Journals
The Institution of Engineering and Technology

An Efficient Digital Implementation of a Multi-Precision Square-Root Algorithm

ISSN 1751-8644
doi: 0000000000
www.ietdl.org

Alexander E. Beasley[1*], *Robert J. Watson*[1], *Christopher T. Clarke*[1]

[1] *Department of Electrical and Electronic Engineering, University of Bath, Claverton Down, Bath, UK*
** E-mail: A.E.Beasley@bath.ac.uk*

Abstract: It is essential for modern high-performance computing systems and signal processing that a basic set of mathematical functions can be performed. Addition, subtraction and multiplication are well understood but less work has been reported on square-rooting. The square-root function is particularly time and resource consuming. Traditional non-restoring algorithms produce a mantissa of half the length of the input mantissa, causing a loss of precision. This paper presents a method for increasing the accuracy of the non-restoring algorithm. The algorithm is shown to work for all IEEE-754R standard floating-point numbers. Error analysis shows an improvement in the normalised error of the algorithm of 57-fold for half-precision inputs and 134×10^6-fold for double-precision inputs. This translates to a maximum error of 1 ULP. Resource and performance optimised variants are presented and throughput is analysed. Performance optimised implementations on an Intel Stratix V device achieve throughputs of 717 MFLOPs. Resource optimised implementations on a low-cost device requires only 127 Adaptive Logic Modules and 232 registers with a throughput of 8.56 MFLOPs. All implementations are DSP block and memory free, saving valuable resources. The maximum throughput of the presented design is 15.5 times greater than that proposed by Pimentel et. al. and two orders of magnitude greater than typical multiply-accumulate methods.

1 Introduction

Modern applications: such as Computer Aided Design (CAD) and 3D graphics rendering, increasingly need support for a variety of floating-point operations. It is also becoming increasingly common to use half-precision floating-point numbers, as described by the amended IEEE-754R standard, in applications such as artificial intelligence and graphics rendering. Multiplication and addition are considered to be used more frequently whereas division and other operations are less frequent. Hence, development into optimisation for floating-point operations reflects this. Pimentel *et. al.* demonstrated an increase in throughput for floating-point operations when using hardware or hardware/software hybrid implementations [1]. They compared a range of implementations for floating-point square-root algorithms on software, hardware and a hybrid of hardware and software. A maximum throughput of 0.8 Million Floating-Point Operations per Second (MFLOPs) for software, 25.5 MFLOPs for a hardware/software hybrid and 46.2 MFLOPs for a hardware implementation was reported.

Finding the square-root of a number either requires an iterative algorithm which is costly in terms of resources and time, or alternatively, a piecewise-polynomial approximation may be used. The latter generally provides a more efficient method for processors. Common methods for square-rooting include first order algorithms (Newton-Raphson [3]), binomial expansions (Goldschmidt's [2]), or Taylor-Series expansion [2].

The Newton-Raphson approximation is widely used to find the square-root of a number. The method requires an initial approximation which must then be iterated on for increased accuracy. Cornea-Hasegan *et. al.* present a mathematical correctness proof for the Newton-Raphson method for floating-point square-root [4]. It is generally accepted that the Newton-Raphson approximation for the square-root is given by (1).

$$R_{i+1} = \frac{1}{2}\left(R_i + \frac{X}{R_i}\right) \tag{1}$$

where R_i is the approximation of the square-root, X is the input number, and R_{i+1} is the improved approximation.

Wang & Schulte present an optimised version of the Newton-Raphson method that approximates $X^{-1/2}$ [3], using (2):

$$R_{i+1} = \frac{R_i}{2}\left(3 - X.R_i^2\right) \tag{2}$$

From (1) and (2), it can be seen that the Newton-Raphson approximation is intensive on resources - requiring multiplication, division and subtraction.

Other square-root algorithms have been presented and implemented on reconfigurable logic devices. Kachhwal & Rout presented an algorithm that uses a non-standard, sub-single precision 24-bit floating-point input, and returned a 16-bit floating-point output. The algorithm uses a so-called Dwandwa Yoga method to determine the square-root of the input. The design was capable of running up to 68.22 MHz [5]. From the information presented in the paper, it can be seen that the operating frequency of the implementation is somewhere in the range of 1 to 1.25 MFLOPs.

There has been considerable research into non-restoring algorithms to find the square-root of a number in hardware implementations. Non-restoring algorithms are based on (3).

$$D = Q^2 + R \tag{3}$$

where D is the input, Q is the quotient of the operation and R is the remainder. There are several different approaches to non-restoring algorithms shown in [6, 7, 8, 9, 10]. Non-restoring algorithms get their name as they are unable to restore the remainder. The majority of these algorithms work by splitting the input into pairs of bits and operating on each pair in turn. At each iteration a partial quotient and partial remainder are created. The iteration continues until all pairs are considered and a final result is calculated. Li & Chu present an example of how a non-restoring algorithm can be implemented in both iteratively and in parallel [9]. Previous works have lacked analysis of the accuracy of the result or maximum throughputs of the designs. Putra's algorithm, proposed in [6], has been found to work for the majority of cases. However, under certain conditions the a negative is produced, which is not allowed mathematically.

Suresh *et. al.* analysed a different non-restoring algorithm that performs a series of comparisons between a register and the input number [7]. Their research focuses primarily on the resource usage when applied to a high-end Xilinx Virtex devices and custom Application Specific Integrated Circuits (ASICs).

Recent work into fast FPGA architectures for square-root and inverse square-roots is presented by Hasnat *et. al.* [11]. A methodology is presented that uses seven 'magic' numbers, found experimentally. These are integrated with a chain of seven multipliers and three adding/subtracting modules. Hasnat *et. al.* claim that the maximum accuracy achievable for the result, given single-precision floating-point with a 23-bit mantissa, is 12 bits.

