bfloat SQRT module documentation

Roberto Rocco, 920721

Leonardo Stagliano, 917310

1. Introduction

This document explains the structure of the sqrt module for the bfloat fpu. The document is structured as follows: Section 2 will explain the algorithm and will show some useful properties that have been exploited during the implementation, Section 3 will analyze in detail the implementation and integration steps, Section 4 will show the result of the tests performed on the module.

2. Algorithm analysis

Floating point numbers are represented following the standard IEEE 754 [1]. The standard specifies 3 fields (sign, exponent, mantissa) and their value concerning the number that must be represented. The formula $x = (-1)^S * (1 + M) * 2^E$ explains how most numbers are mapped into the standard representation.

Given the above representation, it is possible analyze the impact of the square root operation over the fields: $\sqrt{x} = \sqrt{(-1)^S} * \sqrt{(1+M)} * \sqrt{2^E} = \sqrt{(-1)^S} * \sqrt{(1+M)} * 2^{E/2}$. From this formulation few observations can be obtained:

- If $S = 1$ there is no solution (it is impossible to obtain square root of a negative number in $\mathbb{R}$);
- If $S = 0$ it is possible to compute the new values of the fields as follows:

$$\begin{cases} S' = S = 0; \\ M' = \sqrt{(1+M)} - 1; \\ E' = E/2; \end{cases} \tag{1}$$

- Since E' must be an integer number as E, it's mandatory to consider two separate cases and distinguish whether E is odd or even.
  - If even, then $E = 2k$ for some integer k, so $E' = k$ without further problems;
  - If odd, then $E = 2k - 1$ for some integer k. It is then possible to represent the initial number x as $x = (1 + M) * 2^{2k} * 2^{-1} = \frac{(1+M)}{2} * 2^{2k}$, so its square root value becomes $\sqrt{x} = \sqrt{\frac{(1+M)}{2}} * \sqrt{2^{2k}} = \sqrt{\frac{(1+M)}{2}} * 2^k$.
- From all these considerations it is possible to compute the correct values:

$$\begin{cases} NaN & \text{if } S \text{ is 1;} \\ \begin{cases} S' = S = 0; \\ M' = \sqrt{(1+M)} - 1; \\ E' = E/2; \end{cases} & \text{if } E \text{ is even and } S \text{ is 0;} \\ \begin{cases} S' = S = 0; \\ M' = \sqrt{\frac{(1+M)}{2}} - 1; \\ E' = \frac{(E-1)}{2}; \end{cases} & \text{if } E \text{ is odd and } S \text{ is 0;} \end{cases} \tag{2}$$

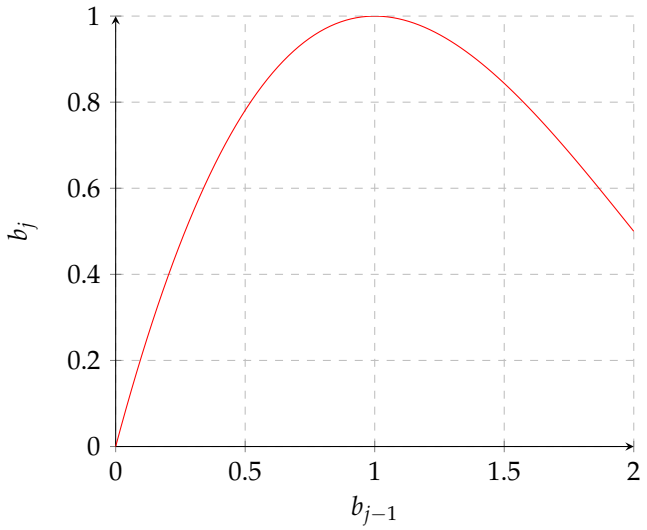

Figure 1: Study on the function b_j.

It's easy to see from the above formula that the only non-trivial field is M'. To compute it, it is possible to exploit the Goldschmidt's algorithm, following the approach proposed in [2]. The algorithm consists in these steps:

- The initial values are
$$b_0 = v; g_0 = v \tag{3}$$
where v represents the value under the square root sign (either $1 + M$ or its half);
- At each iteration j, compute y_j as an estimation of $\frac{1}{\sqrt{b_{j-1}}}$; then
$$b_j = b_{j-1} * {y_j}^2; g_j = g_{j-1} * y_j \tag{4}$$
- Repeat until b_j is equal (or at least close) to 1, g_j will contain the needed square root.

