Algorithms for implementing roots, inverse and inverse roots in hardware

Erik Hertz, Niclas Thuning, Leo Bärring, Bertil Svensson, and Peter Nilsson

Abstract

In applications as in future MIMO communication systems a massive computation of complex matrix operations, such as QR decomposition, is performed. In these matrix operations, the functions roots, inverse and inverse roots are computed in large quantities. Therefore, to obtain high enough performance in such applications, efficient algorithms are highly important. Since these algorithms need to be realized in hardware it must also be ensured that they meet high requirements in terms of small chip area, low computation time and low power consumption. Power consumption is particularly important since many applications are battery powered.
For most unary functions, directly applying an approximation methodology in a straightforward way will not lead to an efficient implementation. Instead, a dedicated algorithm often has to be developed. The functions roots, inverse and inverse roots are in this category. The developed approaches are founded on working in a floating-point format. For the roots functions also a change of number base is used. These procedures not only enable simpler solutions but also increased accuracy, since the approximation algorithm is performed on a mantissa of limited range.

As a summarizing example the inverse square root is chosen. For comparison, the inverse square root is implemented using two methodologies: Harmonized Parabolic Synthesis and Newton-Raphson method. The novel methodology, Harmonized Parabolic Synthesis (HPS), is chosen since it has been demonstrated to provide very efficient approximations. The Newton-Raphson (NR) method is chosen since it is known for providing a very efficient implementation of the inverse square root. It is also commonly used in signal processing applications for computing approximations on fixed-point numbers of a limited range. Four implementations are made; HPS with 32 and 512 interpolation intervals and NR with 1 and 2 iterations. Summarizing the comparisons of the hardware performance, the implementations HPS 32, HPS 512 and NR 1 are comparable when it comes to hardware performance, while NR 2 is much worse. However, HPS 32 stands out in terms of better performance when it comes to the distribution of the error.

Index Terms
Approximation, unary functions, elementary functions, arithmetic computation, root, inverse, inverse roots, harmonized parabolic synthesis, Newton-Raphson method.

1. Introduction
For most unary functions, directly applying an approximation methodology in a straightforward way will not lead to an efficient implementation. Instead, a dedicated algorithm often has to be developed. The functions roots, inverse and inverse roots are computed in large quantity in many complex matrix operations, such as QR decomposition, why these functions are particularly interesting. In applications such as future MIMO communication systems [1] a massive computation of these functions are needed. Therefore, to obtain high enough performance in such applications, efficient algorithms are highly important. Since these algorithms need to be implemented in hardware it must also be ensured that they meet high requirements in terms of small chip area, low computation time and low power consumption. Among these, low power consumption is particularly important, since many applications are battery powered.

The remaining part of this paper is organized as follows: Section 2 describes the algorithms for the functions roots, inverse, and inverse roots; Section 3 presents the implementation of the inverse square root function using two approximation methods, the Harmonized Parabolic Synthesis method and the Newton-Raphson method; Section 4 presents the results of the implementations in terms of both physical characteristics and error characteristics, and Section 5 contains a comparative discussion of the implementations. The comparison is made with respect
to chip area, critical path delay, power consumption and error distribution; Section 6 closes the paper with conclusions.

2. Algorithms for roots, inverse, and inverse roots

For the functions roots, inverse and inverse roots, algorithms have been developed founded on using floating-point numbers. Commonly, algorithms for these functions are developed founded on using fixed-point numbers. Note that the algorithms are developed without regard to a specific approximation methodology. The format of the floating-point numbers used in the algorithms is simplified compared to the IEEE standard for floating-point arithmetic (IEEE 754). The floating-point number format is customized for hardware implementation. The floating-point number consists of a mantissa with the range from 1 to $<2$ and an exponent where the exponent is a scaling of the mantissa. By using floating-point numbers as an internal representation the computation can be divided into separate parts for the mantissa and the exponent. This will reduce the complexity since the approximation is performed on a mantissa of limited range, and the computation of the exponent is a very simple operation. In Table 1 a conversion of a fixed-point number into a floating-point number in base 2 is shown.

<table>
<thead>
<tr>
<th>Base 10</th>
<th>387</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point Base 2</td>
<td>0000001100000011</td>
</tr>
<tr>
<td>Exponent Index</td>
<td>0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1</td>
</tr>
<tr>
<td>Floating-point Base 2</td>
<td>1.100000110000000·2^8</td>
</tr>
</tbody>
</table>

As shown in Table 1, the index of the exponent is given by the most significant 1 in the fixed-point number, which is 8 in this case. In the last line of Table 1 the floating-point number is shown, in which the mantissa is scaled by the exponent.

When computing the function on the mantissa this is performed as an approximation. As shown in Fig. 1 the mantissa of the floating-point number is the input $v$ to the approximation and $z$ is the output from the approximation.

