USING A FLOATING-POINT MULTIPLIER'S INTERNALS FOR HIGH-RADIX DIVISION AND SQUARE ROOT

Eric M. Schwarz
Michael J. Flynn

Technical Report CSL-TR-93-554

January 1993

The work is supported by the IBM Resident Study Program using facilities supported under NSF Contract No. MIP88-22961.

USING A FLOATING-POINT MULTIPLIER'S INTERNALS FOR HIGH-RADIX DIVISION AND SQUARE ROOT

by

Eric M. Schwarz

Michael J. Flynn

Technical Report CSL-TR-93-554

January 1993

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

Abstract

A method for obtaining high-precision approximations of high-order arithmetic operations at low-cost is presented in this study. Specifically, high-precision approximations of the reciprocal (12 bits worst case) and square root (16 bits) operations are obtained using the internal hardware of a floating-point multiplier without the use of look-up tables. The additional combinatorial logic necessary is very small due to the reuse of existing hardware. These low-cost high-precision approximations are used by iterative algorithms to perform the operations of division and square root. The method presented also applies to several other high-order arithmetic operations. Thus, high-radix algorithms for high-order arithmetic operations such as division and square root are possible at low-cost.

Key Words and Phrases: computer arithmetic, division, high-order arithmetic operations, high-radix algorithms, look-up tables, multipliers, reciprocal, starting approximations, square root

Contents

1 Introduction

High-radix algorithms for high-order arithmetic operations typically use a large look-up table implemented as a ROM or PLA to store an initial approximation. The look-up table for each operation requires a large amount of area and may need to be implemented off-chip from the arithmetic unit. Thus, large look-up tables require a large area and possibly an off-chip delay. This study proposes a substitute for the look-up table which only requires a small amount of dedicated hardware and thus easily fits on-chip with the arithmetic unit.

The proposed method expresses an approximation to a high-order operation as a partial product array (PPA). As in binary multiplication, each element of the PPA is a Boolean element, each column has an implied weight (base 2 positional weighted number system), and the elements are to be summed algebraically. The PPA is a two dimensional representation of the formulations. By representing an approximation to an auxiliary operation in this form, it is easy to directly map the array onto a multiplier. The counter tree and adder of a multiplier are reused to sum the many Boolean elements. Thus existing hardware is used to obtain high-precision approximations.

The proposed method is based on a method by Renato Stefanelli [Ste72] and enhancements by David Mandelbaum [Man90a, Man90b, Man91, MM91] and the authors of this study [SF91a, SF91b, SF92a, SF92b]. Stefanelli investigated directly back-solving a multiplication's equations to develop equations for division. The division operation can be expressed as a multiplication equation with two known variables and one unknown: $Q = R/D$, therefore $Q * D = R$, where Q the quotient is the unknown, and D the divisor and R the dividend are known. The multiplication is then expressed in the form of a PPA. Q is in a redundant notation and is chosen such that there is no carry propagation between columns of the PPA. Each column forms a separate linear equation which can be solved for a digit of the quotient. This method can be used for any operation which can be expressed as a series of multiplications and additions. Many high-order arithmetic operations are formulated in this manner: reciprocal [Ste72, Man90b, SF91a], division [Ste72, Man90a, Man90b, SF91a, SF91b], square root [Man91], log and exponential [MM91], and trigonometric operations [SF92a, SF92b]. In the present study a refined algorithm is introduced and applied to the reciprocal and square root, operations but also can be applied to any of these operations.

This study describes the general method for approximating a high-order arithmetic operation in three steps: 1)deriving a PPA, 2)implementing it on an existing design of a multiplier, and 3)including the approximation in a double-precision iterative algorithm. The first step, deriving the PPAs, is divided into two substeps. Each substep is described in algorithm form. Algorithm 1 derives a signed PPA[1] to approximate the given operation. Then algorithm 2 adapts the signed PPA onto the unsigned PPA of a multiplier. Next, the implementation is described by first detailing the PPAs and implementations of multipliers and then describing the necessary adjustments. The third step of the general method is to include the approximation into an operation specific double-precision algorithm. First, this step is detailed for the reciprocal approximation and then for the square root. Their PPAs are described, along with their accuracies, and then double precision, iterative algorithms are compared. And finally, the contributions of this study are summarized. Thus, this study derives a PPA, shows its implementation, and then describes high-precision algorithms. This study presents a low-cost method of performing high-radix division, square root, and other high-order arithmetic operations on a floating-point multiplier.