The effort proposed the use of the tangent to the curve in the point with $b_j = 1$ as a way to obtain y_j. This way the value of y_j becomes equal to:
$$y_j = \frac{3 - b_{j-1}}{2} \tag{5}$$

To analyze the precision of the approximation, it's mandatory to define the domain of b_j. In particular, it's possible to demonstrate the following statement:

Statement 1. *Using as approximation formula 5,* $0.5 \leq b_j < 2 \; \forall j$

Proof. To demonstrate the statement it is possible to proceed by induction. When $j = 0$, $b_j = v$ and v is either $1 + M$ or its half. M is a fractional value, so $0 \leq M < 1$. Given this, it's easy to assert that $0.5 \leq v < 2$. To demonstrate that b_j is in the range given that b_{j-1} is within, it's necessary to analyze their correlation. Following the algorithm, it is possible to compute that $b_j = b_{j-1} * \left(\frac{3-b_{j-1}}{2}\right)^2$. This function in the interval [0.5; 2) has values smaller than 1 and bigger than 0.5, as can be seen in figure 1. □

Once the domain is obtained, it's possible to evaluate the approximation by comparing it with the correct value. Figure 2 plots the two values in the domain. As can be seen in the graph, the approximation is good since the distance between the two curves in the domain is small.

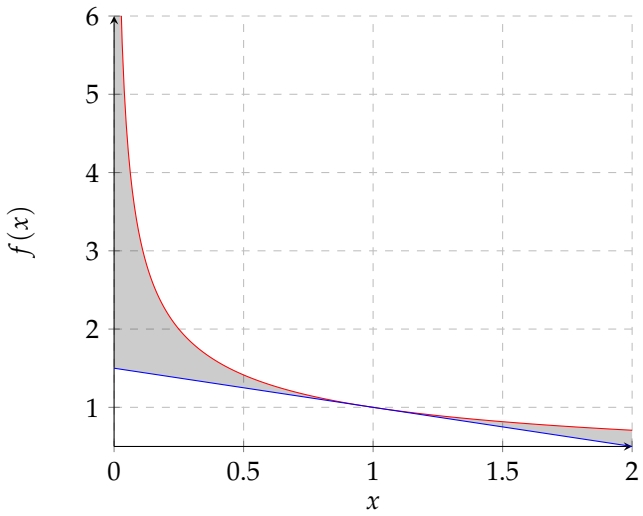

Figure 2: Comparison between first approximation and real value.

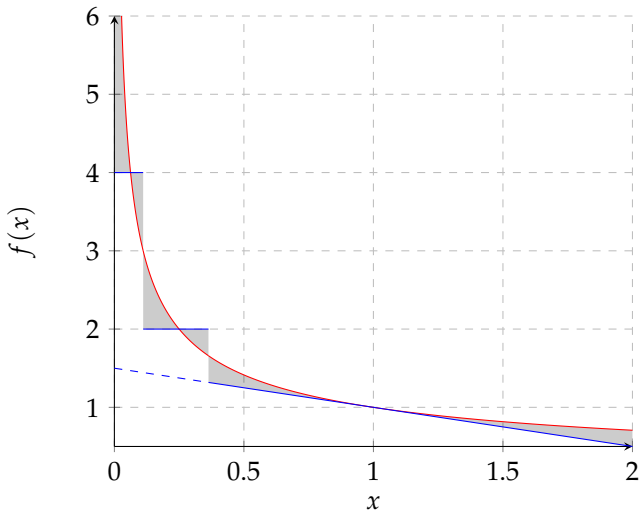

Figure 3: Comparison between second approximation and real value.

2.1. Special cases

The IEEE 754 standard specifies also the representation of values that cannot be obtained in the previous way. It includes infinites, zeros, NaN and denormalized numbers. The first three can be threated without relevant problems (only positive values are considered, since negative ones will become NaN):

- $\sqrt{0} = 0$;
- $\sqrt{\infty} = \infty$;
- $\sqrt{NaN} = NaN$.