![Fig. 1. Block diagram of input and output of the approximation.](image-url)
2.1. Algorithms for computing roots

Besides that the algorithms for computing roots is founded on floating-point numbers it is also based on changing the number base of the exponent. The number base used when performing an approximation of a root depends on the order, \(d\), of the root. The base used in the approximation is \(2^d\). Although the number base is changed, the binary number base is retained in the representation of the mantissa. The algorithms computing roots can therefore only compute roots of order \(d\), where \(d\) is a natural number. The purpose of changing number base is that after computing the root the base of the exponent always will be 2, as shown in (1).

\[
d^{\frac{1}{2^d}} = 2
\]  

(1)

As shown in (2) the mantissa consists of \(d\) integer bits \(M\) and \(h\) fractional bits \(m\).

\[
M_{d-1}...M_0.m_{-1}m_{-2}...m_{-h}
\]  

(2)

The range of the mantissa will be \(1 \leq \text{mantissa} < 2^d\).

2.1.1. Algorithm for computing the square root

When computing the square root \(d = 2\), which gives that the floating-point uses an exponent with base 4. Table 2 shows a conversion from a fixed-point number in number base 2 into a floating-point number with exponent base 4 and binary number representation of the mantissa.

<table>
<thead>
<tr>
<th>Base 10</th>
<th>387</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point Base 2</td>
<td>0000000110000011</td>
</tr>
<tr>
<td>Exponent Index</td>
<td>00 00 00 01 10 00 00 11</td>
</tr>
<tr>
<td></td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Floating-point Base 4</td>
<td>01.10000011000000 \cdot 4^4</td>
</tr>
</tbody>
</table>

As shown in Table 2 when deciding the index of the exponent the fixed-point number in number base 2 is transformed into sequential pairings of number base 2 digits. In Table 2 the most significant pair of digits is found for index 4. The pair of digits for index 4 is the integer bits in the mantissa and the remaining pairs of digits are the fractional bits of the mantissa. In the last line of Table 2 the floating-point number in number base 4 is shown.

The computation to perform when computing the square root, \(z\), is shown in (3).

\[
z = \sqrt[M_{d-1}M_0.m_{-1}m_{-2}...m_{-h} \cdot 4^{\text{index}}]
\]  

(3)

In (4), (3) is simplified to get the exponent in number base 2.
As shown in (4), computing the exponent is made through a simple change of number base.

When computing the approximation of the square root, this is performed only on the range $1 \leq v < 4$, as shown in Fig. 2.

![The square root function.](image)

A summary of the algorithm for computing the square root function is shown in the block diagram in Fig. 3.

![Block diagram of the square root algorithm.](image)

As shown in Fig. 3 the starting point is a floating-point number in number base 4. The exponent and mantissa are computed separately. When computing the exponent the number base is changed from 4 to 2 without changing the index. When computing the mantissa it is assumed that the incoming mantissa is in the number base 4. On the incoming mantissa an approximation of the square root function is performed. After the approximation the result $z$ is in the range $1 \leq z < 2$ which is the desired range of the mantissa with the number base 2.
2.2. Algorithms for computing the inverse

The algorithms for computing the inverse is also founded on using floating-point number.

The computation to perform to compute the inverse, \( z \), is shown in (5).

\[
z = \frac{1}{M_0,m_{-1}m_{-2}m_{-3}...m_{-h},2^{\text{index}}}
\]

(5)

In (6), (5) is simplified to get the exponent in the numerator, thus with negative \( \text{index} \).

\[
z = \frac{1}{M_0,m_{-1}m_{-2}m_{-3}...m_{-h}} \cdot 2^{-\text{index}}
\]

(6)

As shown in (6), when computing the exponent only the sign of \( \text{index} \) is changed.

The approximation of the inverse is performed in the range \( 1 \leq v < 2 \), as shown in Fig. 4.

Fig. 4. The inverse function.

A summary of the algorithm for computing the inverse function is shown in the block diagram of Fig. 5.
As shown in Fig. 5 the starting point is a floating-point number. The exponent and mantissa are computed separately. When computing the exponent the sign of the index is changed and subtracted with 1 depending on the result of the approximation. When computing the mantissa the range after approximation is $0.5 < z \leq 1$. This implies that when $z < 1$ a multiplication with 2 has to be performed on the mantissa.

2.3. Algorithms for computing inverse roots

The algorithm for computing inverse roots is combining the algorithms for roots and inverse described in Section 2.1 and 2.2.

