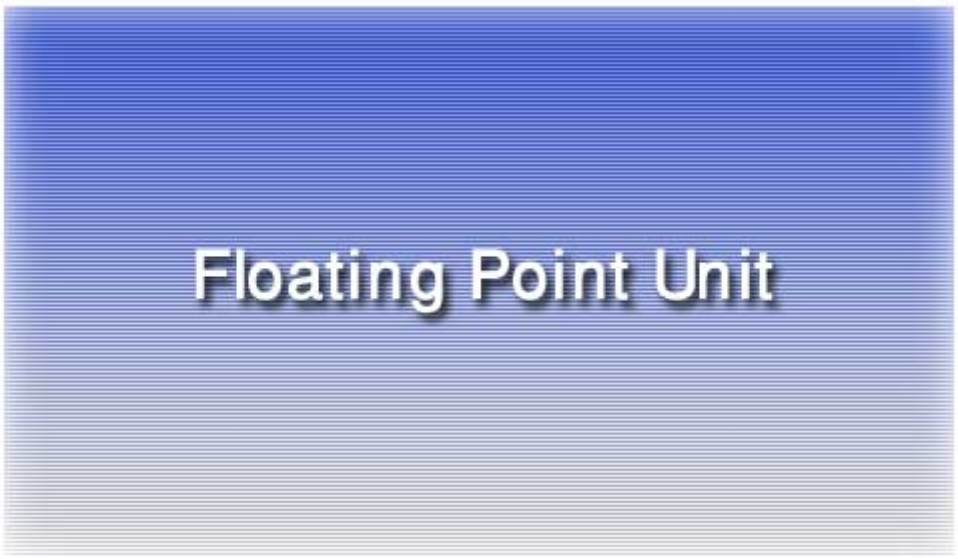


Jidan Al-Eryani

jidan@gmx.net

Contents

1. Introduction

The floating point unit (FPU) implemented during this project, is a 32-bit processing unit which allows arithmetic operations on floating point numbers. The FPU complies fully with the IEEE 754 Standard [1].
The FPU supports the following arithmetic operations:

1. Add
2. Subtract
3. Multiply
4. Divide
5. Square Root

For each operation the following rounding modes are supported:

1. Round to nearest even
2. Round to zero
3. Round up
4. Round down

The FPU was written in VHDL with top priority to be able to run at approximately 100-MHz and at the same time as small as possible. Meeting both goals at the same time was very difficult and tradeoffs were made.
In the following sections I will explain the theory behind the FPU core and describe its implementation on hardware.

2. Floating point numbers

The floating-point representation is one way to represent real numbers. A floating-point number *n* is represented with an exponent *e* and a mantissa *m*, so that:

$$n = b^e \times m, \ldots$$where b is the base number (also called radix)

So for example, if we choose the number n=17 and the base b=10, the floating-point representation of 17 would be:

$$17 = 10^1 \text{ x } 1.7$$

Another way to represent real numbers is to use fixed-point number representation. A fixed-point number with 4 digits after the decimal point could be used to represent numbers such as: 1.0001, 12.1019, 34.0000, etc. Both representations are used depending on the situation. For the implementation on hardware, the base-2 exponents are used, since digital systems work with binary numbers.
Using base-2 arithmetic brings problems with it, so for example fractional powers of 10 like 0.1 or 0.01 cannot exactly be represented with the floating-point format, while with fixed-point format, the decimal point can be thought away (provided the value is within the range) giving an exact representation. Fixed-point arithmetic, which is faster than floating-point arithmetic, can then be used. This is one of the reasons why fixed-point representations are used for financial and commercial applications.
The floating-point format can represent a wide range of scale without losing precision, while the fixed-point format has a fixed window of representation. So for example in a 32-bit floating-point representation, numbers from $3.4 \text{ x } 10^{38}$ to $1.4 \text{ x } 10^{-45}$ can be represented with ease, which is one of the reasons why floating-point representation is the most common solution.
Floating-point representations also include special values like infinity, Not-a-Number (e.g. result of square root of a negative number).

3. IEEE Standard 754 for Binary Floating-Point Arithmetic

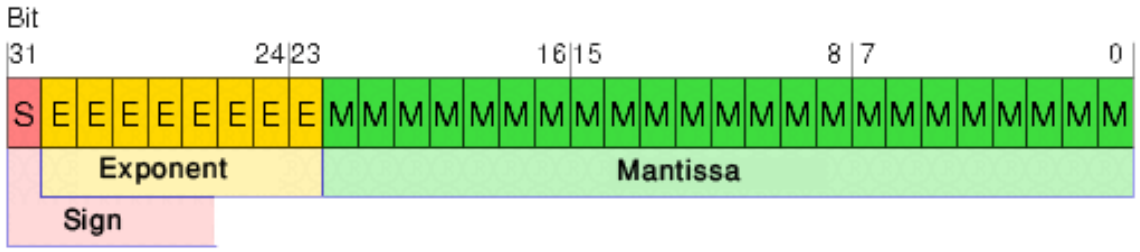
3.1 Formats

The IEEE (Institute of Electrical and Electronics Engineers) has produced a Standard to define floating-point representation and arithmetic. Although there are other representations, it is the most common representation used for floating point numbers.
The standard brought out by the IEEE come to be known as **IEEE 754**.
The standard specifies [1]:

