 16, at the first division step, the divisor D is subtracted from the constant with a value of two. The result is stored in the reminder R. At the second step, both the dividend N and the divisor D are multiplied with the reminder. Multiplication is performed in parallel to save calculation time. However this increases the usage of the FPGA DSP slices. Then the new divisor's result (after the multiplication step) is used in the second iteration, when is is subtracted again from a constant with a value of two. Then the cycle repeats as many times as needed, until the divisor D(x) will reach its maximum value, which is close to a value of one. After that the fractional division result is outputted as well as the resulting division exponent. The number of iterations depends on the input operands' widths. The division unit has been written in a parameterisable style and it adjusts its internal registers' and buses' widths according to the constants defined in the FPU package. The division unit testing is described in chapter 4.1.4 of this thesis.

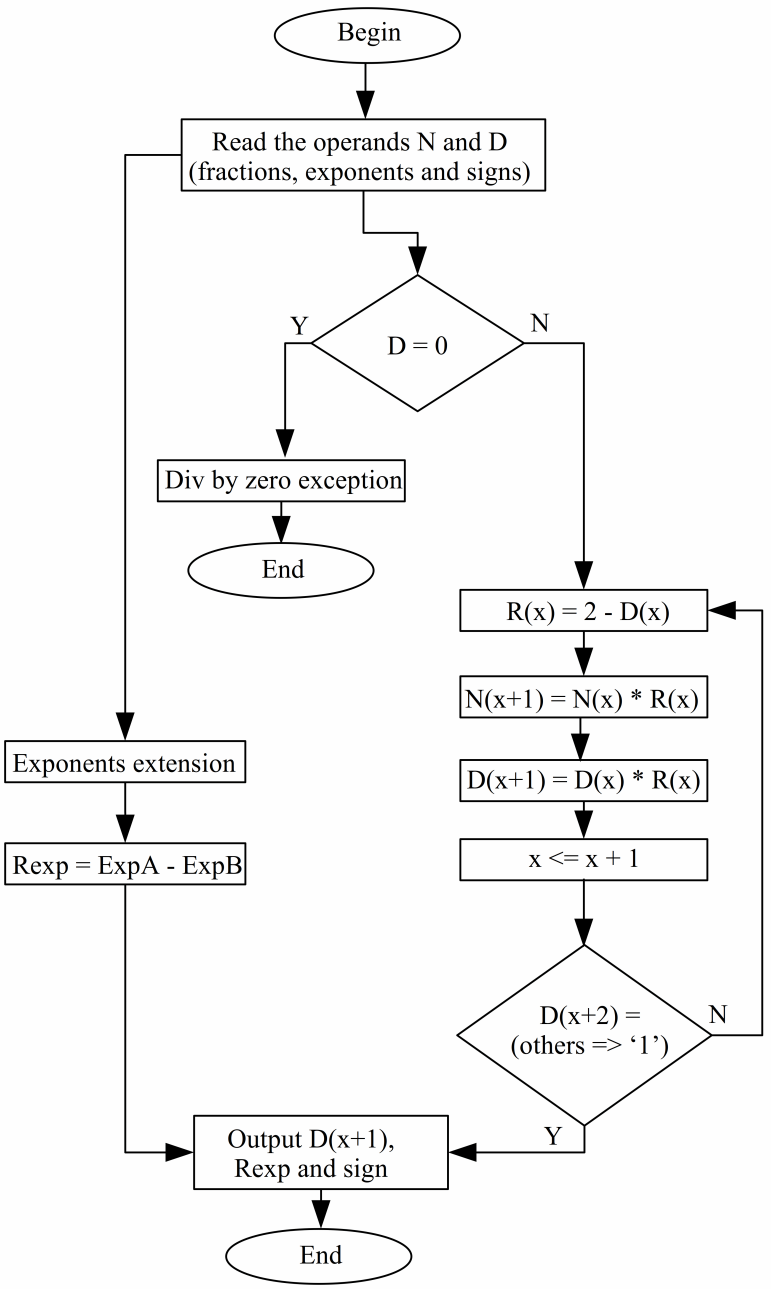


Figure 16. Division algorithm

3.2.7 Generic barrel shifter description

A barrel shifter is a combinational digital circuit, which can shift a data word by a specified number of bit positions. Due to the absence of sequential logic in the barrel shifter, the shift operation does not require synchronization signals. The generic barrel shifter is the heart of the FPU and it can parametrize its input and output widths and the control shifting input width after the synthesis according to the constants, which are defined in the FPU package. Barrel shifter has been designed using the barrel shifter's

example from [7] on page 566. This example has been modified in order to meet the generic FPU requirements. The top level hierarchical block of the barrel shifter is shown in figure 17. The input *shiftIn[n+4..0]* takes the initial binary data word, which is needed to be shifted. The width of the data word is defined by the width of the input operand' extended fractions. The width of *n* is always equal to *[FractionalWidth+4]*. The input *numShifts[mlog-1..0]* specifies the number of required shifts, which are performed for the input *shiftIn[n+4..0]*. The number *mlog* is the logarithm of base two of the number *n*, rounded up to the nearest larger whole number. This number defines the number of the multiplexer layers inside the barrel shifter.

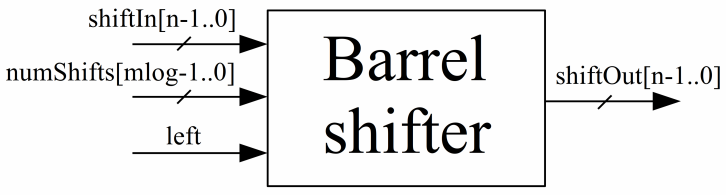


Figure 17. Barrel shifter's top hierarchical block

The *mlog* is defined in the FPU package as a fr_log_width constant as follows:

```
constant fr_log_width: integer := integer(ceil(log2(real(FractWidth))));
```

The signal *left* defines whether a shift to the left or shift to the right will be performed. If this signal has a high level, then the selected direction of shifting is to the left. When the *left* has a low level, then shifting to the right is selected. Finally, an output *shiftOut[n-1..0]* provides a shifted data word. All the barrel shifter inputs are connected to the internal FPU multiplexer, which is commutated by the control FSM, which has been described in part 3.2.10 of this thesis.

The generic barrel shifter's internal structure is shown in figure 18. Depending on the fraction width, which is defined in the FPU package, a different number of shifting slices will be synthesized inside the barrel shifter. Each slice consists of two multiplexers, which are connected in such a way that every slice is able to shift an input data word to the left or right by only a fixed number of bit positions. Each slice shifts the input for a number of bit positions, which is equal to *2**n*, where *n* is the number of the control shift input bit, starting from 0. This means that the shifting slice 0 shifts the input data word by one position. For the next shifting slices a shifting step is increased

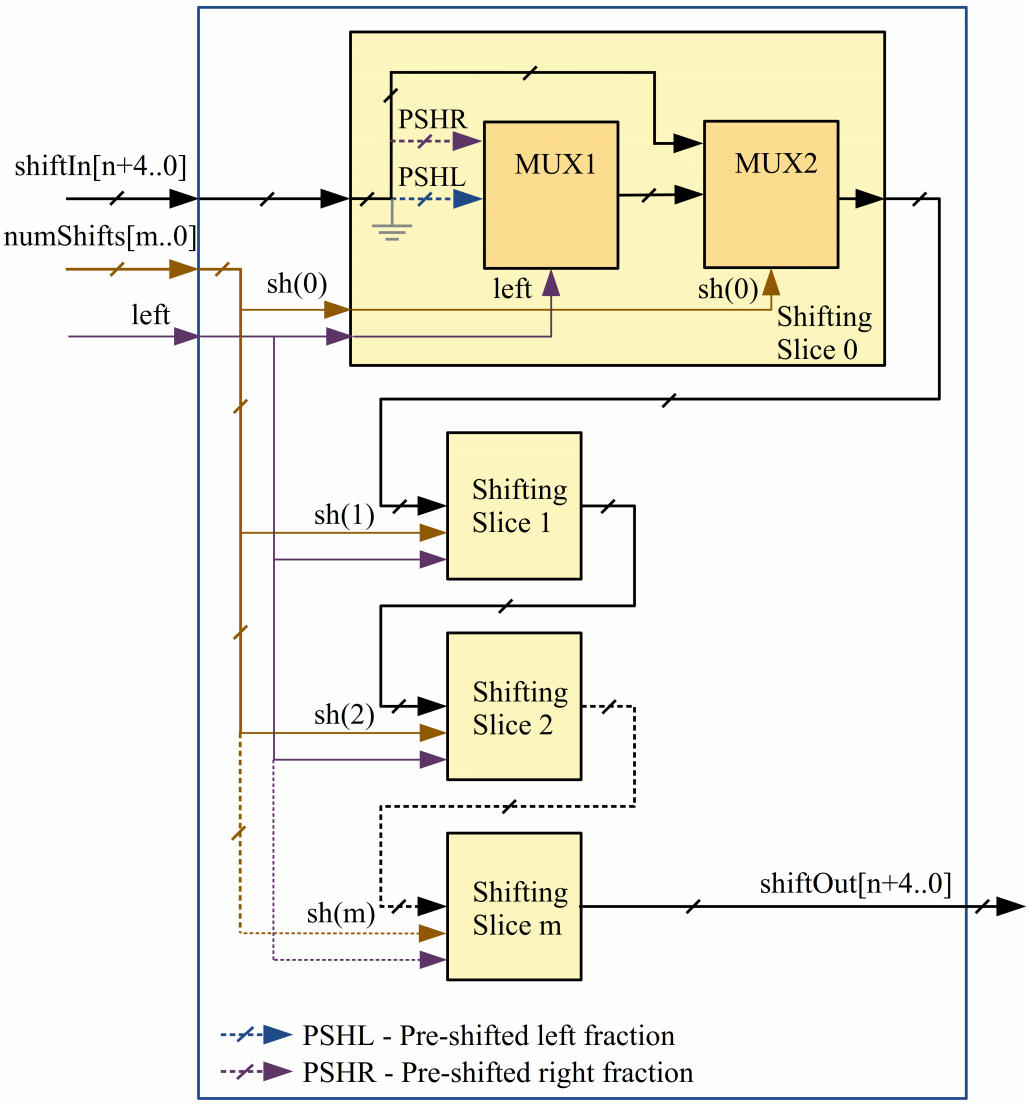


Figure 18. Generic barrel shifter's internal structure

by two times from the number of shifts in the previous slice. All the internal shifting slices in the barrel shifter are identical, except for the inputs connection order of the first multiplexer *MUX1* in each shifting slice. In order to understand the mechanism of shifting, the shifting slice 0 has been more detailed in figure 18. According to the drawing, each shifting slice consists of two multiplexers with the same input data width. The first multiplexer *MUX1* commutates the two data inputs, which are connected to the pre-shifted to the right and pre-shifted to the left input data word. The shift, which is performed by each of barrel shifter's slices is a logical shift. This means that 'empty' places after shifting of a data word are filled with zeros. This is opposite to the arithmetical shift, when the least significant bit acquires the value of the most significant bit or vice versa, whether the arithmetical shifts to the left or to the right are performed respectively. The first multiplexer *MUX1* selects the pre-shifted to the left

input and passes its value to the output, which is connected with the second multiplexer *MUX2*. The first multiplexer *MUX1* is controlled by the barrel shifter's input *left.* The second multiplexer *MUX2* is controlled by one of the bits of the input *numShifts[m..0]*. Each bit of this input corresponds to the shifting slice with the same number. The second multiplexer *MUX2* passes a shifted data word to its output, when its controlling input acquires a high level. If it has a low level, then the slice's input data word is passed unchanged to the slice's output, which in turn is connected to the next slice in the presented slice chain. All other slices operate in the same manner. Finally, the resulting shifted data word is derived from the slice *m* to the barrel shifter's output *shiftOut[n+4..0]*. This output is connected to both denormalization and post-normalization units. Due to the simplicity of the barrel shifter's algorithm and its combinational logic nature, its algorithm diagram has not been shown. The generic barrel shifter testing has been described in section 4.1.6 of this thesis.

3.2.8 FPU multiplexer description

The FPU has an internal separate multiplexer, which is used in order to commutate control and data signals between the barrel shifter and other FPU peripherals, which are the denormalization and post-normalization units. Furthermore, the multiplexer has a dedicated channel for selecting one of the exponential parts of the calculation result, which are derived from the denormalizer, multiplier and divider.

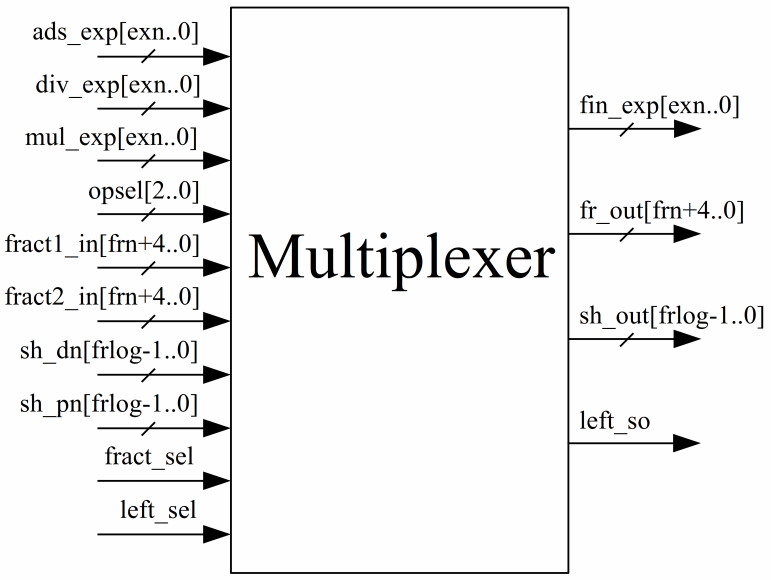


Figure 19. Multiplexer's top hierarchical block

This internal FPU multiplexer significantly reduces the number of the post-normalizer's inputs and gives the physical possibility to have the only one barrel shifter instead of two. The top level hierarchical multiplexer's block is shown in figure 19.

Table 10. Multiplexer input and output signals

Signal	**Type**	**Description**
ads_exp[exn..0]	Input, bus	The exponential part of the result, which is derived from the addition/subtraction unit
div_exp[exn..0]	Input, bus	The exponential part of the result, which is derived from the division unit
mul_exp[exn..0]	Input, bus	The exponential part of the result, which is derived from the multiplication unit
opsel[2..0]	Input, bus	Selects the final exponent for the post-normalizer input according to the operation being performed
fract1_in[frn+4..0]	Input, bus	Fraction which is derived from the denormalization unit
fract2_in[frn+4..0]	Input, bus	Fraction which is derived from the post-normalization unit
sh_dn[frlog-1..0]	Input, bus	This input is connected to the denormalizer and defines the number of the required shifts
sh_pn[frlog-1..0]	Input, bus	This input is connected to the post-normalizer and defines the number of the required shifts
fract_sel	Input, single	Selects the denormalizer fraction (when it is '0') or the post-normalizer fraction (when it is '1')
left_sel	Input, single	Selects between the sifting right or left. The fraction is shifted left when this input acquires a high level
fin_exp[exn..0]	Output, bus	The final exponent, which is passed to the post-normalization unit
fr_out[frn+4..0]	Output, bus	Fraction, which is passed to the post-normalization unit
sh_out[frlog-1..0]	Output, bus	Shift controlling outputs for the barrel shifter. This signal bus defines for how many bit positions a fraction must be shifted.
left_so	Output, single	Shifting direction selection output. It is connected directly to the barrel shifter.

A list of all the multiplexer's inputs and outputs is presented in table 10. The important thing is that the *left_sel* input affects the output signal *left_so* only if the *fract_sel* signal has a high level and, therefore, the post-normalization unit is connected to the barrel shifter through the multiplexer. In this case, the value of the signal *left_sel* is copied to the *left_so* output. The number *exn* is equal to the width of the input operands' exponential parts.

The number *frn* is the fractions' width and *frlog* is the constant, which is calculated by the VHDL code expression, which is listed in section 3.2.7 of this thesis. The multiplexer is controlled by the internal FSM, which has been described in section 3.2.8, and by the post-normalization unit, which has been described in the next chapter 3.2.9. The FSM affects the *fract_sel* input state, while the post-normalizer can select the shifting direction by the *left_sel* signal, whether the shifting to the right or to the left is required.

3.2.9 Post-normalization unit description

A post-normalization unit, which is shown in figure 20, performs normalization of the FP intermediate results of addition, subtraction, multiplication and division operations. Also, the post-normalization unit performs the input operands' normalization. The normalized result is outputted from the FPU. Exceptions are signalled if they occur.

The post-normalization unit is connected internally to other FPU peripherals. Some of the inputs and outputs are connected directly with the FPU input and outputs through the auxiliary internal digital logic. Due to the significant number of the post-normalization unit's inputs and outputs, they have been split in two tables. Table 11 corresponds to the inputs, whereas table 12 shows the outputs.

In these two tables a constant *frn* defines the fractional width, a constant *exn* is the exponential width and, finally, the width of the controlling shifting output is encoded by a constant *frlog*. The FP result, which is provided to the post-normalization unit from the different sources is selected by the *opsel[2..0]* input. These sources are both input operands ports and internal computational blocks. The current result's source is defined by the *opsel[2..0]* signal's value. The relation between this signal and the

selected intermediate FP result, which will be used for the post-normalization, is shown in table 13.

The post-normalization unit receives the result of the calculation or one of the input operands and, depending on this received value, decides whether this value requires the post-normalization or not. If the fraction is not normalized, then the post-normalization unit calculates by how many bit positions the fraction must be shifted to the left or to the right in order to obtain the value *n*, which is satisfied with an expression $0.5 \leqslant n < 1$.