[1]A signed PPA differs from a PPA since its Boolean elements can be signed. They can be negative or positive instead of only positive(or unsigned), but all signs must be known a priori.

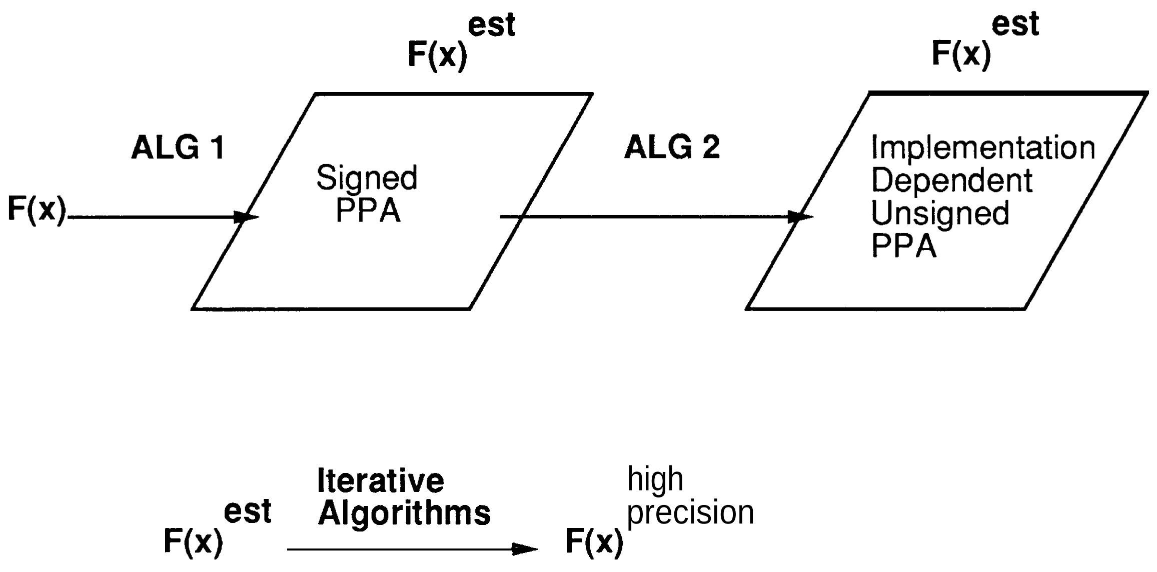


Figure 1: Overview of Derivation of PPA Describing Approximation to an Operation

2 Derivation of the PPA

This section derives a PPA which approximates a high-order arithmetic operation. An overview of this method is given in Figure 1. Given a high-order arithmetic operation, $F(x)$, a PPA is derived which describes an approximation to the operation, $F(x)^{est}$. The PPA produced is dependent on the multiplier that is to be reused. In the following section, details are given of typical multiplier implementations and the minor adjustments needed to sum the auxiliary operation's array. The approximation produced by the PPA can be used in iterative algorithms to produce a high-precision result. This is illustrated in the bottom line of Figure 1. A comparison will be given between several iterative algorithms for double precision division and square root. This paper will show how to produce the PPAs, how to implement them on a multiplier, and how these approximations fit into an overall algorithm for division and square root. This section shows the first step which is deriving a PPA to superimpose on a multiplier's PPA.

The derivation of the PPA, which describes an approximation to a high-order arithmetic operation, is accomplished by a series of steps and sizings which are iterated on to find a reasonable PPA. The problem of finding the optimal PPA which is defined to bc the PPA that produces the smallest and fastest implementation, and produces the minimum worst case error, is an NP-complete problem. Therefore heuristics are used to produce a good but not necessarily optimal PPA. The algorithms are applied for different number of bits estimated and different number of compensating elements until a suitable PPA is found. There are two algorithms that are applied to derive a PPA as is shown in the top of Figure 1. Algorithm 1 approximates a high-order arithmetic operation with equations in the form of a signed PPA. Then algorithm 2 adapts the signed PPA onto the PPA of a given multiplier. The two algorithms are used to create an array that can be placed on top of a given multiplier's array. These two algorithms are repeated with different parameters to determine the most suitable PPA. Thus, this section describes the derivation of the PPA.