1) Basic and extended floating-point number formats
2) Add, subtract, multiply, divide, square root, remainder, and compare operations
3) Conversions between integer and floating-point formats
4) Conversions between different floating-point formats
5) Conversions between basic format floating-point numbers and decimal strings
6) Floating-point exceptions and their handling, including non numbers (NaNs)

When it comes to their precision and width in bits, the standard defines two groups: **basic-** and **extended format**. The extended format is implementation dependent and doesn't concern this project.
The basic format is further divided into **single-precision** format with 32-bits wide, and **double-precision** format with 64-bits wide. The three basic components are the sign, exponent, and mantissa. The storage layout for single-precision is show below:

Single precision



The most significant bit starts from the left.

The double-precision doesn't concern this project and therefore will not be discussed further.

The number represented by the single-precision format is:

value = $(-1)^s 2^e \times 1.f$ (normalized) when E > 0 else
= $(-1)^s 2^{-126} \times 0.f$ (denormalized)

where

f = $(b_{23}^{-1}+b_{22}^{-2}+ b_i^n +\ldots+b_0^{-23})$ where b_i^n =1 or 0
s = sign (0 is positive; 1 is negative)
E =biased exponent; E_{max}=255 , E_{min}=0. E=255 and E=0 are used to represent special values.
e =unbiased exponent; e = E – 127(bias)

A bias of 127 is added to the actual exponent to make negative exponents possible without using a sign bit. So for example if the value 100 is stored in the exponent placeholder, the exponent is actually -27 (100 – 127). Not the whole range of E is used to represent numbers. As you may have seen from the above formula, the leading fraction bit before the decimal point is actually implicit (not given) and can be 1 or 0 depending on the exponent and therefore saving one bit. Below is a table with the corresponding values for a given representation to help better understand what was explained above:

Sign(s)	Exponent(e)	Fraction	Value
0	00000000	00000000000000000000000	+0 (positive zero)
1	00000000	00000000000000000000000	-0 (negative zero)
1	00000000	10000000000000000000000	$-2^{0-127}x0.(2^{-1})=$ $-2^{0-127}x$ 0.5
0	00000000	00000000000000000000001	$+2^{0-127}x0.(2^{-23})$ (smallest value)
0	00000001	01000000000000000000000	$+2^{1-127}x1.(2^{-2})=$ $+2^{1-127}x1.25$
0	10000001	00000000000000000000000	$+2^{129-127}x1.0=$ 4
0	11111111	00000000000000000000000	+ infinity
1	11111111	00000000000000000000000	- infinity
0	11111111	10000000000000000000000	Not a Number(NaN)
1	11111111	10000100010000000001100	Not a Number(NaN)

3.2 Exceptions

The IEEE standard defines five types of exceptions that should be signaled through a one bit status flag when encountered.

3.2.1 Invalid Operation

Some arithmetic operations are invalid, such as a division by zero or square root of a negative number. The result of an invalid operation shall be a NaN. There are two types of NaN, quiet NaN (QNaN) and signaling NaN (SNaN). They have the following format, where s is the sign bit:

```
QNaN      = s 11111111  10000000000000000000000
SNaN      = s 11111111  00000000000000000000001
```

The result of every invalid operation shall be a QNaN string with a QNaN or SNaN exception. The SNaN string can never be the result of any operation, only the SNaN exception can be signaled and this happens whenever one of the input operand is a SNaN string otherwise the QNaN exception will be signaled. The SNaN exception can for example be used to signal operations with uninitialized operands, if we set the uninitialized operands to SNaN. However this is not the subject of this standard.

The following are some arithmetic operations which are invalid operations and that give as a result a QNaN string and that signal a QNaN exception:

1) Any operation on a NaN
2) Addition or subtraction: $\infty + (-\infty)$
3) Multiplication: $\pm 0 \times \pm \infty$
4) Division: $\pm 0 / \pm 0$ or $\pm \infty / \pm \infty$
5) Square root: if the operand is less than zero

3.2.2 Division by Zero

The division of any number by zero other than zero itself gives infinity as a result. The addition or multiplication of two numbers may also give infinity as a result. So to differentiate between the two cases, a divide-by-zero exception was implemented.

3.2.3 Inexact

This exception should be signaled whenever the result of an arithmetic operation is not exact due to the restricted exponent and/or precision range.

