Politenico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

Embedded Systems Project

Square Root for bfloat16

Authors:
Francesco Monti
Fabio Nappi
Davide Piovani

Supervisor:
Dr. Davide Zoni Dr.
Andrea Galimberti

April 25, 2020

Abstract

Machine Learning workloads are computationally intensive and their programs may require to run for hours or days. In order to achieve high performance and therefore reduce the time in use, power consumption and hardware complexity, Google has very recently developed a custom floating point format called "Brain Floating Point Format" or "bfloat16". Various institutions have recognized the potentiality of this new format, and started to develop FPU supporting it. In this document, we present the implementation of a particular mathematical operation for bloat16, the Square Root.

Contents

1 Introduction

In order to represent real numbers in computer memory, a common approach is to use a floating point format. Depending on the type of encoding used, these formats occupy different memory sizes, usually 32-bits (Single precision floating point, or binary32) or 64-bits (Double precision floating point, or binary64).
The value F given by the floating point format, of any kind, can be expressed as:

$$F = (-1)^S * M * 2^E$$

where

- S is the Sign (0 for positive, 1 for negative)
- E is the Exponent
- M is the Mantissa (also called Significand, or fractional part)

Floating-point formats and arithmetic operations are specified by the IEEE 754 standard (https://standards.ieee.org/standard/754-2019.html).
Under this standard (n is the number of bits reserved to the exponent):

- The exponent E is unsigned, with values between 0 and $(2^n - 1)$
- The exponent E is biased, meaning that is encoded using an offset-binary representation, with the zero offset being 2^{n-1}–1
- The mantissa has an hidden bit, always set to 1 (except for subnormal numbers); therefore, the actual mantissa is 1.M
- The standard also introduces some special values, for different purposes, such as Infinite (Inf) and Not-a-Number (NaN)

The single and double precision floating point representations are the two most used in modern computers and mobile phones. These devices usually aren't particularly constrained by memory space and may actually need the accuracy offered by a 32-bit or even 64-bit representation.
On the other hand, many programs and situations don't need that much

accuracy in order to perform their tasks. So, we may use the half-precision floating-point format, also called binary16. For example, Embedded Systems are tipically memory and power constrained and usually don't need that much precision while performing their operations.
The binary16 standard requires 1 bit for the sign, 5 bits for the exponent and 10 bits for the mantissa. This means that we can only represent numbers between 2^{-14} and 2^{15}.
This small range of possible numbers can be a strong limitation. Therefore, another standard has been recently developed by Google, the bfloat16 (Brain Floating Point) floating-point format. This standard requires 1 bit for the sign, 8 bits for the exponent and 7 for the mantissa.
This format is a truncated version of the binary32 with the intent of accelerating machine learning and near-sensor computing. It preserves the approximate dynamic range of 32-bit floating-point numbers by retaining 8 exponent bits, but supports only an 8-bit precision rather than the 24-bit significand of the binary32 format.

