Politecnico di Milano - Department of Electronics, Information and Bioengineering

POLITECNICO MILANO 1863

BFloat16 SQRT using Goldschmidt's algorithm

[System Verilog]

Student(1)	Enrico Melacarne
	10494317
Student(2)	Mykhaylo Kotsaba
	10525073
Course	Embedded Systems
Academic Year	2019-2020
Advisor	Andrea Galimberti
	Davide Zoni
Professor	William Fornaciari

April 20, 2020

Contents

1 Introduction

This is the final project for the Embedded Systems course at Politecnico di Milano. The goal is the implementation of both square root and inverse square root functions in 16-bit bFloat format using System Verilog. The module realized is integrated in the Floating-Point Unit developed by our advisors Andrea Galimberti and Davide Zoni [1].
Our work consisted of writing the lampFPU_sqrt and lampFPU_fractSqrt module. Then we added features to the testbench and the DPI file to verify the behavior of the whole FPU module in sqrt and inverse sqrt operations. In particular Behavioral and Post-Synthesis Functional simulations where taken into account.

Target device: xc7a100tcsg324-1 (FPGA) **Frequency:** 100 MHz

1.1 Format

Before starting it's appropriate to describe the format. BFloat16 is a truncated 16-bit version of the 32-bit IEEE 754 single-precision floating-point format (Float32), with the intent of accelerating machine learning and near-sensor computing. It preserves the range of 32-bit floating-point numbers by retaining 8 exponent bits but supports only an 8-bit precision rather than the 24-bit significant of the Float32 format.
It is so composed S|E|M:

- S The sign uses 1 bit: [0], meaning the number is positive or [1], meaning the number is negative.
- E The exponent uses 8 bits, [0; 255] but it's biased to -127 value. So, the real range is from [-127; 128].
- M The mantissa uses 7 bits, but we are neglecting the first '1' bit. So, the real mantissa is '1' followed by 7 bits of mantissa [1.M].

1.2 Goldschmidt's Algorithm

To perform the square root, it was adopted the Goldschmidt's iterative algorithm [2][3].

$$b_0 = S \;\; ; \;\; Y_0 \approx 1/\sqrt{S} \;\; ; \;\; y_0 = Y_0 \;\; ; \; x_0 = S y_0$$

$$b_{n+1} = b_n Y_n^2 \;\; ; \;\; Y_{n+1} = \frac{3 - b_{n+1}}{2} \;\; ; \;\; x_{n+1} = x_n Y_{n+1} \;\; ; \;\; y_{n+1} = y_n Y_{n+1}$$

The iteration converges to

$$\lim_{n\to\infty} x_n = \sqrt{S} \; ; \; \lim_{n\to\infty} y_n = 1/\sqrt{S}$$

The b_0 is the input mantissa, Y_0 is the approximation of the inverse square root, in our code it was used a Lookup Table (LUT), which is a function in the lampFPU_pkg, to perform immediately the firsts bits of the result. Increasing in this way time performances.

2 Design and implementation

The two operations are executed by the same two modules. The first one takes care of sign, exponent and special cases computation while adapts the mantissa which is passed to the second module, where the iterative algorithm is implemented.

2.1 lampFPU_sqrt module

This module takes in input the operand from the top module, including dedicated signals to detect special cases and to take care of denormal numbers. The value computed is then normalized and returned to the FPU top module for rounding, which can be in truncate mode or nearest mode. Truncate mode simply takes the most significant 8 bits while the nearest mode performs computation and rounds to the nearest bit.
When we calculate the square root of a floating-point number, the exponent value should be divided by 2. This is not enough for our hardware implementation as we have an unbiased binary representation of the exponent. Furthermore we should take in account two other issues:

1. Effect of halving on odd numbers when integer registers are used.
2. Exponent computation in case of denormal numbers

All following considerations are made for the square-root function, same arguments holds for the inverse square-root of which we report just the results.
Exponent formula for unbiased input: As previously anticipated, when calculating the square-root of a number the input exponent should be divided by two. But at the same time we should consider that the real exponent is a biased version of the exponent coded in our register:

$$E_{r,o} = \frac{(E_{r,i})}{2}$$

$$E_r = E - BIAS$$

Where E$_r$ are the real values of exponents and E are the values of the FPU registers, making the substitution for both the E$_{r,o}$ and E$_{r,I}$ we obtain:

$$E_o - BIAS = \frac{(E_i - BIAS)}{2} \Rightarrow E_o = \frac{(E_i + BIAS)}{2}$$

Odd numbers halving issue: As already pointed out we lose the last bit if the E$_i$ + BIAS sum is an odd number, this cause a sqrt(2) error on the final computation. We found a general formula which relies on the possibility to divide by 2 the input mantissa. The formula we have found is always valid for the exponent computation in the square-root function. First of all is necessary to consider that the BIAS value is always an odd number, while le input exponent could be even or odd:

$$BIAS \ = \ 2b \ + 1; \quad E_i = 2n + 1 \vee E_i = 2n$$

It is important to notice that:

$$b = \frac{BIAS - 1}{2}$$

$$n = E_i \gg 1 \, (always!)$$

In case of odd input exponent:

$$\mathrm{Eo} = \frac{E_i + BIAS}{2} = \frac{2n+1}{2} + \frac{2b+1}{2} = \mathrm{n} + \frac{1}{2} + \mathrm{b} + \frac{1}{2} = \mathrm{n} + \mathrm{b} + 1$$

$$\sqrt{\mathrm{F}2^{\mathrm{E_i}}} = \sqrt{\mathrm{F}}2^{n+b+1}$$

In case of even input exponent:

$$Eo = \frac{E_i + BIAS}{2} = \frac{2n}{2} + \frac{2b+1}{2} = n + b + \frac{1}{2} + \left(\frac{1}{2} - \frac{1}{2}\right) = n + b + 1 - \frac{1}{2}$$

$$\sqrt{F2^{E_i}} = \sqrt{F}2^{n+b+1}2^{-\frac{1}{2}} = \sqrt{\frac{F}{2}2^{n+b+1}}$$

The issue is then solved just by using the same formula, but shifting the input mantissa in case of even input exponent. The input of the module performing Goldschmidt algorithm has been extended in order to maintain all the mantissa's bits of resolution.
Denormal number extension: We want now to demonstrate that the result just obtained is valid also when we want to compute the exponent of denormal numbers. Denormal number have the form E=0x00 and F>0x00. FPU's top module takes care for the initial preprocessing of this kind of numbers: incoming E value is set to E=0x01 and F is shifted in order to have the usual binary form F = 1XXX XXXX, where X means "don't care", the number of shifts executed is saved in the Number of Leading Zeros register "nlz". Our formula has to take into account the number of leading zeros subtracting nlz from the exponent $\mathrm{E}_{i,}$ which is fixed to 1. Again we have two cases, when "nlz" is odd and when "nlz" is even. In formulas:

$$BIAS \ = \ 2b \ - \ 1; \quad E_i = 1$$

$$nlz \ = \ 2z \ + 1 \vee nlz \ = \ 2z$$

In case of odd input nlz:

$$\mathrm{Eo} = \frac{E_i - nlz + BIAS}{2} = \frac{1 - 2z - 1}{2} + \frac{2b+1}{2} = -\mathrm{z} + \mathrm{b} + \frac{1}{2} + \left(\frac{1}{2} - \frac{1}{2}\right) = -\mathrm{z} + \mathrm{b} + 1 - \frac{1}{2}$$

$$\sqrt{F2^{E_i}} = \sqrt{F}2^{-z+b+1}2^{-\frac{1}{2}} = \sqrt{\frac{F}{2}2^{-z+b+1}}$$

In case of even input nlz:

$$\mathrm{Eo} = \frac{E_i - nlz + BIAS}{2} = \frac{1 - 2\mathrm{z}}{2} + \frac{2b+1}{2} = -\mathrm{z} + \frac{1}{2} + \mathrm{b} + \frac{1}{2} = -\mathrm{z} + \mathrm{b} + 1$$

$$\sqrt{F2^{E_i}} = \sqrt{F}2^{-z+b+1}$$

We have now highlighted that the input mantissa should be divided by two in case of odd input nlz and even input exponent. Now is important to notice that:

$$b = \frac{BIAS - 1}{2}$$

$$(E_i - nlz) \gg 1 = n \quad \text{if nlz} = 0 \text{ and } \mathrm{E}_i > 0 \text{ (standard case)}$$
$$(E_i - nlz) \gg 1 = -z \quad \text{if nlz} > 0 \text{ and } \mathrm{E}_i = 1 \text{ (denormal case)}$$

We can derive the final expression of exponent for the square root operation:

$$\mathrm{Eo}^{\mathrm{sqrt}} = (E_i - nlz) \gg 1 + \frac{BIAS-1}{2}$$

Following a similar argumentation we obtained also the inverse square root general formula:

$$\mathrm{Eo}^{\mathrm{invSqrt}} = -(E_i - nlz) \gg 1 + 3\frac{BIAS-1}{2}$$

In both cases the input mantissa should be divided by 2 if nlz is odd or if the input exponent is even, while it should be taken as is if the nlz is even and the input exponent is odd.