3.2.4 Underflow

Two events cause the underflow exception to be signaled, tininess and loss of accuracy. Tininess is detected after or before rounding when a result lies between $\pm 2^{E_{min}}$. Loss of accuracy is detected when the result is simply inexact or only when a denormalization loss occurs. The implementer has the choice to choose how these events are detected. They should be the same for all operations. The implemented FPU core signals an underflow exception whenever tininess is detected after rounding and at the same time the result is inexact.

3.2.5 Overflow

The overflow exception is signaled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signaled when one of the operands is infinity, because infinity arithmetic is always exact. Division by zero also doesn't trigger this exception.

3.2.6 Infinity

This exception is signaled whenever the result is infinity without regard to how that occurred. This exception is not defined in the standard and was added to detect faster infinity results.

3.2.7 Zero

This exception is signaled whenever the result is zero without regard to how that occurred. This exception is not defined in the standard and was added to detect faster zero results.

3.3 Rounding Modes

Since the result precision is not infinite, sometimes rounding is necessary. To increase the precision of the result and to enable round-to-nearest-even rounding mode, three bits were added internally and temporally to the actual fraction: *guard*, *round*, and *sticky* bit. While guard and round bits are normal storage holders, the sticky bit is turned '1' when ever a '1' is shifted out of range.
As an example we take a 5-bits binary number: 1.1001. If we left-shift the number four positions, the number will be 0.0001, no rounding is possible and the result will no be accurate. Now, let's say we add the three extra bits. After left-shifting the number four positions, the number will be 0.0001 101 (remember, the last bit is '1' because a '1' was shifted out). If we round it back to 5-bits it will yield: 0.0010, therefore giving a more accurate result.

The standard specifies four rounding modes:

3.3.1 Round to nearest even

This is the standard default rounding. The value is rounded up or down to the nearest infinitely precise result. If the value is exactly halfway between two infinitely precise results, then it should be rounded up to the nearest infinitely precise even.

For example:

Unrounded	Rounded
3.4	3
5.6	6
3.5	4
2.5	2

3.3.2 Round-to-Zero

Basically in this mode the number will not be rounded. The excess bits will simply get truncated, e.g. 3.47 will be truncated to 3.4.

3.3.3 Round-Up

The number will be rounded up towards $+\infty$, e.g. 3.2 will be rounded to 4, while -3.2 to -3.

3.3.4 Round-Down

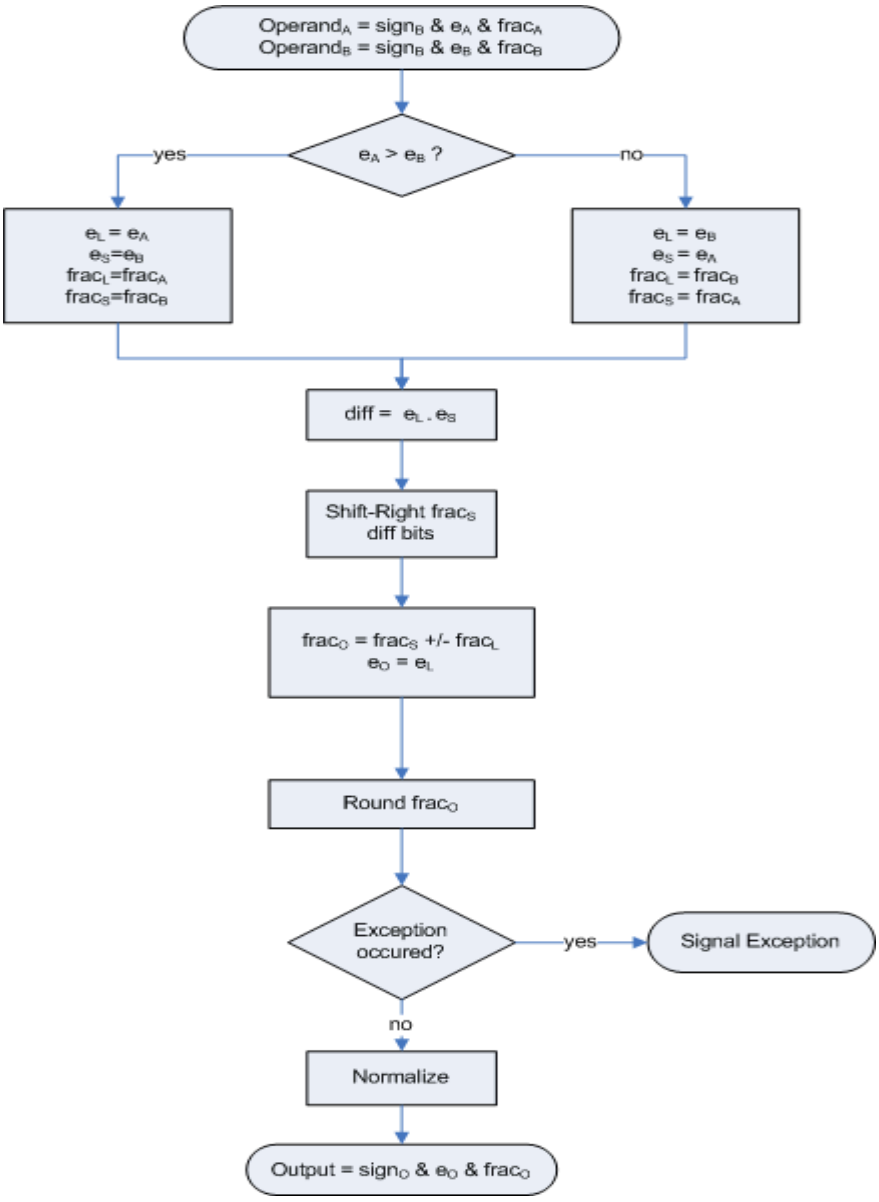
The opposite of round-up, the number will be rounded up towards $-\infty$, e.g. 3,2 will be rounded to 3, while -3,2 to -4.