2.1 Algorithm 1: Describing an Operation as a Signed PPA

The first step in deriving a PPA is to derive a signed PPA to approximate an operation. The algorithm is based on Stefanelli's method[Ste72] of describing division as the inverse of multiplication. First the operation is described as a multiplication (or a series of multiplications and additions). Then, this multiplication (or series) is expanded bit by bit into a partial product array (PPA). The unknown operand is in a redundant notation and chosen so that the PPA is limited to have no carry propagation between its columns. Thus, each column forms a separate linear equation which can be back-solved for a digit of the unknown operand. The resulting formulations of several digits are placed into a new PPA and reduced. Then the error of this PPA is analyzed, and additional Boolean elements are added to compensate for the worst regions of error. The steps are summarized below:

1. Express operation as a multiplication(or series).
2. Expand multiplication(or series) into a PPA,
3. Back-Solve the PPA for digits of the unknown operand,
4. Form a new PPA and reduce, and
5. Add error compensating elements.

As an illustration of this algorithm, these steps are described for a five bit approximation of the reciprocal operation.

2.1.1 Express operation as a multiplicat ion(or series):

The following equation expresses the reciprocal operation as a multiplication:

$$Q = 1.0/D \Rightarrow D * Q = 1.0 \ or \ D * Q = 0.111\cdots .$$

Either of the last two expressions can be used, but it has been determined through simulation that the last expression gives more accurate results.

2.1.2 Expand multiplication(or series) into a PPA:

The multiplication of step one is expanded into a binary partial product array. The divisor is assumed to be normalized ($0.5 \leq D < 1.0$) and unsigned to simplify the formulations:

$$\begin{aligned} D &= -d_0 + \sum_{i=1}^{N} d_i * 2^{-i} \\ D &= (0.1)_2 + \sum_{i=2}^{N} d_i * 2^{-i} \\ D &= (0.1 d_2 d_3 \cdots)_2 \\ \textit{and} & \\ Q &= \sum_{i=0}^{N} q_i * 2^{-i} \end{aligned}$$

Note that D is in binary notation and Q is redundant (each q_i is an element from the set of integers for the reciprocal operation, but for other operations it is from the set of real numbers). Using this notation a partial product array is formed.

$$\begin{array}{c|c|c|c|c|c|ccccc}
 & & & & 0. & 1 & d_2 & d_3 & d_4 & d_5 & \cdots \\
 & & & X & q_0. & 41 & 42 & q_3 & q_4 & \cdots & \\
\hline
 & & & & & & \vdots & \vdots & \vdots & \vdots & \cdots \\
 & & & & & 44 & d_2q_4 & d_3q_4 & d_4q_4 & d_5q_4 & \cdots \\
 & & & & 43 & d_2q_3 & d_3q_3 & d_4q_3 & d_5q_3 & \cdots & \\
 & & & q_2 & d_2q_2 & d_3q_2 & d_4q_2 & d_5q_2 & \cdots & & \\
 & & q_1 & d_2q_1 & d_3q_1 & d_4q_1 & d_5q_1 & \cdots & & & \\
 & 40 & d_2q_0 & d_3q_0 & d_4q_0 & d_5q_0 & \cdots & & & & \\
\hline
0. & 1 & 1 & 1 & 1 & 1 & \cdots & & & &
\end{array}$$

2.1.3 Back-Solve the PPA for digits of the unknown operand:

This step solves several columns of the PPA for digits of the unknown operand. Not all of the columns are back-solved because the formulations become too complex to implement with a reasonable amount of hardware. In the worst case the complexity of a digit increases exponentially for each lesser significant digit, but in practice the complexity appears to be linear[SF91b]. To solve the PPA the quotient digits are chosen such that there are no carries between columns of the array. This results in the columns forming separate equations as is denoted in the previous figure by vertical lines. For this example of the algorithm, five digits of quotient are back-solved. Thus, the first five columns form equations which are solved to yield the following five digits of the quotient:

$$\begin{aligned}
q_0 &= 1 \\
41 &= 1 - d_2 \\
q_2 &= 1 - d_3 \\
q_3 &= 1 - d_2 + 2d_2d_3 - d_3 - d_4 \\
q_4 &= 1 - d_2d_3 - d_4 + 2d_2d_4 - d_5.
\end{aligned}$$