Overflow, underflow, denormal outputs and normalization step:
It is important to notice that the two formulas just obtained can not bring to overflow or underflow. This is more in general a lucky peculiarity of the sqrt and inverse-sqrt operations. In fact to the range of numbers in input correspond a narrower output range. Moreover there is no input number, even denormal inputs, that can lead to a denormal output. These statements are easy to check, we need just to substitute the minimum and maximum values of exponent and mantissa in the previous formulas:

$$\mathrm{Eo}_{max}^{sqrt} = \left(E_{i,max} - nlz_{min}\right) \gg 1 + \frac{BIAS-1}{2} = (255-0) \gg 1 + 63 = 190 \in [0:255]$$

$$Eo_{min}^{sqrt} = \left(E_{i,min} - nlz_{max}\right) \gg 1 + \frac{BIAS-1}{2} = (1-7) \gg 1 + 63 = 60 \in [0:255]$$

$$\mathrm{Eo}_{\mathrm{max}}^{invSqrt} = -\left(E_{i,min} - nlz_{maz}\right) \gg 1 + 3\frac{BIAS-1}{2} = -3 + 3 * 63 = 186 \in [0:255]$$

$$\mathrm{Eo}_{min}^{invSqrt} = -\left(E_{i,max} - nlz_{min}\right) \gg 1 + 3\frac{BIAS-1}{2} = -127 + 3 * 63 = 62 \in [0:255]$$

This property of sqrt and inverse-sqrt operations show itself also on the fractional part computation. Having [0.5;2) as fractional part input range we end up with narrower output range for both the operations:

$$[0.5;2) \Rightarrow sqrt \Rightarrow (0.7;1,42)$$

$$[0.5;2) \Rightarrow invSqrt \Rightarrow (0.7;1,42)$$

The importance of this result is that we can divide in two simple cases the normalization process of the result. In fact there are just two forms for the "most significant nibble" of Goldschmidt algorithm's output: 00.1x and 01.xx. In the first case we need a left shift of the result of just one bit, and a decrease of the Eo value of 1:

$$Eo_{postNorm} = Eo - 1$$

This variation is not enough to generate an underflow being the previously calculated value of $\mathrm{E}_{o,min} = 60$.
In the second case both output exponent and output mantissa are not modified.
Normalization is practically done in our digital design taking directly the twelve MSBits of the fractSqrt module output and computing the stickybit from the LSB and the discarded bits.
Special cases: finally the computed values are returned to the top module unless a special condition is detected, for example if the operand in input is negative the output is NaN. These conditions are detected for both the operations by the same function which is defined in the package and called in this module.