The computation to perform to compute the inverse, $z$, is shown in (7).

$$z = \frac{1}{d[M_{d-1} \cdots M_0.m_{-1}m_{-2} \cdots m_{-h}] \cdot 2^{d \cdot \text{index}}} \quad (7)$$

As shown in (7) the inverse root $d$ is performed on a floating-point number with the number base $2^d$ and in binary number representation. As shown in (7) the mantissa consist of $d$ integer bits $M$ and fractional bits $m$ where $h$ is the number of bits used in the fractional part.

In (8), (7) is simplified to get the exponent in the numerator, thus in base 2 with negative index.

$$z = \frac{1}{d[M_{d-1} \cdots M_0.m_{-1}m_{-2} \cdots m_{-h}] \cdot 2^{-d \cdot \text{index}}} \quad (8)$$

As shown in (8), when computing the exponent only the sign of index is changed. Since the integer part of the mantissa always has to be 1 or larger the range of the mantissa will be $1 \leq \text{mantissa} < 2^d$. 

When computing the approximation of the inverse square root, this is performed only on the range $1 \leq v < 4$, as shown in Fig. 6.

![Graph showing the inverse square root function](image.png)

Fig. 6. The inverse square root function.

A summary of the algorithm for computing the inverse square root function is shown in the block diagram of Fig. 7.

![Block diagram of the inverse square root algorithm](image2.png)

Fig. 7. Block diagram of the inverse square root algorithm.

As shown in Fig. 7 the starting point is a floating-point number. Also as shown in Fig. 7 the exponent and mantissa are computed separately. When computing the mantissa the range after approximation is $0.5 < z \leq 1$. A special case is when $z = 1$, since then the mantissa in the appropriate format. In most cases, however, when $z < 1$ a multiplication with 2 has to be performed to get the mantissa in the appropriate format. When computing the exponent, initially the sign of the index is changed. Depending on the result when computing the approximation of the mantissa, when $z = 1$ the exponent remains untouched whereas when $z < 1$ the exponent is subtracted with 1.
3. Implementations of the inverse square root

As a summarizing example of the three algorithms described the implementation of the inverse square root is chosen. It is chosen since it includes both the square root and the inverse described in this paper. For comparison the inverse square root is implemented using two methodologies: Harmonized Parabolic Synthesis [2] and Newton-Raphson method [3] [4]. The novel methodology, Harmonized Parabolic Synthesis is chosen since it has been demonstrated to provide very efficient approximations. The Newton-Raphson method is chosen since it is known for providing a very efficient implementation of the inverse square root. It is also commonly used in signal processing applications for computing approximations on fixed-point numbers of a limited range. To facilitate the correct evaluation of the two methodologies the distribution of the error for both is set to be a normal distribution.

When implementing the inverse square root algorithm, only an approximation on the mantissa of the floating-point number is implemented. This is namely the only part that is different for the two implementations. The approximation is performed in the interval from 1.0 to nearly 4.0. The input to the approximation is a 15 bit fixed-point number and the output is also a fixed-point number with a target accuracy of 15 bits and with a normal distributed error. A bit-accurate MatLab model has been developed for simulations of the approximations and when for analysis of the performance of the approximations.

The implementations are performed as an ASIC with a ST Microelectronics 65nm General Purpose Standard-\(V_T\) (1.1V) technology. For all implementations Synopsys Design Compiler [5], Synopsys Primetime [6] and Mentor Graphics Modelsim [7] are used.

3.1. Implementation using the harmonized parabolic synthesis methodology

The Harmonized Parabolic Synthesis methodology is founded on multiplication of two factors, both in the form of second-order parabolic functions, called the first sub-function, \(s_1(x)\), and the second sub-function, \(s_2(x)\). When recombined, as shown in (9), they approximate the original function \(f_{org}(x)\).

\[
f_{org}(x) \approx s_1(x) \cdot s_2(x)
\]  

(9)

The first sub-function, \(s_1(x)\), is shown in (10).

\[
s_1(x) = x + c_1 \cdot (x - x^2)
\]  

(10)

In (10), the coefficient \(c_1\) determines the amplitude of the non-linear part. The second sub-function, \(s_2(x)\), is developed as a second-degree interpolation, as shown in (11), where the number of intervals in the interpolation decides the order of the accuracy of the approximation.

\[
s_2(x) = l_{2,i} + j_{2,i} \cdot x_w - c_{2,i} \cdot x_w^2
\]  

(11)
In (11), the coefficient $l_{2,i}$ is the starting point of an interval, coefficient $j_{2,i}$ is the sum of the coefficients of the linear and non-linear parts and $c_{2,i}$ is the coefficient of the non-linear part. To simplify the coding of the interval number, $i$, in hardware, the number of equal-range intervals in the second sub-function are chosen as $2^w$, where $w$ is a natural number.