The shifting direction is defined by the fraction's value. In cases, when overflow or underflow situations occur (if the value of the fraction is greater or equal to '1'), the fraction will be always shifted to the right in order to fit into the given FP format. The fraction is shifted to the left in cases, when its value is less than 0.5. Depending on how the fraction is shifted, the exponent is incremented when the fraction is shifted to the right, or decremented when the fraction is shifted left. The exponential value is

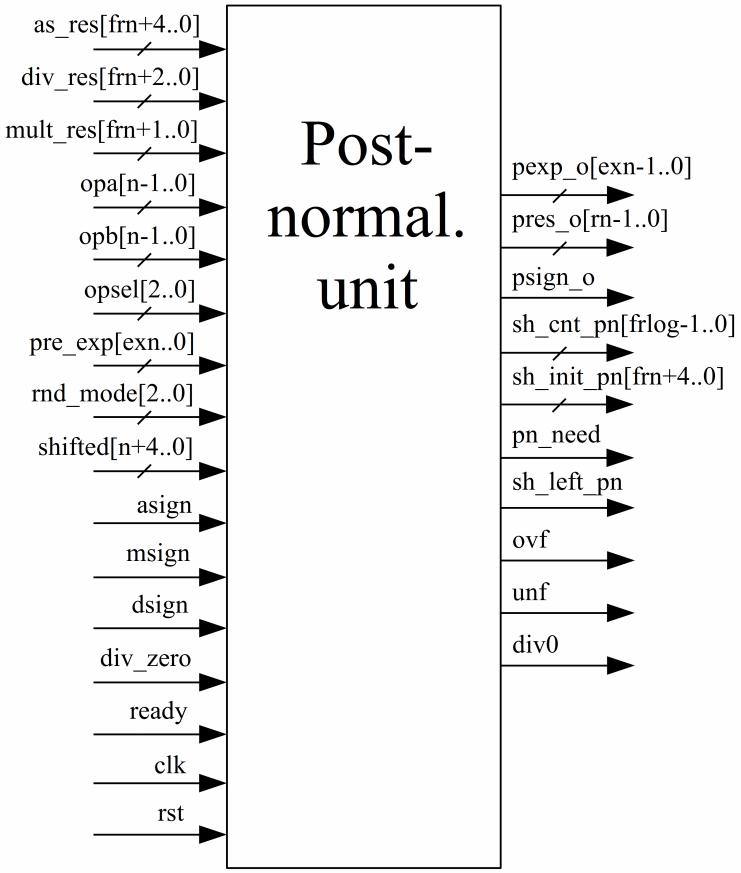


Figure 20. Post-normalizer's top hierarchical block

increased by a number, which is equal to the number of needed shifts to the left or right.

According to the tables 11 and 12, the width of the calculation results from different sources is equal to *[frn+4..0]*. The three most significant bits are used for storing the fractional number's whole part of the the FP number result, whilst the two least significant bits in this range are used for rounding purposes. The post-normalization unit controls the barrel shifter's direction input through the FPU internal multiplexer by using the output signal *sh_left_pn*. When this signal has a high level, the

Table 11. Post-normalization unit input signals

Signal	**Type**	**Description**
as_res[frn+4..0]	Input, bus	Fractional part of the addition/subtraction result
div_res[frn+2..0]	Input, bus	Fractional part of the division result
mult_res[frn+1..0]	Input, bus	Fractional part of the multiplication result
opa[n-1..0]	Input, bus	Input operand A
opb[n-1..0]	Input, bus	Input operand B
opsel[2..0]	Input, bus	Selects the fractional and exponential parts of the computational results or input operands
pre_exp[exn..0]	Input, bus	Exponent, which derives from the multiplexer
rnd_mode[2..0]	Input, bus	Selects the rounding mode for the resulted fraction
shifted[n+4..0]	Input, bus	Shifted fraction from the barrel shifter's output
asign	Input, single	Sign bit from the addition/subtraction unit
msign	Input, single	Sign bit from the multiplication unit
dsign	Input, single	Sign bit from the division unit
div_zero	Input, single	Division by zero exception input. Derives from the divider
ready	Input, single	This signal is derived from the control FSM and signals that the post-normalization result can be outputted
clk	Input, single	Synchronization signal
rst	Input, single	Reset signal

Table 12. Post-normalization unit input and output signals

Signal	**Type**	**Description**
pexp_o[exn-1..0]	Output, bus	Exponential part of the FPU result
pres_o[frn-1..0]	Output, bus	Fractional part of the FPU result
sh_cnt_pn[frlog-1..0]	Output, bus	This signal is connected to the multiplexer and it defines the number of shifts for the fraction
sh_init_pn[frn+4..0]	Output, bus	Initial fraction, which is needed to be shifted
psign_o	Output, single	Sign of the FPU result
sh_left_pn	Output, single	This output is connected with the multiplexer and indicates whereas the fraction is needed to be shifted left, when is '1', or right, when it is '0'
pn_need	Output, single	This output indicates does the FPU resulted fraction requires the post-normalization. The post-normalization is needed, when this signal is '1'. This signal is connected with the control FSM
div0	Output, single	Division by zero exception flag. It is connected to the FPU output
ovf	Output, single	Overflow exception flag. It is connected to the FPU output
unf	Output, single	Underflow exception flag. It is connected to the FPU output

exponential part of the result is shifted left by the number of shifts, which are defined by the output signal bus *sh_init_pn[frn+4..0]*. Shifting of the fraction to both left and right is possible only during the post-normalization procedure due to the fact that shifting to the left is not required for denormalization.

The overflow and underflow exceptions are generated by the post-normalization unit itself. Depending on the exponential value of the FP result after the post-normalization, one of these exceptions is raised. If the resulted exponent, which has the width *n*, acquires a value after the post-normalization, which is larger than 2^n-1 or less than $-(2^n)$, then the overflow or underflow exception is signalled respectively. The algorithm of the post-normalization unit is shown in figure 21.

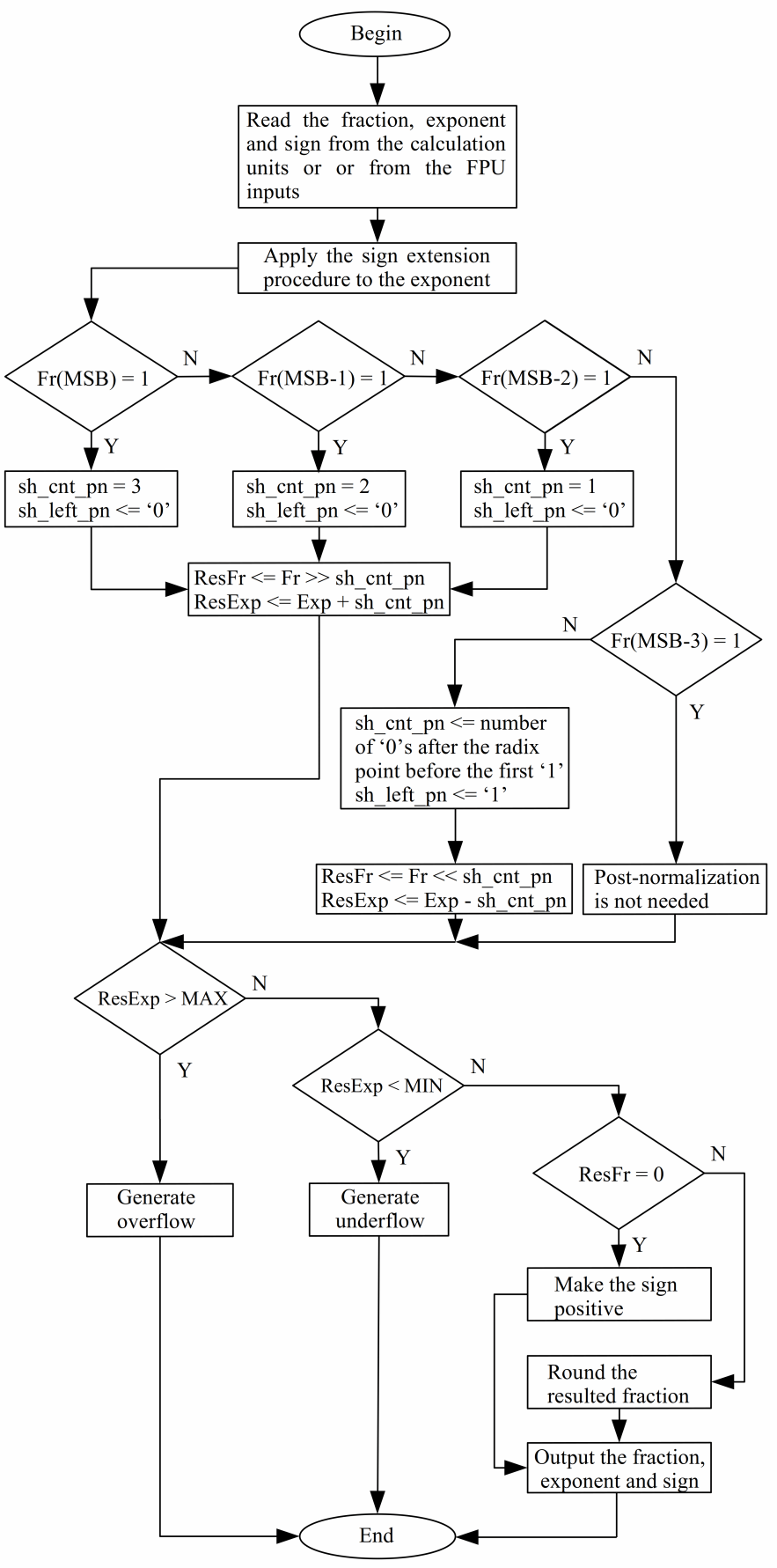


Figure 21. Post-normalization unit algorithm

Table 13. Selected result's source for the post-normalization

Input opsel[2..0]	**Result's source**
000/001	Addition/subtraction unit
010	Multiplication unit
011	Division unit
100	Not used, reserved
101	Not used, reserved
110	Operand's A input
111	Operand's B input

The FPU supports five rounding modes, which have been listed in table 7 in part 3.2.1 of this thesis. The normalized and rounded fraction is derived from the post-normalizer and outputted from the FPU, concatenating with the exponential part of the FP result and its sign. This occurs at the rising edge of the output signal *ready*. This signal is produced by the control FSM, which has been described in chapter 3.2.10. The exception signals are derived from the post-normalization unit and outputted from the FPU.

3.2.10 Control FSM description

A control FSM performs synchronization between different internal FPU blocks. The control FSM top hierarchical block is shown in figure 22. The FSM controls the multiplexer's inputs in order to connect the barrel shifter's input to the fraction, which derives from either denormalization or the post-normalization unit. Furthermore, the control FSM generates the ready signal *fsm_rdy*, which triggers the post-normalizer to store the ready calculation or normalization result and its sign to the FPU output port. Depending on the operation, which is being performed by the FPU, the FSM controls the fraction's selection output *fsm_sel*, which connects the barrel shifter to the denormalization and post-normalization units at the specific clock cycle. This is defined by the FPU synchronization signal *clk* and starts from the moment, when one of the FPU commands has been triggered by the rising edge of the FPU input signal *start*. The rising edge at this input initiates the FSM operation. During the addition or subtraction

operation, the control FSM ensures that the denormalizer is connected to the barrel shifter first, owing to the necessity of denormalization before the calculations. After that the FSM connects the post-normalization unit to the barrel shifter. For other operations the denormalizer's fraction is always disconnected from the barrel shifter and the FSM generates the output signal *fsm_rdy* only, which affects the post-normalizer as was mentioned above.

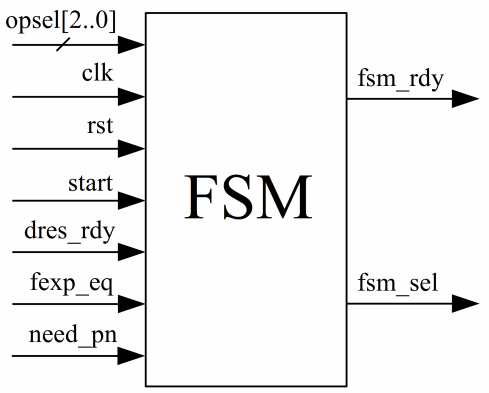


Figure 22. Control FSM top hierarchical block

The control FSM input and output signals are listed in table 14. The FSM algorithm varies upon a selected FPU operation. In general, the FSM has its five internal states. These states are: *idle*, *calc_st1*, *calc_st2*, *postn_st1* and *postn_st2*.

The FSM begins its operation from the state *idle* and it concerns any FPU operation. The states *calc_st1* and *calc_st2* are used while performing the calculations, whereas *postn_st1* and *postn_st2* are the states of the control FSM, when the FPU post-normalizes the result of calculations. Due to the complexity of the control FSM algorithm, it has been split into the three diagrams. Each diagram illustrates different groups of the FPU operations' flow. These groups are the addition/subtraction, multiplication/division and, finally, the numbers' normalization. Such a distinction is explained by the operations, which have been grouped according to their similarity. The FSM output signals *fsm_sel* and *fsm_rdy* directly depend on the internal FSM signals *fsel1*, *fsel2*, *psel1, psel2* and *sel*. Each diagram shows the formulas how the signals *fsm_sel* and *fsm_rdy* are affected by the internal signals. Different groups of the operations require the FSM to switch between the different FSM states, which sequence is specific for each case and it is shown in the following three diagrams.

Table 14. Control FSM input and output signals

Signal	**Type**	**Description**
opsel[2..0]	Input, bus	This signal indicates to the FSM, which FPU operation is being performed
clk	Input, single	Synchronization signal
rst	Input, single	Reset signal
start	Input, single	This signal indicates to the FSM that one of the FPU operation has been initiated
dres_rdy	Input, single	Division result ready input. Connected with the divider's output *ready*.
fexp_eq	Input, single	Exponents equation input. Connected with the output *dexp_eq* of the denormalization unit
need_pn	Input, single	This input signal indicates the necessity of the post-normalization for the FPU operation result. Connected with the output *pn_need* of the post-normalization unit
fsm_rdy	Output, single	FPU result ready signal. It triggers the post-normalization unit to output the final result from the FPU
fsm_sel	Output, single	Fraction selection signal. This signal is connected to the multiplexer, which switches the barrel shifter's input between the denormalizer and post-normalizer

The first diagram, which is shown in figure 23, illustrates the FSM algorithm in cases when the FPU performs the addition or subtraction. In this diagram all states of the FSM are used, given that the FP addition and subtraction operations usually require both denormalization and post-normalization, depending on the input operands' values.

According to this diagram, after the signal *start* has been applied, the control FSM compares whether the operands' exponents are equal or not. If they are equal, then the signal *fsel1* will acquire a high value at the next FSM state *calc_st1*. During this state the FSM checks the condition, when *sel = '1'*, *need_pn = '0'* and *fexp_eq = '1'*. If this condition is met, then the FSM switches to the next state *post_st2*, the signal *psel1* will acquire a high level, while the signal *fsel2* will become low. If the condition is not satisfied, then the FSM checks the exponents' equity. If the exponents are equal, then the signals *fsel1* and *psel1* obtain low levels at the next FSM state *post_st1*. If

exponents are not equal, then the next FSM state will be switched to the *calc_st2* and the signals *fsel2* and *psel1* will obtain the values '1' and '0' respectively.

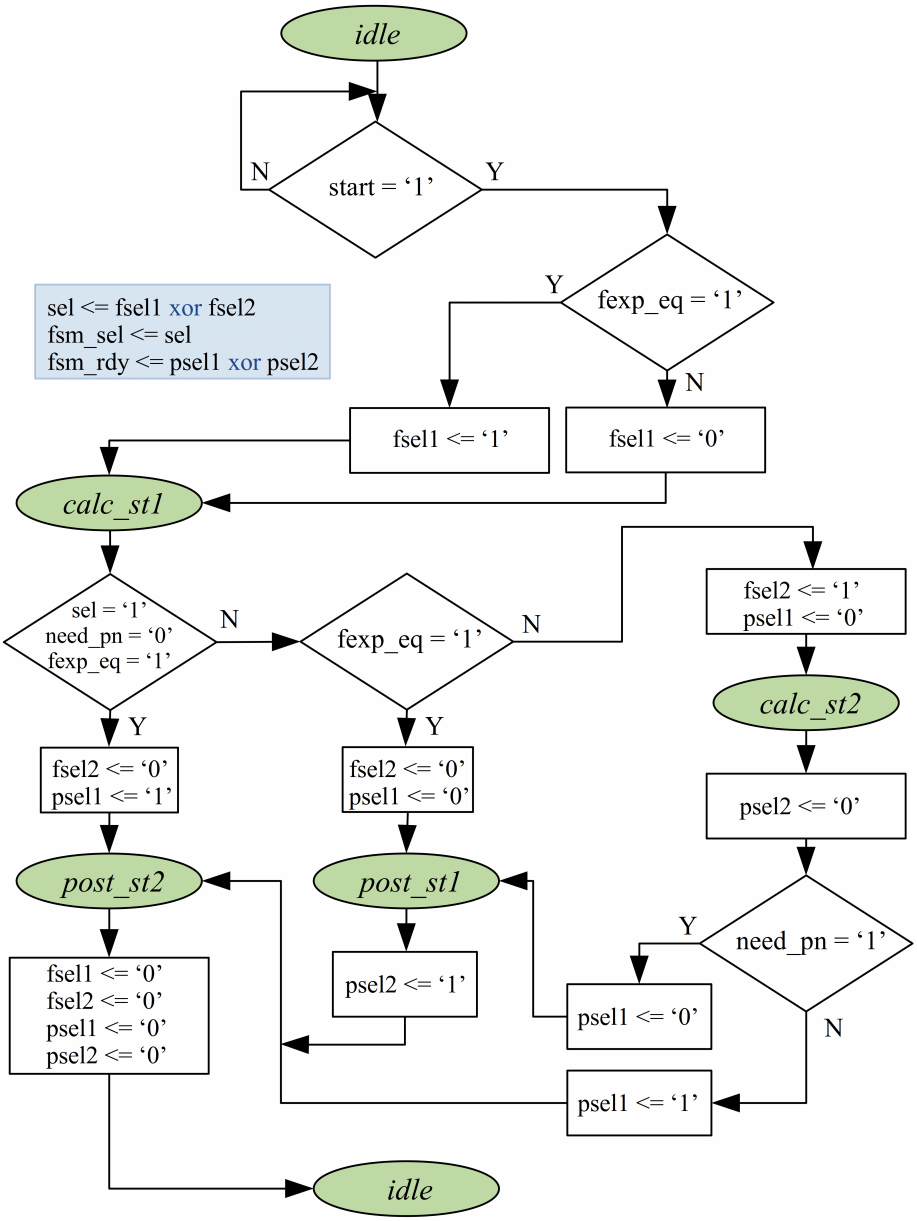


Figure 23. FSM algorithm for the addition and subtraction

At the state *calc_st2* the FSM checks the value of the signal *need_pn*. If its value is high, then the signal *psel1* will acquire a high level at the next FSM state *post_st2*. In cases, when the value of the signal *need_pn* is low, the signal *psel1* will obtain a low level at the next FSM state *post_st1*. The signal's *psel2* value will be changed to '0' at any of these two states *post_st1 or post_st2*. According to the remaining part of the shown diagram, at the state *post_st1* the value of the signal *psel2*

is going to be changed to ‘1’ at the next FSM state *post_st2*. Finally, at the FSM state *post_st2*, all the internal signals’ next values will be reset when the FSM reaches the state *idle*. There is no a clear distinction between additions and subtractions in the diagram, because these are the interchangeable operations as has been described in chapter 3.2.3.