This paper proposes a method for significantly increasing the accuracy of the result of a traditional non-restoring square-root algorithm. The use of non-restoring algorithms provides an implementation that is free of both Digital Signal Processing (DSP) and memory blocks, unlike offerings found in [12, 13]. DSP blocks can be found on many modern FPGA devices. However, they are a valuable resource that might be better used elsewhere in a design. The use of DSP blocks can also increase design area and reduce throughput. Methods proposed in this paper produce a square-root block that has an issue rate of one (a new answer on every clock cycle) and have an accuracy of one Unit of Least Precision (ULP). The algorithm is based on the one proposed by Putra in [6], applying an addition to remove the problems caused by negative remainders. Implementations of the new algorithm are subjected to both resource and performance optimisations. Resource, performance and error metrics for the algorithm and its implementation are shown. Comparisons are made between the error of a traditional non-restoring algorithm and the proposed algorithm. The paper demonstrates the new algorithm is suitable for half-, single- and double-precision floating-point numbers, as per the IEEE-754R standard [14].

The remainder of this paper is arranged as follows: Section 2 presents the theory of traditional non-restoring algorithms and how the new algorithm is derived and integrated into the floating-point number system. Section 3 discusses implementation of the algorithm in hardware. Section 4 presents the implementation performance metrics and compares the accuracy of the non-restoring algorithms. Finally the paper concludes in section 5.

2 Theory

Non-restoring algorithms, such as those proposed by Putra in [6] and Yamin & Wanming in [10], are usually used to calculate the square-root of a fixed-point binary number. The algorithm is summarised by the following steps:

1. Initialise all variables
2. Split the radicand into pairs of bits - prepend a 0 if the radicand is an odd number of bits
3. Start processing with the Most Significant Bit (MSB) pair
4. Perform comparisons with the partial remainder and the partial factor to determine how to set the partial factor, partial remainder and quotient for the next iteration
5. Repeat stage 4 until the entire number has been processed

By de-constructing floating-point numbers, it is possible to apply this process to determine their square-root.

2.1 Floating-point square-root

A floating-point number can be represented as three parts: the sign, the exponent, and the mantissa. The true value of the floating-point number is given by (4). The bias is an offset that allows the exponent to be signed.

$$\left\{\text{Sign}, \text{Mantissa} \times \text{Base}^{\text{Exponent}+\text{Bias}}\right\} \tag{4}$$

Separating the number into its component parts allows the square-root to be calculated using a non-restoring algorithm. Removing the exponent and sign leaves the true mantissa which is a fixed-point number.

To get the exponent of the result the original exponent needs to be divided by two. If the exponent (plus the bias) of the input number is odd; the exponent and the mantissa must be manipulated as per Listing 1. In Listing 1, E is the exponent, M is the mantissa and b is the bias of the floating-point number. It is not possible to find the square-root of the number if the input exponent is odd. Dividing the odd exponent would be synonymous to attempting to place bits in 'half bit' locations. Instead by either adding or subtracting one to an odd exponent - then bit-shifting the mantissa down or up by one respectively - the new exponent can now be calculated.

```
1.  if ($E \bmod 2$ != 0)      $E = E - 1$, $M = M << 1$
                            OR $E = E + 1$, $M = M >> 1$,
    else                       $E = E$, $M = M$,
2.  Always                     $E = E - b$,
3.  if ($E \geq 0$)            $E = (E/2) + b$,
    else                       $E = (E/2) + (b - 1)$,
```

Listing 1: Method for calculating the value of the exponent of the resultant floating-point number.

Bit-shifting the mantissa up could cause a loss of data from over-flowing over the top. Additionally, the latency of the non-restoring algorithm will increase as it is a function of length of the input.

Once the new exponent and mantissa have been found they can be concatenated to create the output root number. A comparison of the sign is used to check for negative input numbers, which have complex roots. The output of the module reflects this.

2.2 Putra's non-restoring fixed-point square-root algorithm

Traditional non-restoring algorithms iterate over the input number. The number is considered as pairs of bits, each pair is considered individually. The result of each iteration gives a single bit for the output, hence there is a loss of accuracy. Non-restoring algorithms work on fixed-point binary numbers. Putra's algorithm [6] is detailed in Listing 2.

```
1.  if ($D_{WIDTH} \bmod 2$ != 0)   $D=\{0,D\}$,
    else                           $D = D$,
2.  Always                         $Q_0 = 0$, $F_0 = 0$,
3.  Always                         $t = 0$, $i = D_{WIDTH}$,
4.  Always                         $R_t = D_{i:i-1}$,
5.  iterate from $i = D_{WIDTH}$ to 0,
6.  if($(F_t << 1)|1 < R_t$)
                       $Q_{t+1} = (Q_t<<1)|1$,
                       $F_{t+1} = ((F_t + F_t[0])<<1)|1$,
    else
                       $Q_{t+1} = (Q_t<<1)|0$,
                       $F_{t+1} = ((F_t + F_t[0])<<1)|0$,
7.  Always
          $R_{t+1} = (R_t - (F_{t+1} \times F_{t+1}[0]))<<2 | D_{i-2:i-3}$,
8.  Always                         $t = t + 2$, $i = i - 2$,
9.  Repeat steps 6 to 8 until $i = 0$
```

Listing 2: Non-restoring square-root algorithm detailed by Putra.