2.1.4 Form a new PPA and reduce:

The next step is to form a new PPA. The equations of each digit are placed in separate columns as is shown below:

$$\begin{array}{ccccc}
40 & q_1 & q_2 & 43 & 44 \\
 & & & -d_4 & -d_5 \\
 & & & -d_3 & 2d_2d_4 \\
 & & & 2d_2d_3 & -d_4 \\
 & -d_2 & -d_3 & -d_2 & -d_2d_3 \\
1 & 1 & 1 & 1 & 1
\end{array}$$

Then these equations are reduced. This step of the algorithm differs from other authors' methods. This step takes advantage of both Boolean and algebraic cquivalencies. The elements in the array are Boolean elements and their Boolean equations can be made more complex by combining several elements together. Each Boolean element can be implemented simply with Boolean logic gates. The overall array represents an algebraic summing of these Boolean elements. Algebraic equivalencies are also used. Thus, equivalencies are used to create a good balance of both notations.

The following arc some of the equivalencies [Sch89, SF91b] used to reduce the array:

1. Algebraic Expansion: $3a = 2a +$ a (note that coefficient of 2 is implied by column weight, and that all coefficients in the array are expanded into their binary components),
2. Algebraic Reduction: $2a - a = a$,
3. Boolean Reduction: a $+ b - ab = a|b$,
4. Boolean Reduction: a $- ab = a(1 - b) = a\overline{b}$ and
5. Boolean Reduction: $a + b - 2ub = a \oplus b$,

where a and b are signed binary variables. The juxtaposition of two or more binary variables is considered to be the logical AND of these variables, "|" the logical OR, "$\oplus$" the exclusive OR, "+" addition, "$-$" subtraction, and "$\overline{b}$" inversion. Many orderings can be found for using these equivalencies (or steps of reductions). One algorithm would be to: apply the first step to the whole array starting with the least significant columns, apply step two, apply step three, apply step four, and then apply step five. Loop on steps two through five until no further reduction is achieved. This is a simple algorithm with no back tracking which does produce reasonable results. Other more complex algorithms might use a smart expansion such as expand a $= 2a -$ a only if both new elements, $2a$ and $-a$, can be recombined by other rules. Applying the simple algorithm to the PPA for a five bit estimate of the reciprocal yields:

$$\begin{array}{ccccc}
40 & q_1 & q_2 & 43 & q_4 \\
 & & & & \overline{d_5} \\
 & & & -(d_2\ |d_4) & -d_4 \\
1 & \overline{d_2} & (d_2|\overline{d_3}) & \overline{d_3} & -d_2d_3
\end{array}$$

There has been a reduction from 15 total elements and 5 rows in the maximum column to 8 and 3 respectively. The reduction creates a partial product array with true and complement, positive and negative signed Boolean elements.

2.1.5 Add error compensating elements:

The approximation of the reciprocal operation using the preceding PPA has a good average number of bits correct but has a large worst case error. From simulating the most significant 20 bits of the divisor the average number of bits correct is determined to be 5.75 bits. The number of bits correct is defined to be equal to the negative of the log base two error (for N bits correct there is one integer bit and $N - 1$ fractional bits, where the absolute signed error is less than $\pm 2^{-N}$) [2]. Also, the worst case error is determined from simulation by using interval arithmetic. The error is taken of an actual data point, X, and its calculated reciprocal, F(X), and also for the next data point, X $+ 2^{-S}$, using the same calculated reciprocal, F(X) not $F(X + 2^{-S})$. The worst

[2]The relative error is also limited by the same amount since the function ranges from 1.0 to 2.0. Relative error $\frac{abs.\ error}{f(x)}$