The second diagram, which is shown in figure 24, describes the FSM algorithm for the multiplication and division operations. For these two FPU operations the FSM state *calc_st1* is not required. After the input signal *start* has been applied, the FPU is switched to the state *calc_st2* and the signal *fsel1* is set. The signal *psel1* is held in a zero state. If the division operation is selected, then the control FSM checks continuously the division ready flag *dres_rdy*. When this flag switches to a high level, the FSM checks if the division result requires the post-normalization by reading the signal’s *need_pn* value. If the division result does not require the post-normalization, then the FSM is switched to the state *post_st2* and the signals *fsel1* and *psel2* acquire a high level, while the other signals are held in zero. If the post-normalization is required, then the FSM is switched to the state *post_st1* and only the signal *fsel1* obtains a high level. At the state *post_st1* if the division operation is ongoing then the FSM is switched to the state *post_st2*, while setting the same internal controlling signals as if the result did not require the post-normalization. If at the state *post_st1* the multiplication operation is being performed, then the signal *need_pn* is checked. If this signal, then the signal *psel2* acquires a high level, when it is switched to the state *post_st2*. Otherwise, the FSM is switched to the state *post_st2* while both signals *psel1* and *psel2* obtain low levels. The state *post_st2* ensures that all the internal controlling signals are reset, when the FSM has changed its state to the *idle* state. When the multiplication or division is executed again, the scenario described above is repeated again.

The absence of the state *calc_st1* is explained by the fact that the denormalization is not required for these two operations. However, applying the normalized operands is highly recommended in order to preserve the result’s precision.

A diagram, which is depicted in figure 25, explains the FSM algorithm when the FPU normalizes one of the input operands. This algorithm is straightforward and contains only two operational states *post_st1* and *post_st2*.

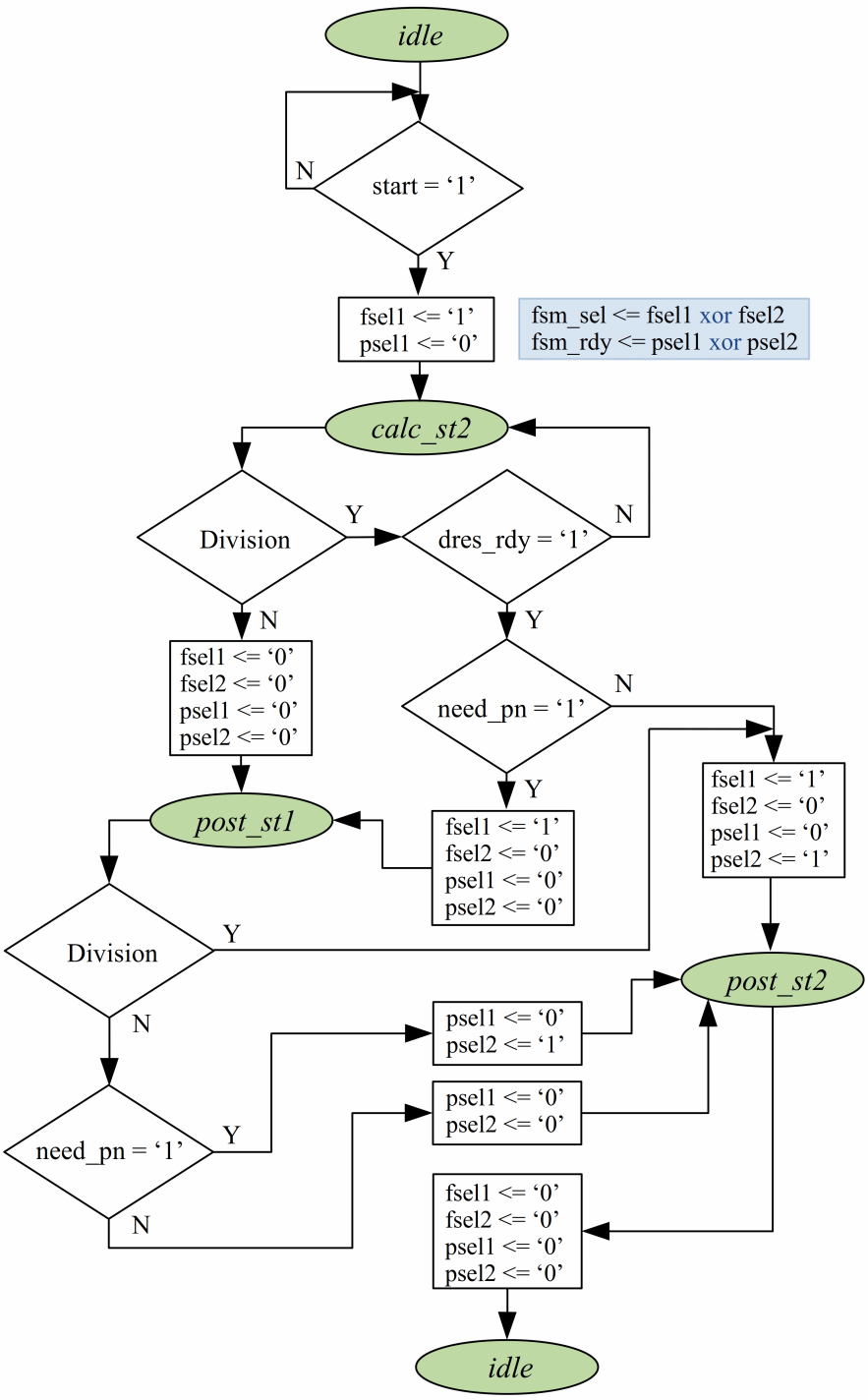


Figure 24. FSM algorithm for the multiplication an division

After the signal *start* has been applied and the normalization operation has been selected, the FSM is switched to the state *post_st1*, while changing the signal's *fsel1* value to '1'. Then the control FSM toggles to the state *post_st2*, setting the signal *psel2* to the high level, while keeping the other signals low. At the state *post_st2* the FSM ensures that all the internal controlling signals will be reset at the state *idle*.

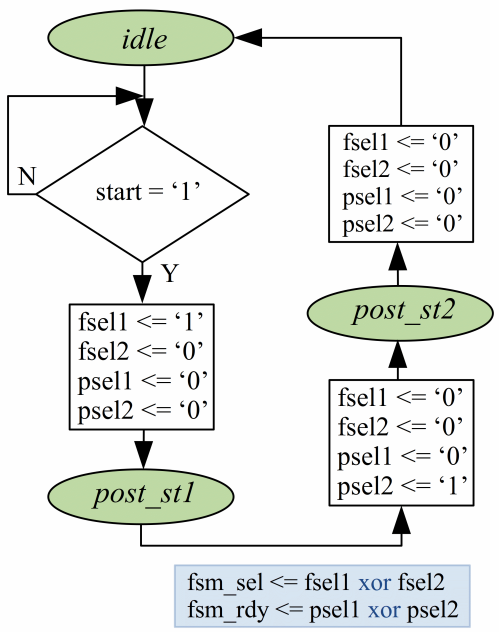


Figure 25. FSM algorithm for the operands' normalization

If any of the FPU commands are executed again, then one of the described algorithms is repeated again. The FSM is not sensitive to any input controlling signals, when one of the FSM algorithms is ongoing. If some internal signals have not been shown, then they are considered as these, which acquire a low level at the next FSM state.

3.3 Synthesis Results and Timing Constraints

The designed FPU is able to be synthesized with different FP formats, allowing the designer to select the best trade-off between precision and the occupied area in the FPGA. The resulted IP core can be configured before the synthesis. The environment ISE Project Navigator has been used for the FPU synthesis. According to the technical task of this thesis, which has been set, the widths of fractional and exponential parts of the FPU operands can be defined in the FPU package, which corresponds to the file *fpu_pack.vhdl* in the FPU design. In order to estimate the resources usage, the FPU has been synthesized according to the five FP defined standards: half precision, single precision, double precision, extended precision and 128-bit FP. The result of this synthesis is shown in table 15, which contains the most important hardware units that have been extensively used. Also, the multipliers, adders, counters and FSMs are inferred in the FPU core synthesis, but in the table they are not shown. Due to the quite

small percentage of the logic utilization, which targets these hardware resources, they are ignored in the FPU synthesis summary table. The number of GPIOs, which are necessary for the FPU implementation on the real hardware platforms depends on the FPU input operands' width, and, consequently on the width of the result.

Table 15. FPU synthesis summary

	16 bit FP	**32 bit FP**	**64 bit FP**	**80 bit FP**	**128 bit FP**
Multipliers	3	3	3	3	3
Adders/Subtractors	10	10	10	10	10
Slice Registers	233	423	875	1068	1646
Comparators	7	8	8	8	8
Multiplexers	156	197	286	253	327
DSP48E1s	3	10	39	44	106
Slice LUTs	495	914	2287	2742	5500
Used LUT-FF pairs	207	389	800	976	1506

Different FPU formats entail different usage of the internal FPGA hardware resources and, due to this, different timing delays affect the maximum FPU operating frequency. In the technical task the aim was to create as small design as possible, compromising with the maximum design operational speed. In order to achieve this, the speed grade -1 has been selected in the ISE design properties.

Table 16. FPU timing summary

FPU format	**Minimum period**	**Maximum frequency**
16 bits	6.608 ns	151.332 MHz
32 bits	10.564 ns	94.661 MHz
64 bits	14.612 ns	68.436 MHz
80 bits	15.415 ns	64.871 MHz
128 bits	26.989 ns	37.053MHz

Table 16 contains timing summary for the FPU designs with different FP format. According to this table, the FPU designs with greater precision are restricted to operate at higher frequencies. This is due to the signal propagation delays. According to the previous two tables, the resulting generic FPU design is enough efficient and can be used in other designs as the separate internal computational block.

However, in cases, when a design is required to be fast, it cannot be too compact. Fast data processing requires a pipeline, which is needed in order to increase the computational throughput. The additional extra computational units, which perform calculations in parallel, will be inferred in the synthesis process. This usually needs the preliminary analysis of the required extra hardware with its binding to the computational cycles, number of which must minimal. Due to this, in order to achieve the better FPU performance comparing with the achieved one in the thesis, the pipeline for multiplication and division operations could be created. Another FPU improvement, which could increase its throughput, is to pipeline the FPU computational and de/post-normalization steps. This means that the computational and de/post-normalization operations could be executed at the same time.

On the other hand, the pipelining allows to balance critical combinational path of the design. The increasing combinational delay in the barrel shifter and multiplier is the main reason why the pipelining can be used. Pipelining them allows to increase the clock frequency at the cost of increasing the number of clock steps for some FPU parts and/or operations. In addition, having higher clock frequencies can simplify to some extent the design of other modules in FPGA, which are not part of the FPU.

Conclusions

In this part of the thesis have been described the FPU itself and its internal parts such as the adder/subtractor, multiplier, divider, denormalizer, post-normalizer, barrel shifter, multiplexer and control FSM. The operation algorithms and the input and output signals have been presented for every hierarchical block. This data introduces the relatively simplified internal FPU structure, which gives understanding of its hardware features and the internal data flow. Due to the hardware complexity of almost all internal units, their detailed internal structure is not shown, except the barrel shifter, which is the major valuable part of the generic FPU design. The simulation results of the FPU internal blocks have been described in the next part of the thesis.

The generic FPU design has been successfully created and simulated using the "black box" principle in VHDL, which allows for separating the complex hardware algorithms into more simplified ones. This entails the possibility of design separation and the simulation of its separate internal hardware units, which in turn can be combined into one complex design.

Creating the design as parameterisable is highly attractive and practical for different applications. Indeed, the code re-usability will increase to a certain extent, if the VHDL generic features are used as much as possible. This allows digital designers to configure their designs according to the specific project needs, eliminating the necessity of modifying the large and complex code pieces.

4 Generic Floating-Point Unit Simulation and Testing

This section describes the generic FPU simulation and testing flow. Each FPU internal block has been simulated and verified in order to validate its proper operation and, consequently, the correctness of the FPU behaviour. The timing diagrams have also been included in this section. The clock cycle usage for each FPU operation has been estimated. In addition, some of the FPU operations have been validated with different FP formats in part 4.2 of this thesis. This ensures that the FPU design is generic and fully complies with the thesis technical task. The FPU design contains a generic testbench, which allows the FPU to be simulated and tested before its real implementation. This testbench validates the FPU operation at the synthesis level.

4.1 Testing Methodology

The FPU is tested by applying the appropriate signals to its inputs. The internal units are tested by using their internal intermediate signals, which are derived from each block. The sequence of the signals being applied is described further. The response from the FPU outputs is checked and the specific assertions are made. This allows one to ensure that the FPU provides the correct calculation results before the real implementation. For this purpose, ISE Project Navigator tool has been used in order to create a testbench in VHDL for the test signals generation and checking the output responses from the FPU. The testbench verifies the addition, subtraction, multiplication, division and normalization operations. Also it checks whether the overflow, underflow and division by zero exceptions arise or not.

The testbench has been written in the generic form, which allows its usage with testing of different FPU designs with various FP formats. Although the division result cannot be checked by the testbench assertions accurately due to its iterative division nature, all other operations are successfully verified with different widths of the fraction

and exponent. However, the division unit has been simulated using the VHDL *real* data type and the division results, which have been obtained, are valid.

Each following chapter describes how the different FPU internal parts have been tested. In this part of the thesis, a single precision FP format has been used in order to test the internal FPU blocks' operation. By contrast, in part 4.2 of this thesis different FP formats have been used in order to verify the FPU parametrizability feature, which demonstrates the flexibility of the FPU design.

4.1.1 Denormalization unit testing

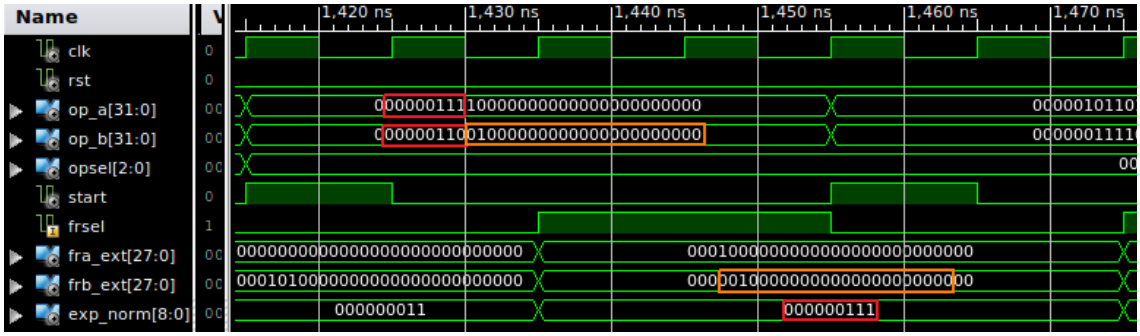
A denormalization unit is tested by applying the operands with different exponential parts to the FPU inputs. The denormalization unit provides the initial data for the addition/subtraction unit. The common exponent is calculated by the denormalization unit. The denormalizer always shifts one of the fractions to the right. A set of the FP operands, which have been used for the denormalization unit testing are shown in table 17.