4. Arithmetic on floating point numbers

In the following sections, the basic algorithms for arithmetic operations will be outlined. For more exact detail please see the VHDL code, the code was commented as much as possible.

4.1 Addition and Subtraction

Addition and Subtraction operations on floating-point numbers are a lot more complex than that on integers. The basic algorithm for adding or subtracting FP numbers is shown in the following flow diagram.



An example is given below to demonstrate the basic steps for adding/subtracting two FP numbers.
Let's say we want to add two 5-digits binary FP numbers:

$$\begin{array}{ll} & 2^4 \times 1.1001 \\ + & 2^2 \times 1.0010 \\ \hline \end{array}$$

Step1: get the number with the larger exponent and subtract it from the smaller exponent.

$e_L = 2^4$, $e_S = 2^2$, so diff = 4 -2 = 2

Step 2: shift the fraction with the smaller exponent diff positions to the right. We can now leave out the exponent since they are both equal. This gives us the following:

$$\begin{array}{ll} & 1.1001 \quad 000 \\ + & 0.0100 \quad 100 \\ \hline \end{array}$$

Step 3: Add both fractions

$$\begin{array}{ll} & 1.1001 \quad 000 \\ + & 0.0100 \quad 100 \\ \hline & 1.1101 \quad 100 \end{array}$$