The case of denormalized numbers is trickier: those numbers are mapped following the formula $x = (-1)^S * M * 2^{-126}$. From that, it's easy to come up to a formulation similar to the one of Formula 2:

$$\begin{cases} NaN & \text{if } S \text{ is } 1; \\ \begin{cases} S' = S = 0; \\ M' = \sqrt{M}; \\ E' = E/2 = -63; \end{cases} & \text{if } S \text{ is } 0; \end{cases} \tag{6}$$

It shall be noted, however, that the result of the operation produces a number that is no more denormalized: some additional operations may be needed to make it coherent with the normal representation. The procedure for the calculation of the square root is similar except for the starting domain: rather than going from 0.5 to 2 excluded, it goes from 0 excluded to 1 excluded. Statement 1 can be adapted to include those points, since the demonstration would be the same. It becomes:

Statement 2. *Using as approximation formula 5,* $0 < b_j < 2 \; \forall j$

The new domain introduces problems with the current approximation chosen: in the interval (0; 0.5) the error is relevant. A bad approximation would lead to the result after more iterations, so it would slow down the component. A better approximation is needed, at least for some part of the domain. A good approximation

function must present a low error and must be simple to compute: complex operations shall be avoided since they may require a lot of time to complete and would slow down the component.

Given all the above requirements, and given that the function 5 solves the problem in the domain [0.5;2), it's natural to use it together with some others that would handle the problematic part. The solution adopted uses the following function:

$$y_i = \begin{cases} 4 & \text{if } b_{j-1} \leq \frac{1}{9} \\ 2 & \text{if } \frac{1}{9} < b_{j-1} < 0.36325 \\ \frac{3-b_{j-1}}{2} & \text{if } b_{j-1} \geq 0.36325 \end{cases} \tag{7}$$

It shall be remarked that the values $\frac{1}{9}$ and 0.36325 are chosen to reduce to the minimum the distance between the function and its approximation. Domain analysis must be done again since the approximation function is different.

Statement 3. *Using as approximation formula 7,* $0 < b_j < 2 \; \forall j$

Proof. It's still possible to proceed by induction. The base case is the same as Statement 2. Proving that b_j is within the range if b_{j-1} is part of it can be done splitting the three cases:

- if $b_{j-1} \geq 0.36325$, the situation is analogous to Statement 2;
- if $\frac{1}{9} < b_{j-1} < 0.36325$, then $b_j = 4b_{j-1}$. Since $b_{j-1} > 0$, $b_j > b_{j-1} > 0$. The maximum value b_j can reach is $4 * 0.36325 = 1.453 < 2$;
- in a similar way, if $\frac{1}{9} \geq b_{j-1}$, then $b_j = 16b_{j-1}$. The maximum value b_j can reach is $16 * \frac{1}{9} = 1.\overline{7} < 2$.

□

It's possible to plot this new approximation function and compare it with the other. This has been done in Figure 3, where it can be seen that the grey area (corresponding to the error) is smaller compared with the one in Figure 2. Moreover, the constant values are very fast to use and the impact on the component is small.

2.2. Inverse calculation

The algorithm proposed in [2] allows also the calculation of the inverse of the result. To obtain such a result, the algorithm must be slightly changed to introduce another variable. The new variable is called i_j[1]:

$$\begin{cases} i_0 = 1 \\ i_j = i_{j-1} * y_j \end{cases} \tag{8}$$

The needed value will be contained in i_j when the algorithm ends. This calculation doesn't change the previous considerations on the approximations.

3. Implementation

In the following section, all the implementation details of the above algorithms will be analyzed, including how it was possible to introduce some mathematical approximations that have a meaningful performance impact. The first part will provide some details on the bfloat representation and will be followed by a punctual exposition of the implementation of the algorithm and some additional notions on truncation and integration.

[1]It shall be noted that in [2] it is named y_j and the variable that here has its name is called Y_j. It has been decided to change the name to avoid confusion between the two.

3.1. Data representation

The operand of the module is stored and elaborated through the `bfloat16` representation which is based on the IEEE 754 standard on 32 bits as described in [1]. The modification concerning the usual representation on 32 bits regards mainly the number of bits in which each field is codified, `bfloat16` numbers are represented as follows:

- S (sign): 1 bit
- E (exponent): 8 bits
- M (mantissa): 7 bits

Since the number of bits for the exponent is the same as in the original IEEE 754 specification over 32 bits, all the considerations about the bias of such field of the number remain valid.
One important detail about this representation is that the 7 bits of the mantissa represent only the fractional part of the number: to obtain the real mantissa, the value shall be preceded by either an integer 1 or 0, depending on the normalization of the number. The module operates on the entire number so the fields that are provided in the input have the very same dimensions of the representation above, except for the mantissa which has 8 bits to include the implicit integer digit.