Table 17. FP operands for the denormalization unit testing

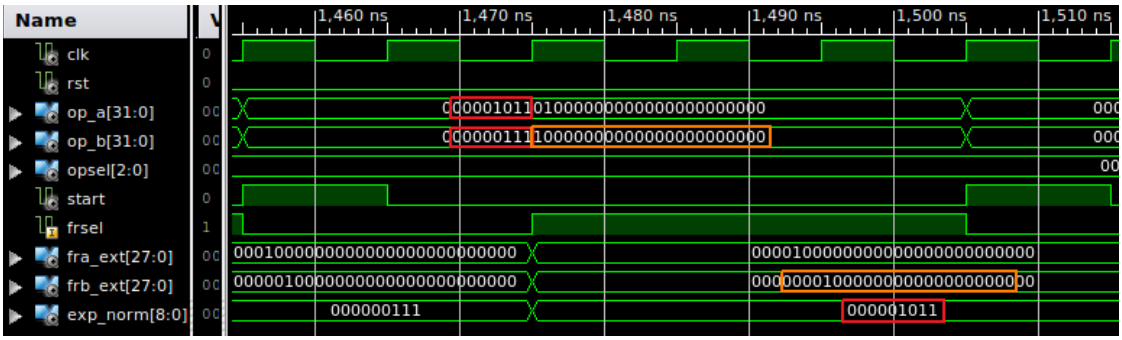
Denorm. example	**Operand A**		**Operand B**		**Fraction A**	**Fraction B**	**Common exponent**
	Fraction	**Exponent**	**Fraction**	**Exponent**			
a	0.5	7	0.25	6	0.5	0.125	7
b	0.25	11	0.5	7	0.25	0.031250	11
c	0.75	10	0.5	12	0.187500	0.5	12
d	0.625	9	0.25	9	0.625	0.625	9
e	0.875	-7	0.5	-5	0.218750	0.5	-5

The denormalization examples *a* - *c* have been simulated and shown in figure 26. For the example *a* the colour orange highlights that the fraction with a value 0.25 is shifted by one bit position to the right in order to align the exponents. The exponents before the denormalization and the final common exponent are marked with the colour red. As has been illustrated, the final common exponent is always equal to the larger exponent of the input operands. This targets all the next denormalization examples,

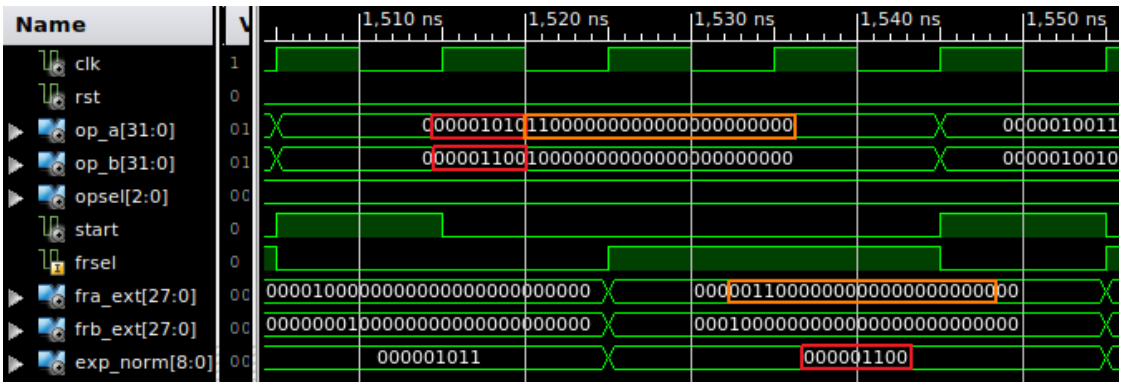
except the fourth example, where both operands' exponents are equal. The denormalization examples *d* and *e* are shown in figure 27.



a) $A = 0.5 \times 2^{7}, B = 0.25 \times 2^{6} \rightarrow B = 0.125 \times 2^{7}$



b) $A = 0.25 \times 2^{11}, B = 0.5 \times 2^{7} \rightarrow B = 0.031250 \times 2^{11}$

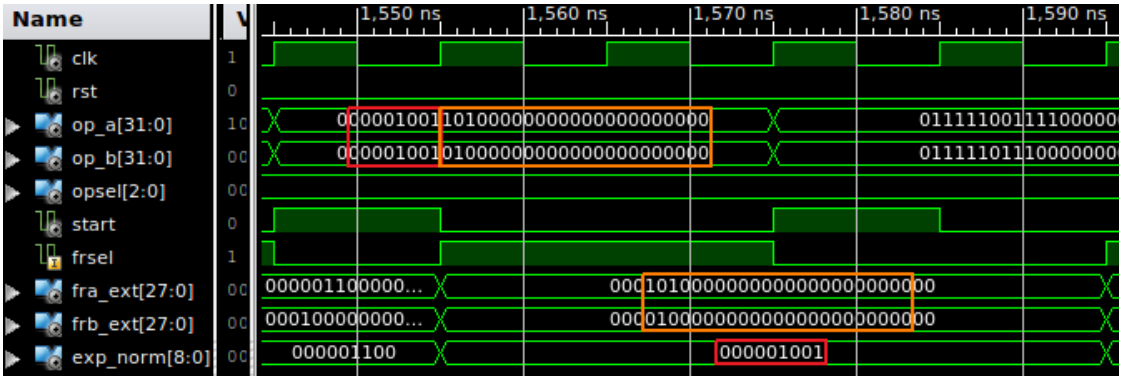


c) $A = 0.75 \times 2^{10} \rightarrow A = 0.1875 \times 2^{12}, B = 0.5 \times 2^{12}$

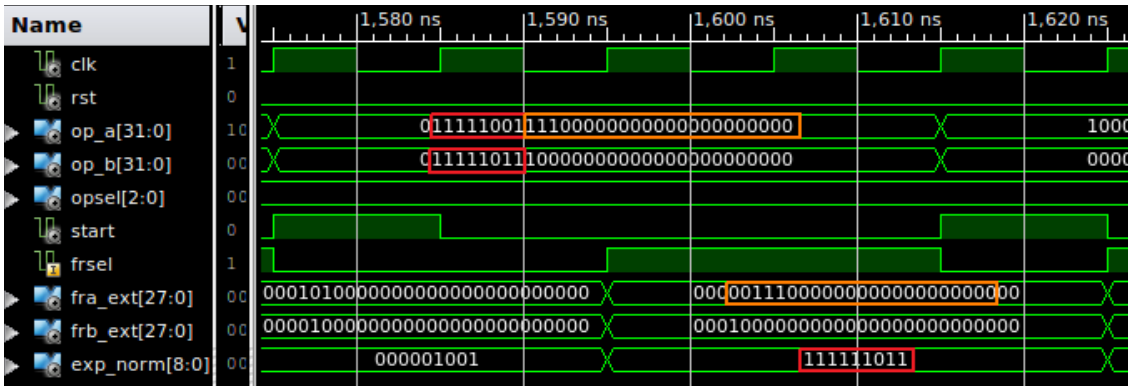
Figure 26. Denormalization examples a - c

In order to test the denormalization operation in the FPU, the addition or subtraction operation must be initiated by a single high pulse at the *start* input. This entails the operands being denormalized before the addition or subtraction can be performed. The exponents equity affects the number of clock cycles *clk*, which are required for the denormalization unit to perform the denormalization. If exponents of

the input operands are different (examples *a*, *b*, *c* and *e*), then the denormalization is started taking two clock cycles to be finished.



d) $A = 0.625 \times 2^{9}, B = 0.25 \times 2^{9}$



e) $A = 0.875 \times 2^{-7} \rightarrow A = 0.21875 \times 2^{-5}, B = 0.5 \times 2^{-5}$

Figure 27. Denormalization examples d - e

However, if the exponents are equal (example *d*), then the denormalization is not performed and only one clock cycle is spent, storing the operands to the denormalization unit output at the rising edge of the signal *frsel*. This signal is derived from the control FSM output and shown in all the the simulation diagrams above.

4.1.2 Addition/subtraction unit testing

Testing of the addition/subtraction unit can be performed with the testing of the entire FPU, due to the fact that the FPU output result is much easier to observe then the internal signals. Furthermore, this ensures that other FPU internal blocks work correctly. The input signal *opsel[2..0]* has a value “000” for the addition and “001” for the subtraction operations respectively. The list of operands, which have been used for the addition and subtraction simulation is shown in table 18. Although the testbench

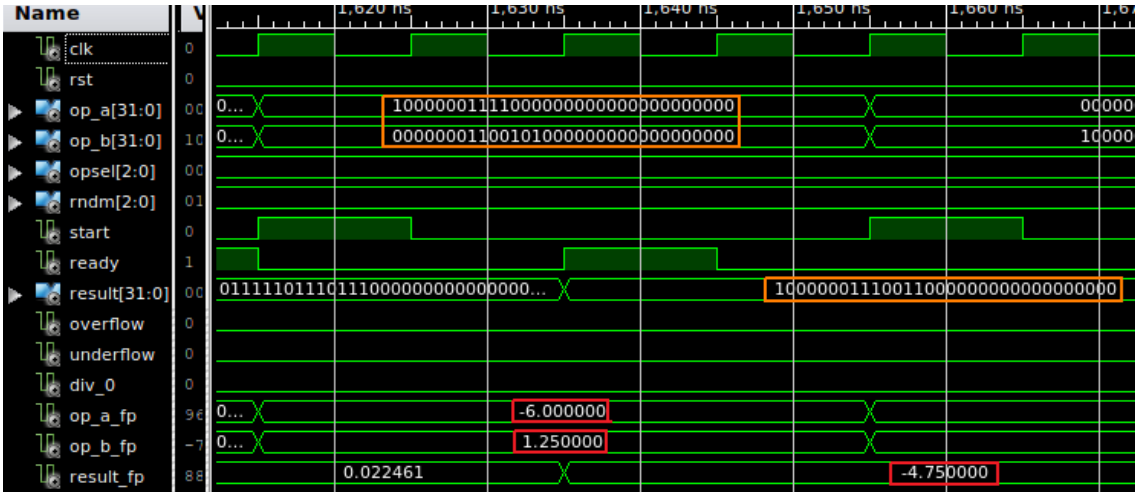
contains more addition and subtraction examples, in order to demonstrate the adder/subtractor operation only some of them have been shown in the thesis report.

Table 18. FP operands for the addition/subtraction unit testing

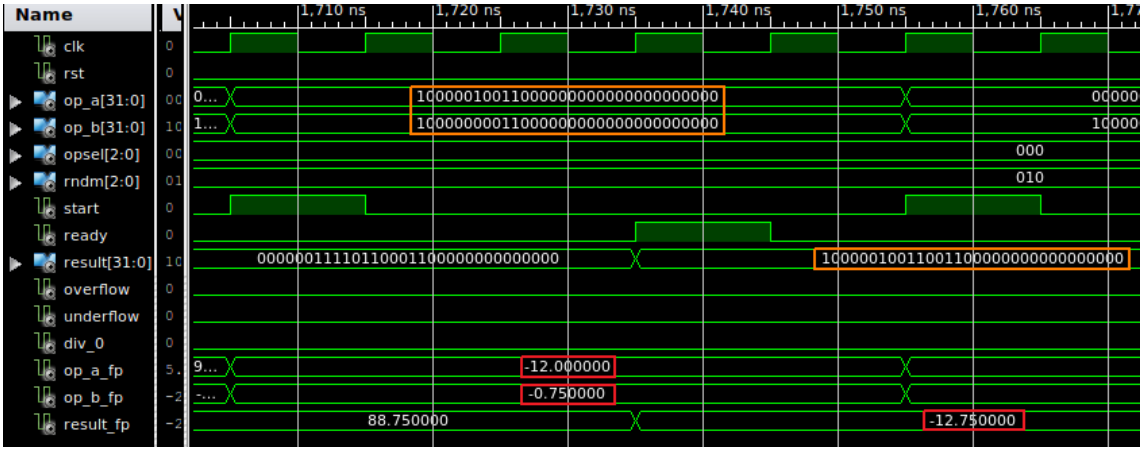
N	**Operation**	**Operand A**	**Operand B**	**Result**
a	(-6) + 1.25 = (-4.75)	(-0.75) * 2^3	0.15625 * 2^3	(-0.59375) * 2^3
b	(-12) + (-0.75) = (-12.75)	(-0.75) * 2^4	(-0.09375) * 2^3	(-0.796875) * 2^4
c	48 - 112 = (-64)	0.75 * 2^6	0.875 * 2^7	(-0.5) * 2^7
d	(-1.5) - (-7) = 5.5	(-0.75) * 2^1	(-0.875) * 2^3	0.59375 * 2^3
e	“+infinity” + “+infinity”	0.875 * 2^127	0.625 * 2^127	Overflow
f	“- infinity” - “- infinity”	0.5 * 2^(-128)	0.25 * 2^(-128)	Underflow

The addition/subtraction simulation results are shown in figures 28 and 29. The examples *a* and *b* are addition examples, whereas examples *c* and *d* correspond to subtraction. The colour orange highlights the binary input operands and the addition/subtraction results, whereas the corresponding real numbers are shown with the colour red. If the addition or subtraction result is larger or less than maximum or minimum FPU value, which can be stored after the post-normalization, then the underflow or underflow exception occurs respectively. The overflow and underflow exceptions are simulated in the examples *e* and *f* in figure 29. The operations are initiated at the rising edge of the *start* signal. The result of the addition or subtraction always becomes available at the same time, as when the FPU output signal *ready* produces a rising edge.

The number of clock cycles, which are needed for the addition or subtraction operations, depends on the input operands and the intermediate addition/subtraction result. Due to this, the time of the calculation can vary from two to four clock cycles. As has been described in the previous section, denormalization requires 1-2 clock cycles. In addition, the post-normalizer requires two clock cycles if the post-normalization is needed. Otherwise the post-normalization is not performed and the result is outputted one clock cycle after the post-normalization has been started.



a) $(-0.75)\times 2^3 + 0.15625\times 2^3 = (-0.59375)\times 2^3$



b) $(-0.75)\times 2^4 + (-0.09375)\times 2^3 = (-0.796875)\times 2^4$

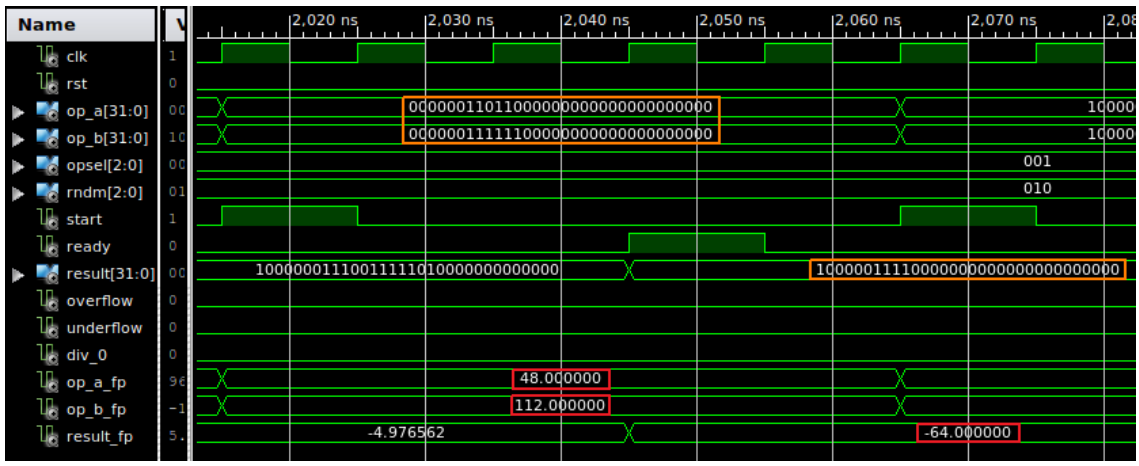
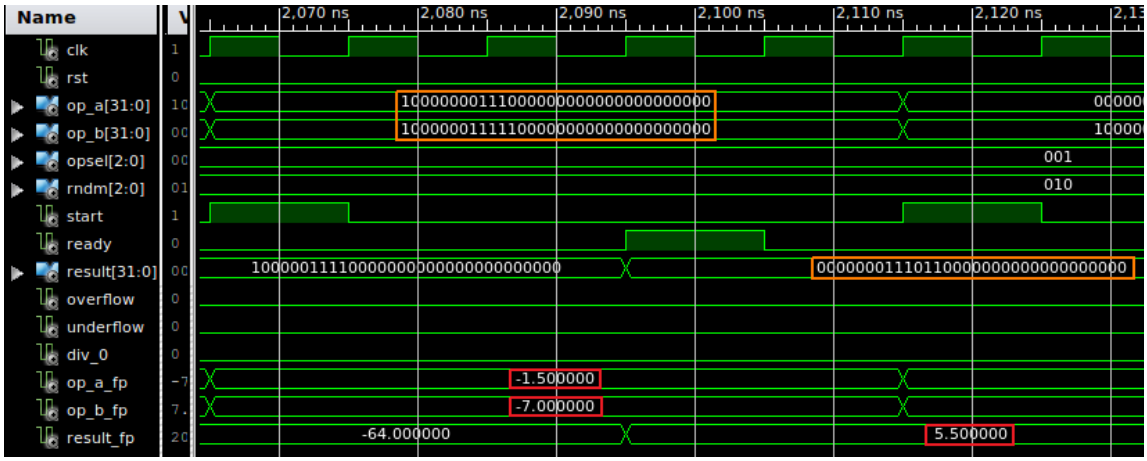
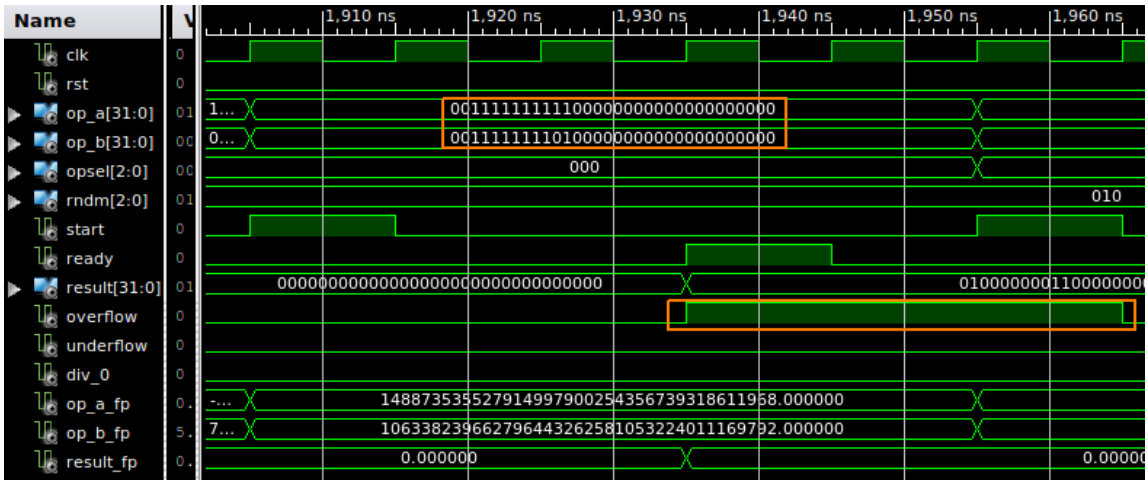