In Listing 2, D is the input number (radicand), $D_{i:i-1}$ represents the current sub group of the input number, F_t is the partial factor, R_t is the partial remainder, Q_t is the quotient, t is the time step and i is the bit index. The algorithm is repeated until entire number is parsed at which time the outputs will satisfy (3).

2.3 Proposed amendments to the fixed-point square-root algorithm

Whilst the resultant values of Q and R conform to the relationship $D = Q^2 + R$, under certain conditions a negative remainder is calculated. In order to conform to strict mathematical rules a negative remainder is not allowed. The result of a negative remainder is a larger error in the value of Q. A number of additional steps are proposed by Listing 3. Applying these proposed additional steps to the algorithm prevents calculating negative remainder and increases the accuracy of the algorithm. The proposed amendments are placed between steps 6 and 7 of the original algorithm (Listing 2).

```
      ...
6.    $if((F_t << 1)|1 < R_t)$,
6a.        $if((F_t << 1)|1 > R_t)$
                $Q_{t+1} = (Q_t{<<}1)|0$,
                $F_{t+1} = ((F_t + F_t[0]){<<}1)|0$,
           else
                $Q_{t+1} = (Q_t{<<}1)|1$,
                $F_{t+1} = ((F_t + F_t[0]){<<}1)|1$,
      else
           $Q_{t+1} = (Q_t{<<}1)|0$,
           $F_{t+1} = ((F_t + F_t[0]){<<}1)|0$,
7.    Always
         $R_{t+1} = (R_t - (F_{t+1} \times F_{t+1}[0])){<<}2 \mid D_{i-2:i-3}$,
      ...
```

Listing 3: Additions to the non-restoring algorithm to prevent negative remainder values.

The steps introduced by this method maintain the key features of the non-restoring algorithms. Only very simple logic functions, such as bit-shifts, comparators and fixed-point adders, are used.

2.4 Increasing the precision of the traditional non-restoring algorithm

Floating-point numbers use the exponent to move the fixed-point mantissa up or down in order to represent as much precision about a number as possible. The mantissa is shifted until the first 'one' is at the top of the register. Consequently, the magnitude of the error from an incorrect bit in the mantissa is large.

Non-restoring algorithms introduce a large error in the result when compared to other iterative approaches, such as Newton-Raphson. This is an artefact of the bit pairing system. Each pair of bits only gives a single bit for the output, therefore the resulting mantissa has only half the number of bits calculated. The error caused from this approach can be removed.

For a mantissa of fixed length, $n + 1$ bits (where the '$+1$' is the implied bit required to make the true mantissa), the number of bits in the quotient will be $n/2$. However, if the mantissa is padded before the non-restoring algorithm is performed, then more bits of the quotient can be calculated. To achieve a quotient of the same length as the original mantissa, the mantissa must be padded until its length is $2(n + 1)$ bits. The lowest significant bits of the padded mantissa are set as zeros. This is similar to the effect of a processor performing an iterative algorithm, such as Newton-Raphson, until it has calculated a value for each bit in the mantissa of a number.

2.5 Performance optimised versus resource optimised design

The latency (τ), in clock cycles, for calculating the square-root of a floating-point number using non-restoring means is dominated by the non-restoring algorithm. Implementing the design on a highly parallelised platform, such as an FPGA, allows the exponent and mantissa calculations can be performed at the same time. Calculating the new exponent will always take the same number of clock cycles, whereas the latency for the mantissa calculations is dependant on the length of the mantissa. The number of clock cycles for the exponent calculations is far fewer than the mantissa.

The algorithm proposed in this paper was implemented as both resource and performance optimised designs. The resource optimised design reuses the same hardware to iterate over the input mantissa. The performance optimised design unrolls the calculations into a pipeline. Even though this is more resource intense, an issue rate of one is achieved - a new answer is given on every clock cycle. The speed of the design is further increased by adding pipelining stages. Pipelining stages reduce routing delay, allowing for a faster clock to be used. This gives a higher overall throughput. The formulas to calculate the latency of each design are summarised in Table 1.

Consider the resource optimised implementation. A standard non-restoring algorithm the latency is given by:

$$\tau = (n + 1) + 1 \tag{5}$$

if the length of the true mantissa is odd, otherwise:

$$\tau = (n + 1) + 2 \tag{6}$$

where $n + 1$ is the number of bits of the true mantissa.

When the accuracy of the algorithm is increased the latency is calculated by:

$$\tau = 2(n + 1) + 1 \tag{7}$$

for both an odd and even input mantissa length.

Latency calculations also consider the time taken to process the input mantissa and set up the radicand. For example, to calculate the square-root of a single-precision floating-point number, the proposed method takes 49 clock cycles and the traditional algorithm takes 26 clock cycles. The length of the true mantissa is even for single-precision floating-point.

The latency of the performance optimised designs are also influenced by the number of pipelining stages (N_p). For the traditional algorithm the latency is given by:

$$\tau = (n + 1) + \left(\frac{n}{2} \times N_p\right) \tag{8}$$

if the length of the true mantissa is odd, otherwise:

$$\tau = (n) + \left(\frac{n-1}{2} \times N_p\right) \tag{9}$$

for a true mantissa with an even number of bits.

Similarly, the latency of the performance optimised implementation with increased accuracy can be derived from (8) and (9). The latency is now given by:

$$\tau = (2(n + 1) + 1) + ((n + 1) \times N_p) \tag{10}$$

or

$$\tau = (2(n + 1)) + (n \times N_p) \tag{11}$$

for an odd or even width true mantissa respectively.

Again, applying this to a single-precision floating-point number results in a latency of 48 clock cycles for the new algorithm. This is compared to the traditional algorithm that would take 34 clock cycles, assuming there are no pipelining stages.