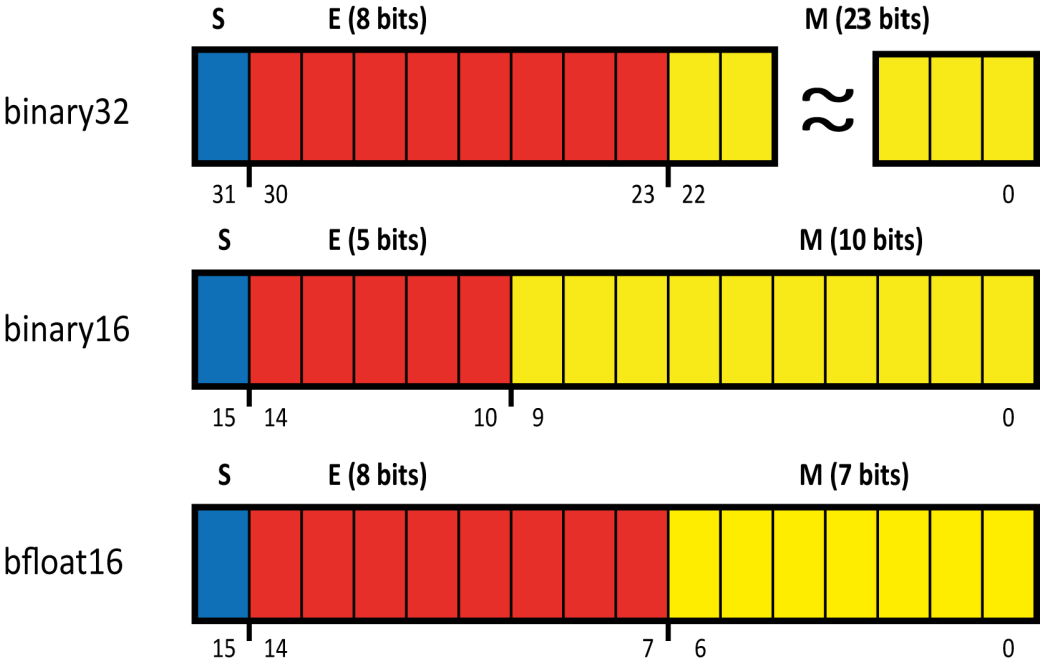

Figure 2: How the bits are assigned in the different standards.
From top to bottom: binary32, binary16, bfloat16

Heap Lab has already developed an open source FPU (https://gitlab.com/davide.zoni/bfloat_fpu_systemverilog) supporting the most common operations for bfloat16 operands.
However, it still misses some other important operations, such as the fused Multiply-Add and the Square Root. We are requested to implement one of these operations; in particular we have chosen to develop the Square Root algorithm.

More formally, the goals of our project are:

- Add support for Square Root operation (SQRT)
- Add support for the Inverse Square Root operation (INVSQRT)
- Integrate our design with the already developed FPU
- Verify the implementation correctness, comparing the results produced by our design with the ones obtained from the C function *sqrt*

The rest of this document is organized as follow: in section 2, we explain the algorithm used for computing the Square Root of a real number. In section 3 we describe our design decisions and our implementation. In section 4, we show the benchmarks used to test the correctness of our implementation and an analysis of the algorithm's efficiency. Finally, in section 5 we offer a conclusion for our work.

2 Goldschmidt's algorithm

The Square Root is quite complicated from a computational point of view; indeed, the most efficient known algorithms require an iterative approach to compute the result of this operation, involving a chain of multiplications and other operations. The two most famous algorithms are the Newton-Raphson method and Goldschmidt's algorithm. The latter has been chosen for bfloat16, since it's suitable for an efficient hardware implementation.
Given a Significand S, Goldschmidt's algorithm can be used to compute $\sqrt{S}$. With only few small variations, it's also possible to compute $\frac{1}{\sqrt{S}}$.
The objective is to find a series of R_0, R_1, ... , R_n such that the following product tends to 1:

$$B_n = S * R_0^2 * R_1^2 * ... * R_{n-1}^2 \approx 1$$

Then, we can approximate the Square Root as:

$$X_n = S * R_0 * R_1 * ... * R_n \approx \sqrt{S}$$

And the Inverse Square Root as:

$$Y_n = R_0 * R_1 * ... * R_n \approx \frac{1}{\sqrt{S}}$$

Goldschmidt's algorithm initializes its variables as follows:

- $B_0 = \mathrm{S}$
- $Y_0 = R_0 = \frac{3-S}{2}$
- $X_0 = \mathrm{S} * R_0$

Each Goldschmidt's iteration firstly consists in computing:

1. $B_i = B_{i-1} * R_{i-1}^2$
2. $R_i = \frac{3-B_i}{2}$

And subsequently it updates the Square Root and the Inverse Square Root approximations as follows:

1. $X_i = X_{i-1} * R_i$
2. $Y_i = Y_{i-1} * R_i$

Iterations stop when R reaches a good approximation of 1.