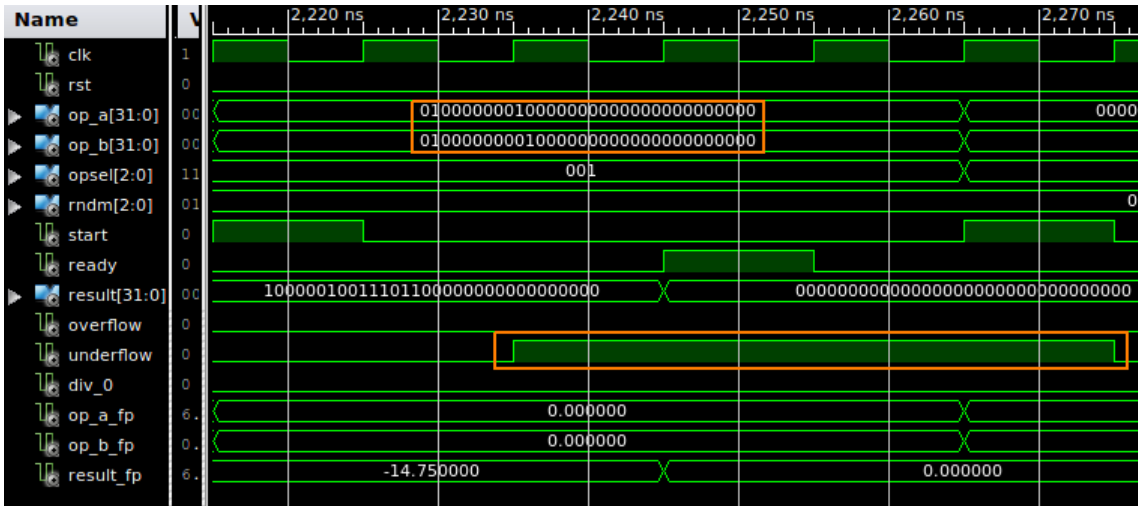
Figure 28. Addition/Subtraction examples a – c



d) $(-0.75)\times 2^{1}-(-0.875)\times 2^{3}=0.59375\times 2^{3}$



e) $0.875\times 2^{127}+0.625\times 2^{127}=overflow$



f) $0.5\times 2^{-128}-0.25\times 2^{-128}=underflow$

Figure 29. Addition/Subtraction examples d - f

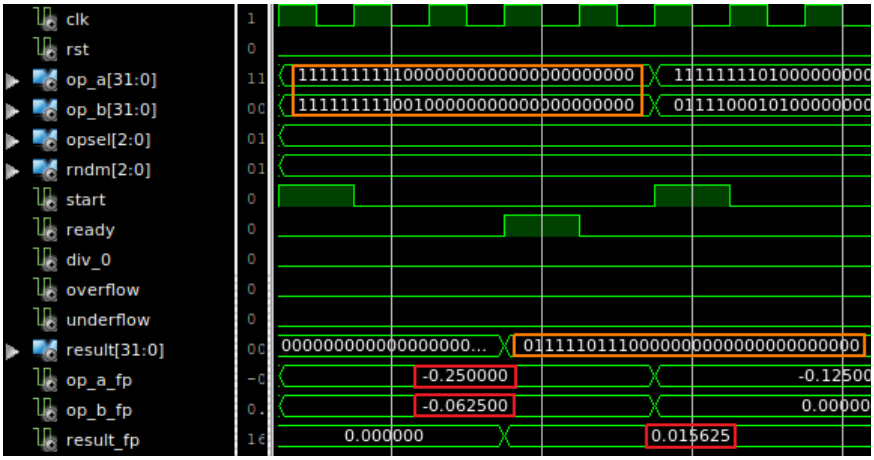
When exceptions arise, the FPU output result must be ignored. The overflow exception occurs only when the final adder/subtractor's operation is addition, whereas the underflow relates to subtraction. The relation between the required and the actual addition/subtraction operation has been described in detail in part 3.2.4 of this thesis.

4.1.3 Multiplication unit testing

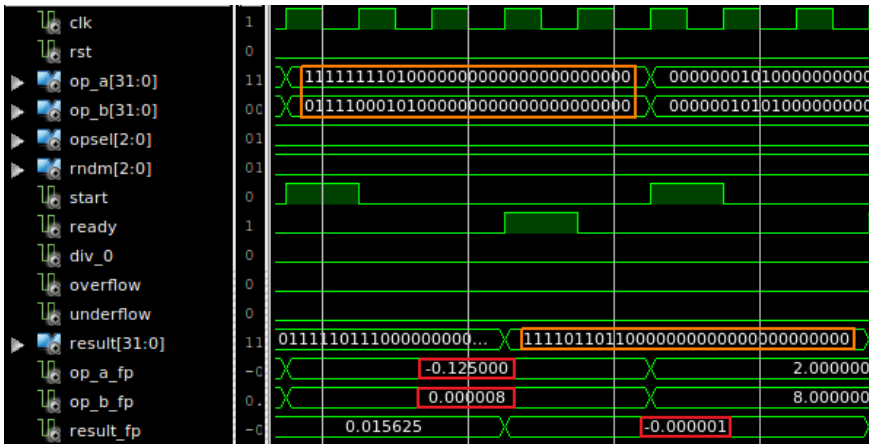
In order to test the multiplication unit, the *opsel[2..0]* signal must have a value "010". This switches the FPU to the multiplication mode. The rising edge at the input *start* triggers the multiplication operation. A set of numbers, which have been used in order to verify the multiplier, are shown in table 19. The simulation results for the multiplication examples are shown in figures 30 and 31. The two last examples *e* and *f* correspond to the overflow and underflow exceptions.

Table 19. FP operands for the multiplier testing

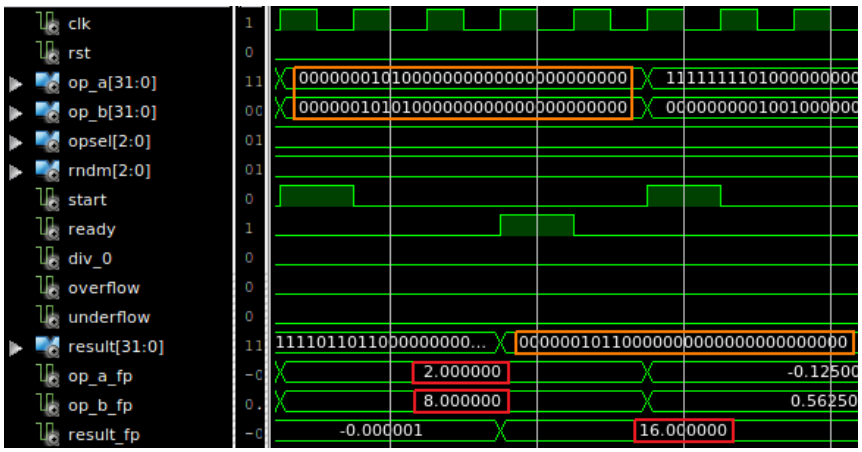
Example	**Form**	**Operand A**		**Operand B**		**Result**	
		Fraction	**Exponent**	**Fraction**	**Exponent**	**Fraction**	**Exponent**
a	Binary	(-0.5)	(-1)	(-0.125)	-1	0.5	(-5)
	Decimal	(-0.25)		(-0.0625)		0.015625	
b	Binary	(-0.5)	(-1)	0.25	-15	(-0.5)	(-19)
	Decimal	(-0.25)		0.000008		(-0.000002)	
c	Binary	0.5	2	0.25	5	0.5	5
	Decimal	2		8		16	
d	Binary	(-0.25)	20	(-0.375)	12	0.75	29
	Decimal	(-262144)		(-1536)		402653184	
e	Binary	0.5	127 (MAX)	0.5	2	Overflow	
	Decimal	→ "infinity"		2			
f	Binary	0.25	-128 (MIN)	0.5	-3	Underflow	
	Decimal	→ "- infinity"		0.0625			



a) $(-0.5)\times 2^{-1}\times 0.125\times 2^{-1}=0.5\times 2^{-5}$

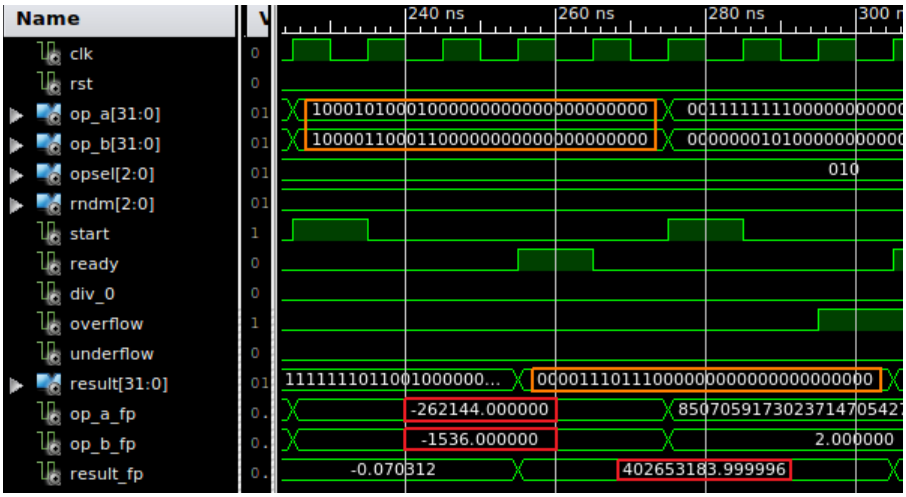


b) $(-0.5)\times 2^{-1}\times 0.25\times 2^{-15}=(-0.5)\times 2^{-19}$

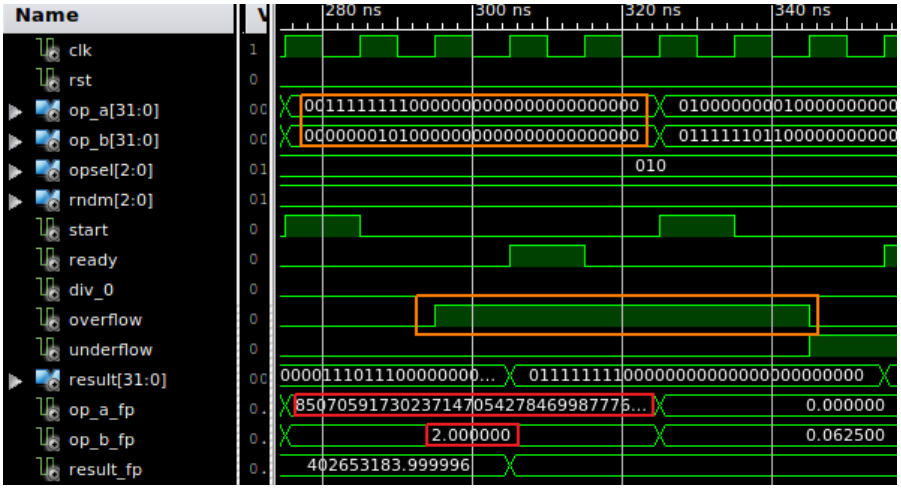


c) $0.5\times 2^{2}\times 0.25\times 2^{5}=0.5\times 2^{5}$

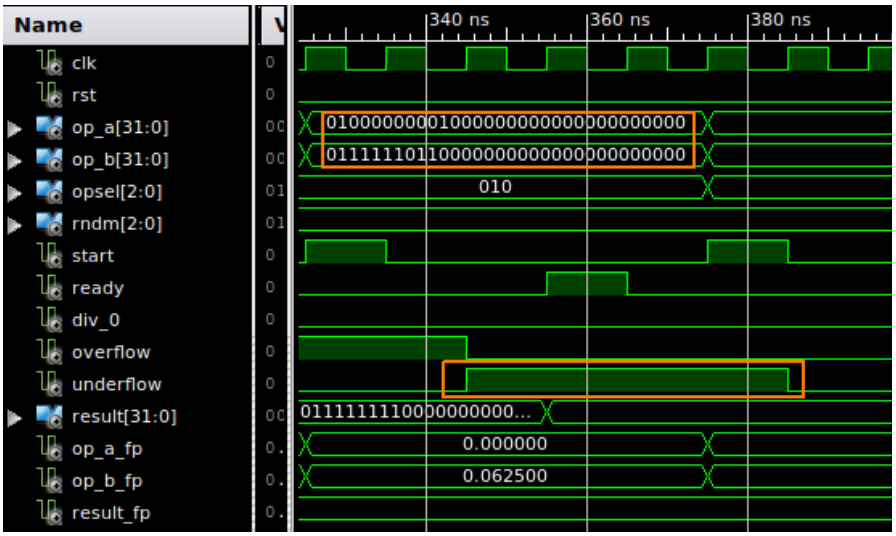
Figure 30. Multiplication examples a - c



d) $(-0.25)\times 2^{20}\times(-0.375)\times 2^{12}=0.75\times 2^{29}$



e) $0.5\times 2^{127}\times 0.5\times 2^{2}=overflow$



f) $0.25\times 2^{-128}\times 0.5\times 2^{-3}=underflow$

Figure 31. Multiplication examples d - f

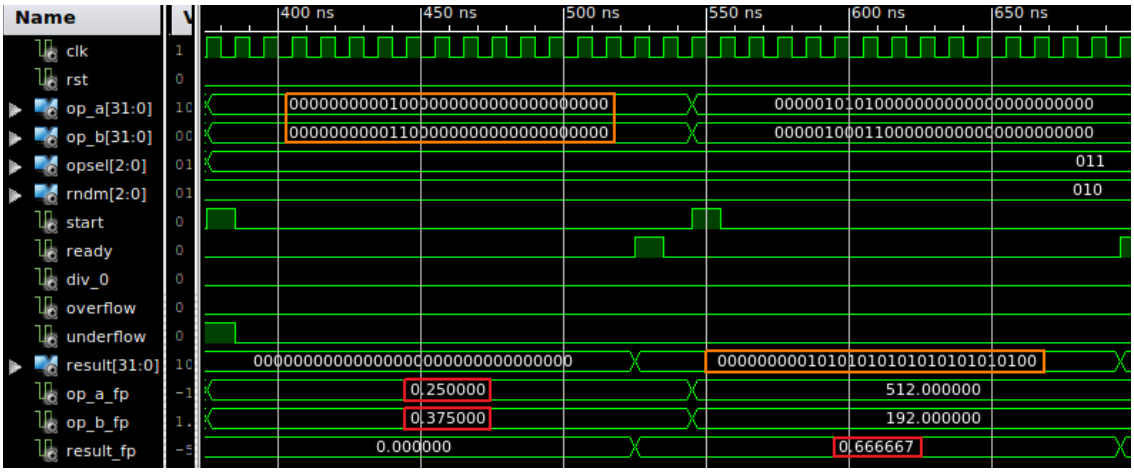
The FP multiplication is performed by the FPU in three clock cycles, including the post-normalization. A rising edge at the output *ready* indicates that the multiplication result can be read from the FPU output. The overflow and underflow conditions are signalled with an active high logical level after two clock cycles.

4.1.4 Division unit testing

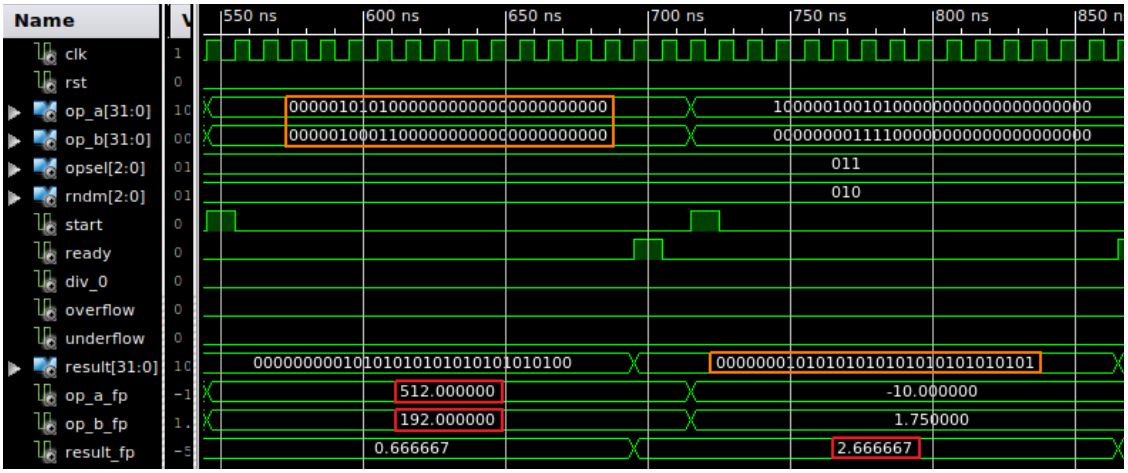
The division unit is tested by applying the input operands and triggering the division operation. In order to perform this, the input *opsel[2..0]* must be set to the value "011" and a single high pulse at the input *start* has to be applied. Due to the division algorithm, which has been used in the FPU, the division result is produced after some number of clock cycles. This number depends on the operands' width and their values. When the division is finished, the output signal *ready* produces a high pulse. Table 20 contains several division examples for the division testing and checking of its exceptional conditions such as the overflow, underflow and division by zero.