3 Implementation

Figure 1 shows the architecture of the square-root core. Two implementations of this core have been considered, a resource optimised and a performance optimised implementation. The resource optimised design implements the logic for the bit-select and comparators once and iterates over the entire input using this logic as a loop.

Table 1 Latency calculations for resource and performance optimised implementations of the non-restoring square-root module. Latency (τ) is given in clock cycles.

		Traditional algorithm	Improved accuracy algorithm
Resource optimised	Odd mantissa	$(n+1)+1$	$2(n+1)+1$
	Even mantissa	$(n+1)+2$	$2(n+1)+1$
Performance optimised	Odd mantissa	$(n+1)+((n/2)\times N_p)$	$(2(n+1)+1)+((n+1)\times N_p)$
	Even mantissa	$(n)+(((n-1)/2)\times N_p)$	$(2(n+1))+(n\times N_p)$

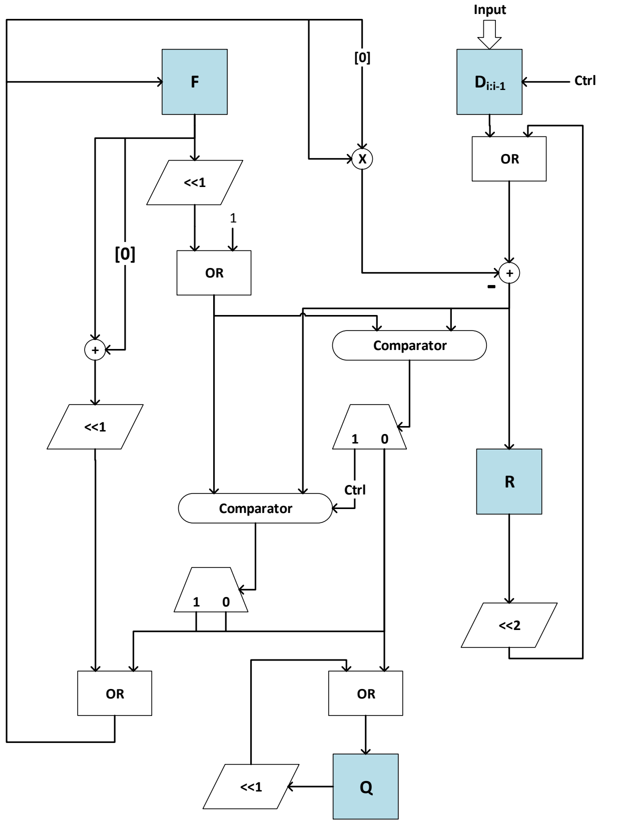

Fig. 1: *Architecture of the proposed DSP- and memory-free square-root implementation.*

Consequently the latency of this core reduces the throughput. The throughput is the clock frequency divided by the latency.

Optimising for performance implements a number of stages, one after the other. Pipelining stages can be inserted as necessary to further increase the clock speed. Since there is no reusing of logic in this implementation the core can operate with an issue rate of one. The throughput is equal to the clock frequency.

The floating-point square-root algorithm was implemented using a Hardware Description Language (HDL). The design was synthesised for both a low-cost Intel Cyclone V [15] device and a high-performance Intel Stratix V [16] device. Simulation and verification of the design was achieved with an automated test bench. The design was configured onto a DE10-Standard board from TerASIC [17] with an Intel Cyclone V 5CSXFC6D6F31C6N FPGA device for confirmation of real world operation.

The algorithm was implemented in both the resource and performance optimised versions for comparison of metrics. Both designs parallelise the processing of the exponent and the mantissa to reduce the overall latency. Details of latency calculations for different size inputs are discussed in Section 2.5.

In order to comply with IEEE-754R if the module is passed a negative number it will return a ‘QNAN’ on the output [14].

4 Results

All designs were synthesised using the Quartus II 15.0 toolkit (Intel), and timing statistics were analysed using TimeQuest. Table 2 shows a summary of the resource use and maximum operating frequency of the different designs on the different platforms for half-precision, single-precision and double-precision floating-point inputs. The metrics given are for the proposed new increased accuracy implementation of a non-restoring algorithm. Power measurements for the traditional non-restoring algorithm, the proposed new non-restoring algorithm, and Intel floating-point square-root cores were taken using the Intel PowerPlay analysis tool.

On a high-performance device, the proposed design has a throughput of 717 MFLOPs for half-, single-, and double-precision respectively, limited only by the maximum switching speed of the FPGA fabric [18 pp. 38]. The resource optimised implementation on a low-cost device required only 127 Adaptive Logic Modules (ALMs) and 232 registers, 239 ALMs and 452 registers and 456 ALMs and 901 registers for half-, single- and double-precision respectively.

The maximum throughput of this design exceeds the throughput of the designs presented in [7, 11], which used Xilinx Virtex-5 and Virtex-6 devices. Additionally, the design presented by Hasnat *et. al.* that used magic numbers required 12% of the high-end Xilinx Virtex-5 flip-flop logic. The designs presented here used only 7% of the high-end Intel Stratix V (5SGXEA7K2F40C2) device, when arranged for maximum throughput.

The addition of pipeline stages in the design increases the performance, albeit at a cost. Pipeline stages increases the number of required resources. Each pipeline stage has to register all the data moving through the pipeline. The more stages there are the more registers that need to be used. However, adding pipelining stages allow the fitter to reduce the routing delay, therefore a higher frequency clock can be used. The unrolled design produces a new result on every clock cycle. Therefore, using pipelining to increase the clock frequency increases the designs overall throughput.