We wanted to show that the Goldschmidt's algorithm is very efficient, as it requires only a very small number of iterations to reach the desired precision. To do so, we quickly implemented the algorithm in Matlab and tracked the values of B_i, R_i, Y_i and X_i for ten iterations.
The code (3) is simply the software implementation of the algorithm, and we run it twice; the first time, we put $S = 1{,}5625$, which yelds an exact Square Root (4). The second time, we used $S = 1{,}5$, which instead yelds an irrational number as solution (5).
In both cases, the algorithm converges in 5 or 6 iterations. Given that we only use 8 bits for the Mantissa, and not 24 like Matlab, we'll reach the desired precision in even fewer iterations.

2.1 Additional notes

In this section we presented only a quick and high-level description of Goldschmidt's algorithm. Implementation details were not discussed, but they are quite important. For example:

- Goldschmidt's algoritmh is not self-correcting, meaning we can't fully trust the LSB.
- The values (significant included) are in the bfloat16 format, and must be consequently managed
- The full square root algorithm requires also to compute the final exponent and eventually round the result

These and other details are more widely discussed in the next section.

```
5 -    format long
6
7 -    n = 10;
8 -    B = zeros(n, 1);
9 -    y = zeros(n, 1);
10 -   R = zeros(n, 1);
11 -   x = zeros(n, 1);
12
13 -   S = 1.5625;
14 -   B(1) = S;
15 -   R(1) = (3 - S) / 2;
16 -   y(1) = R(1);
17 -   x(1) = S * y(1);
18
19 -   for i=2:n
20 -       B(i) = B(i-1) * R(i-1)^2;
21 -       R(i) = (3 - B(i)) / 2;
22 -       y(i) = y(i-1) * R(i);
23 -       x(i) = x(i-1) * R(i);
24 -   end
25
26 -   [B R y x]
```

Figure 3: Goldschmidt's algorithm implemented in Matlab

```
ans =

   1.562500000000000   0.718750000000000   0.718750000000000   1.123046875000000
   0.807189941406250   1.096405029296875   0.788041114807129   1.231314241886139
   0.970326247853848   1.014836876073076   0.799733183168011   1.249583098700017
   0.999333069156312   1.000333465421844   0.799999866531299   1.249999791455155
   0.999999666328275   1.000000166835862   0.799999999999967   1.249999999999948
   0.999999999999917   1.000000000000042   0.800000000000000   1.250000000000000
   1.000000000000000   1.000000000000000   0.800000000000000   1.250000000000000
   1.000000000000000   1.000000000000000   0.800000000000000   1.250000000000000
   1.000000000000000   1.000000000000000   0.800000000000000   1.250000000000000
   1.000000000000000   1.000000000000000   0.800000000000000   1.250000000000000
```

Figure 4: Results with S = 1,5625
Columns from left to right represent the values of B, R, Y, X

```
ans =

   1.500000000000000   0.750000000000000   0.750000000000000   1.125000000000000
   0.843750000000000   1.078125000000000   0.808593750000000   1.212890625000000
   0.980735778808594   1.009632110595703   0.816382214426994   1.224573321640491
   0.999719880049084   1.000140059975458   0.816496556899911   1.224744835349867
   0.999999941144115   1.000000029427943   0.816496580927725   1.224744871391588
   0.999999999999998   1.000000000000001   0.816496580927726   1.224744871391589
   1.000000000000000   1.000000000000000   0.816496580927726   1.224744871391589
   1.000000000000000   1.000000000000000   0.816496580927726   1.224744871391589
   1.000000000000000   1.000000000000000   0.816496580927726   1.224744871391589
   1.000000000000000   1.000000000000000   0.816496580927726   1.224744871391589
```

Figure 5: Results with S = 1,5
Columns from left to right represent the values of B, R, Y, X