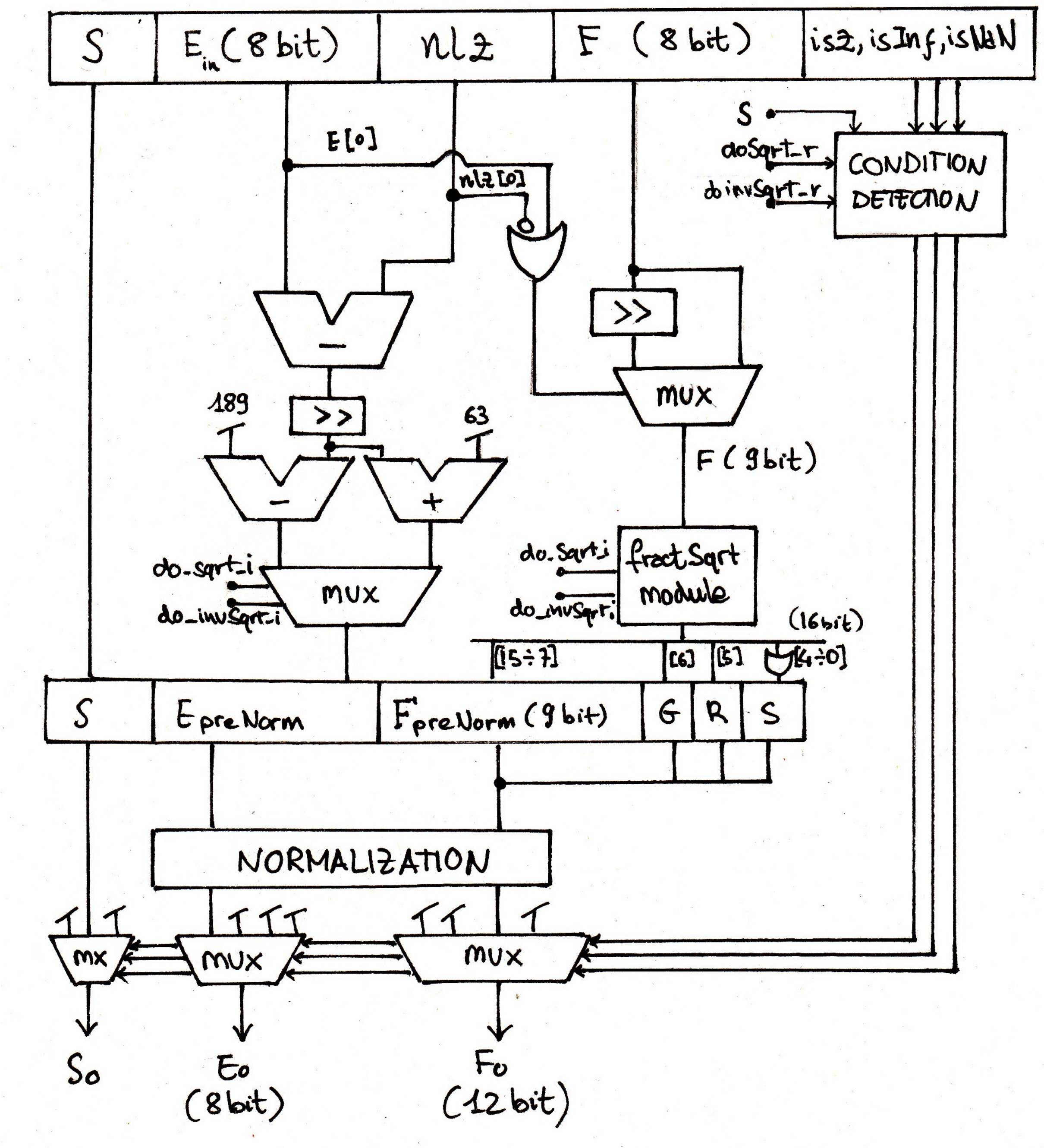
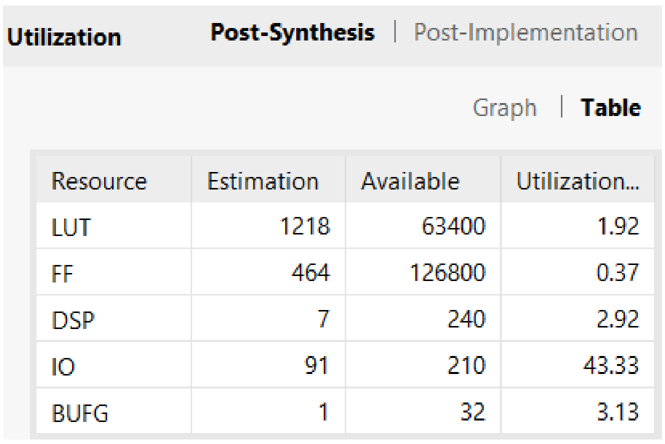


Figure 1: Schematic of the lampFPU_sqrt module

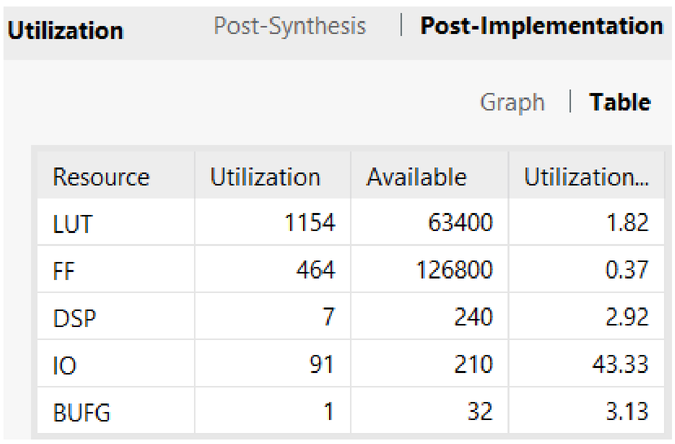
2.2 lampFPU_fractSqrt module

This module is the heart of the square root operation. Here we perform the algorithm on the mantissa and return the result, when the validity bit is set to one, to the sqrt module. In input we have 9 bits of mantissa in two formats, depends on the precedent shifts, 0.1MMMMMMMM or 1.MMMMMMM0. The result is instead 16 bits long XX.XXXXXXXXXXXXXX format, point in the second position.