Intel provide a floating-point square Megafunction capable of single-, double-, and extended-precision, (extended-precision is not included as part of IEEE-754R). The Megafunction has two latency options for both single- and double-precision. Table 3 presents the resource requirements and performance of the lowest latency options for all Intel Intellectual Property (IP). For the high-performing Intel Stratix V device the metrics are given in [19 pp. 18-7]. Metrics for the Intel Cyclone V device come from the compilation report using Quartus II 15.0. Unlike the performance optimised design, the Megafunction is unable to provide an issue rate of one. Therefore, the throughput of the Intel block is significantly lower than that of the proposed design.

The resources for the Intel Megafunction and the proposed algorithm can be compared. Table 2 shows that the single-precision resource optimised design requires additional resources compared to the equivalent Intel Megafunction. However, the double-precision Intel implementation for the Cyclone V device requires more resources than the proposed double-precision resource optimised implementation. The proposed performance optimised designs have a larger resource cost than the Megafunctions, however the throughput of the proposed algorithm is greater than the Megafunction’s by two orders of magnitude. Additionally, the Megafunction for the Stratix V device uses DSPs, whereas the proposed algorithm is a

Table 2 Resource usage (number of logic modules, ALMs and registers) and performance statistics for the pipelined and resource shared designs synthesised on both a Cyclone V and a Stratix V device

		Cyclone V		Stratix V	
		Resource Optimised	Performance Optimised	Resource Optimised	Performance Optimised
Half-Precision	$f_{\max}$ (MHz)	214	236.52	456.83	717
	Throughput (MFLOPs)	8.56	236.52	18.27	717
	ALMs	127	728	126	734
	Registers	232	2,465	230	2,417
Single-Precision	$f_{\max}$ (MHz)	178.25	194.1	717	717
	Throughput (MFLOPs)	3.50	194.1	28.68	717
	ALMs	239	3,585	240	3,510
	Registers	452	11,564	436	11,389
Double-Precision	$f_{\max}$ (MHz)	133.92	124.29	717	717
	Throughput (MFLOPs)	1.25	124.29	6.7	717
	ALMs	456	16,902	460	16,975
	Registers	901	58,404	909	58,099

Table 3 Resource usage (number of logic modules, ALMs and registers) and performance statistics for the Intel Megafunction square-root IP core.

		Single-Precision	Double-Precision
Cyclone V	$f_{\max}$ (MHz)	135.94	104.88
	Throughput (MFLOPs)	8.50	3.50
	ALMs	192	888
	Registers	396	1783
	DSP	0	0
	Latency	16	30
Stratix V	$f_{\max}$ (MHz)	393.7	274.12
	Throughput (MFLOPs)	65	16
	ALMs	112	458
	Registers	136	1060
	DSP	2	9
	Latency	6	17

DSP-free implementation. Using DSP blocks increase the used physical area of a design. Additionally, DSP blocks can increase the system latency or reduce $f_{\max}$, the maximum clocking frequency of a design. The absence of DSP blocks is a key characteristic for the proposed design.

Multiplicative square-root algorithms are another approach for determining the square-root of a number. These can be implemented effectively using FPGAs, as shown in [12]. The multiplicative square-root algorithm presented in [12] was synthesised for Xilinx Virtex-4 and Virtex-5 devices, requiring a number of DSP blocks and BRAM. Similarly, the multiply-accumulate methods described in [13] rely on DSP blocks integrated within the FPGA. The square-root method if [13] has latencies of 35 and 59 clock cycles for single- and double-precision respectively. Whereas the performance optimised design presented in this paper has an issue rate of one, therefore achieving a throughput that is two orders of magnitude greater.

Automation allowed extensive testing of the system. Initial tests swept the input to the module over a large range in fixed increments. An analysis of the error of the results was used to confirm correct operation of the square-root function. Tests were also performed to provide an exhaustive search of the possible input values. A Linear Feedback Shift-Register (LFSR) pseudorandomly generated input numbers, hence limiting the potential for the module to appear working when only given sequential data. The tap points for the LFSR were given as per the table found in [20]. The error of the traditional and proposed non-restoring square-root algorithms was compared. The normalised error was calculated by comparing the output of the hardware, S_h, with the result calculated using an Intel Core i7-7700HQ processor, S_p, performing a square-root operation in double-precision. The error (ϵ) is given by (12).

$$\epsilon = \frac{S_p - S_h}{S_p} \tag{12}$$

Relative error is also calculated in terms of ULPs (Units of Least Precision). A ULP is the difference in the floating-point results given by both a processor and the FPGA in terms of the significance of the lowest significant bit of the mantissa for a given exponent.

Plots for relative and normalised error can be seen in Figures 2 and 3 for the traditional algorithm and the improved accuracy algorithm respectively. The solid line shows the relative error (ULPs) and the dashed line shows the normalised error. The data has been split into bins and the maximum error of each bin has been plotted.

The error for the original non-restoring algorithm, with the proposed solution for negative remainders, is large. For the half-precision FPGA implementation, shown in Figure 2a, there is a worst case normalised error of 0.058; single-precision is 4.88×10^{-4}; and a double-precision is 2.98×10^{-8}.

Increasing the precision has increased the number of bits calculated in the mantissa, hence there is a better accuracy for higher precision numbers. The mantissa of a floating-point number is bound between zero and one, and the contents is adjusted based on the exponent to get the represented number. Due to the construction of a floating-point number, the error in the output of the square-root module has a cyclic nature. This can be observed in the error plots. For a given exponent, the contents of the mantissa is cycled through to get all possible representable values. This happens for all exponent values, giving rise to the cyclic nature of the error in the plots.

The relative error is expressed in ULPs. A half-precision number produces a maximum relative error of 64 ULPs, single-precision of 4,096 ULPs and double-precision of 134,217,726 ULPs.