Table 20. FP operands for the divider testing

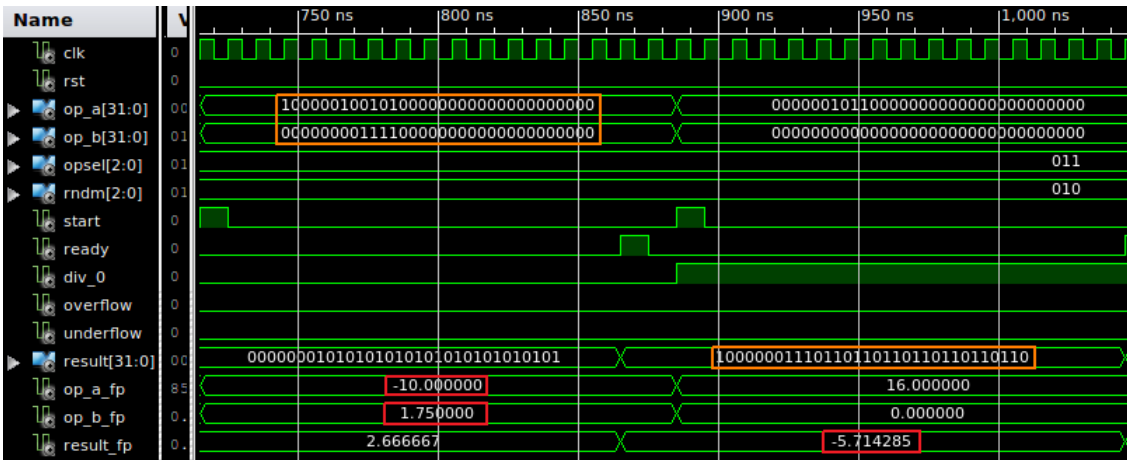
Division example	**Dividend**		**Divisor**		**Result**	
	Fraction	**Exponent**	**Fraction**	**Exponent**	**Fraction**	**Exponent**
a	0.25	0	0.375	0	0.666667	0
	0.25		0.375		0.666667	
b	0.5	10	0.75	8	0.66667	2
	512		192		2.666667	
c	(-0.625)	4	0.875	1	(-0.714286)	3
	(-10)		1.75		(-5.714233)	
d	0.5	5	0	0	Division by zero	
	16		0			
e	0.5	127 (MAX)	0.875	(-1)	Overflow	
	→ "infinity"		0.4375			
f	0.75	(-128) (MIN)	0.625	3	Underflow	
	→ '0'		5			



a) $0.25\times 2^{0}\div 0.375\times 2^{0}=0.666667\times 2^{0}$

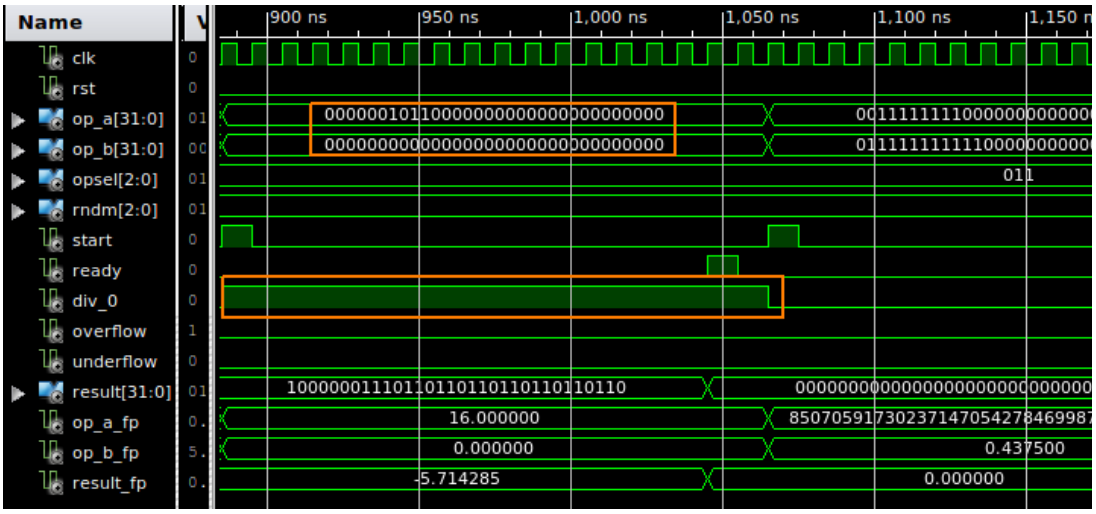


b) $0.5\times 2^{10}\div 0.75\times 2^{8}=0.666667\times 2^{2}$

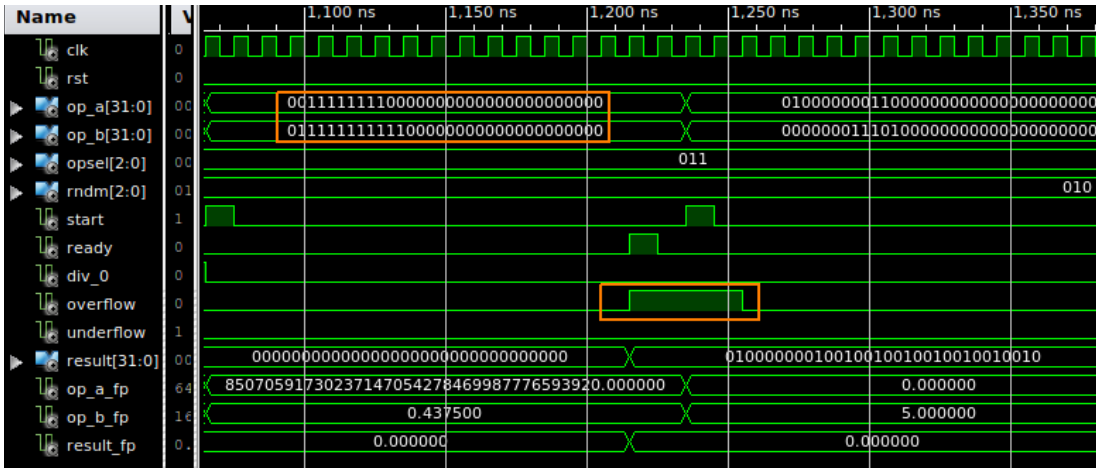


c) $(-0.625)\times 2^{4}\div 0.875\times 2^{1}=(-0.714286)\times 2^{3}$

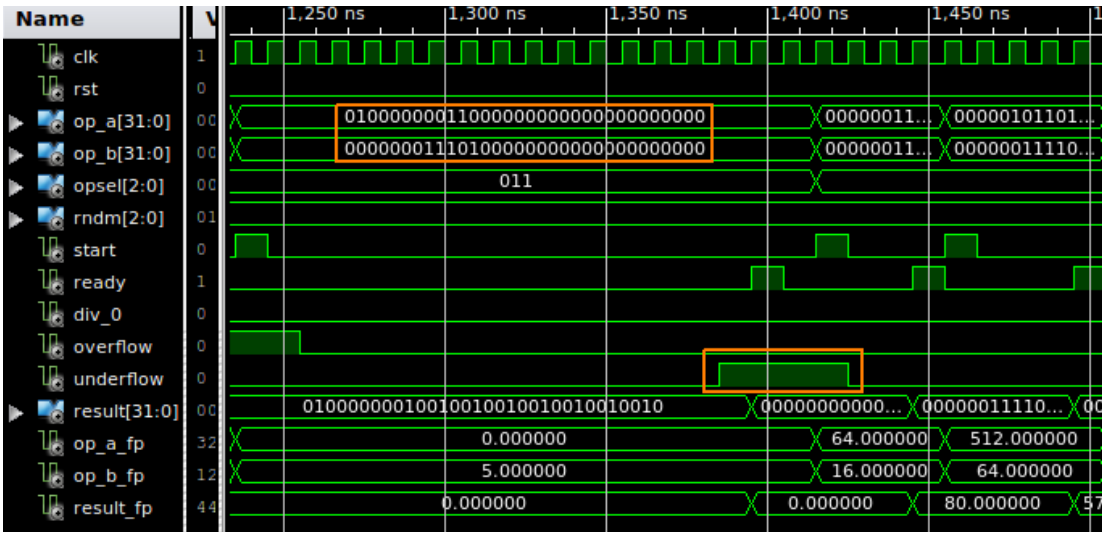
Figure 32. Division examples a - c



d) $0.5\times 2^{5}\div 0 = \textit{division by zero}$



e) $0.5\times 2^{127}\div 0.875\times 2^{-1} = \textit{overflow}$



f) $0.75\times 2^{-128}\div 0.625\times 2^{3} = \textit{underflow}$

Figure 33. Division examples d - f

The first three FP division examples *a* - *c*, which are shown in figure 32, are regular division operations. These examples have been tested graphically, due to the complexity of the exact result's prediction. The division unit confidently divides FP numbers from the half FP precision to the extended FP precision and even more.

The three last FP division examples *d* - *f*, which are shown in figure 33, illustrate the cases when different exceptions are generated. The overflow and underflow exceptions are signalled with delay. This is due to the fact that the overflow and underflow exceptions are generated in the post-normalization unit after the division intermediate result is ready. By contrast, the division by zero exception is indicated by the divider itself and that is why it is signalled by the FPU immediately.

Table 21. FP division performance

N	**FP format, bits**	**Fraction, bits**	**Exponent, bits**	**Division clock cycles**
1	16	10	5	9
2	24	16	7	14
3	32	23	8	15
4	40	30	9	19
5	48	36	11	23
6	56	44	11	27
7	64	52	11	31
5	80	64	15	37
6	128	112	15	61

In this chapter, all the FP division examples, which have been simulated and illustrated, are in the single precision FP format. However, different FP formats require various number of cycles, which are required in order to perform the FP division. This information is listed in table 21. In order to verify this, the various FP formats have been synthesized and the number of the required clock cycles have been measured for each case. The number of the clock cycles, which are required, has been measured from the rising edge of the *start* signal to the rising edge of the output signal *ready*.

4.1.5 Post-normalization unit testing

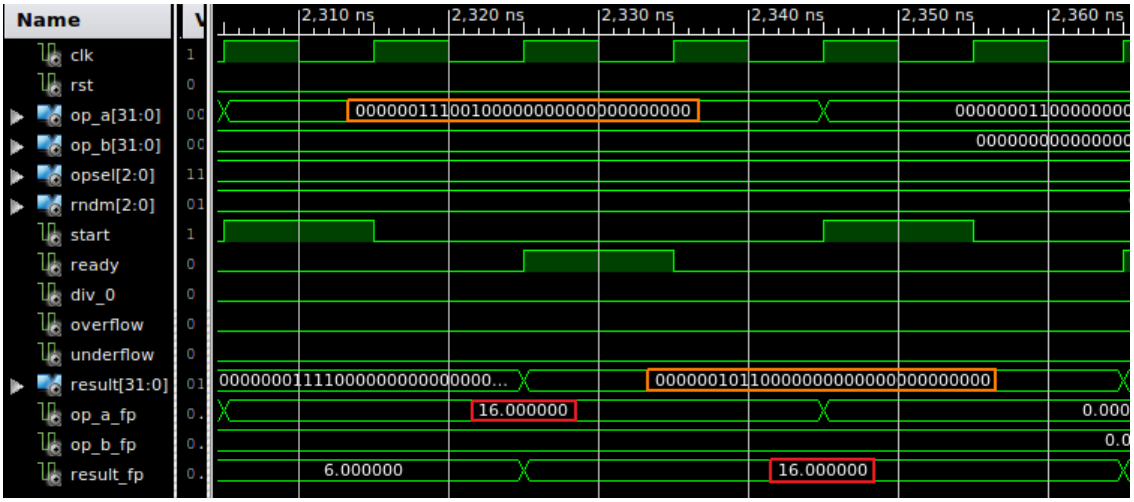
The post-normalization unit can be tested with an intermediate result from any computational unit or from the FPU inputs. Due to the fact that the previous simulation examples have demonstrated the proper operation of the post-normalization unit with only the FPU computational blocks, the input operands' normalization and the rounding modes have been simulated in this chapter. The examples *a* - *c*, which are listed in table 22, are shown in figure 34.

Table 22. FP operands for the post-normalizer testing

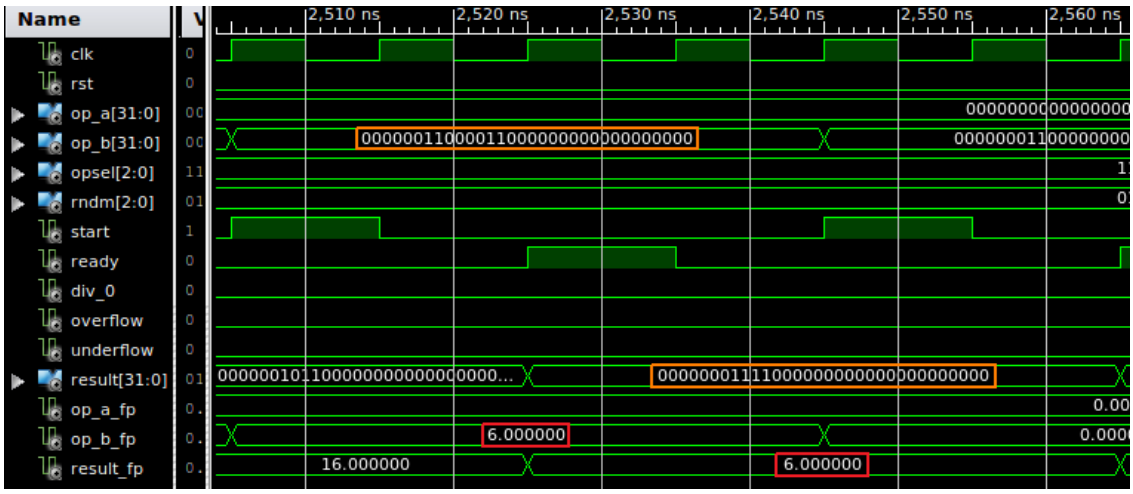
Normalization example	**Operand A**		**Operand B**		**Result**	
	Fraction	**Bin. Exp.**	**Fraction**	**Bin. Exp.**	**Fraction**	**Bin. Exp.**
a	0.125	7	X	X	0.5	5
	16		X		16	
b	X	X	0.09375	6	0.75	3
	X		6		6	
c	the smallest fraction	-128	X	X	Underflow	
	the smallest number		X			

All the normalization examples, except the third example, show the same real values for both the input and output. However, the binary values are different and the results are normalized. The third example simulates the underflow situation for the FPU operand A. This situation takes place when the final exponential value must be less than a value, which can be stored by a chosen FP format.

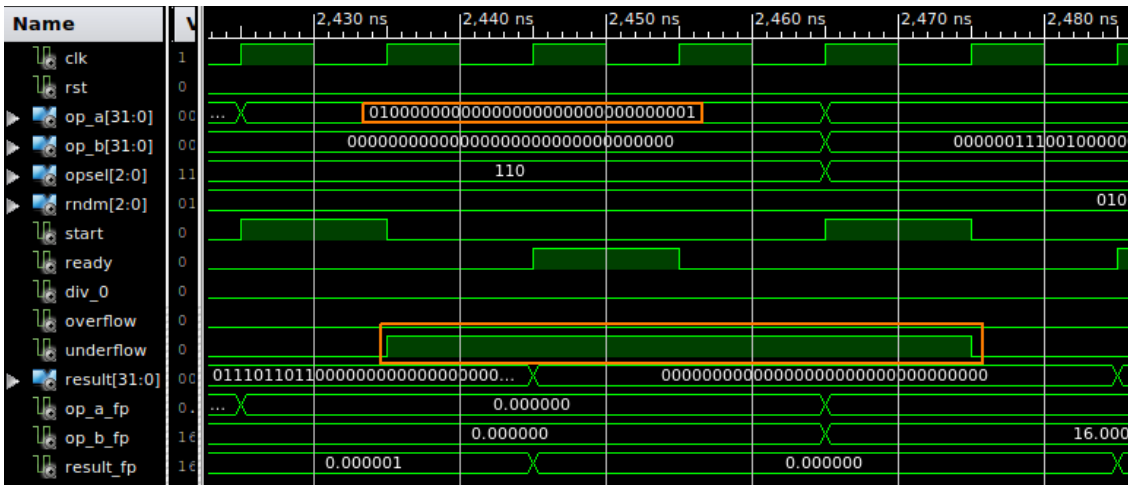
In order to test the rounding modes in the post-normalization unit, the addition and subtraction operations have been selected. This involves a using of two least significant extra bits in the result's fraction. Although the number of simulations in the testbench for the rounding modes testing is about 40, many of them are interchangeable.



a) $A = 0.125 \times 2^{7} \rightarrow A = 0.5 \times 2^{5}$



b) $A = 0.09375 \times 2^{6} \rightarrow A = 0.75 \times 2^{3}$



c) $A = 0.00..01 \times 2^{-128} \rightarrow underflow$

Figure 34. Normalization examples a - c

For performing this experiment, an operand's A value were kept fixed, but the operand's B two least significant bits varied among values "00", "01", "10" and "11". The simulation examples, which have been listed in this section, show the internal FPU signals and their binary values. However, due to the simulation tool limitations, the FPU results, which are shown as real values, are rounded by the simulator roughly. Owing to this, the binary numbers have been considered as a measure of rounding correctness. Table 23 shows different rounding modes on the real decimal numbers. These examples demonstrate the interchangeability of the simulation results, which have been obtained. The signal *fres[27..0]*, which is shown in each figure is the fractional intermediate result before its rounding. For the rounding simulation simplification, this result have been chosen as always normalized. The rounding procedure is always performed as the last FPU operation step, before the FPU result is being outputted.