The algorithm was implemented like a state machine with three states. In the IDLE we just wait until a signal to do the square root or the inverse square root doesn't arrive. When this happens, we save the initial values, perform and move to the next state. Since having a double multiplication, the timing parameter WNS was negative at the first implementation when our state machine had only two states. So, we have decided to divide the iteration into two states. One performs b and result and the other does the remaining operations. In this way, the WNS came out positive on VIVADO 2018.2 and VIVADO 2019.2. The number of clock cycles is double, but it respects the timing request.
To elaborate the multiplications, we make use of auxiliary registers, which length is the sum of the multiplier and multiplying registers length. But since every cycle this would lead to a very large register, we take only the necessary number of bits from the auxiliary register. We use it for the y square, b and result.



Resource	Estimation	Available	Utilization...
LUT	1218	63400	1.92
FF	464	126800	0.37
DSP	7	240	2.92
IO	91	210	43.33
BUFG	1	32	3.13



Resource	Utilization	Available	Utilization...
LUT	1154	63400	1.82
FF	464	126800	0.37
DSP	7	240	2.92
IO	91	210	43.33
BUFG	1	32	3.13

Figure 2: Summary of the resource's utilization. On the left (1a) we can observe the results of the Post-Synthesis and on the right (1b) the results of the Post-Implementation.