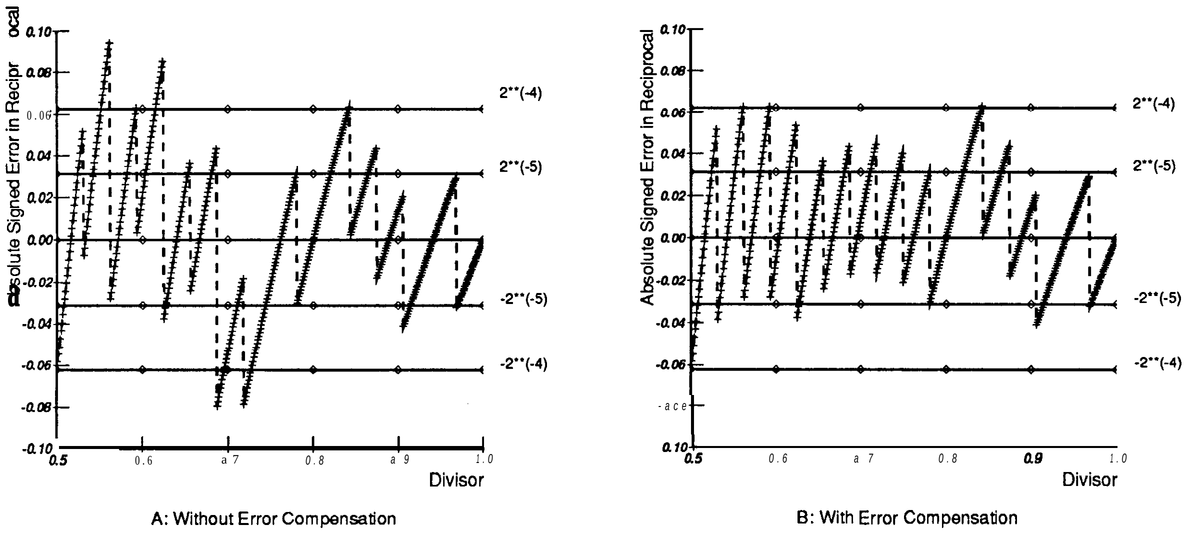


Figure 2: Absolute Signed Error in 5 bit Reciprocal Approximation

case error is determined to be 3.36 bits correct. Mandelbaum [Man90b] first noticed that there is some significant regions of error and he added a PLA to store the reciprocal estimate for these regions. There is another way of reducing the error in these regions. This study suggests adding error compensating Boolean elements to array. By having the correction in the array there is no added latency, no special control circuitry, and the added hardware is small.

To determine the elements to be added, the absolute signed error (computed value minus true value) is plotted versus the divisor and is shown in Figure 2.a. Several elements can be added to the PPA to compensate for errors. Lines have been added to the plot: the top and bottom lines are the lines $\pm 2^{-4}$ and the middle line is 0 and the inner lines are $\pm 2^{-5}$. These lines are useful in determining the effect of a correction term added to the columns of q_4 and q_5 in the PPA. Many terms could be added to reduce the error but there is a tradeoff between starting with a better approximation and adding more correction terms. Because of this there are many possible PPAs. For this example, two compensating elements are chosen: subtracting 2^{-5} for a divisor between 0.53125 and 0.5625, and between 0.59375 and 0.625 (which can be represent by $\bar{d}_2 \bar{d}_3 d_5$); and adding 2^{-4} between 0.6875 and 0.75 $(\overline{d_2} d_3 d_4)$. The PPA with compensating elements after applying reductions is shown below:

$$\begin{array}{cccccc}
q_0 & q_1 & q_2 & q_3 & q_4 & \\
 & & & & & \\
 & & & & \overline{d_5} & \\
 & & & & \overline{d_2} d_3 d_4 & \\
 & & & -(d_2|d_4) & -d_4 & \\
1 & \overline{d_2} & (d_2|\overline{d_3}) & \overline{d_3} & -d_2 d_3 & -\bar{d}_2 \bar{d}_3 d_5
\end{array}$$

This array has 10 total elements and a maximum column height of 4. For larger arrays and larger number of compensating elements, it is common to have recombination of some of these additional elements. In this example there is no recombination but the maximum column only increased by one element. The resulting error of the PPA is shown in Figure 2.b. Numerically the average bits correct has increased to 6.09 bits and the worst case error has improved to 3.92 bits which is an improvement of 0.34 bits on average and 0.56 bits worst case. Thus, a significant improvement has resulted from the addition of two elements. By adding the corrections directly to the array rather than using a PLA, there is a savings in hardware, latency, and control.

Thus, a method has been described for expressing an approximation to a high-order arithmetic operation as a signed PPA. The algorithm differs from other authors due to the additional equivalencies used and the adding of error compensating elements. This has resulted in a small signed PPA which describes an approximation to a high-order operation. This technique can be used for estimating a different number of bits, or using a different number of compensating elements. Thus, there are many PPAs that can be derived from this algorithm for the same operation.