3 Design and Implementation

For our project we used the Hardware Description Language (HDL) SystemVerilog. SystemVerilog allows to create a device-independent representation of digital logic and is the de-facto standard for digital design and especially for verification (together with VHDL). In SystemVerilog, the *modules* are the basic building blocks. SystemVerilog design consists in interconnected modules. Our implementation is based on two new interconnected modules together with the original FPU, properly modified:

- *lampFPU_sqrt.sv* is the top module, called from the external component of the FPU; it mainly computes the exponent of the number we want the Square Root (or Inv Square Root) of, manages some special cases (Square Root of NaN, Square Root of negative number, etc.) and instantiates *lampFPU_fractSqrt.sv*

- *lampFPU_fractSqrt.sv* implements Goldschmidt's algorithm for the Mantissa of a bfloat16 number.

- Our source code also uses some functions and constants in order to make the design parametric. We imported the already existing package *lampFPU_pkg.sv* containing some useful parameters. We then added some other values and functions we needed. We highlight that in doing so we greatly accelerated the integration process of our modules with the rest of the FPU.

- We modified *lampFPU_top.sv* in order to properly manage the square root operation, adding the required internal wires, instantiating our new module *lampFPU_sqrt.sv* and binding the inputs and outputs to it. We assigned the square root to the operation code 10 and to the inverse square root to 11.

- The folder Source_Code also contains other SystemVerilog files already implemented for the bfloat16 FPU. We need these modules because *lampFPU_top.sv* instantiates them.

The Square Root algorithm requires to compute not only the Mantissa, but the exponent too. *lampFPU_sqrt.sv* and *lampFPU_fractSqrt.sv* need to collaborate in order to obtain the correct result.

- If the exponent is even, we simply divide it by 2 in the external module *lampFPU_sqrt.sv* (so we right shift the exponent).

- If the exponent is odd, simply right shifting would not produce the right result. In order to compute the correct final value, we make the exponent even by exploiting:

$$F = (-1)^S * M * 2^E = (-1)^S * (M * 2) * 2^{E-1}$$

We can now divide by 2 the exponent and we inform the internal module that the exponent was odd with the input *is_exp_odd_i*.
We then exploit a well-known property of the Square Root:

$$\sqrt{2 * M} = \sqrt{2} * \sqrt{M}$$

So, we actually pass M (and not $2 * M$) to the internal module, but we multiply the result computed by Goldschmidt's algorithm by $\sqrt{2}$ before returning it.
The same mathematical properties are applied when we want to compute the inverse square root, but in this case, we exploit:

$$\sqrt{\frac{1}{2 * M}} = \frac{1}{\sqrt{2} * \sqrt{M}}$$

3.1 lampFPU_fractSqrt.sv

We firstly describe the lampFPU_fractSqrt.sv module.
In general, *lampFPU_fractSqrt.sv* takes as input an 8-bits Mantissa M between 1.0 (8b'100000000) and 1.9921875 (8b'11111111) and some flags indicating when to start the computation (*doSqrt_i*), whether to compute the square root or the inverse square root (invSqrt_i), if we are in a special case (*special_case_i*).
The module signals it finished its work with the output valid_o, and returns 16 bits representing $\sqrt{M}$ or $\frac{1}{\sqrt{M}}$. The Mantissa length is greater than expected by the bfloat16 standard, but we'll use this enhanced precision in order to properly round the result when needed.

The sequential logic of *lampFPU_fractSqrt.sv* implements a synchronous reset of the module, setting to zero all the internal wires if the input *rst* is

1. Otherwise, it saves the values computed by the combinational logic into internal registers.

The combinational logic of *lampFPU_fractSqrt.sv* implements a Finite State Machine with four different states:

1. Every times it starts, we compute the various values required by the algorithm, but we don't save them until we're in the right state.

2. In the IDLE state, we initialize our internal variables as described in Section 2, only if the external input *doSqrt_i* signals to start the computation and we're not in one of the special cases (Square Root of NaN, Square Root of Zero...). To note, we declare the number 3 as `17'b11000000000000000`; since our R occupies 16 bits and is between 1.0 and 1.999969482421875, in order to represent the number 3 we need to add an additional bit.