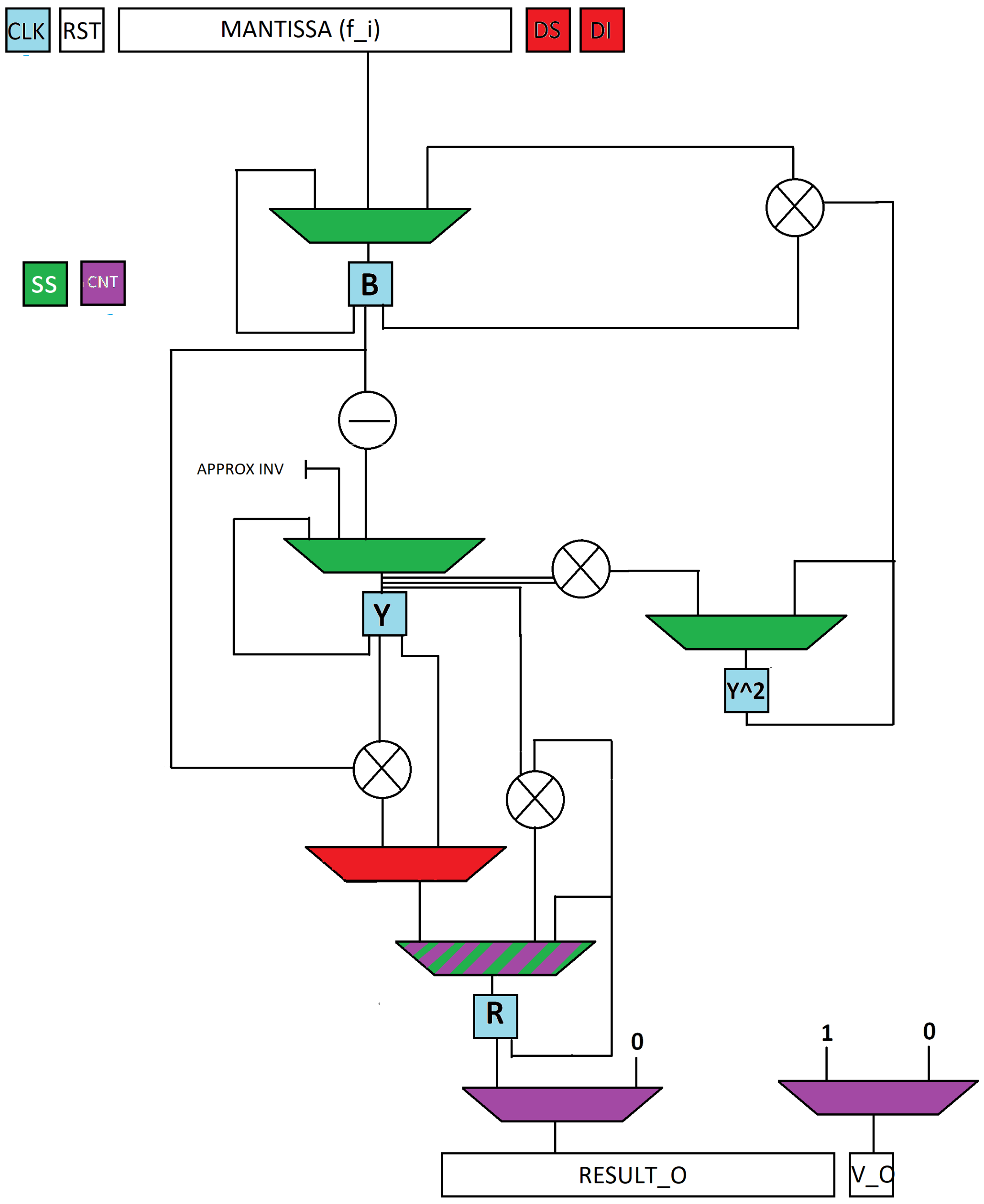


Figure 3: Schematic of the lampFPU_fract module. The trapezoids are multiplexers.(-) is the substraction and division. (X) is the multiplication. The colors indicate the selecting variable of the multiplexer. The FF are light blue because they are updated at every clock.

3 Experimental evaluation

To test the module, it was written an appropriate testbench that makes use of c code to calculate the square root and compare it whit the FPU results. To use the dpi, it's necessary to run on TLC console some commands before. Otherwise, the simulation won't start.

xsc <path_directory_c_file>/dpi_lampFPU.c

set_property -name {xsim.elaborate.xelab.more_options} -value {–sv_lib dpi -sv_root <path_directory_a_file } -objects [get_filesets sim_1]

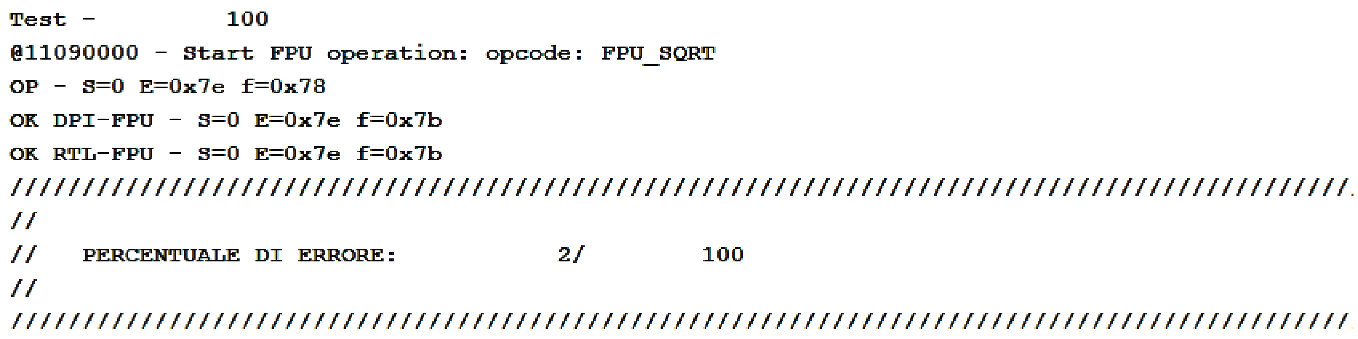
It is also necessary to change the path to the svdpi.h file in the dpi_lampFPU.c file. It depends on your Installation folder and version of the software.

#include "C:\Xilinx\Vivado\2017.3\data\xsim\include\svdpi.h"

3.1 Test bench and dpi

In the DPI we have added two functions, one to calculate the square root and one to calculate the inverse square root. The functions receive 16 bits, operate in 32 bits and return 16 bits. The mantissa is truncated to 7 bits. This may influence a lot the percentage of errors in the test bench when we operate in the Nearest rounding mode. Errors of one bit, not more.

