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

February 27, 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.
In section 1 we offer a quick introduction to floating point number and bfloat16, together with the goals of our project. 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. Finally, in section 5 we offer a conclusion for our work.

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 being used, those 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)

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; 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 in order to perform their tasks.
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 number 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.

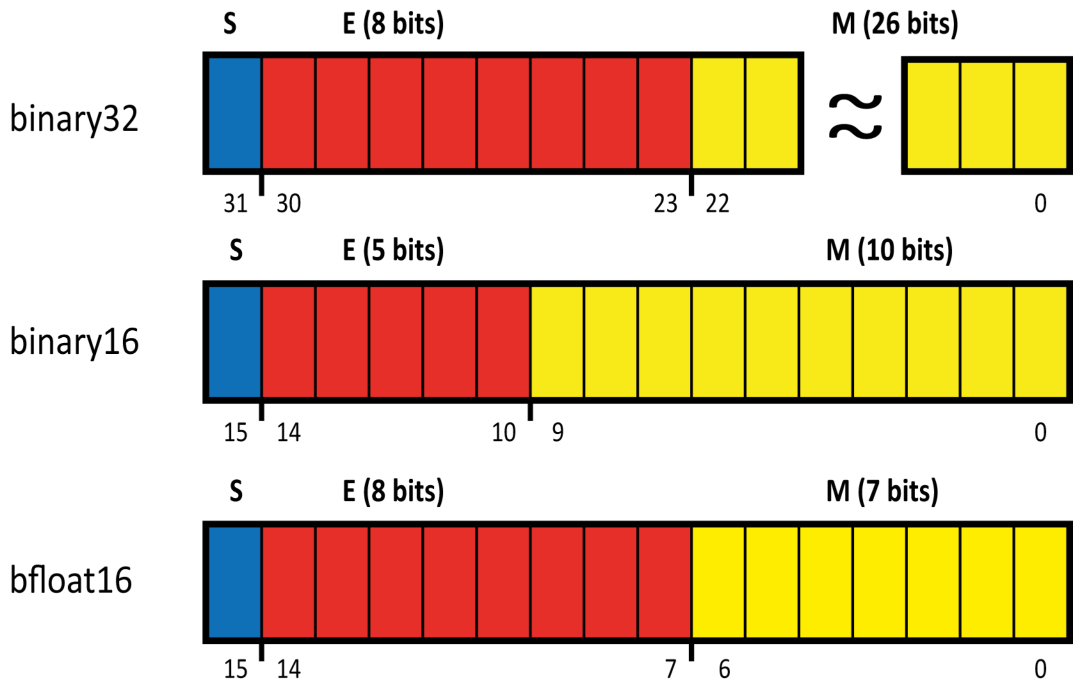


Figure 2: How the bits are assigned in the different standards.
From top to bottom: binary32, binary16, bfloat16

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
- Verify the implementation correctness, comparing the results produced by our design with the ones obtained from the C function sqrt
- Add support for the INVSQRT operation

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 follow:

- B_0 = S
- $Y_0 = R_0 = \frac{3-S}{2}$
- X_0 = 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 Reciprocal 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.