To facilitate the hardware implementation of the inverse square root, a normalization to satisfy the criterion that the values are in the interval $0 \leq x < 1.0$ on the $x$-axis and $0 \leq y < 1.0$ on the $y$-axis has to be performed. As shown in Fig. 8 the inverse square root function, $f(v)$, is in the interval $1.0 \leq v < 4.0$ on the $v$-axis and $1 \leq y < 0.5$ on the $f(v)$-axis. This means that a transformation on the function $f(v)$ has to be performed to generate the normalized function, $f_{org}(x)$, as shown in Fig. 8.

![Fig. 8. The function $f(v)$ before normalization and the normalized function, $f_{org}(x)$.

To move the function $f(v)$ which is in the interval $1.0 \leq v < 4.0$ into the normalized interval on the $x$-axis $0 \leq x < 1.0$ is carried out by the substitution of $v = 3x + 1$. The developed expression of the preprocessing function is shown in (12).

$$x = \frac{v - 1}{3}$$

(12)

After the substitution the function $f(v)$ also has to be moved into the normalized interval $0 \leq y < 1.0$ on the $y$-axis. As shown in (13) initially the function has to be multiplied with $2$ and then subtracted with $1$ achieve the normalized function, $f_{org}(x)$.

$$f_{org}(x) = \frac{1}{\sqrt{3x + 1}} \cdot 2 - 1 = \frac{2}{\sqrt{3x + 1}} - 1$$

(13)

The result $y$ from the processing part is in the interval $0 \leq y < 1.0$ and the desired output $z$ from the approximation shall be in the interval $0.5 < z \leq 1.0$. A transformation of $y$ is therefore needed. As shown in (14), initially $1$ is added to $y$ which gives an offset of $1$. To get the result into the desired interval is has to be divided with $2$ as shown in (14). The developed expression of the post-processing function is shown in (14).
\[ z = \frac{y + 1}{2} \]  

(14)

As shown in Fig. 8 the original function, \( f_{\text{org}}(x) \), starts in (0.0,1.0) and ends in (1.0,0.0). \( x \) has therefore to be substituted with \( 1 - x \) as shown in (15). The result after the substitution is also shown in (15).

\[
s_1(x) = (1 - x) + c_1 \cdot ((1 - x) - (1 - x)^2) = \\
1 - x + c_1 \cdot (1 - x - (1 - 2x + x^2)) = \\
1 - x + c_1 \cdot (x - x^2)
\]  

(15)

The only modification needed of (10) is to replace \( x \) with \( 1 - x \). As shown in (15) will this only affect the linear part, while the non-linear part remains unchanged.

When proceeding in the development of the algorithm of the processing stage the coefficients in (15) and (11) have to be decided but before that the number of intervals needed in (11) also have to be decided. The foundation of the Harmonized Parabolic Synthesis methodology is to approach the development with a holistic view. This is done by developing the two sub-functions simultaneously. When developing an approximation of an original function the first and second sub-functions are looked upon as one device. The methodology is to develop the first sub-function in such a way that the product of the two sub-functions gives a good conformity to the original function. The approach when developing the first sub-function, \( s_1(x) \), is not based on independent analytical calculations since it is dependent on the performance of the second sub-function, \( s_2(x) \). Therefore, the coefficient \( c_1 \) in the first sub-function has to be determined by, for different values of the coefficients, calculating the maximum absolute error, \( f_{\text{error}}(x) \), between the approximation and the original function according to (16).

\[
f_{\text{error}}(x) = \left| s_1(x) \cdot s_2(x) - f_{\text{org}}(x) \right|
\]  

(16)

To perform the calculation of the absolute error, \( f_{\text{error}}(x) \), the second sub-function, \( s_2(x) \), has to be made dependent on the coefficient \( c_1 \) in the first sub-function, \( s_1(x) \) as shown in [2]. The calculation is interesting only as an indication of how the absolute error, \( f_{\text{error}}(x) \), depends on the coefficient \( c_1 \). When choosing the coefficient \( c_1 \) it has to be made with regard to both the characteristics of the error and the efficiency of the hardware implementation. The number of intervals in the second sub-function, \( s_2(x) \), needs to be increased to achieve the intended accuracy; this has also to be taken into account when performing the calculation of the error. In Fig. 9 the bit accuracy for the inverse square root function is shown when using 2, 4, 8, 16 and 32 intervals in the second sub-function, \( s_2(x) \) with different values of the coefficient, \( c_1 \).
Fig. 9. The bit accuracy depending on $c_1$ for 2, 4, 8, 16 and 32 intervals in the second sub-function, $s_2(x)$.