In the test bench, we have added TASK_testSqrt that generates the test for the square root and for the inverse square root also. The single task performs 100 times. Generates random numbers and capture the results, displaying the numbers in the console. In the end, it prints out the percentage of errors.



```
Test -          100
@11090000 - Start FPU operation: opcode: FPU_SQRT
OP - S=0 E=0x7e f=0x78
OK DPI-FPU - S=0 E=0x7e f=0x7b
OK RTL-FPU - S=0 E=0x7e f=0x7b
///////////////////////////////////////////////////////////////////////////////////////////
//
//   PERCENTUALE DI ERRORE:          2/        100
//
///////////////////////////////////////////////////////////////////////////////////////////
```

Figure 4: Console output of the test bench

3.2 Accuracy Results

We have performed the test over 1000 times for both the square root and inverse square root. At the end of the block test, we can compute the percentage of errors. In the worst-case, it's just a bit of difference from the real one and it depends also by the selected rounding mode. Different type of rounding gives us a different percentage of errors like we can observe from the graph below. For a number of iterations under three, it's not working.

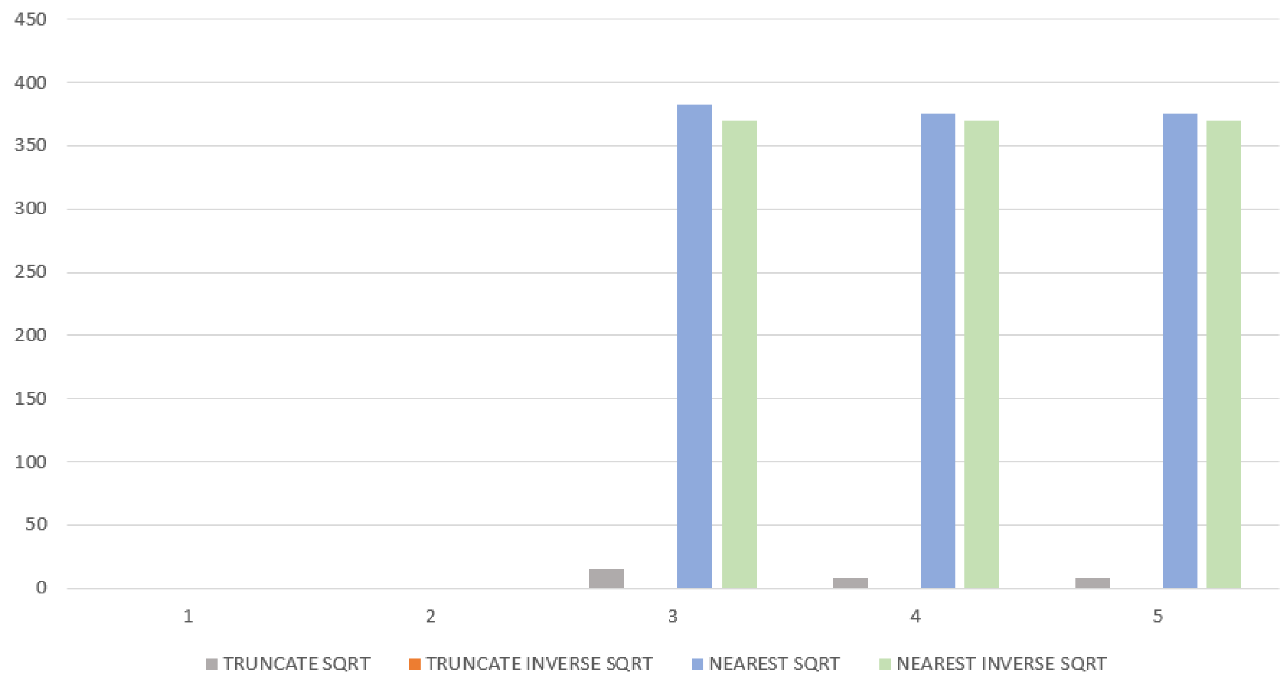
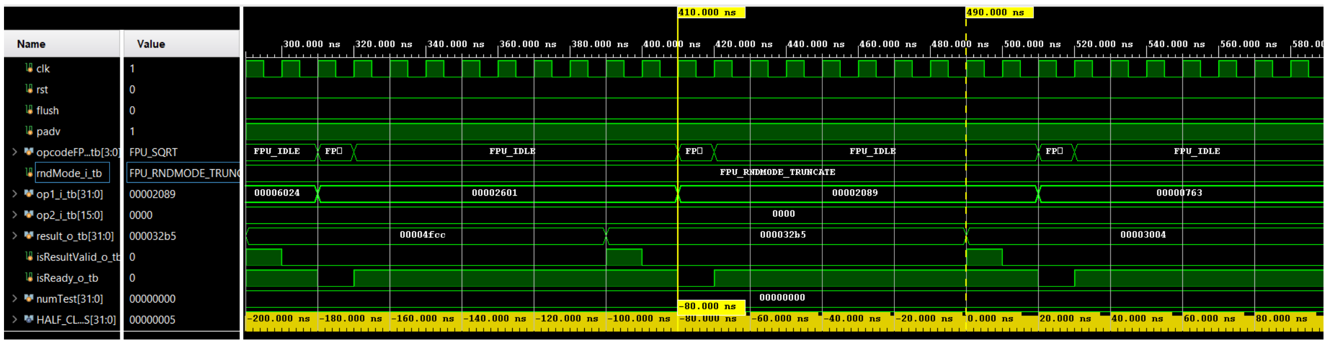