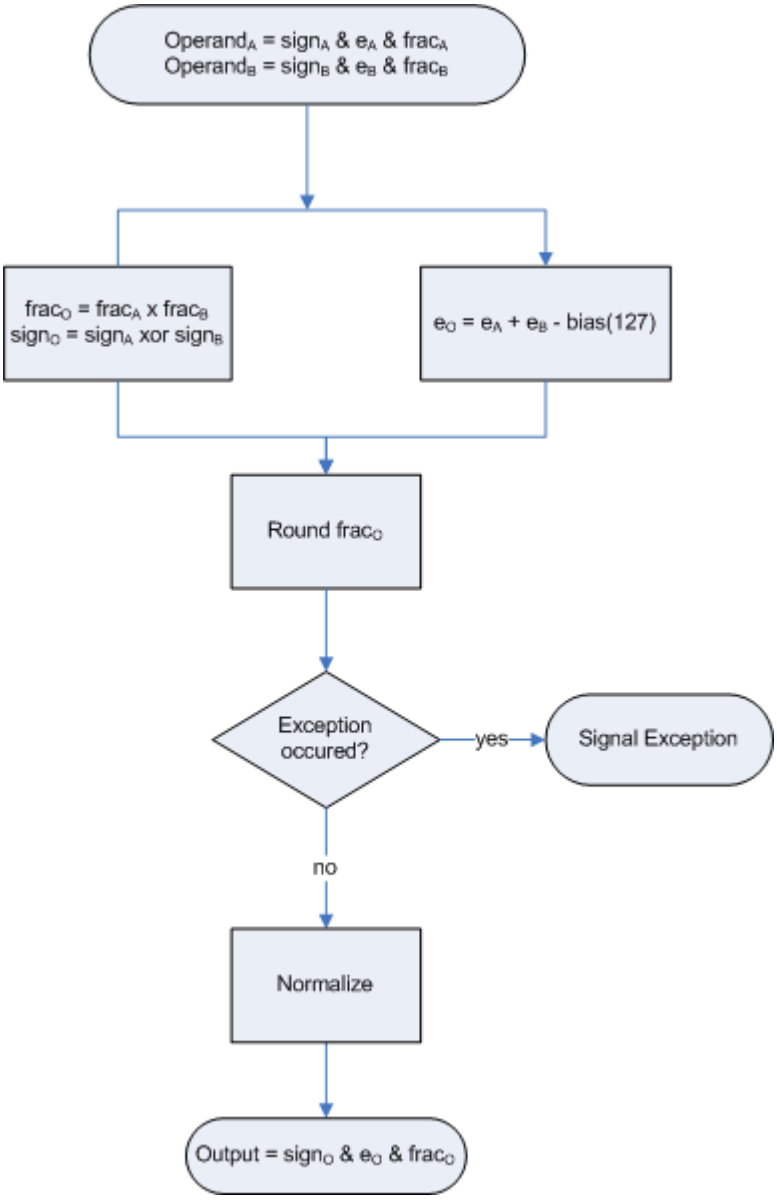
Step 4: Round-to-nearest-even

1.1110

Step 5: Result

$2^4 \times 1.1110$

4.2 Multiplication



The multiplication was done parallel to save clock cycles, at the cost of hardware.
If done serial it would have taken 32 clock cycles (without pre-, post-normalization) instead of the actual 5 clock cycles needed. Disadvantage, the hardware needed for the parallel 32-bit multiplier is approximately 3 times that of serial.
To demonstrate the basic steps, let's say we want to multiply two 5-digits FP numbers:

$$\begin{array}{ll} & 2^{100} \times 1.1001 \\ \times & 2^{110} \times 1.0010 \\ \hline \end{array}$$

Step 1: multiply fractions and calculate the result exponent.

$$\begin{array}{rl} & 1.1001 \\ \times & 1.0010 \\ \hline & 1.11000010 \end{array}$$

so $frac_O = 1.11000010$ and $e_O = 2^{100+110-bias} = 2^{83}$

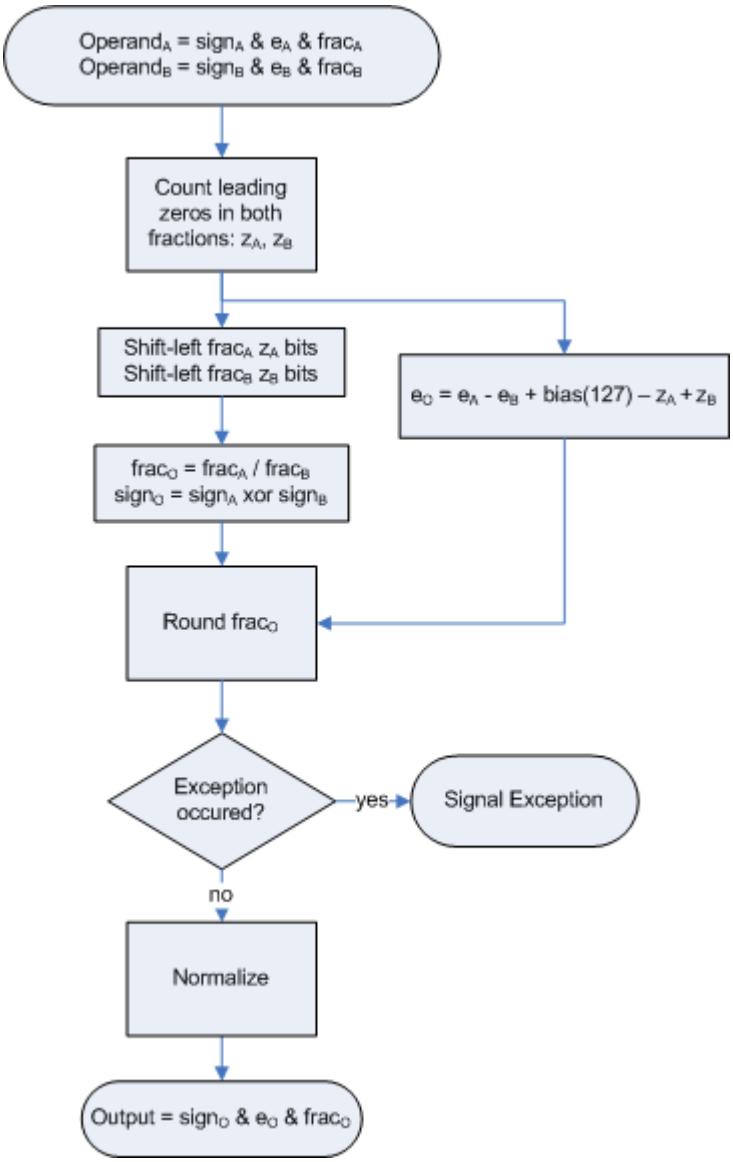
Step 2: Round the fraction to nearest-even

$frac_O = 1.1100$

Step 3: Result

$2^{83} \times 1.1100$

4.3 Division



The division was done serially using the basic algorithm taught in most schools, which is division through multiple subtractions. Since divisions are not needed as often as multiplications (divisions can be done also through multiplications!), it was implemented as serial and in the process saving some hardware area.
To demonstrate the basic steps of division, let's say we want to divide two 5-digits FP numbers:

$$\begin{array}{ll} & 2^{110} \times 1.0000 \\ \div & 2^{100} \times 0.0011 \\ \hline \end{array}$$

Step 1: count leading zeros in both fractions.