Values of the accuracy in bits for $c_1$ values less than -0.5 are not shown since the results are not defined because of divisions by numbers near 0 when performing the calculations. Based on Fig. 9, values of the coefficient $c_1$ are chosen to allow an efficient implementation of the hardware. As shown in (15), $c_1$ is fed into a multiplier; therefore choosing a value that is a power of two is desirable. In Fig. 9, the desired accuracy at 15 bit as indicated. To accomplish this, at least 32 intervals are necessary. As shown in Fig. 9, choosing $c_1 = 0.0$ for 32 intervals is interesting since this will exclude the multiplication in (15) and thereby reducing the hardware. In the implementation study presented later in this paper, we have also included a design with more than 32 intervals. With higher number of intervals the internal word lengths in the architecture can be shortened. Thereby it is possible to improve chip area and speed in a design. Table 3 shows preliminary study of chip area and critical path delay dependent of the number of intervals.

**TABLE 3:** Preliminary study of the dependency of the number of intervals in the design

<table>
<thead>
<tr>
<th>Number of Intervals</th>
<th>Chip Area</th>
<th>Critical Path Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>16300 $\mu$m$^2$</td>
<td>4.3 ns</td>
</tr>
<tr>
<td>64</td>
<td>15700 $\mu$m$^2$</td>
<td>4.2 ns</td>
</tr>
<tr>
<td>128</td>
<td>15900 $\mu$m$^2$</td>
<td>4.3 ns</td>
</tr>
<tr>
<td>256</td>
<td>17600 $\mu$m$^2$</td>
<td>4.1 ns</td>
</tr>
<tr>
<td>512</td>
<td>19500 $\mu$m$^2$</td>
<td>3.9 ns</td>
</tr>
</tbody>
</table>

Since the implementation using 512 intervals in Table 3 is the fastest and that the increase in chip area is relatively small, it was used in this survey.
3.1.1. Architecture

As a comparison, two implementations have been made, one with 32 intervals and one with 512 intervals.

Finding the optimal set of coefficients can be done as an iterative procedure starting from an initial set. After deciding the initial coefficients in the first and second sub-function the next step is to optimize the architecture. The method for the optimization is to decide the word lengths used in the architecture and to optimize the coefficients. The optimization is performed in an iterative trial and error manner. The evaluation of different coefficient values should be performed in parallel with the evaluation of the word length, since the truncation error effects have impact on the performance of calculations in the design. The strategy is to adjust coefficients and word lengths in the design for best accuracy and distribution of the error. The values of the coefficients are specified in [8].

In practice, the simulation of the approximation is performed with a bit-accurate C model and the performance of the approximation is analyzed in MatLab.

The architecture consists of the three parts, preprocessing, processing, and post-processing. The preprocessing and the post-processing are implementations of (12) and (14) whereas the processing is implemented as an approximation of (13). Pre- and postprocessing are not influenced by the number of intervals used in the processing part.

Preprocessing

The preprocessing part is implemented according to (12), as shown in Fig. 10. In Fig. 10  \( c = \frac{1}{3} \).

![Fig. 10. The preprocessing stage with word lengths annotated with names.](image)

After optimization, the word lengths of the data paths in the preprocessing are as shown in Table 4.

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>15</td>
</tr>
<tr>
<td>v2</td>
<td>15</td>
</tr>
<tr>
<td>c1</td>
<td>14</td>
</tr>
<tr>
<td>x1</td>
<td>16</td>
</tr>
</tbody>
</table>
**Processing**

The implementation of the processing part is made as a product of the first sub-function (15) and second the sub-function (11). Since the coefficient $c_1 = 0.0$ the first sub-function is reduced to $1 - x$ as shown in Fig. 11.

![Diagram](image)

**Fig. 11.** The processing stage with word lengths annotated with names.

After optimization in the way described above, the word lengths of the data paths of the implementations with 32 and 512 intervals, described in Fig. 11, are as shown in Table 5.

**TABLE 5: The word lengths in the processing stage for both 32 and 512 intervals**

<table>
<thead>
<tr>
<th>Data path</th>
<th>32 intervals</th>
<th>512 intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>xs</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>xw</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>x2</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>i1</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>c2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>cx</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>l2</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>j2</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>jx</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>j1</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>jlc</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>y1</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
Postprocessing

The postprocessing part is implemented according to (14), as shown in Fig. 12. Note that the division with 2 in (14) was replaced with one right shift. The word lengths of the data paths in Fig. 12 are as shown in Table 6.