3. SQRT_B is the output state; if R_i is 1, we stop the computation, return the result (either the square root or the inverse square root) and signal the end by setting *valid_next* as 1. If the exponent of the bfloat16 we want the square root of is odd, we first multiply X_i by $\sqrt{2}$ and Y_i by $\frac{1}{\sqrt{2}}$. Goldschmidt's algorithm doesn't yeld a precise LSB, but we return x_tmp or y_tmp, which have more precision than our final result. We delegate the rounding to *lampFPU_top.sv*.
   On the other hand, if the estimation of the square root is not good enough, we perform another iteration of the algorithm, by updating *b_next* and moving to the next state.

4. In the SQRT_R state, we update R_i as requested by the algorithm.

5. In the SQRT_XY state, we update X_i and Y_i as requested by the algorithm and move back to SQRT_B.

3.2 lampFPU_sqrt.sv

We now describe the lampFPU_sqrt.sv module.
In general, *lampFPU_sqrt.sv* module instantiates *lampFPU_fractSqrt.sv*, computes the final exponent and manages some special cases.
The square root is a particular operation, since it can't produce neither an

overflow nor an underflow. They occur when positive numbers exceed the maximum value or negative numbers exceed the maximum negative value that can be represented. For the square root, we have that:

- If $X > 1$, then $1 < \sqrt{X} < X$
- If $0 < X < 1$, then $0 < X < \sqrt{X} < 1$
- If $X < 0$, then $\sqrt{X}$ is NaN

This means that if X is a number that can be represented in bfloat16, its square root can always be represented in bfloat16. For this reason, *lampFPU_sqrt.sv* doesn't need to manage overflow or underflow cases.

lampFPU_sqrt.sv takes as inputs the operand we want the square root of, already divided in signum, exponent and mantissa. Some flags states whether the input is NaN, Zero or Infinity. The module returns the square root of the inputs, still split in signum, exponent and mantissa, as well as a flag signaling if the result needs to be rounded.

The sequential logic of *lampFPU_sqrt.sv* resets to zero the module's internal wires and the outputs when the input *rst* is 1. Otherwise, it saves some of the inputs into internal registers and updates the outputs.

The combinational logic of *lampFPU_sqrt.sv* firstly checks if we're trying to compute the Square Root or Inverse Square Root of a special case, using the function *FUNC_calcInfNanZeroResSqrt*. In this case, the output doesn't need to be rounded. We have the following:

- $\sqrt{+0} = +0$
- $\sqrt{-0} = -0$
- $\sqrt{NaN} = NaN$
- $\sqrt{+\infty} = +\infty$
- $\sqrt{-\infty} = NaN$
- $\sqrt{-X} = NaN$ (X is any positive number)
- $\frac{1}{\sqrt{+0}} = NaN$

- $\frac{1}{\sqrt{-0}} = NaN$
- $\frac{1}{\sqrt{NaN}} = NaN$
- $\frac{1}{\sqrt{+\infty}} = +0$
- $\frac{1}{\sqrt{-\infty}} = NaN$
- $\frac{1}{\sqrt{-X}} = NaN$ (X is any positive number)

If we are not in a special case, we compute the exponent and wait until the internal module computes the square root or inverse square root of the Mantissa (we uses *srm_valid* for it) and then we return the final value as outputs. To note, the signal *f_res_o* has a dimension of 12 bits, that are:

- The MSB is the overflow bit; as discussed before, this bit will always be zero;
- Bits from 10^{th} to 3^{rd} represent the "real" mantissa, as expected by bfloat16 standard;
- The Ground bit is the bit immediately on the right of the Mantissa LSB, so it's the 2^{nd} bit of *f_res_o*;
- The Round bit is on the right of the Ground bit, so it's the 1^{st} bit of *f_res_o*;
- The Sticky bit is computed as the logical OR of 3 bits of *f_initial* (which has 16 bits of precision), and is the LSB (the 0^{th} bit) of *f_res_o*.