Firstly it is useful to analyze the module interface:

```
input                                   clk,
input                                   rst,
input logic                             doSqrt_i,
input logic [(7+1)-1:0]                 extF_op1_i,
input logic [8-1:0]                     e_op1_i,
input logic                             s_op1_i,
input logic                             isZ_op1_i,
input logic                             isInf_op1_i,
input logic                             isSNaN_op1_i,
input logic                             isQNaN_op1_i,
input logic                             isOpInv_i,

output logic                            s_res_o,
output logic [8-1:0]                    e_res_o,
output logic [(7+2+3)-1:0]              f_res_o,
output logic                            valid_o,
output logic                            isToRound_o
```

A brief explanation of the functionality of each input is provided in the following list:

- `clk`: the clock signal;
- `rst`: the reset signal;
- `doSqrt_i`: the signal that indicate whether the inputs are stable and the calculation can start;
- `extF_op1_i`: the signal which contains the implicit bit and the mantissa of the operand;
- `e_op1_i`: the signal which contains the exponent of the operand;
- `s_op1_i`: the signal which contains the sign of the operand;
- `isZ_op1_i`, `isInf_op1_i`, `isNaN_op1_i`, `isQNaN_op1_i`: signaling bits to handle exceptional cases;
- `isOpInv_i`: signaling bit to toggle the calculation of the inverse square root;

The following list does the same for the output:

- `s_res_o`: the output sign

- `e_res_o`: the output exponent
- `f_res_o`: the output fractional number, which includes: the mantissa (7 bits), the implicit integer digit (1 bit) and the bits for rounding (3 bits)
- `valid_o`: a signaling bit for stable output availability
- `isToRound_o`: a signaling bit that is low only in the exceptional cases (Zero, Inf, NaN)

During the computation, the operands that contribute at the calculation of the resulting (extended) mantissa may need additional bits to perform a correct rounding in the top module. The following line of code shows an example of SystemVerilog logic declaration:

```
logic [(7+1+3)-1:0]           b_r, b_next;
```

As can be seen above, the b register has 7 bits (the same number of bits that composes the mantissa of the input), plus 1 for the integer digit and other three bits (guard, sticky and round bits) for rounding purposes. All the other registers dimensions can be explained in an analogous way, with the exception of some that may contain results of multiplications: in those cases, the dimension may be multiplied by a factor correlated to the number of operands involved. The exception is showed in the below example:

```
logic [2*(7+1+3)-1:0]         y_square_r, y_square_next;
```

The `y_square_r` register contains the result of the multiplication of 2 numbers that are contained into register of the same dimension of `b_r`, so it's size will be double with respect to `b_r`.

3.2. Algorithm realization

To better understand the implementation, it is useful to first explain the meaning of the registers and wires later mentioned. Those refer to the mathematical counterparts that are present in [2] and explored in Section 2. To save partial results across the iterations, the storing feature of the Verilog registers is exploited.

- `ss_r`: the state register;
- `b_r, y_r, g_r`: they correspond to the b, y, g operands;
- `i_r`: used to store the partial mantissa of the inverse square root of the input operand;
- `s_r`: stores the sign of the result (immediately calculated);
- `e_r`: stores the exponent of the result (immediately calculated);
- `iteration_r`: stores the number of times the Goldschmidt algorithm is applied;
- `i_ib_r`: it is used to handle the possible increment (across multiple iterations) of digits needed to store the integer part of the inverse;
- `isSNaN_r, isQNaN_r, isZ_r, isInf_r, isInv_r`: used to store the homonyms input signal to achieve stability of the input interface of the module;
- `g_temp_r`: a wire that is used for the partial computation of the g number. It allows to perform a correct truncation across iterations, handling the extra bits of the product of the previous value of g and y inside its larger dimension;
- `i_temp_r`: a wire used for the mantissa computation of the inverse result, to be able to perform truncation as above;
- `y_temp_r`: a wire to perform the truncation of the y at each iteration as expressed in 5;

- `y_square_r`: as the name says, it is used to be able to to calculate the next value to be stored in the register b, whom calculation requires y squared 4;
- `lzeros_r, lzeros_inv_r`: used to perform correct shifting of the results before the output phase and consequently involved in the adjustment of the exponent of the result;
- `e_div2_r`: used to handle the adjustment of the bias of the exponent once halved by the shifting detailed below;

The algorithm has been realized using a state machine. It features four states: `IDLE`, `WORK`, `CALC`, `RESULT`. The first one will wait for the proper signal to start preparing all the registers for the computation and move to the next state, following the analysis of the inputs as in the equation 2. The sign of the operand is checked to pull up the NaN signal in case it would be negative, and if the condition doesn't hold all the other "side-cases" (Zero, NaN, Inf codifications) are checked to skip the real computation and jump directly to the state in which the result is ready. In case no exceptional cases are encountered the following steps are performed:

- The extended mantissa of the operand is stored into the b register that will also include 3 additional bits to perform truncation (later discussed in this document). It is important to notice that the operand is properly shifted to handle the case in which an odd exponent is given in input since it would lead to a fractional exponent in the result as it has to be halved.

```
b_next = (e_op1_i[0] || (e_op1_i == '0)) ? {extF_op1_i, 3'd0} : {1'd0, extF_op1_i, 2'd0};
```

  As can be seen from the code above, the shift operation is not described explicitly by the usage of the Verilog operator `»`, instead the constructor operator is used to arrange the bits in the way they have to be stored into the register to correctly represent the desired number.

- The partial exponent is calculated by shifting the original of one position to the right, the odd case is already considered because the possible implicit truncation is handled as explained above.
- The bias of the exponent is restored (since it was halved too) by adding the half of the bias to the partial result previously obtained. The final exponent is thus stored into the register.

```
e_r_next = ~(isOpInv_i) ? e_div2_r + 64 : 190 - e_div2_r;
```

  As shown above, in the "normal" square root operation the bias of the exponent is restored by adding its half plus one. In the inverse case, it is necessary not only to obtain the number with the correct bias but also its opposite.

- The `i_r` register is initialized with the string `10000000000`, which codifies 1.0 in the extended representation of the implicit 1 and the fractional part extended with the three extra bits used for truncation.
- As a consequence of what is said above, the i_ib register has to be set to 1 since the number of integers to represent the integer part of the mantissa of the result is exactly one.
- The register that keeps track of the number of iterations is initialized to 0.

In the next state, WORK, many operations useful for the following state execution are performed. Sometimes, the execution will be handled completely in this state for an iteration: this happens when y is equal to 2 or 4 following the formula 7. This special case will be analyzed first, and will be followed by the normal flow which leads to the `CALC` state.

- The first thing checked is if `y_r` is approximated as a constant or the tangent. In the first case the y register is replaced by one of the two constants: those are chosen to gain advantages in terms of computational complexity of the operations (divisions mostly), since having constants (2 and 4 in the specific case) that are power of 2 leads to the possibility of doing a shift operation instead of a multiplication or a division. This case is handled by shifting the b and g register by an appropriate number of positions that depends on the constants. The inverse case is considered by simply adding the shifting constant to the `i_bi_r` register.

```
//Y is clamped to 4
b_next = b_r << SHIFT_CONST4*2;
g_next = g_r << SHIFT_CONST4;
i_ib_next = i_ib_r + SHIFT_CONST4;
```

- The second case performs only the calculation of the new value of y and y square applying the formula 5.

```
y_temp_r = 12'b110000000000 - ({1'b0, b_r});
y_next = {y_temp_r[11:2], y_temp_r[1]};
y_square_next = {y_temp_r[11:2], y_temp_r[1]} * {y_temp_r[11:2], y_temp_r[1]};
ss_next = CALC;
```

The state CALC is responsible for the finalization of all the computations needed to compute an iteration. It's reached only if the `WORK` state used the tangent.

- The `y_square_r` register is used in combination with the current value of b to compute the value of b for the next iteration. Slicing is required, since the multiplication of those three values has three integer digits as result, but the representation requires only a single integer bit.

```
b_partial_r = b_r * y_square_r;
b_next = b_partial_r[30 -: 11];
```

  The new value of the b_r register is obtained taking 11 bits (its dimension) starting from the bit 31: this means that, since the b_partial_r register is 33 bits long, the first two bits (that represents 2 of the 3 integer digit) are discarded.

- The register that contains g is updated following the formula 4. Here the truncation to the intermediate result of the multiplication is applied.

- The remaining operations are all about the inverse calculation: the partial product of `y_r` register times the `i_r` register is computed and processed to eliminate possible zeros in the most significant bits; the `i_ib_r` register is updated accordingly; the final result is, as usual, obtained after a truncation.

```
i_temp_r = (i_r * y_r);
lzeros_inv_r = FUNC_numLeadingZeros(i_temp_r[21-:8]);
i_ib_next = i_ib_r + 1 - lzeros_inv_r;
i_temp_r = i_temp_r << lzeros_inv_r;
i_next = {i_temp_r[21 -:10], |i_temp_r[10-:STICKYANALYSIS]};
```

The next state will be selected according to few conditions. If the register `b_r` contains `10000000000` or the maximum number of iterations is reached, then the module is ready to produce its result (or has no more time to get a better one): execution will continue to the `RESULT` state. If those conditions don't apply, then computation must proceed and the next state will be `CALC` again.