![Diagram of postprocessing stage with word lengths annotated with names.](image)

**Fig. 12.** The postprocessing stage with word lengths annotated with names.

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>y1</td>
<td>16</td>
</tr>
<tr>
<td>z1</td>
<td>17</td>
</tr>
<tr>
<td>z2</td>
<td>18</td>
</tr>
</tbody>
</table>

**Table 6:** The word lengths in the postprocessing stage for both 32 and 512 intervals

3.2. Implementation using the Newton-Raphson method

The Newton-Raphson method is a method to find a root of an arbitrary function, \( f(y) = 0 \), numerically. Successively, through iterations, the method finds an increasingly accurate approximation. The general iteration formula for the Newton-Raphson method is shown in (17).

\[
y_i = y_{i-1} - \frac{f(y_{i-1})}{f'(y_{i-1})}
\]

(17)

In (17), \( i \) is the index for the iteration and the iteration is initiated by a presupplied guess \( y_0 \). To perform iteration the derivative in (17) must meet the condition that \( f'(y) \neq 0 \).

Initially the function to perform the approximation of the inverse square root of \( v \) is developed. In (18) the expression to the function in (19) is developed.

\[
y_{i-1} = \frac{1}{\sqrt{v}} \Leftrightarrow \sqrt{v} = \frac{1}{y_{i-1}} \Leftrightarrow v = \left(\frac{1}{y_{i-1}}\right)^2 \Leftrightarrow \frac{1}{y_{i-1}^2} - v = 0
\]

(18)

\[
f\left(\frac{1}{\sqrt{v}}\right) = \frac{1}{y_{i-1}^2} - v
\]

(19)

In (20) is the derivative (19) computed.
With (19) and (20) inserted in (17) the iteration formula is developed in (23).

\[ f'(\frac{1}{\sqrt{v}}) = \frac{2}{y_i^3 - 1} \]  

(20)

The iterations in the Newton-Raphson method are initiated by a presupplied guess of \( y_0 \). To improve the convergence, thus reducing the number of iterations, the guess should be as close as possible to the actual result. To accomplish this, a look-up table storing initial values is used. Where the addressing is done through using the operand \( v \).

The strategy used for determining the initial values stored in the look-up table is illustrated in Fig. 13.

![Fig. 13. The target function together with the strategy for the initial values.](image)

The strategy is trying to solve several problems. As shown in Fig. 13, the range width within which an initial value is valid depends on the slope of the function on which the approximation is performed. The steeper the slope the shorter the range within which an initial value is valid. This is to get a uniform convergence of the different intervals. Besides this, the initial values in the look-up table are situated both above and below the function. This accomplishes both that the convergence is improved and that the error is distributed around zero.

A drawback with this strategy is that when the input \( v = 1 \) the output \( y \neq 1 \). To solve this the initial value \( y = 1 \) is stored in the look-up table when \( v = 1 \).

**3.2.1. Architecture**

To investigate the influence of the number of iterations on an implementation, two implementations have been made, one using one iteration and one using two. Also for these implementations the desired accuracy was 15 bit. After optimization in the way described above,
it was found that the implementation using one iteration needs 95 entries and the implementation using two iterations needs 14 entries.

**One iteration**
The algorithm to implement is shown in (21), giving the architecture shown in Fig. 14. The size of the look-up table (LUT) is 95 initial values.

![Diagram](image1.png)

Fig. 14. The processing stage with word lengths annotated with names.

After optimization, the word lengths of the data paths marked in Fig. 14 are as shown in Table 7.

**TABLE 7: The word lengths in the implementation using one iteration**

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>15</td>
</tr>
<tr>
<td>l1</td>
<td>15</td>
</tr>
<tr>
<td>l2</td>
<td>16</td>
</tr>
<tr>
<td>lv</td>
<td>19</td>
</tr>
<tr>
<td>s3</td>
<td>19</td>
</tr>
<tr>
<td>d2</td>
<td>19</td>
</tr>
<tr>
<td>z1</td>
<td>18</td>
</tr>
</tbody>
</table>

**Two iterations**
The second implementation, uses two iterations of (21) and has a look-up table (LUT) with 14 initial values, as shown in Fig. 14.

![Diagram](image2.png)

Fig. 15. The processing stage with word lengths annotated with names.
After optimization, the word lengths of the data paths marked in Fig. 15 are as shown in Table 8.

### Table 8: The word lengths in the implementation using two iterations

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>15</td>
</tr>
<tr>
<td>l1</td>
<td>16</td>
</tr>
<tr>
<td>l2</td>
<td>16</td>
</tr>
<tr>
<td>lv</td>
<td>16</td>
</tr>
<tr>
<td>s3</td>
<td>16</td>
</tr>
<tr>
<td>d2</td>
<td>16</td>
</tr>
<tr>
<td>i1</td>
<td>16</td>
</tr>
<tr>
<td>i2</td>
<td>17</td>
</tr>
<tr>
<td>iv</td>
<td>17</td>
</tr>
<tr>
<td>m3</td>
<td>18</td>
</tr>
<tr>
<td>q2</td>
<td>18</td>
</tr>
<tr>
<td>z2</td>
<td>17</td>
</tr>
</tbody>
</table>

### 4. Results

In this section the results of the implementations of the inverse square root using the Harmonized Parabolic Synthesis methodology and the Newton-Raphson method are described. Four implementations are made, two following the Harmonized Parabolic Synthesis methodology using 32 and 512 intervals, respectively, and two following the Newton-Raphson method with one and two iterations, respectively. We consider only the results of the approximation of the mantissa. The characteristics and distribution of the error are compared with respect to maximum error, symmetric distribution around zero and narrow gravity center around zero of the distribution. The implementation aspects are compared with respect to chip area, critical path delay and power consumption.