$z_A = 0$, $z_B = 3$

Step 2: shift-left the fractions according to z_A, z_B . Calculate the result exponent

$frac_A = 10000\ 00000$
$frac_B = 00000\ 11000$

$e_O = 2^{110-100+bias-0+3} = 2^{140}$

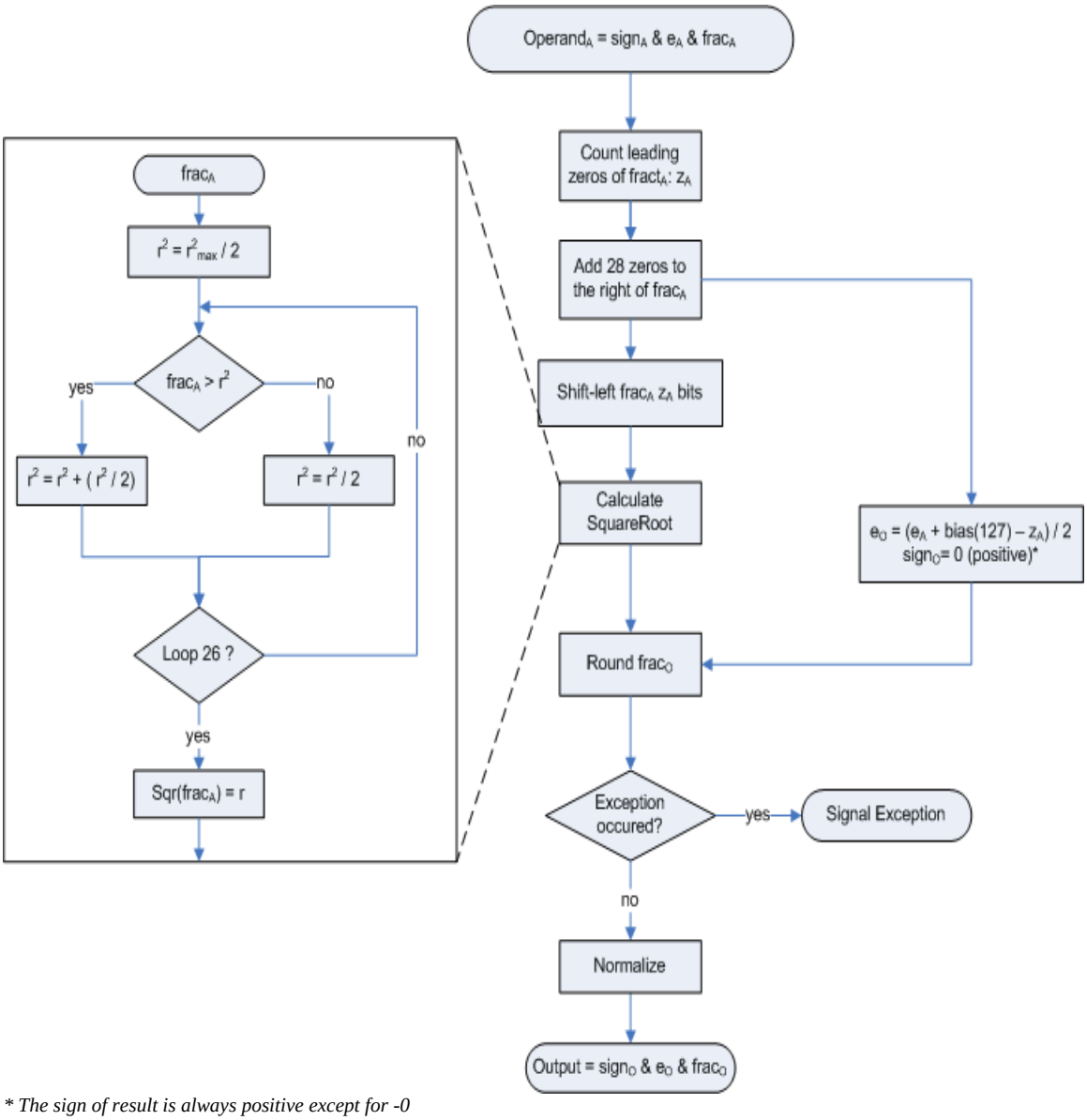
Step 3: divide both fractions

$$\begin{array}{rr} & 100000{,}0000 \\ \div & 000001{,}1000 \\ \hline & 1{,}0101 \end{array}$$