The last state of the implementation is **RESULT**, which is responsible for the management of the output.

- The corner cases are checked and the result is prepared to be compliant with the BFLOAT representation. (Zero, Inf, NaN);

- In case the required operation is the inverse of the square root, it is not necessary to perform any post processing on the i register, since the possible presence of leading zeros is already handled in the previous states, so it represents the final mantissa of the result. The `i_ib_r` register is used to compute the correct exponent by adding to it the content of the `i_ib_r` register.

```
s_res_o = s_r;
e_res_o = e_r + i_ib_r - 1;
f_res_o = {1'b0, i_r};
```

- The normal square root result is prepared through a post processing that counts the number of leading zeros, shifts the mantissa properly and updates the exponent accordingly.

```
lzeros_r = FUNC_numLeadingZeros(g_r[10-:8]);
s_res_o = s_r;
e_res_o = e_r - lzeros_r;
f_res_o = {1'b0, g_r << lzeros_r};
```

The additional zero added as most significant bit in the mantissa of the result is due to the need of being compliant with the representation of the top module, as it signals a possible overflow, never present in our cases.

Execution then continues into the `IDLE` state.

3.3. Rounding and truncation

Some operations executed by our module make the result bigger than the register that will hold the value: multiplications tend to expose this behavior. To make the result fit, the truncation must be performed, but it introduces rounding error. To reduce its impact, guard, round and sticky bits have been introduced. Moreover, to allow better optimization, it is possible to configure the module to change the span over which the sticky bit is computed: the `STICKYANALYSIS` constant is used for this. A simple example of this concept can be seen in the code snippet below:

```
g_temp_r = (g_r * y_r);
g_next = {g_temp_r[20 -:10], |g_temp_r[10-:STICKYANALYSIS]};
```

`g_r` is updated storing a truncated and rounded version of the product that has as operands `g_r` itself and `y_r`. The intermediate result has 22 bits (each operand has 11 bits) in which 2 bits represent the integer part of the obtained number and the other 20 bits give its fractional part. To store a value in `g_r` we have to go back to a representation on 11 bits so we take just one integer bit (the less significant since the most one will always be 0) and the fractional bits are taken in this way: the first 10 are taken exactly from the intermediate result, the last bit is obtained by applying a bitwise-or to the next `STICKYANALYIS` bits starting from the number 10. This implies that the "extra" fractional part that results from the product is approximated considering only some of the most significant bits, the others are simply discarded.

The rounding operations on the final result will be completed in a more general module.

3.4. Integration with the current FPU

The integration of the module with the other components that make up the entire FPU can be summarized by inspecting the following Verilog instantiation taken from the top module:

```
lampFPU_sqrt
        lampFPU_sqrt0 (
            .clk                    (clk),
            .rst                    (rst),
            //  inputs
            .doSqrt_i               (doSqrt_r),
            .extF_op1_i             (extF_op1_r),
            .e_op1_i                (e_op1_r),
            .s_op1_i                (s_op1_r),
            .isZ_op1_i          (isZ_op1_r),
            .isInf_op1_i            (isInf_op1_r),
            .isSNaN_op1_i           (isSNAN_op1_r),
            .isQNaN_op1_i           (isQNAN_op1_r),
            .isOpInv_i              (isOpInv_r),
            //  outputs
            .s_res_o                (sqrt_s_res),
            .e_res_o                (sqrt_e_res),
            .f_res_o                (sqrt_f_res),
```

```
            .valid_o                (sqrt_valid),
            .isToRound_o            (sqrt_isToRound)
        );
```

- The clk and the rst signal are connected to the clock and reset port of the sqrt module.
- The input signal that triggers the functioning of the FP unit is connected to the port that starts the computation of the sqrt module.
- The `extF_op1_i` port is connected to the `extF_op1_r` register that we can find in the pre-processing part of the top module as the one that contains the extended mantissa of the original operand, the extension simply adds the implicit integer digit to the fractional part of the number (0 is denormalized, 1 otherwise).
- An analogous reasoning can be done for `e_op1_r`, `s_op1_r`, `isZ_op1_i`, `isInf_op1_i`, `isSNaN_op1_i` and `isQNaN_op1_i` as they connect the registers that contain the sign and the exponent of the operand to the corresponding ports of the module. All the registers involved are, once again, taken from the pre-processing part of the top module which analyzes the operand in input with respect to the standard representation.
- `isOpInv_r` is connected to the port of the module which states whether either the square root or the inverse square root is required. This register is set to 1 in case the Opcode given in input to the FU specifies the inverse square root operation.
- The output ports `s_res_o`, `e_res_o`, `f_res_o` are connected to the wires used in the post-processing part of the top module, this means that they bring the outputs to the rounding logic that prepares the data before the final result is given in output.
- `valid_o` port is connected to the wire that will drive the top module to effectively output the final value.
- `isToRound_o` port is connected to the signal that triggers the rounding process in case of necessity.