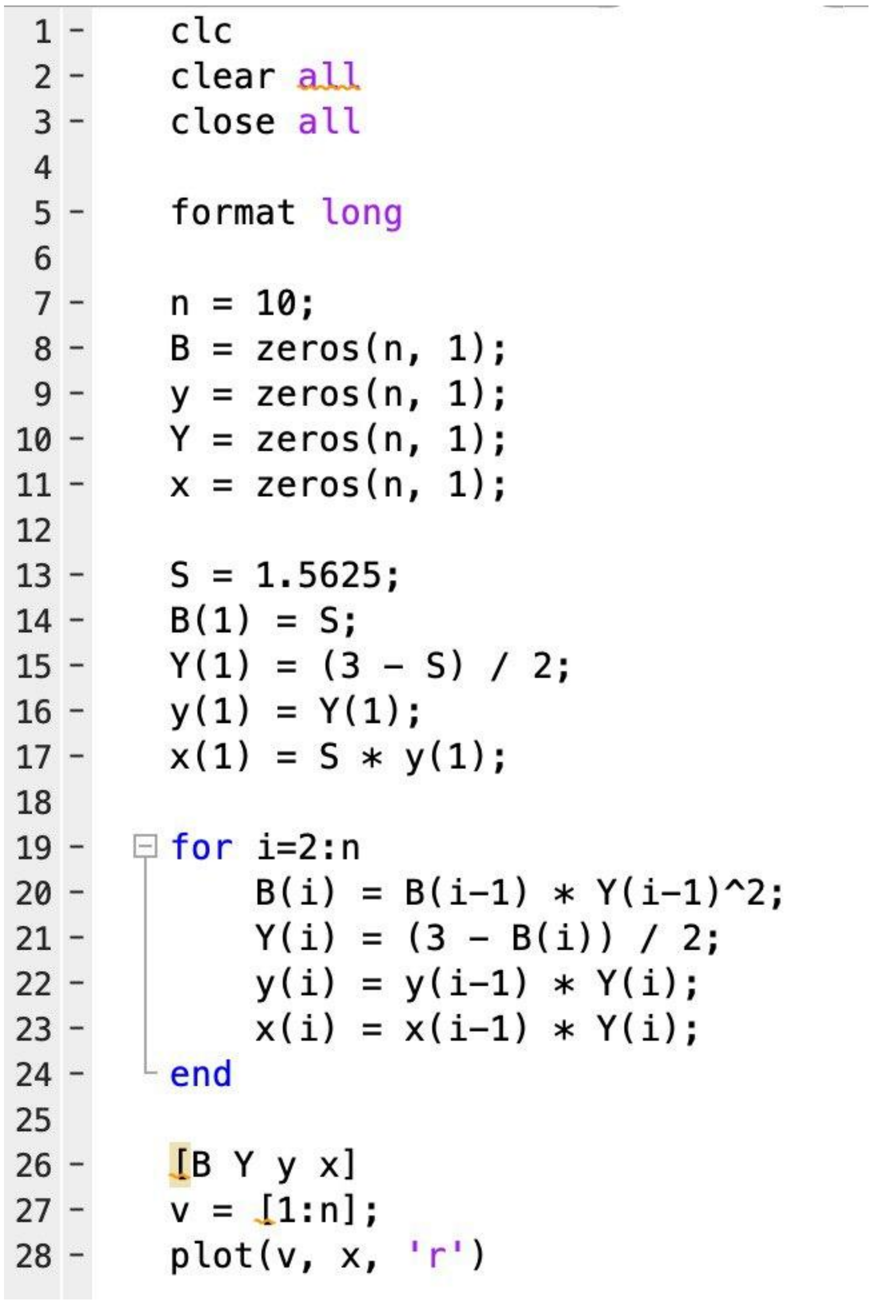
We wanted to show that the Goldschmidt's algorithm is very efficient, as it requires only a very small number of iterations to reach the desired precision. To do so, we quickly implemented the algorithm in Matlab and tracked the values of B_i, R_i, Y_i and X_i for ten iterations.
The code (3) is simply 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:

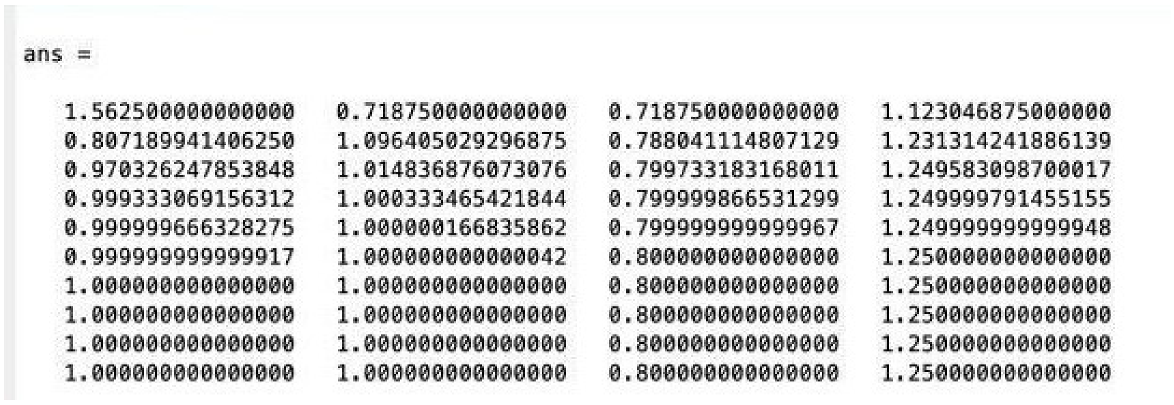
- Goldschmidt's algoritmh is not self-correcting, meaning we can't fully trust the LSB.
- The values (significant included) are in the bfloat16 format, and must be consequently managed (e.g.: the value 3 used in the algorithm is represented as 9b'110000000).
- The full square root algorithm requires also to compute the final exponent.