Step 4: result

$1{,}0101 \times 2^{140}$

4.4 Square-Root

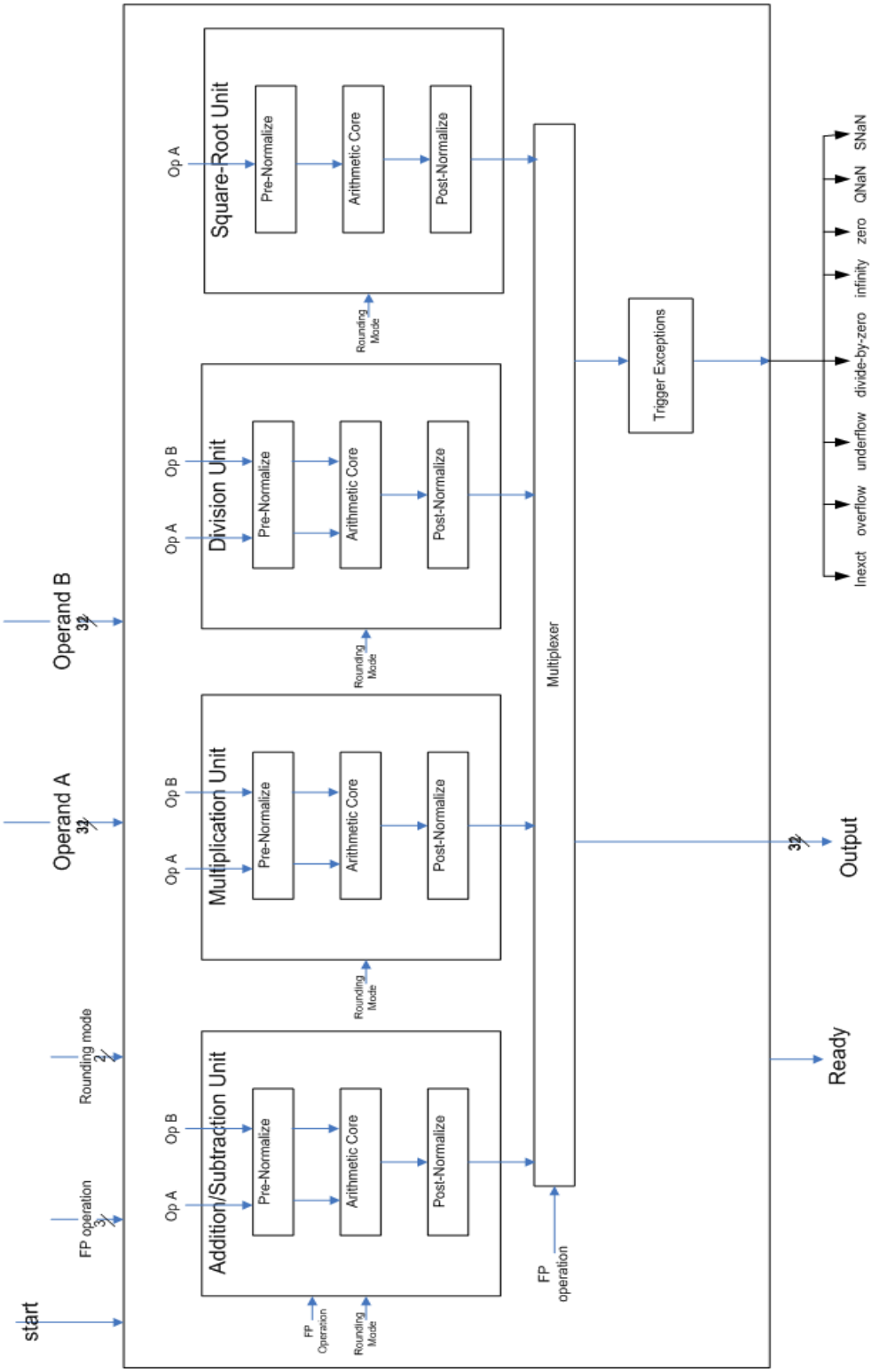


** The sign of result is always positive except for -0*

The square root is calculated using an iterative algorithm, which needs the same number of loops as the precision of the result. The square-root algorithm used here doesn't need any multipliers or divisors, because all multiplications were replaced with left-shifts and all divisions with right-shifts. This makes the algorithm very efficient and fast for hardware implementations.

4. Hardware implementation

The FPU core basic architecture is shown below:



The FPU core was designed to be as modular as possible. The current core supports five arithmetic operations:

1. Add
2. Subtract
3. Multiply
4. Divide
5. Square Root

To save logic elements on the chip, one can disable the arithmetic units that are not needed by modifying the output multiplexer code, since all units are totally independent from each other. Future arithmetic units can be added very easily just by instantiating the unit and connecting its output to the output multiplexer.
All arithmetic operations have these three stages:

1. **Pre-normalize**: the operands are transformed into formats that makes them easy and efficient to handle internally.

2. **Arithmetic core**: the basic arithmetic operations are done here.

3. **Post-normalize**: the result will be normalized if possible (leading bit before decimal point is 1, if possible) and then transformed into the format specified by the IEEE standard.