#### 4.1. Characteristics and distribution of the error

The characteristics and the distribution of the error are studied in the interval from 1.0 to nearly 4.0.

##### 4.1.1. The harmonized parabolic synthesis methodology using 32 intervals

Fig. 16 shows the distribution of the error of the implementation using the Harmonized Parabolic Synthesis methodology with 32 intervals in the second sub-function.
As shown in Fig. 16, the error is evenly distributed around zero, which is confirmed by the related histogram of the distribution of the error in Fig. 17. The distribution is well centered around zero, which is an advantageous characteristic for the following calculations in an algorithm. The advantageous characteristics of the error are confirmed in Table 9 later in this section by the fact that the standard deviation and the root mean square value are equal.

4.1.2. The harmonized parabolic synthesis methodology using 512 intervals

Fig. 18 shows the distribution of the error of the implementation using the Harmonized Parabolic Synthesis methodology with 512 intervals in the second sub-function.
Again, as shown in Fig. 18 the error is evenly distributed around zero. This is also confirmed by the related histogram in Fig. 19.

Fig. 19 shows a triangle-like shape of the distribution of the error around zero. This indicates a synthesis of two rectangular error distributions, probably from the two look-up tables $l_{2,i}$ and $j_{2,i}$ in Fig. 20. The error is distributed around zero, which is an advantageous characteristic for the following calculations in an algorithm. The advantageous characteristics of the error are confirmed in Table 9 by the fact that the standard deviation and the root mean square value are equal.

4.1.3. The Newton-Raphson method using one iteration

Fig. 20 shows the distribution of the error of the implementation using the Newton-Raphson method with one iteration and with a look-up table with 95 initial values.
As shown in Fig. 20 the error tends not to be evenly distributed around zero which is confirmed by the related histogram in Fig. 21.

Fig. 21 shows that the distribution of the error has a displacement towards the right side of the histogram. The error is not sufficiently evenly distributed around zero for it not to be a disadvantage in the following calculations. This distortion is confirmed by the difference between the standard deviation and the root mean square values, given in Table 9.

4.1.4. The Newton-Raphson method using two iterations

Fig. 22 shows the distribution of the error of the implementation using the Newton-Raphson method with two iterations and with a look-up table with 14 initial values.
As shown in Fig. 22 there is a very small tendency to the error not to be evenly distributed around zero. In the related histogram in Fig. 23 however, the tendency is hard to see.

Fig. 23 shows not clearly that the distribution of the error has a displacement towards the right side of the histogram. This small distortion is however confirmed by the difference between the standard deviation and the root mean square values in Table 9. But since the maximum error is so small this will not have any influence on the following calculations.

4.1.5. Error characteristics
The error characteristics of the four designs are summarized in Table 9.
Table 9 shows some general differences between the two implementation methodologies. The mean error and median error are much smaller for the implementation using the Harmonized Parabolic Synthesis methodology. Similar there are no differences in the standard deviation and the root mean square values. The better error characteristics of the designs using the Harmonized Parabolic Synthesis methodology depends on that this methodology has greater potential in the optimization of the design depending on its structure. The Newton-Raphson method is lacking these opportunities to control the error characteristics.
4.2. Hardware performance

The implementations are made as an ASIC with a ST Microelectronics 65nm General Purpose Standard-$V_T$ (1.1V) technology. Their power consumption at 10 MHz is shown in Table 10 together with chip area and critical path delay.

<table>
<thead>
<tr>
<th>Implementation Methodology</th>
<th>Chip Area</th>
<th>Critical Path Delay</th>
<th>Energy Consumption per Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harmonized Parabolic Synthesis 32 intervals</td>
<td>13900 $\mu$m$^2$ 100%</td>
<td>4.3 ns 110%</td>
<td>0.002073 nW 129%</td>
</tr>
<tr>
<td>Harmonized Parabolic Synthesis 512 intervals</td>
<td>17100 $\mu$m$^2$ 123%</td>
<td>3.9 ns 100%</td>
<td>0.001895 nW 118%</td>
</tr>
<tr>
<td>Newton-Raphson method 1 iteration</td>
<td>15600 $\mu$m$^2$ 112%</td>
<td>4.6 ns 118%</td>
<td>0.001610 nW 100%</td>
</tr>
<tr>
<td>Newton-Raphson method 2 iterations</td>
<td>22800 $\mu$m$^2$ 164%</td>
<td>6.7 ns 172%</td>
<td>0.006378 nW 396%</td>
</tr>
</tbody>
</table>