Figure 5: Comparison of errors. Different rounding mode and number of iterations. (1000 tests)

In the waveform below we can observe the behavioral simulation of the sqrt. With three iterations we can observe a total computation time of eight clock cycles, 80ns.

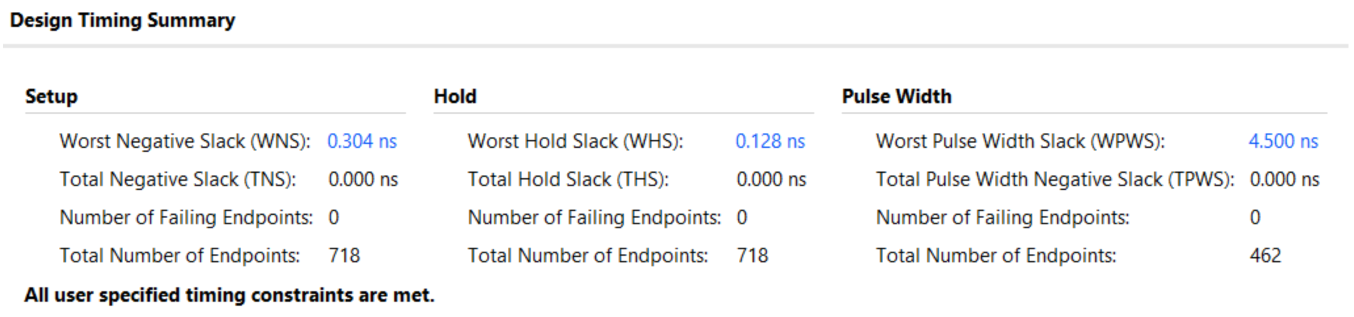


Name	Value
clk	1
rst	0
flush	0
padv	1
opcodeFP...tb[3:0]	FPU_SQRT
rndMode_i_tb	FPU_RNDMODE_TRUNC
op1_i_tb[31:0]	00002089
op2_i_tb[15:0]	0000
result_o_tb[31:0]	000032b5
isResultValid_o_tb	0
isReady_o_tb	0
numTest[31:0]	00000000
HALF_CL...S[31:0]	00000005

Figure 6: Waveform of the behavioral simulation.

3.3 Timing Results

Like said before, we operate at 100Mhz and so the timing is very important to us. To control the timing issues, we have made the implementation and seen the Worst Negative Slack (WNS) parameter. With a double multiplication, the WNS was negative and we have seen that the problem was there. Only when we changed the state machine, dividing the long chain of multiplications, we had a positive WNS.


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.304 ns	Worst Hold Slack (WHS):	0.128 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	718	Total Number of Endpoints:	718	Total Number of Endpoints:	462

All user specified timing constraints are met.

Figure 7: Post-Implementation WNS

To enable the timing report, it was necessary creating a constraint file to generate a clock.

```
create_clock -period 10.000 -name clk -waveform {0.000 5.000} [get_ports clk]
```

References

[1] D.Zoni A.Galimberti. Bfloat16fpu. *https://gitlab.com/davide.zoni/bfloat_fpu_systemverilog*.

[2] Goldschmidt's Algorithm. Wikipedia. *https://en.wikipedia.org/wiki/Methods_of_computing_square_roots*.

[3] Peter Markstein. Software division and square root using goldschmidt's algorithms.