A common post-normalization unit for all arithmetic operations was not used, although it was possible to combine them all in one unit. It was not done so because:

- Post-normalizations differ from one arithmetic operation to another, e.g. the post-normalization unit for addition/subtraction needs 259 logic elements (LCs) while multiplication needs 889 LCs.
- Most importantly, less clock cycles are needed for some operations
- Hardware can be saved if not all operations are wanted

Through pipelining the FPU core was able to reach higher f_{max} at the cost of throughput (more clock cycles). The number of clock cycles that the FPU needs for each arithmetic operation is listed below:

Operation	**Number of clock cycles**
Addition	7
Subtraction	7
Multiplication	12
Division	35
Square-root	35

By lowering the amount of pipelining, the clock cycles needed can be reduced, but at the same time f_{max} decreases. To reduce the clock cycles needed and therefore increase the speed of processing without much effecting f_{max}, the precision can be decreased. So for

example when dividing, the serial divider needs 26 clock cycles (again, without pre-, post-normalization) for the 24-bits precision result and the extra 3-bits to enable rounding. So if we reduced the precision to let's say 10-bits, we can reduce the needed clock cycles to 13 clock cycles. The same thing can be done with the square root operation. Decreasing the precision will also save hardware area.

4.1 Interface

Input signals:

Signal Name	Width	Description
clk_i	1	clock signal
opa_i	32	operand A
opb_i	32	operand B
fpu_op_i	3	FPU operations: 000 = add, 001 = subtract, 010 = multiply, 011 = divide, 100 = square root 101 = unused 110 = unused 111 = unused
rmode_i	2	Rounding modes: 00 = round to nearest even 01 = round to zero 10 = round up 11 = round down
start_i	1	Start signal

Output signals:

Signal Name	Width	Description
output_o		output
ready_o		ready signal
Exceptions		
ine_o	1	inexact
overflow_o	1	overflow
underflow_o	1	underflow
div_zero_o	1	divide by zero
inf_o	1	infinity
zero_o	1	zero
qnan_o	1	QNaN
snan_o	1	SNaN

4.2 Compilation and Synthesis

The FPU core was compiled and synthesized successfully with *Altera Quartus II v.5* and *Synplify Pro 8.1*. The *Cyclone I–EP1C6Q240C6* was the intended FPGA.
The order in which the files shall be compiled are:

```
fpupack.vhd
pre_norm_addsub.vhd
addsub_28.vhd
post_norm_addsub.vhd
pre_norm_mul.vhd
mul_24.vhd
post_norm_mul.vhd
pre_norm_div.vhd
serial_div.vhd
post_norm_div.vhd
pre_norm_sqrt.vhd
sqrt.vhd
post_norm_sqrt.vhd
comppack.vhd
fpu.vhd
```

The number of Logic elements needed for each unit is shown below.

Altera Quartus II v.5

f_{max}: 100 MHz

Number of logic elements:

Addition unit:	684
Multiplication unit:	1530
Division unit:	928
Square-root unit:	919
Top unit:	326
Total:	4387

4.3 Test and verification

The FPU was tested with test cases created using SoftFloat (http://www.jhauser.us/arithmetic/SoftFloat.html). SoftFloat is a software implementation of floating-point that conforms to the IEC/IEEE Standard for Binary Floating-Point Arithmetic. The FPU was tested in ModelSim with 100000 test cases for each arithmetic operation and for each rounding mode. This comes up to 2 million test cases. The instructions for how to create the test cases and test the FPU core, can be found in the readme file in folder test_bench.

The FPU mastered also successfully the hardware test. The FPU was implemented in the *Cyclone I–EP1C6Q240C* FPGA chip and was then connected to the Java processor JOP (www.jopdesign.com) to do some floating-point calculations.

4.4 FPU comparsion

I compared the FPU presented here with Usselmann's FPU (http://www.opencores.com/projects.cgi/web/fpu/overview), since it was the only open source FPU known to me. Both FPU's were tested with *Altera Quartus II v.5* using *Cyclone I–EP1C6Q240C6*. Summery of the most important parameters are shown in the table below.

	FPU #1 (presented here)	**FPU #2 (Usselmann)**
Nr. of logic elements	*3468	7392
f_{max}	100 MHz	6.17 MHz
Clock Cycles		
Addition/Subtraction	7	3
Multiplication	12	3
Division	35	3
Square-root	35	NA

** Without the square-unit*

5. Conclusion

An FPU was implemented, which successfully achieved the goals stated at the beginning which were:

- 100 MHz operating frequency
- Few clock cycles
- Few logic elements

Further, the FPU was tested, verified, and implemented in hardware successfully.

6. References

1. IEEE computer society: IEEE Standard 754 for Binary Floating-Point Arithmetic, 1985.

2. David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic, 1991.

3. W. Kahan: IEEE Standard 754 for Binary Floating-Point Arithmetic, 1996.

7. Updates

30/01/2006
Added serial multiplier to the parallel multiplier already implemented to reduce the number of logic elements needed. By changing 2 constants in fpu.vhd one of the multipliers can be chosen.

28/03/2006
Tested the FPU with 2 million test cases and corrected few bugs.