2.2 Algorithm *2:* Adapting Signed PPA to the Multiplier's PPA

The next step in the derivation is to adapt the signed PPA from algorithm 1 to be unsigned and to fit on a given multiplier's PPA. This step could be skipped if a dedicated counter tree were to be built for each application. The counters would need to be Pezaris type counters [Pez71] to accept signed Boolean elements. Many previous studies [SF91a, SF91b, SF92a, SF92b, Man91, MM91] have assumed such an implementation but due to hardware costs these studies are limited to lower precision approximations. To save hardware and allow higher precision the signed arrays are adapted to be added on a multiplier [SF92c]. Algorithm 2 describes the method for adaptation which consists of three steps:

1. Complement negative elements and subtract one,
2. Sum all constant terms and add the result (two's complement if negative), and
3. Adjust array to match the multiplier.

These steps result in an array that can be mapped onto an array of a multiplier. Typical multiplier arrays are described in the next section and this section assumes the signed PPA will fit within the unsigned multiplier's PPA with only minor adjustments.

2.2.1 Complement negative elements and subtract one:

The first step in adapting the signed PPA to be summed on an unsigned PPA is to eliminate any negative variables by complementing them and subtracting one from the appropriate column. The reason this transformation is valid is because: -a = $\overline{a} - 1$, where a is a signed Boolean element. An example is shown using the signed PPA of the five bit reciprocal approximation with compensation.

$$
\begin{array}{rcccccc}
 & q_0 & q_1 & q_2 & q_3 & q_4 & \\
\\
\textit{Before}: & & & & & \overline{d_5} & \\
 & & & & & \overline{d_2}d_3d_4 & \\
 & & & & -(d_2|d_4) & -d_4 & \\
 & 1 & \overline{d_2} & (d_2|\overline{d_3}) & \$3 & -d_2d_3 & -\bar{d}_2\,\bar{d}_3d_5 \\
\hline
\\
\textit{After}: & & & & & \overline{d_5} & \\
 & & & & & \overline{d_2}d_3\overline{d_4} & \\
 & & & & \bar{d}_2\,\bar{d}_4 & \overline{d_4} & \\
 & 1 & \overline{d_2} & (d_2|\overline{d_3}) & \overline{d_3} & (\bar{d}_2|\bar{d}_3) & (d_2|d_3|\overline{d_5}) \\
 & & & & -1 & -2 & -1
\end{array}
$$

2.2.2 Reduce constants:

In this step all the constants are reduced to one row which is then added back to the array. Thus, in the worst case one row is added to adapt **a** signed array to be unsigned. The constants in the previous example are reduced: $(1,0,0,-1,-2,-1)_2 = (1,0,-1,0,0,-1)_2 = (0,1,0,1,1,1)_2$. If the result was negative then the two's complement of it would be added back to the array. The following is the result of this step:

$$\begin{array}{ccccccc}
q_0 & q_1 & & q_2 & q_3 & q_4 & \\
 & & & & & 1 & \\
 & & & & & \overline{d_5} & \\
 & & & & \bar{1} & \overline{d_2}d_3d_4 & \\
 & 1 & & \bar{d_2}\,\bar{d_4} & & \$4 & 1 \\
 & \overline{d_2} & (d_2|\overline{d_3}) & \overline{d_3} & (\bar{d_2}|\bar{d_3}) & & (d_2|d_3|\overline{d_5})
\end{array}$$

2.2.3 Adjust array to match the multiplier:

This step changes the shape, or aspect ratio, of the application's array to be similar to the multiplier's array by minor modifications. Two types of modifications that are commonly used are: 1)shifting the application's array to be superimposed on a different set of columns of the multiplier's array, and 2)shifting specific elements in oversized columns of the application's array to lesser significant columns and replicating them. The first modification is concerned with the positioning of the superimposed array. Column one of the application's array can be shifted to be superimposed on any column of the multiplier array. The second modification is a minor adjustment of one element rather than the whole array. An element can be shifted to a lesser significant column given that it is replicated the appropriate amount. If the element is moved to next less significant column it must be replicated twice due to the equivalency: $a = a/2 + a/2$ where the fractional constant is implied by the column weight. Thus, there are two methods of adjustment presented that perform a coarse or fine adjustment of the application's array.