As shown in Table 10, the Harmonized Parabolic Synthesis methodology with 32 intervals has the smallest chip area. The Harmonized Parabolic Synthesis methodology with 512 intervals has the shortest critical path delay and the energy consumption per sample is smallest for the Newton-Raphson method with one iteration. The Newton-Raphson method with two iterations is outperformed by the others in all aspects.

5. Comparing implementations

This section summarizes the comparison of the four implementations of the inverse square root function done. Two of them used the Harmonized Parabolic Synthesis methodology, and two of them used the Newton-Raphson method. With the Harmonized Parabolic Synthesis methodology the implementations use 32 (HPS 32) and 512 (HPS 512) intervals in the second sub-function. In the implementations use the Newton-Raphson method, one (NR 1) and two (NR 2) iteration stages are used.

Quantitative comparisons of different approximation methodologies should always be done in the context in which the approximations are going to be used. In the absence of this, the comparison of the implementations will therefore be made in the form of reasoning.

Chip area
As shown in Table 10 HPS 32 has the smallest chip area, while it is slightly larger for NR 1 and HPS 512 and additionally larger for NR 2.
Critical path delay
Table 10 also shows that HPS 512 has the shortest critical path delay while it is slightly larger for HPS 32 and NR 1 and, again, additionally larger for NR 2.

Energy consumption per sample
When it comes to energy consumption per sample, Table 10 shows that it is best for NR 1 while it is slightly higher for HPS 512 and HPS 32 and much higher for NR 2.

The overall comparison of the hardware gives that NR 2 is worst-performing in all categories. For HPS 32, HPS 512 and NR 1 each of them are best in one of the categories. The difference in performance between these three implementations is generally small; it is therefore difficult to appoint a winner.

Characteristics and the distribution of the error
In the Sections 4.1.1, 4.1.2 and 4.1.5 it was shown that both HPS 32 and HPS 512 have a distribution of the error that is centered around zero. It was also shown that the distribution of the error for HPS 32 is better than for HPS 512 since the standard deviation is smaller. A comparatively smaller standard deviation implies that the distribution of the error as more narrow around zero.

The Sections 4.1.3, 4.1.4 and 4.1.5 show that both NR1 and NR 2 have a distribution of the error that is not well centered around zero. This may in the following calculations result in an accumulated error which can ruin the computation. Since it is hard to center the distribution of the error around zero when designing using the Newton-Raphson method. It is common to increase the accuracy in order to reduce the effects of the accumulated error.

The small maximum error for NR 2 (according to Table 8) comes with the prize of larger chip area, longer critical path delay and higher power consumption per sample (according to Table 9).

Summary of comparisons
When making a summary of the comparisons of the hardware performance and the distribution of the error the three implementations HPS 32, HPS 512 and NR 1 are comparable when it comes to hardware performance. However, HPS 32 distinguishes itself by better performance when it comes to the distribution of the error. As mentioned initially in this section, the context in which the implementation is going to be used is decisive for selection. The different performance parameters need therefore to be weighed against each other.

6. Conclusion
In many future applications, such as in the MIMO communication systems, a massive computation of complex matrix operations needs to be performed. In these matrix operations, such as QR decomposition, the functions roots, inverse and inverse roots are computed in large quantities. For these functions an implementation in a straightforward way applying an approximation methodology will not lead to an efficient implementation. This paper introduces novel dedicated algorithm for these functions founded on the use of floating-point format and, for the roots functions, also a change of number base. Using the floating-point format enables simpler
solutions but also increased accuracy, since the approximation algorithm is performed on a mantissa of limited range. Also the change of number base when computing the roots enables simpler solutions when computing the approximation. The most important characteristics of the developed dedicated algorithms are fast computation, low power, and simple and efficient hardware implementation.

In this paper, two implementations of each of two methodologies, the Harmonized Parabolic Synthesis methodology and the Newton-Raphson methodology, are made. In the Harmonized Parabolic Synthesis methodology the accuracy is scaling with the number of intervals used in the interpolation part. In the Newton-Raphson method the accuracy is scaling with the number of guesses stored in the look-up table and the number of iterations. For each methodology, one of the two implementations was made allowing higher accuracy than the required. A deduction that could be made from this was that, with increased accuracy, the size of the hardware is growing much slower when using the Harmonized Parabolic Synthesis methodology than when using the Newton-Raphson methodology.

References