Table 23. Rounding mode examples (decimal)

Rounding mode / Values	**opsel[2..0]**	**+10.5**	**+11.5**	**-10.5**	**-11.5**
TowardsZero (a)	000	+10	+11	-10	-11
to nearest/TiedToEven (b)	001	+10	+12	-10	-12
to nearest/TiedAwayFromZero (c)	010	+11	+12	-11	-12
TowardPlusInfinity (d)	011	+11	+12	-10	-11
TowardMinusInfinity (e)	100	+10	+11	-11	-12

The simulated examples are different from the examples, which are shown in the table 23. This is due to simplification of the numbers' observation. The simulated numbers have much greater precision.

The data, which is shown in table 24, demonstrates how the binary least significant bits are rounded before the fraction's truncation. Figure 35 shows the three first rounding examples *a* - *c*. According to this figure, for the *TowardsZero* rounding mode (example *a*) the two least significant bits are simply cut off, due to the positive fraction and the selected rounding mode. The next example *b*, which have been shown in the same figure, relates to the rounding *to nearest/TiedToEven* rounding mode. The rounding mode *to the nearest/TiedAwayFromZero* has been shown in this figure.

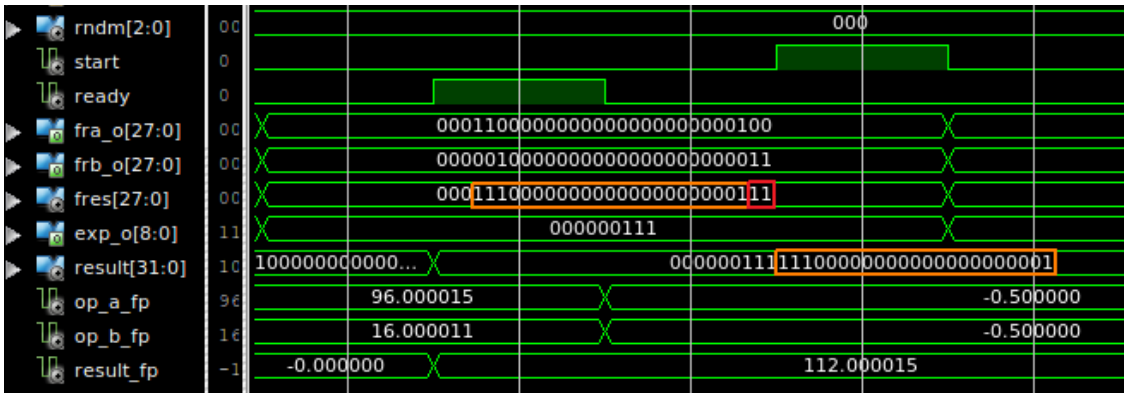
Finally, the rounding modes *TowardPlusInfinity* and *TowardMinusInfinity* have been illustrated in figure 36.

Table 24. Rounding mode examples (binary)

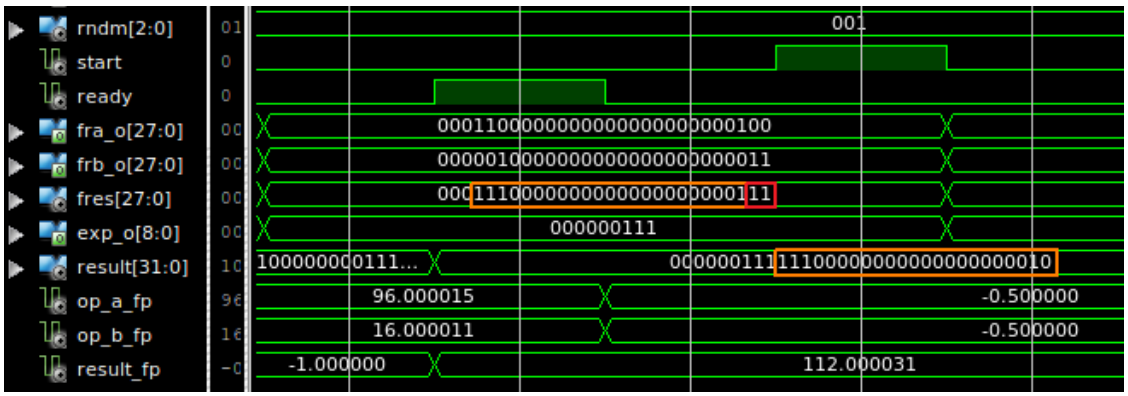
Rounding mode / Values	**opsel [2..0]**	**..00.00**	**+..00.01**	**+..00.10**	**+..00.11**	**-..00.01**	**-..00.10**	**-..00.11**
TowardsZero (a)	000	..00	+..00	+..00	+..00	-..00	-..00	-..00
to nearest/ TiedToEven (b)	001	..00	+..00	+..10	+..10	-..00	-..10	-..10
to nearest/ TiedAway FromZero (c)	010	..00	+..00	+..01	+..01	-..00	-..01	-..01
Toward PlusInfinity (d)	011	..00	+..01	+..01	+..01	-..00	-..00	-..00
Toward MinusInfinity (e)	100	..00	+..00	+..00	+..00	-..01	-..01	-..01

The final rounding result also relies on the fraction's sign and that is why different rounding operations seem to produce the same final fractional result. This situation has been shown in tables 23 and 24 above.

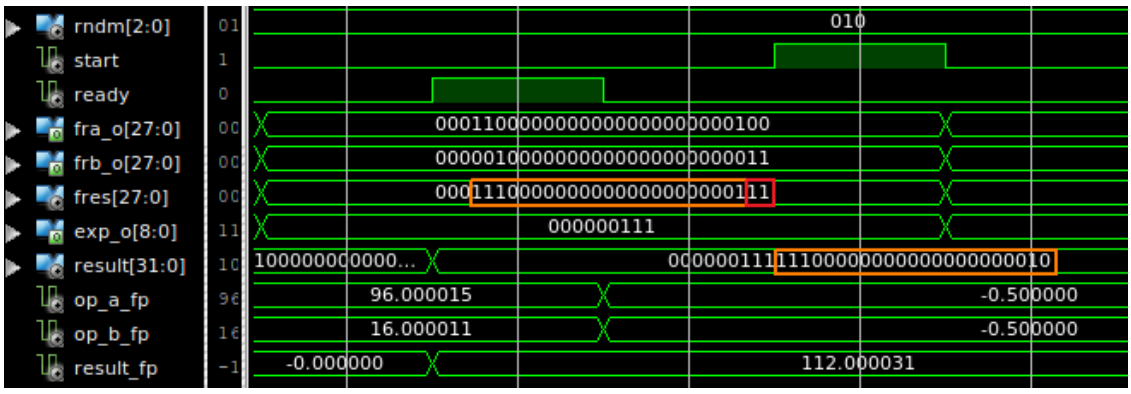
The rounding modes affect the final result's accuracy to some extent in cases, when the chosen FPU precision is relatively small. This is due to the fact that the most significant bits of the fraction have the highest value for encoding of the final fractional result in a binary form. For cases when the FPU precision is larger, the two least significant bits of the fractional result can be ignored, because they do not contribute significantly to a resulting floating point value.



a) TowardsZero rounding mode



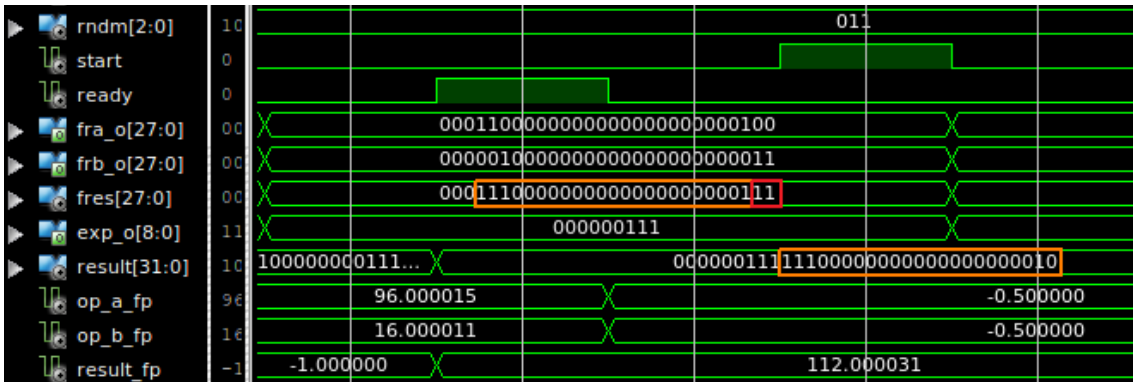
b) To nearest/TiedToEven rounding mode



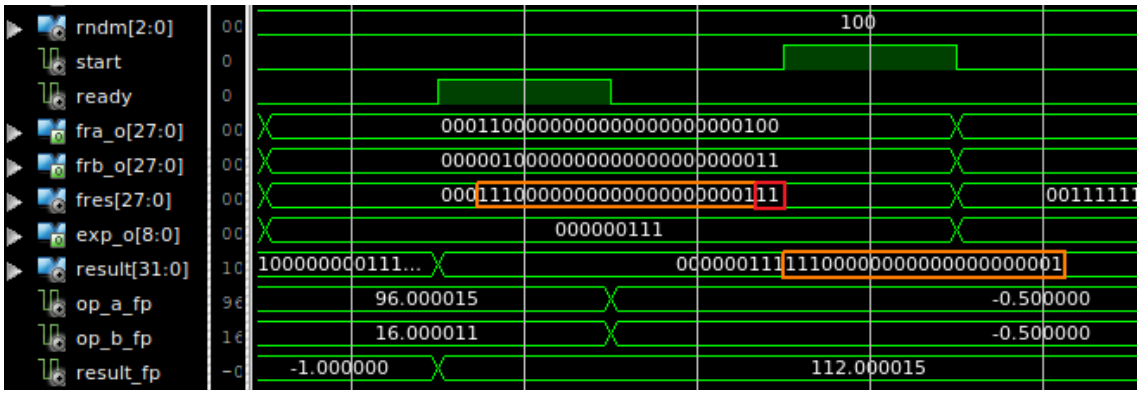
c) To nearest/TiedAwayFromZero rounding mode

Figure 35. Rounding modes a – c

Figures 35 and 36 show all the rounding examples. Due to this, in order to avoid confusion, each rounding mode in tables 23 and 24 has been marked with letters *a* – *e*.



d) TowardPlusInfinity rounding mode



e) TowardMinusInfinity rounding mode

Figure 36. Rounding modes d – e

4.1.6 Generic barrel shifter testing

The barrel shifter has been simulated and tested separately from the FPU design. In order to check the barrel shifter's proper operation, the input signals must be applied to it. The shifted binary word appears at the shifter's output immediately, due to its combinational structure. The two examples with shifting to the left and to the right are shown in figures 37 and 38 respectively. The barrel shifter shifts a binary word to the left, if the input signal *dir_left* has a high level. In order to demonstrate shifting, the input data width has been chosen less than the fractional width in the single precision FP format. The clock is not required for shifting. Clock has been used in simulation figures in order to observe the shifting result at the time axis.

The barrel shifter is the essential hardware unit for the denormalization and post-normalization procedures. Furthermore, the other internal blocks correct simulation results also implicitly indicate that the barrel shifter's operation is valid.

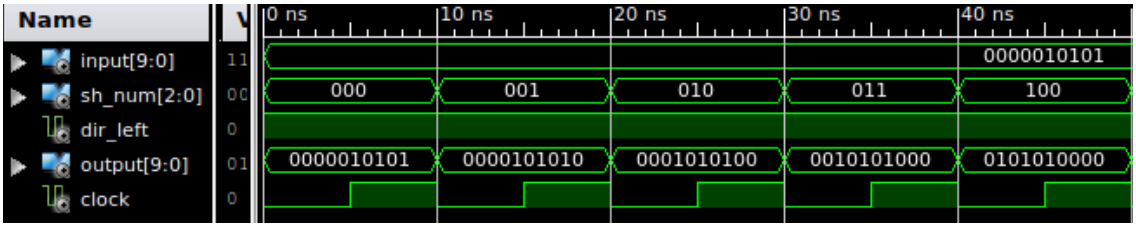


Figure 37. Shifting to the left

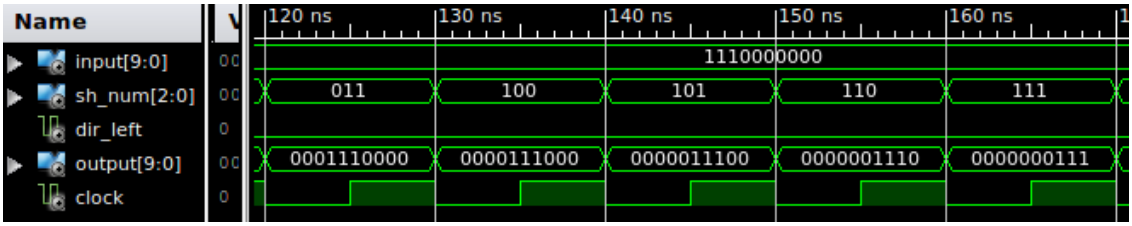
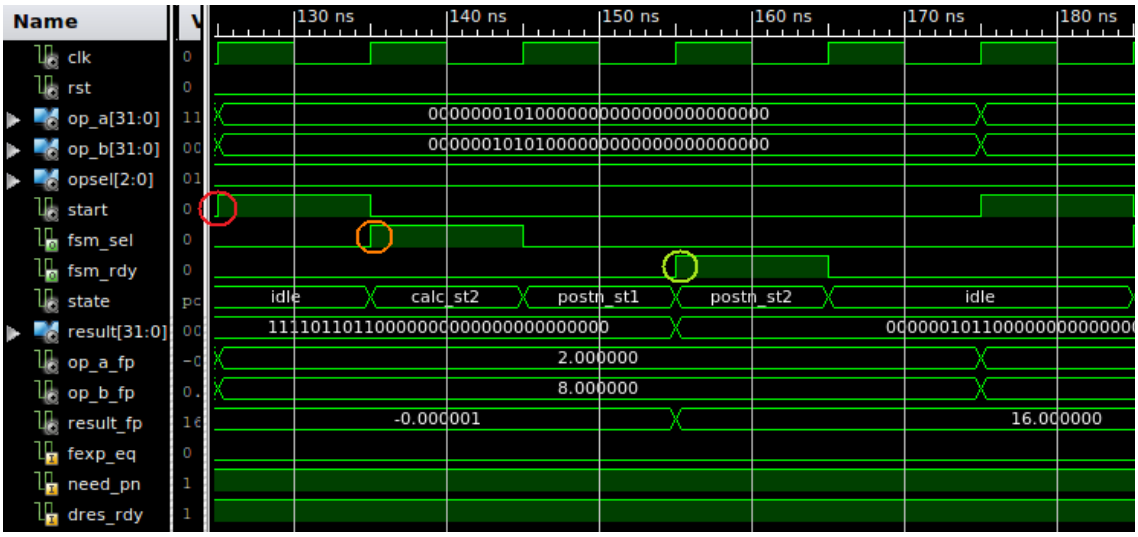


Figure 38. Shifting to the right

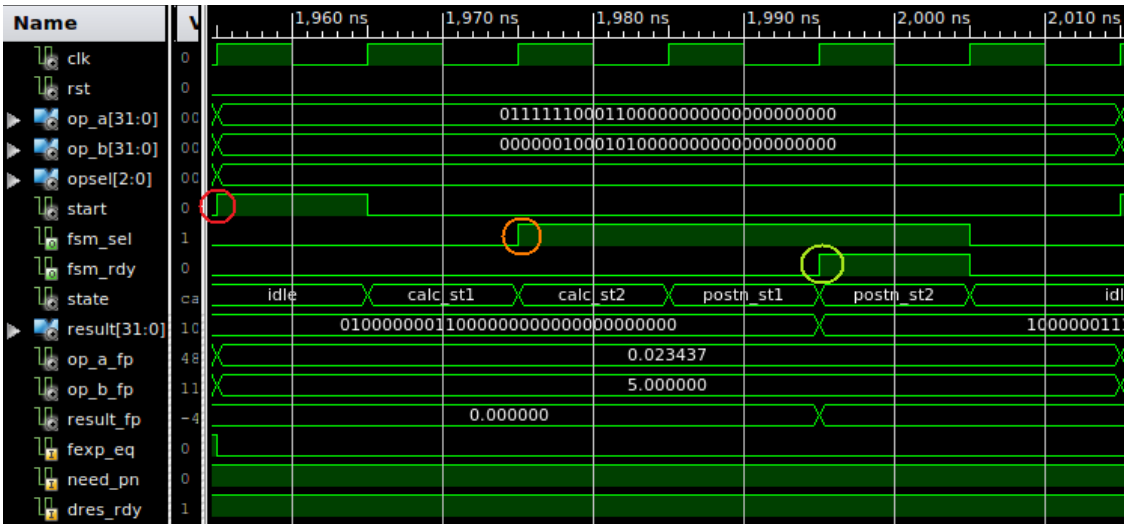
4.1.7 Control FSM testing

The control FSM can be tested by observing its internal signals, due to the fact that the majority of them are not connected with the FPU external inputs and outputs. Only two output FSM signals are called the *fsm_sel* and *fsm_rdy* respectively. These signals are changed by the FSM in the same sequence, but the time when one or the other is changed varies upon the selected operation and an intermediate result's value. A rising edge at the input signal *start* is circled with the colour red. After this signal has been applied, the FSM changes its state according to the diagrams, which have been described in part 3.2.10 of this thesis. The three FSM operation examples for the multiplication, subtraction and normalization are shown in figure 39.