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



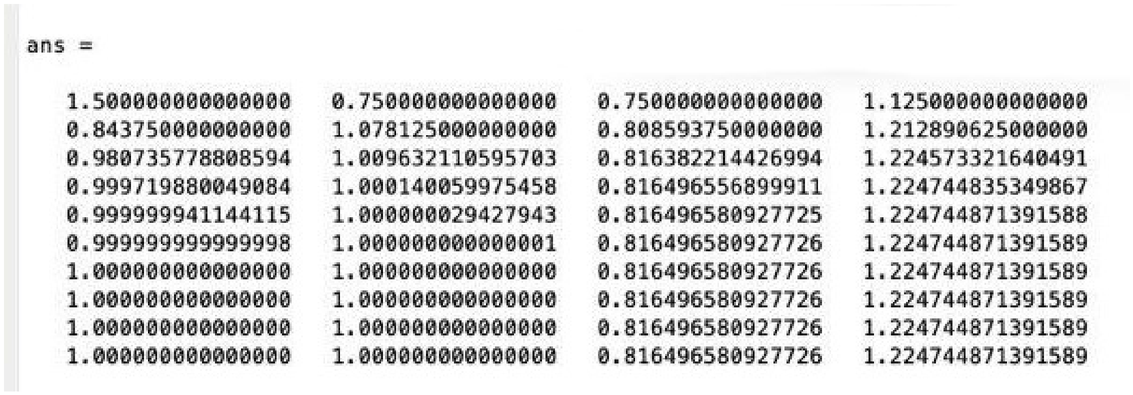
```
clc
clear all
close all

format long

n = 10;
B = zeros(n, 1);
y = zeros(n, 1);
Y = zeros(n, 1);
x = zeros(n, 1);

S = 1.5625;
B(1) = S;
Y(1) = (3 - S) / 2;
y(1) = Y(1);
x(1) = S * y(1);

for i=2:n
    B(i) = B(i-1) * Y(i-1)^2;
    Y(i) = (3 - B(i)) / 2;
    y(i) = y(i-1) * Y(i);
    x(i) = x(i-1) * Y(i);
end

[B Y y x]
v = [1:n];
plot(v, x, 'r')
```

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 expecially 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 interconnected modules:

- *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 of, manages some special cases (Square Root of NaN, Square Root of negative number, etc.) and instantiates SquareRootModule.sv
- *SquareRootModule.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'll greatly accelerate the integration process of our modules with the rest of the FPU.

3.1 SquareRootModule.sv

We firstly describe the SquareRootModule.sv module.

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

1. 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. To note, we declare the number 3 as 9'b110000000; since our Significant occupies 8 bits and is between 1.0 and 1.9921875, in order to represent the number 3 we need to add an additional bit.
2. SQRT_B is the output state; if R_i is 1, we stop the computation, return the result 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 then we took the 8 most significant bits. Since Goldschmidt's algorithm doesn't yeld a precise LSB, we round the result based on the 8th and 7th bit of x_tmp, which has 16 bits of precision. Instead, if the estimation of the square root is not good enough, we perform another iteration of the algorithm, by updating *b_next* and moving in the next state.

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

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

4 Experimental Results

5 Conclusions