As it can be seen from the above explanation the integration of the module has required the introduction of the following wires:

```
logic                                   sqrt_s_res;
logic [(LAMP_FLOAT_E_DW)-1:0]           sqrt_e_res;
logic [(LAMP_FLOAT_F_DW+2+3)-1:0]       sqrt_f_res;
logic                                   sqrt_valid;
logic                                   sqrt_isToRound;
```

Their purpose is to correctly bring the result of the sqrt module to the other logics present in the top implementation.

4. Testing and evaluation

The module was integrated into the tests of the other modules. This allows to check the functioning of the module and evaluate its precision. The testing is analogous to the other modules: it generates a random floating-point number and computes its square root both with the implemented module and the fpu of the simulator (via a c file). It then compares the two results, pointing out an error if they are different. Specific analysis for the denormalized number is also performed to better analyze their impact on the correctness of the module.

Testing is exploited to optimize the value of certain design parameters. The optimization is toward the error percentage reduction: the total absence of errors cannot be achieved, since the simulator operates with 32-bit floats, while the module only uses 16-bit. The optimization must also consider the variables that can affect the precision: rounding policy is one of these.

To prove the validity of the implemented module, its error percentage must be comparable with the one of other similar operations, such as multiplication and division.

The module configurations analyzed are:

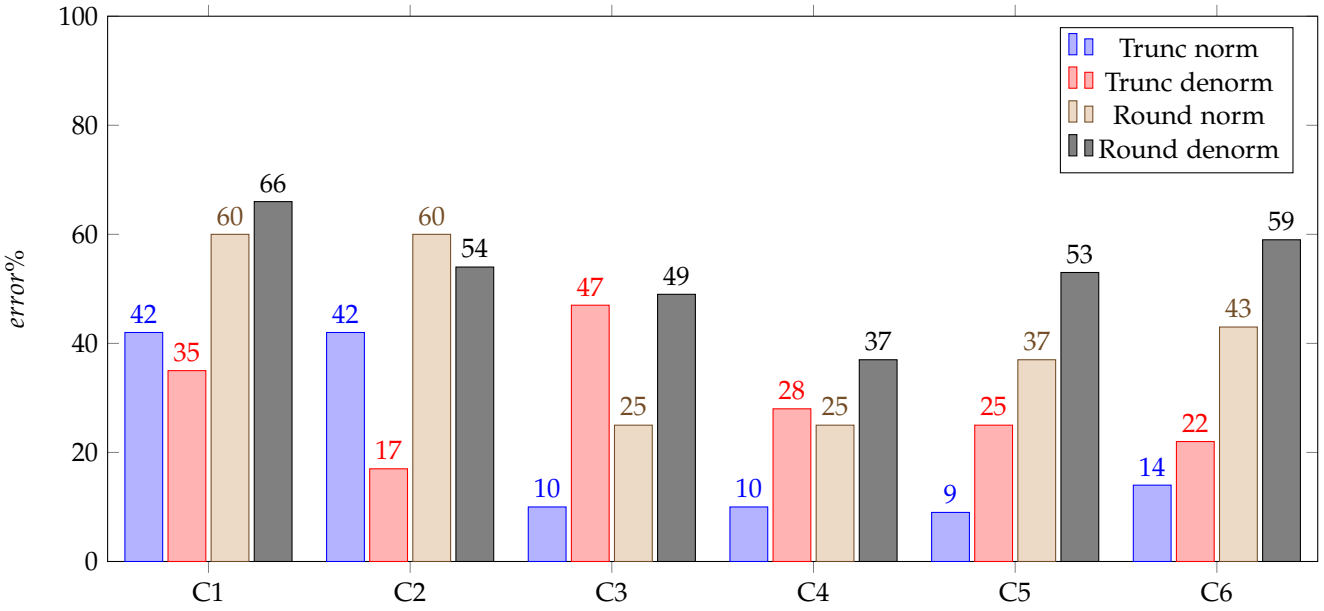

Figure 4: Comparison of the various configurations with SQRT operation.