Increasing the accuracy of the square-root significantly reduces the maximum error. Figure 3 shows the half-precision system will now generate a maximum normalised error of only 0.001 compared to the previous 0.058, this is a 57-fold improvement. Similarly a 4,094-fold improvement for single-precision and a 134×10^6-fold improvement for double-precision have been observed. The maximum relative error (ULPs) for each implementation (half-, single-, and double-precision) is now only one ULP. This complies the IEEE-754R standard for accuracy of floating-point operations [14]. The effect of the proposed method for increasing the accuracy of the algorithm is synonymous to a processor using an iterative algorithm until all bits of the mantissa are calculated.

The improvements in accuracy have come at a cost. Table 4 shows the resource uses and performance of the algorithm before and after the accuracy improvements were made when targeting a Cyclone V device. The improvements require additional resources

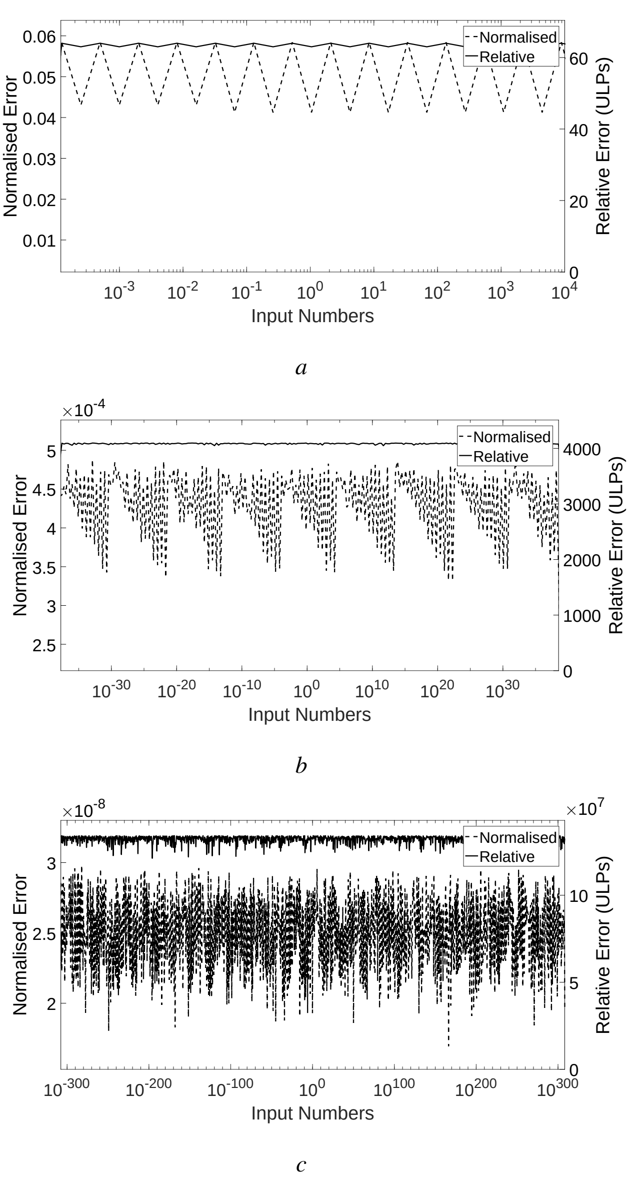

Fig. 2: Normalised error in the result from the non-restoring algorithm compared to the result a processor calculating the square-root of double-precision floats. Normalised error (dashed line). Relative Error in ULPs (solid line).
a Half-precision floating point (normalised error better than 0.058)
b Single-precision floating point (normalised error better than 4.88×10^{-4})
c Double-precision floating point (normalised error better than 2.98×10^{-8})

to account for the additional processing and increased size of the registers for the quotient, partial factor and intermediate values. For the resource optimised implementations this increase is less than twice the original resource count. Performance optimisation requires approximately three times the number of resources, most of which are used to implement the pipeline stages. Without pipelining, the increase in resources used would be closer to the difference seen by the resource optimised design.

Increasing the accuracy has caused a reduction in throughput for the same implementation at a lower accuracy; particularly in the case of a resource optimised design. This is due to the increase in required iterations to perform all calculations for the extra bits of the mantissa. The throughput for the performance optimised implementations still has an issue rate of one.

Table 5 shows the power metrics for floating-point square-root implementations. In the Table, numbers in parenthesis represent the percentage change in power consumption for the new algorithm compared to the traditional algorithm. From Table 5 it can be seen that the power consumption of the non-restoring algorithms is lower than the Intel Megafunctions for the same floating-point precision. In general, increasing the precision of the non-restoring algorithm or adding pipelining resources has a negligible impact on the power consumption of the FPGA (less than 0.1%). The change in the number of resources used can result in a reduction in power consumption due to how the fitter can now pack the hardware. The only exception is the half-precision implementation of the increased accuracy non-restoring algorithm without pipelining resources; this is due to the widths of the registers causing an slight increase in routing complexity in order to map onto the technology.