For the example of the five bit reciprocal estimate the column heights are $(2,1,3,5,2)$. Assume that it is to be adapted to a non-Booth multiplier which has column heights which increase linearly (i.e. $(1,2,3,4,5,6,\cdots)$). Then the first method of modification can be used to sum the reciprocal array on the non-Booth multiplier. By method one, the first column with two elements would be matched with the second column of the multiplier. There is no need to then apply fine adjustments to the array. None of the other columns exceed the multiplier's column heights ($2 \leq 2$, $1 \leq 3$, $3 \leq 4$, $5 \leq 5$, *and* $2 \leq 6$). If the column with five elements had instead had six elements then the constant term "1" could be moved and replicated twice in the next less significant column. For this example, only a proper shifting of the whole array is needed to adapt the array to a non-Booth multiplier.

Thus, the three steps of algorithm 2 are easy to implement and result in an array that has at most one extra row. By applying algorithm 1 and algorithm 2 for different number of bits approximated and different number of compensating elements a variety of PPAs can be derived. The final PPA chosen should fit within the constraints of the given multiplier and produce a reasonable approximation of the high-order operation desired. Thus, a method has been presented for deriving PPAs which describe an approximation to an arithmetic operation and which can be superimposed on a given multiplier's array.

3 Implementation: Reusing a Multiplier

To create a low-cost high-precision approximation to an operation a large amount of hardware needs to be reused from an existing design. In this study, a floating-point multiplier is reused to sum the Boolean elements of the PPA which describes an approximation to an operation. Hardware could be dedicated to the auxiliary operation but these operations are not executed frequently enough to merit this expense. Floating-point multiplication is a frequent operation. Thus, it is common to implement a full direct multiplication's PPA in one iteration. These other operations also can benefit from having a large counter tree and adder. Thus, this study suggests reusing a floating-point multiplier's counter tree and adder to sum the PPA of auxiliary operations.

This section describes typical multipliers, their PPAs, their dataflow, and how to adapt them to sum an auxiliary operation's PPA. In particular, floating point multipliers are considered for adaptation since their PPAs are usually bigger than fixed point multipliers. This study assumes the floating point multiplier to be adapted uses double precision IEEE 754 standard [IEEE85]. Adapting multipliers that use other standards is possible and the IEEE standard multiplier is shown as an example. In general a floating-point multiplier has a sign, exponent, and a magnitude unit. The magnitude unit is by far the largest of the three units and it is the only one discussed in this study. Thus, this study is limited to discussing double precision floating point multipliers that use IEEE 754 standard.

Multipliers can be divided into two types: non-Booth or Booth [Boo51] multipliers. Each type will be discussed separately. Their PPAs differ in the number of partial products; non-Booth having twice as many as a Booth scheme for the same width multiplier. The Booth scheme has signed partial products and uses hardware to encode sign extensions and hot ones. The non-Booth multiplier has unsigned partial products. If the sign extension encoding and hot one encoding hardware of a Booth multiplier are not reused then both PPAs can be considered to be unsigned. Their PPAs differ only in size and shape. Their implementations are slightly different. They both have a counter tree and adder but the Booth multiplier has a longer latency prior to the counter tree. This delay can have some effect on how the auxiliary operation's PPA is adapted to the multiplier. These slight differences will be noted but they do have many similarities. The basic method of adaptation is to multiplex the auxiliary operation's PPA into the multiplier's counter tree. Thus the PPAs, dataflows, and additional hardware for adaptation will be shown for both types of multipliers.

3.1 Non-Booth Multipliers

A non-Booth multiplier performs a direct multiplication. Each bit of one operand, the multiplier, is multiplied by the multiplicand and determines one partial product row. The partial product is equal to zero if the corresponding bit of the multiplier is zero, and is equal to the multiplicand if the bit is a one. For IEEE 754 standard, double precision operands are 53 bits. Thus, the corresponding PPA consists of 53 partial products of 53 bits as shown in Figure 3. This is a very large PPA. For this reason it is the most desirable PPA for a large auxiliary operation's PPA. Thus, an auxiliary operation is limited to a column height of 53 elements for this type of multiplier.