These additional bits are used for rounding, which is delegated to the top module; we inform the top module with the signal *isToRound*. The rounding is performed using the function *FUNC_rndToNearestEven* which implements the following rules:

- If GRS = 00X, then round down
- If GRS = 01X, then round up if sticky bit is 1, round down otherwise
- If GRS = 10X, then round up
- If GRS = 11X, then round up and add 1

4 Experimental Results

4.1 Test Setup

In order to properly validate our implementation, we tested our design using a test-bench; this test-bench can generate random numbers, compute the results of our implementation and then compare them with the results of the C function *sqrtf* (that can be found in *math.h*).
All the other functions and operations already developed for the FPU were tested this way, so we extended two existing files:

- dpi_lampFPU.c is a C file; we added the functions *DPI_sqrt* and *DPI_invSqrt*. Given a value X, They return the value of $\sqrt{X}$ and $\frac{1}{\sqrt{X}}$, as unsigned integer representing a number in binary32.

- tb_lampFPU.sv is a SystemVerilog file; it instantiates the FPU, imports the C functions and compares their results with the ones produced by the top module.

  In order to test our implementation, we added two different tasks:

- *TASK_doSqrt_op* takes as input the number X we want the square root of, calls the right C function defined in dpi_lampFPU.c depending on the opcode, and saves the result. Then, the task rounds this result and compute $\sqrt{X}$ or $\frac{1}{\sqrt{X}}$ using our implementation. Then, the two results are compared. If they're the same, the test is passed, otherwise it fails.

- *TASK_testSqrt* creates a loop with 5000 iterations; in each one, it randomly generates values for the sign, the exponent and the mantissa, and calls *TASK_doSqrt_op*. It also manages all the special cases, explicitly creating a test for each of them.

4.2 Results

We ran our test-bench using the Vivado software, performing a behavioural, post-synthesis and post-implementation simulation. The results in all three cases are the same, and completely validate our implementation; out of 5000 tests performed, we're able to correctly compute the square root of all of them, obtaining zero rejections.
For the inverse square root, instead, we have a very small set of tests rejected

for the behavioural simulation (21 out of 5000, the 0,42%); in all these cases, our implementation returns the right exponent and sign, but has a mantissa one bit greater than expected. The issue probably lies in the rounding function, since the mantissa computed by our implementation is very close to the right value.

4.3 Algorithm Simulation

In this section we show the efficiency of the algorithm for one randomly selected value. In order to keep the description as simple as possible, we will focus only on some signals and their values.
In the following pictures, the colour red is reserved to the clock and reset signals, the light blue to the module lampFPU_sqrt.sv and the green to the module lampFPU_fractSqrt.sv