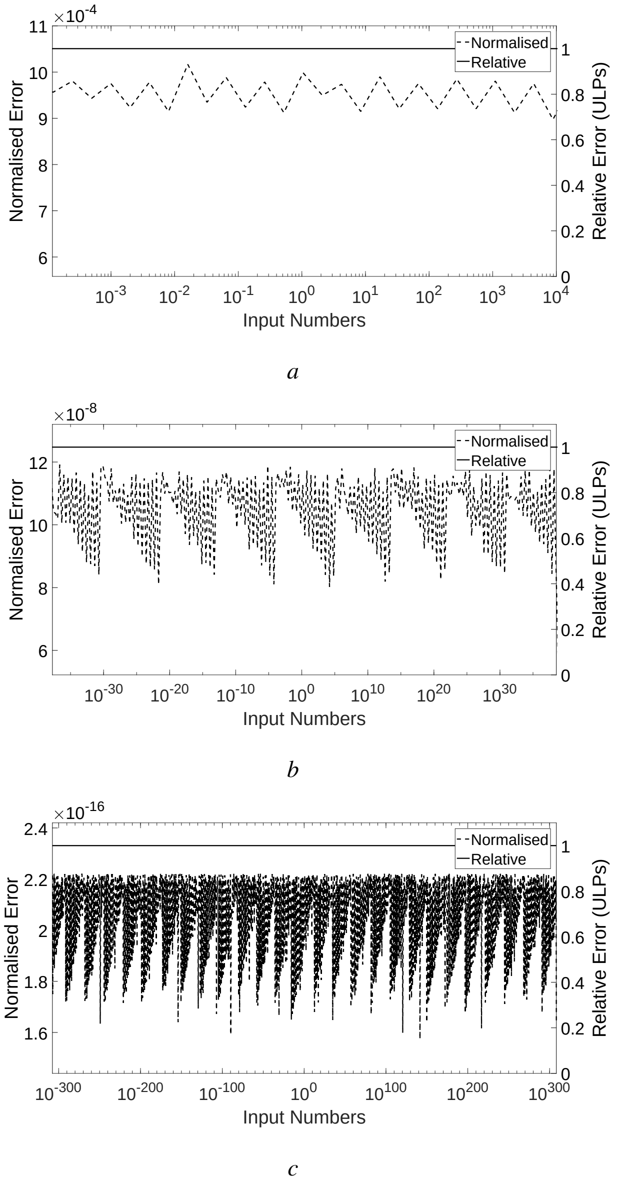

Fig. 3: Normalised error in the result from the non-restoring algorithm after a method to improve the accuracy was implemented compared to the result a processor calculating the square-root of double-precision floats. Normalised error (dashed line). Relative Error in ULPs (solid line).
a Half-precision floating-point (normalised error better than 0.001)
b Single-precision floating-point (normalised error better than 1.19×10^{-7})
c Double-precision floating-point (normalised error better than 2.22×10^{-16})

Table 4 Resource usage (number of logic modules, ALMs and registers) and performance comparisons for the algorithm before and after the improvements in accuracy were made when implemented on a Cyclone V device

		Before Improvement		After Improvement	
		Resource	Performance	Resource	Performance
Half-Precision	$f_{\max}$ (MHz)	290.36	407.5	214	236.52
	Throughput (MFLOPs)	20.74	407.5	8.56	236.52
	ALMs	80	251	127	728
	Registers	153	836	232	2465
Single-Precision	$f_{\max}$ (MHz)	207.7	260.28	178.25	194.1
	Throughput (MFLOPs)	7.42	260.28	3.50	194.1
	ALMs	162	1,143	239	3,585
	Registers	309	3,769	452	11,564
Double-Precision	$f_{\max}$ (MHz)	180.67	185.91	133.92	124.29
	Throughput (MFLOPs)	3.23	185.91	1.25	124.29
	ALMs	318	5,248	456	16,902
	Registers	583	18,416	901	58,404

Table 5 PowerPlay (Quartus) power metrics for floating-point square-root FPGA implementations, % relative to the traditional algorithm.

		Power (mW)	
		Cyclone V	Stratix V
Half-Precision	Traditional Algorithm	421.02	1,032.32
	Pipelined Traditional Algorithm	421.02	1,032.58
	New Algorithm	526.15 (25.0%)	1,173.59 (13.7%)
	Pipelined New Algorithm	421.02 (0%)	1,032.31 (-0.002%)
	Intel Megafunction	-	-
Single-Precision	Traditional Algorithm	422.11	1032.85
	Pipelined Traditional Algorithm	422.23	1032.85
	New Algorithm	422.06 (-0.04%)	1032.84 (-0.001%)
	Pipelined New Algorithm	422.07 (-0.04%)	1032.84 (-0.001%)
	Intel Megafunction	453.41	1,233.06
Double-Precision	Traditional Algorithm	424.60	1,033.91
	Pipelined Traditional Algorithm	424.62	1,033.91
	New Algorithm	424.60 (0%)	1,033.90 (-0.001%)
	Pipelined New Algorithm	424.61 (-0.002%)	1,033.90 (-0.001%)
	Intel Megafunction	554.72	1,488.24

5 Conclusions

An algorithm for calculating the square-root of multi-precision floating-point numbers with increased accuracy over traditional non-restoring algorithms, such as those presented by Putra [6] and Li [10], has been presented and extensively tested. The latency of the algorithm is fixed and predictable based on the length of the mantissa of the input number.

The performance and resource metrics for the traditional non-restoring algorithm and the improved accuracy algorithm have been compared for both resource and performance optimised implementations. Increasing the accuracy results in a relative error of one ULP or fewer, which is what is described by IEEE-754R for floating-point operations. However, this comes at a cost of increased resources and a lower throughput.

Comparisons in the error of the FPGA implementation were made using the result from an IEEE-754R compliant processor. Half-precision has an accuracy increase of 57-fold, single-precision 4,094-fold and double-precision 134×10^6-fold. In all cases the relative error of the proposed algorithm has been reduced to a single ULP. This is ideal of any floating-point mathematical function. The increase in accuracy means the proposed algorithm is now comparable to other approaches such as Newton-Raphson approximation performed by a processor.