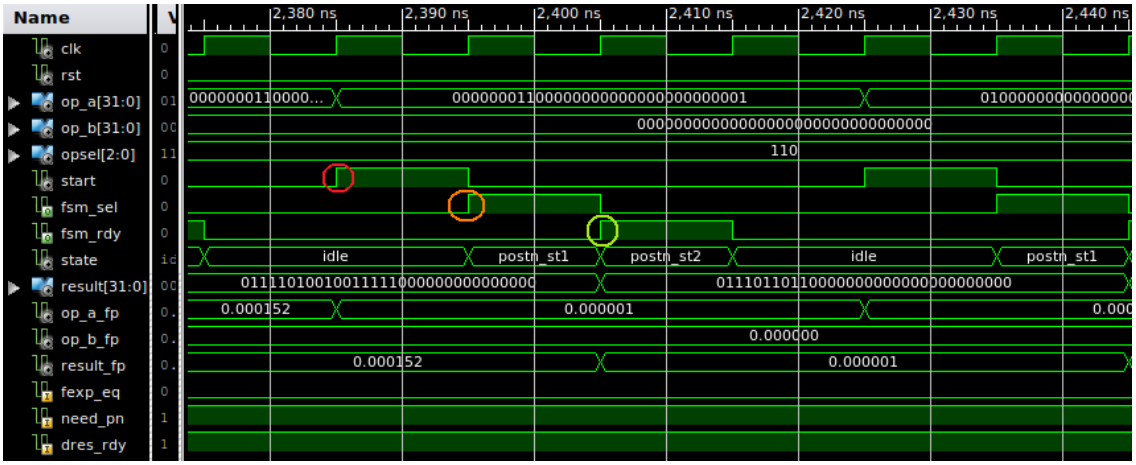
The control signals are switched by the FSM according to the values of signals *opsel[2..0]*, *fexp_eq*, *need_pn*, *dres_rdy* at the different FSM states. These values affect the FSM timing and, consequently, the number of clock cycles which are required in order to perform the specific operation. The detailed FSM algorithm has been described in the previous part of the thesis. The simulation diagrams in this part mainly demonstrate that the FSM is switched between its states according to the required FPU algorithm.



a) Multiplication



b) Subtraction



c) Normalization

Figure 39. FSM simulation

4.2 FPU Testing with Different FP Formats

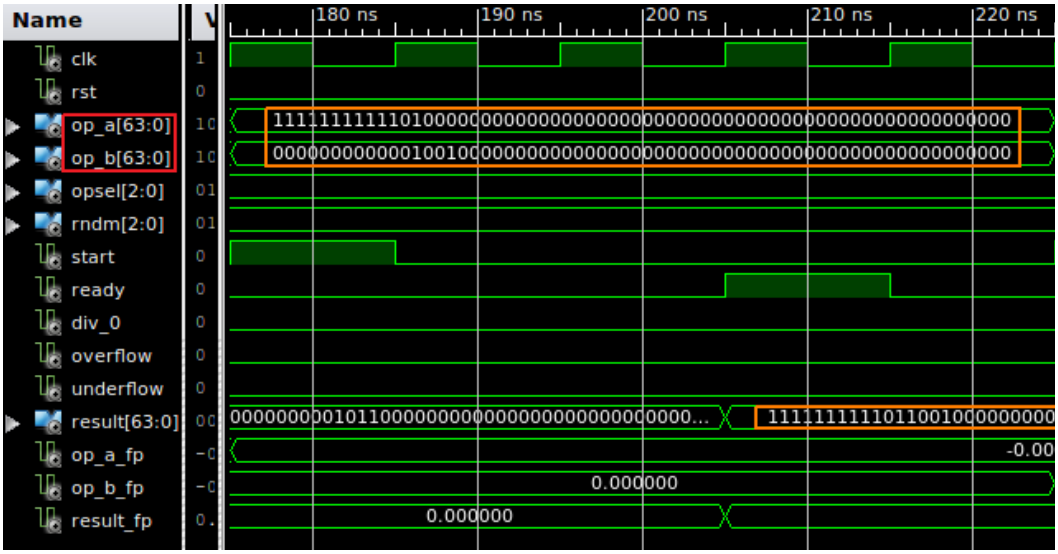
In order to prove the FPU parametrizability, it must be synthesised with different FP formats. In the previous part 4.1, all the FPU calculation and normalization examples have been simulated with a single precision FP format. This has been done in order to simplify and standardize the simulation. This part of the thesis describe the simulated calculation examples, which deal with different FP formats. However, this imposes certain limitation in cases when the FP exponential part exceeds the length of 32 bits. This problem is due to the maximum and minimum VHDL *real* signal possible range. Although, the calculation result can be observed and asserted in a binary form, the exceeding of the given exponential part's width with more than 32 bits will make it impossible to check the result in a decimal form. However, this issue does not affect the FPU operation itself. Table 25 contains the examples, which have been used for testing the FPU parametrizability.

Table 25. Examples of the operations with different FP formats

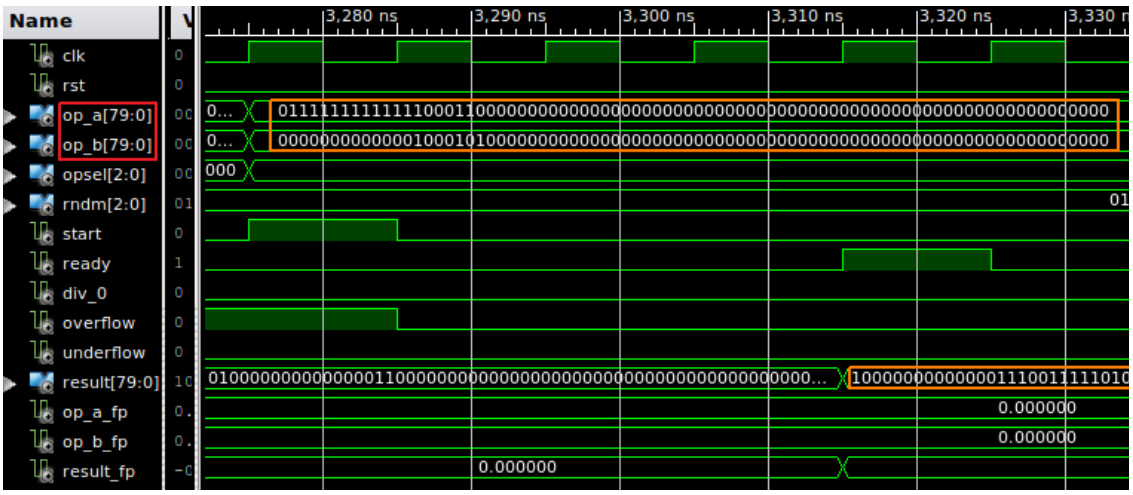
Performed operation	**Fractional width**	**Exponential width**	**Operand A**	**Operand B**	**Result**
Multiplication	52	11	0.5*2^2	0.25*2^5	0.5*2^5
Subtraction	64	15	0.375*2^(-4)	0.3125*2^4	-0.622...*2^3
Multiplication	37	20	-0.5*2^(-2)	0.5625*2^0	-0.5625*2^(-3)
Multiplication	10	5	-0.5*2^(-1)	-0.125*2^(-1)	-0.5*2^(-5)

The simulation examples, which are shown in figures 40-41, illustrate that the FPU can be synthesised with different FP formats. The division examples have not been shown. However, the generic testbench can be run with different fractional and exponential widths in the FPU. In this section the simulation results have been validated. The colour red in the simulation pictures shows the number of bits, which are used in a FP format. The colour orange highlights the input operands and the final

results which have been obtained after the normalization. The correctness of the simulation results has been checked by the ISE simulator.

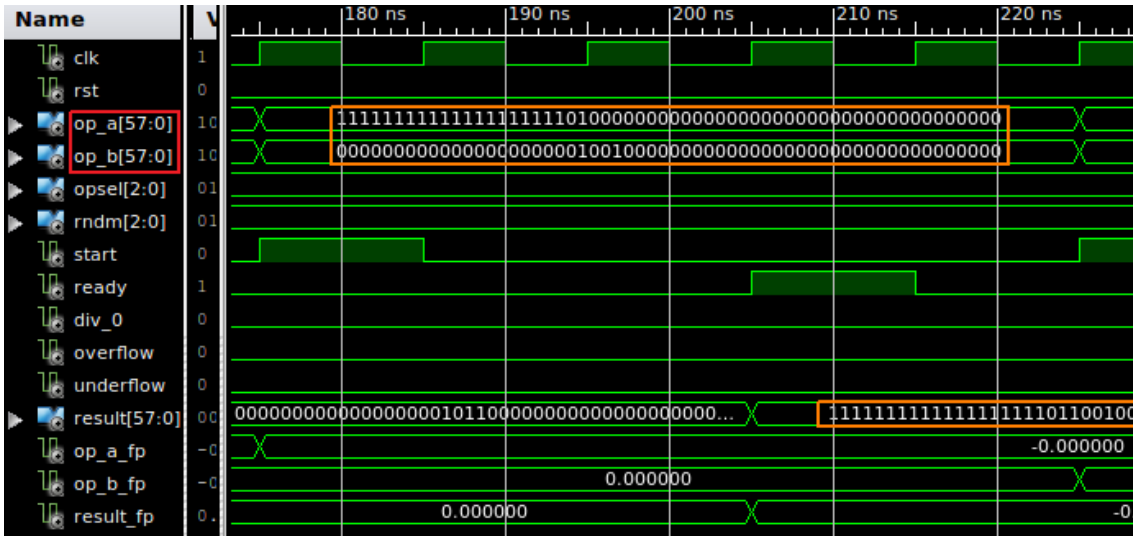


a) Double precision FP addition example

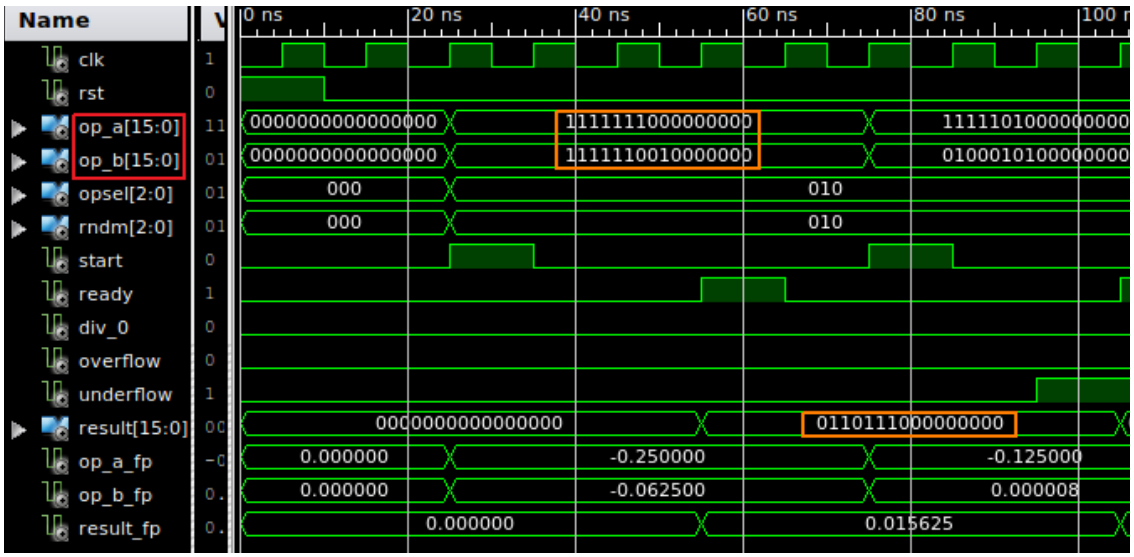


b) Extended precision FP subtraction example

Figure 40. Different FP addition and subtraction examples



a) Custom FP multiplication examples



b) Half precision FP multiplication example

Figure 41. Different FP multiplication examples

Conclusions

In this part of the thesis the simulation and testing of the generic FPU has been performed. The internal FPU blocks have been simulated in order to validate their proper operation. The simulation pictures, which have been presented, show the relation between the expected calculation results, and the results which have been obtained after the FPU simulations. All the FPU simulated results match with the expected results. The exception signals have been checked during the simulations. Due to their simulation results, the FPU exceptions are signalled properly.

The rounding modes, which are performed after the normalization, have been checked in this part. The final rounded fractional values correspond to the five predefined IEEE-754 rounding standards, which have been implemented in the generic FPU design. The FPU parametrizability feature has been simulated and tested in part 4.2 of this thesis. The FPU can be synthesized and simulated with various numbers of bits, which are dedicated for the fractional and exponential parts of the FP format. Although the VHDL *real* data type is restricted to the length of 32 bits and, due to this, certain limitations occur for the decimal numbers' simulation, the simulation results for the binary numbers have been checked and they are valid.

5 Summary

In this master's thesis the generic FPU has been designed, simulated and tested. The FPU is able to be synthesized with different, even non-standard, FP formats. The operations supported by the FPU are: addition, subtraction, multiplication, division and the normalization. The FPU includes the internal denormalization and post-normalization units, which simplify the input FPU operands handling. Due to having these internal blocks, the FPU supports the normalization of the input operands.

Part 1 of the thesis introduced the problems of the existing FPU solutions, which are not parameterisable. The second part provides an overview of the basic FP standard formats as well as highlighting the main IEEE-754 standard [6] features. It also shows how the operations between the FP numbers are performed. The FPU design flow, its major internal signals and algorithms have been described in part 3 of this thesis. At the end of this part the FPU synthesis results with different FP formats have been presented. Also the timing constraints and possible ways of improving the FPU have been described in the end of part 3. Part 4 describes the simulated calculation and normalization examples and the performance results of the designed FPU. The simulation diagrams have been illustrated in order to illustrate and test the FPU operation flow step by step. The FPU VHDL source code is not listed in appendixes, but the actual repository link is provided in references [15].

Although various draw-backs occurred during the FPU implementation and simulation, the technical task of the thesis has been fulfilled. This has been achieved due to a deep analysis of problems described in the introductory part, applying previous experience with VHDL and analysing the existing FPU solutions, which are not satisfactory in terms of their parametrizability. The design VHDL code has been stored into the Bitbucket repository. This FPU design might require further improvements. It can be pipelined in order to achieve better throughput and the new computational blocks for other mathematical functions also can be added.

References

[1] Press, William Henry; Teukolsky, Saul A.; Vetterling, William T.; Flannery, Brian P. (2007). Numerical Recipes - The Art of Scientific Computing (3 ed.). Cambridge University Press. ISBN 978-0-521-88407-5

[2] Kahan, William Morton (2001), University of California at Berkeley,. Why do we need a floating-point arithmetic standard?

[3] Kahan, William Morton (2004), University of California at Berkeley, On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic

[4] Tagliavini, Giuseppe & Mach, Stefan & Rossi, Davide & Marongiu, Andrea & Benin, Luca. (2018). A transprecision floating-point platform for ultra-low power computing. 1051-1056. 10.23919/DATE.2018.8342167. , Published in 2018. Design, Automation & Test in Europe Conference & Exhibition

[5] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu and L. Benini, "A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing," 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, 2018, pp. 1-5. doi: 10.1109/ISCAS.2018.8351816

[6] IEEE Computer Society (August 29, 2008), IEEE Standard for Floating-Point Arithmetic, IEEE, doi:10.1109/IEEESTD.2008.4610935, ISBN 978-0-7381-5752-8, IEEE Std 754-2008

[7] Pong P. Chu. RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability, Wiley-IEEE Press, 2006, ISBN:0471720925 2006

[8] Aharon Yadin. Computer Systems Architecture (Chapman & Hall/CRC Textbooks in Computing) Aug 19, 2016

[9] Meeus, W., Van Beeck, K., Goedemé, T. et al. Design Automation for Embedded Systems (2012) 16: 31. https://doi.org/10.1007/s10617-012-9096-8

[10] Grant Martin, Gary Smith, High-Level Synthesis: Past, Present, and Future, IEEE Design & Test of Computers, 2009

[11] Philippe Coussy; Daniel D. Gajski; Michael Meredith; Andres Takach. An Introduction to High-Level Synthesis, IEEE Design & Test, IEEE Computer Society Press Los Alamitos, CA, USA, July 2009, ISSN:0740-7475

[12] Dirk Jansen et al. (editors). The electronic design automation handbook, Springer, 2014

[13] Pedroni, Volnei A., Circuit design and simulation with VHDL, 2nd ed., MIT Press, 2010.

[14] Kenneth L. Short. VHDL for Engineers 1st Edition, Pearson, 2009

[15] Israel Koren, Computer arithmetic algorithms, 2nd ed., MIT Press, 2002.

[16] Chashurin, T. (2019). Generic Synthesizable Floating-Point Unit (Master's dissertation, Tallinn University of Technology, Tallinn, Estonia). Retrieved from https://bitbucket.org/timofey-chashurin/fpu-generic/src/master/