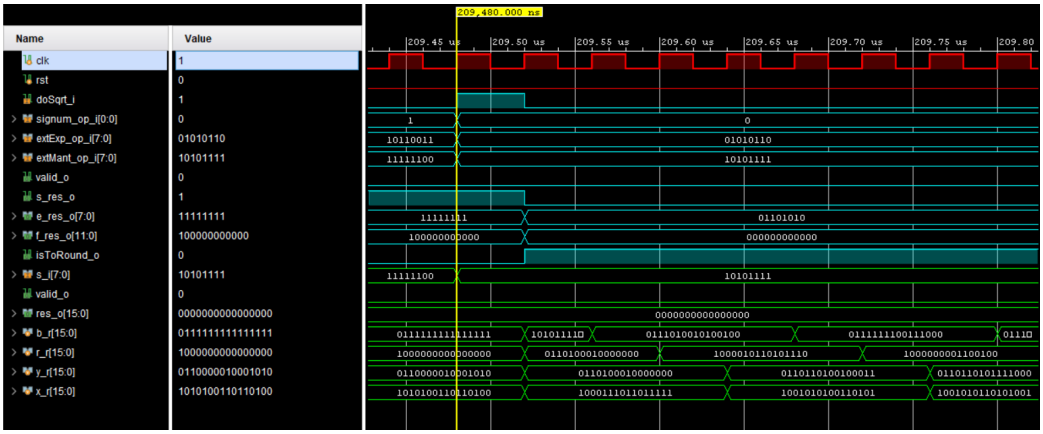

Figure 6: First part of the simulation

- The test start at time 209.48 μs, when the signal doSqrt_i is set to 1. We can see from the blue signals just under doSqrt_i that the input number is positive, has exponent 01010110 and mantissa 10101111, so is the decimal number $6.217249 * 10^{-13}$; the mantissa, considered without the exponent, is the decimal number 1.3671875. We need a clock cycle before this information is propagated to the inner module, lampFPU_fractSqrt.sv (green signals), which still has some values

left from the previous test. We remember that for each internal signal shown in this simulation (for example b_r) there is another one (b_r_next), not visualized for clarity.

- In the next clock cycle (209.52 μs), lampFPU_sqrt.sv recognizes we're not in a special case, so it computes the final exponent and sets *isToRound_o* to 1. Now, this module will wait until the inner one finishes its task.
  In lampFPU_fractSqrt.sv, all the internal registers are initialized. We refer to Section 2 for a complete description of the algorithm; since it's the first iteration, B_0 is set to the extended mantissa input and Y_0 is equal to R_0.

- In the next clock cycles we can see the evolution of the algorithm; depending on the current state, we update b_r, r_r, or y_r and x_r. To note, r_r from the first to the second cycle becomes grater than 1, but then it keeps decreasing.

- After only 10 clock cycles, r_r becomes exactly 1; we need two more clock cycles in order to propagate the computed result. We can see that:
  $$x_r = 1001010110101001$$
  However, the input exponent is odd (01010110, or -41), so we need to multiply $x_r * \sqrt{2}$. At time 209.92μs the inner module sets *valid_o* to 1 and gives us the final result, that is 1101001110100110, or 1.6535034 The Windows calculator instead computes 1.65359457.

- This result is passed to the outer module, that returns it to lampFPU_top.sv, puts the overflow bit to 0, computes the sticky bit and sets *valid_o* to 1. The 12 bits now are:
  $$f\_res\_o = 011010011101$$

- The top module now rounds the results (not shown in the diagram); using the rounding rules described in Section 3, we obtain the final value of the mantissa:
  $$M = (1)1010100 = 1.65625$$

This value is a bit greater than the real one but we have to remember that we only have 8 bits of precision; indeed, the value rounded down instead of up would be:

$$M' = (1)1010011 = 1.6484375$$

The difference between M' and the real value is greater than the difference between M and the real one.

- The test finishes and prints the results (fig.8). We can see that it passes the test.

Figure 7: Second part of the simulation

```
Test-          41
@209400000 - Start FPU operation: opcode:FPU_SQRT
OP1 - S=0 E=0x56 f=0x2f
OK DPI-FPU - S=0 E=0x6a f=0x53
OK DPIROUNDED-FPU - S=0 E=0x6a f=0x54
OK RTL-FPU - S=0 E=0x6a f=0x54
```

Figure 8: Final results of the simulation

4.4 Timing Results

As shown in the algorithm simulation, we're able to properly compute, round and manage our results in only 12 clock cycles. We firstly used the frequency already set in the test-bench module (one clock cycle every 40 ns), but then we tried to lower this value in order to check how fast our modules could operate. We found out that our implementation can still work using a way smaller clock frequency (clock period of 6 ns, more than 6 times faster).

5 Conclusions

In this document, we presented our implementation for the square root and inverse square root operations for bfloat16 numbers. We introduced the problem setting and the mathematical background, and then we presented our implementation. We finally validated our work using a test-bench, and confronting our results with the ones provided by the square root implementation of the C language. The results confirmed the correctness of our code, and also its successful integration with the already developed modules.

Therefore, as a conclusion, we affirm to have reached all the goals for the project.