Implementations for the proposed algorithm provided a throughput of up to 717 MFLOPs when implemented on a high-performance FPGA, such as the Stratix V. Alternatively, a resource optimised implementation uses only 127 ALMs and 232 registers on a low-cost FPGA, such as the Cyclone V. In both cases the system does not require any DSP blocks.

The maximum throughput achieved by the proposed algorithm for a single-precision floating-point numbers on a low cost Cyclone V device is 4.2 times greater than the reported maximum throughput for a hardware implementation and 243 times greater than the software implementation presented by Pimentel *et. al.* in [1]. Comparing the throughput achieved by a Stratix V and the implementations in [1], there is a 15.5 times increase against the hardware implementation and a 896 times increase against the software implementation.

There are a number of multiply-accumulate methods that have been compared against [12, 13]. The throughput of the proposed design is two orders of magnitude greater than the multiply-accumulate methods.

Comparisons have further been made against the resource use for the algorithm proposed by this paper and the floating-point square-root Megafunctions provided by Intel. The Intel Megafunction core for single-precision requires fewer resources in single-precision mode, the double-precision implementations are more resource intense. The Intel Megafunction for performing square-root on the high-performing Stratix V device also uses DSPs. The proposed implementation from this paper is always DSP- and memory cell-free. DSPs and memory cells are a valuable FPGA resource that can be better used elsewhere in designs. The power consumption of the proposed non-restoring implementation has negligible change on the power consumption of the FPGA compared to the traditional non-restoring algorithm. In some cases, the routing of the proposed algorithm offers a reduction in power consumption.

Functionality was verified by implementing the algorithm on an Intel Cyclone V 5CSXFC6D6F31CN.

Acknowledgments

This work was funded by EPSRC grant EP/M50645X/1. We thank Intel Corporation for kindly donating hardware platforms used in this research.

References

[1] Pimentel, J., Bohmenstiehl, B., Baas, B.: 'Hybrid hardware/software floating-point implementations for optimised area and throughput tradeoffs', IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2017, 25, (1), pp. 100-113

[2] Tack-Jun, K., Sondeen, J., Draper, J.: 'Floating-point division and square root using a Taylor-Series expansion algorithm', 50^{th} Midwest Symposium on Circuits and Systems, 2007, pp. 305-308

[3] Liang-Kai, W., Schulte, M.: 'Decimal floating-point square root using Newton-Raphson iteration', IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05), 2005, pp. 309-315

[4] Cornea-Hasegan, M. Golliver, R., Markstein, P.: 'Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms', IEEE Symposium on Computer Arithmentic, 1999, pp. 96-105

[5] Kachhwal, P., Rout, B.: 'Novel square root algorithm and its FPGA implementation', International Conference on Signal Propagation and Computer Technology (ICSPCT 2014), 2014, pp. 158-162

[6] Putra, R.: 'A novel fixed-point square root algorithm and its digital hardware design', International Conference on ICT for Smart Society, 2013, pp. 1-4

[7] Suresh, S., Beldianu, S., Ziavras, S.: 'FPGA and ASIC square root designs for high performance and power efficiency', IEEE 24^{th} International Conference on Application-Specific Systems, Architectures and Processors, 2013, pp. 269-272

[8] Vijeyakumar, K., Sumath, V., Vasakipriya, P., *et al.*: 'FPGA implementation of low power high speed square root circuits', IEEE International Conference on Computational Intelligence and Computing Research, 2012, pp. 1-5

[9] Yamin, L., Wanming, C.: 'Implementation of single precision floating point square root on FPGAs', Proc. The 5^{th} Annual IEEE Symposium on Field Programmable Custom Computing Machines, 1997, pp. 226-232

[10] Yamin, L., Wanming, C.: 'A new non-restoring square root algorithm and its VLSI implementations', Proc. International Conference on Computer Design, VLSI in Computers and Processors, 1996, pp. 538-544

[11] Hasnat, A., Bhattacharyya, T., Dey, A., *et al.*: 'A fast FPGA based architecture for computation of square root and Inverse Square Root,' 2017 Devices for Integrated Circuit (DevIC), Kalyani, 2017, pp. 383-387

[12] de Dinechin, F., Joldes, M., Pasca, B., Revy, G.: 'Multiplicative Square Root Algorithms for FPGAs,' 2010 International Conference on Field Programmable Logic and Applications, Milano, 2010, pp. 574-577

[13] Amaricai, A., Boncalo, O.: 'FPGA implementation of very high radix square root with prescaling,' 2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012), Seville, 2012, pp. 221-224

[14] 'IEEE Standard for Floating-Point Arithmetic' IEEE Std 754-2008, pp. 1-70, 2008

[15] Intel.: 'Cyclone V FPGAs & SoCs,' `https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html`, accessed: 20/04/2018

[16] Intel.: 'Stratix V FPGAs,' `https://www.altera.com/products/fpga/stratix-series/stratix-v/overview.html`, accessed: 20/04/2018

[17] TerASIC.: 'DE10-Standard,' `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English\&CategoryNo=165\&No=1081`, accessed: 20/04/2018

[18] Intel.: 'Stratix V Device Datasheet', `https://www.altera.com/en_US/pdfs/literature/hb/stratix-v/stx5_53001.pdf`, accessed: 21 June 2018

[19] Intel.: 'Floating-Point IP Cores User Guide', `https://www.altera.com/en_US/pdfs/literature/ug/archives/ug-altfp-mfug-15.0.pdf`, accessed: 25 January 2017

[20] Peterson, R.L., Ziemer, and R.E., Borth, D.E.: 'Introduction to Spread Spectrum Communications' (Prentice-Hall, Inc., 1995)