- Configuration 1: all the exceeding bits are used for the computation of the sticky bit, approximation using formula 5;
- Configuration 2: all the exceeding bits are used for the computation of the sticky bit, approximation using formula 7;
- Configuration 3: sticky bit obtained from the truncation of the result (no more sticky), approximation using formula 5;
- Configuration 4: sticky bit obtained from the truncation of the result (no more sticky), approximation using formula 7;
- Configuration 5: first three exceeding bits are used for the computation of the sticky bit, approximation using formula 7;
- Configuration 6: first six exceeding bits are used for the computation of the sticky bit, approximation using formula 7;

Each configuration will be tested four times, analyzing both normal and denormalized numbers with both the rounding policies supported (truncation and round to nearest). Results of those tests can be seen in Figure 4 and 5.

From those figures, it's easy to see that the configurations are not all valid: the first and the second, in particular, have a too high error percentage on all the tests done on the ISQRT operation. Among the others, the fourth configuration is better than the third. The last three configurations have different results: the test on the ISQRT operation prefer the fourth configuration, the ones on SQRT depend on the rounding policy. Since neither the operation nor the rounding policy is fixed a-priori, the adopted configuration shall perform reasonably well on any situation: that's the reason why configuration 4 has been chosen for the module.

Timing analysis is crucial since it must be assured that the module can behave correctly given a certain clock frequency. The post-synthesis timing analysis of the overall FPU containing the SQRT module met all the timing constraints (10ns period clock), reporting a worst negative slack (WNS) of 0.15ns.

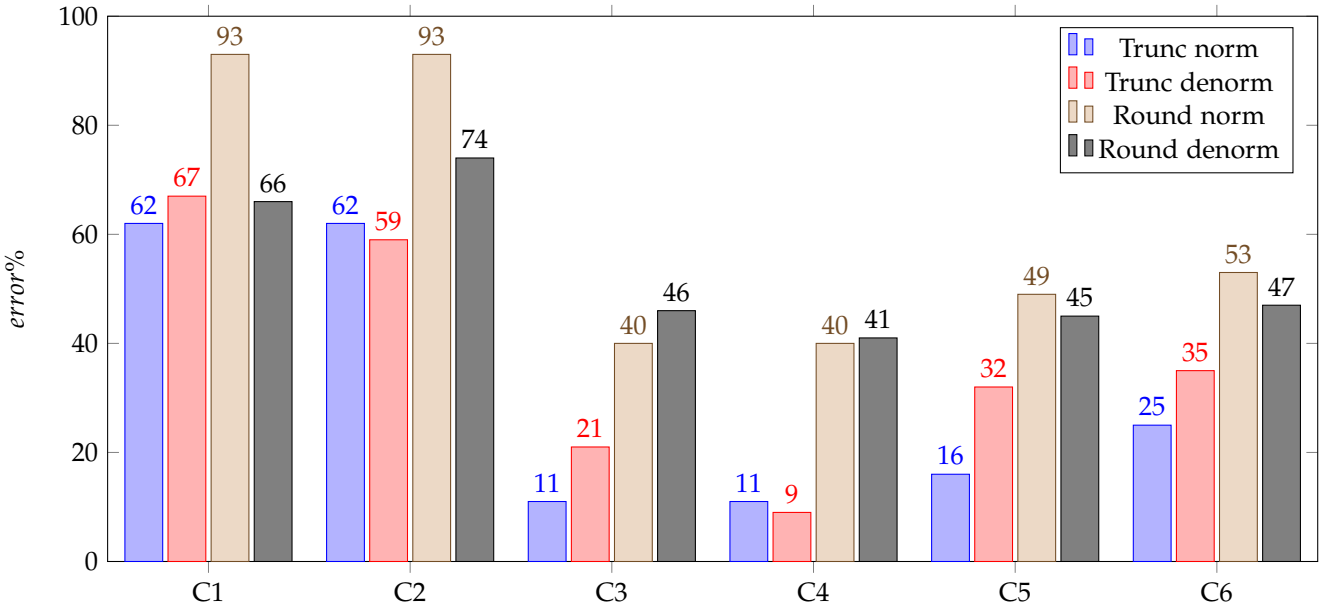

Figure 5: Comparison of the various configurations with ISQRT operation.

References

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.

[2] Markstein, P. Software division and square root using goldschmidt's algorithms. In *Proceedings of the 6th Conference on Real Numbers and Computers (RNC'6)* (2004), vol. 123, pp. 146–157.