The dataflow is shown in Figure 4. Only the magnitude unit of the multiplier is shown in the figure. It consists of three parts: 1) Boolean element creator, 2) counter tree, and 3) adder. The counter tree and adder are reused by the auxiliary operations. To do this, a multiplexor is placed

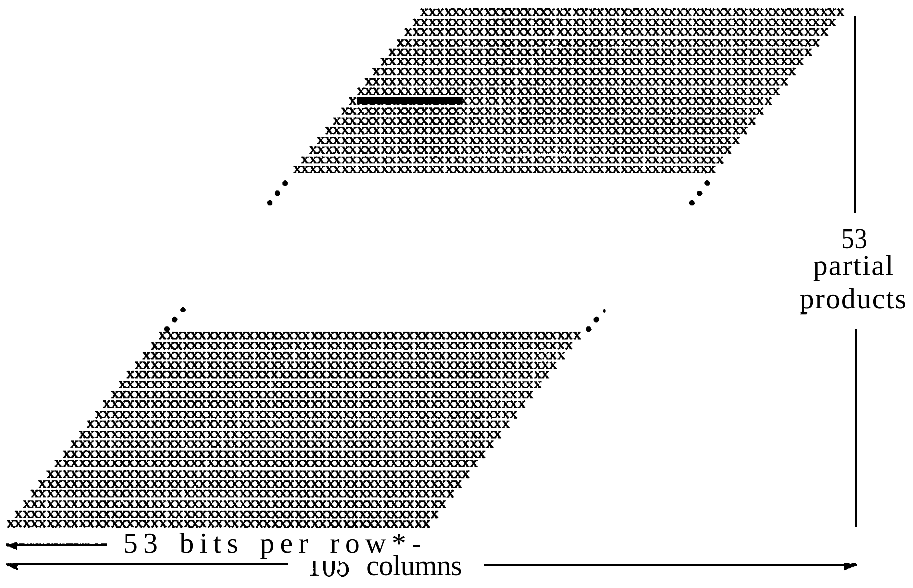


Figure 3: Array of a 53 bit Non-Booth Multiplier

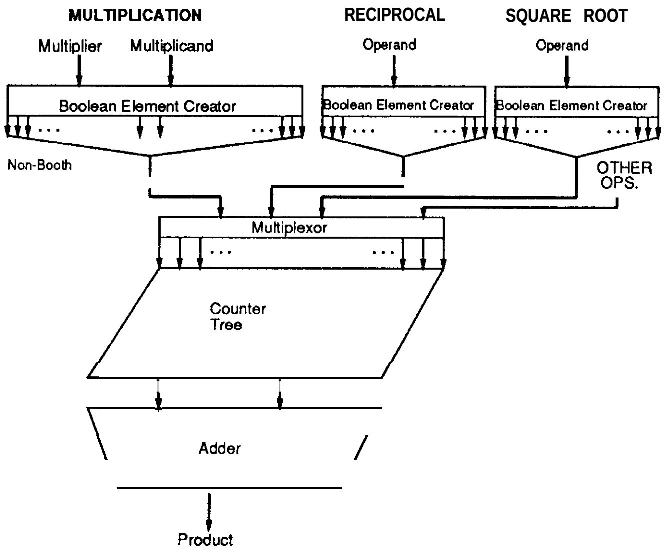


Figure 4: Implementation Using Non-Booth Multiplier

in front of the counter tree. The multiplier's PPA or any auxiliary operation's PPA can be selected to be summed on the counter tree and adder. Each auxiliary operation requires a Boolean element creator which calculates each element in their PPA. Each of these Boolean element creators is very small in comparison to the counter tree and adder. This saves a substantial amount of hardware by reusing the hardware of the multiplier. The only disadvantage is that the latency will increase slightly since a multiplexor is added to the path, and there might be a small delay due to differences in the latency of the Boolean element creators. Thus, the changes to the multiplier are minor in terms of latency and amount of hardware.

The adaptation to a non-Booth multiplier is very simple. A multiplexor is added above the counter tree, and Boolean element creators are added for each auxiliary operation. Typically the auxiliary operation's PPA is much smaller than the multiplier's PPA. The multiplexor only needs to be implemented for each bit of the auxiliary operation's PPA, since the operands of the multiplier can be set to zero causing any unused elements to be zero. The number of gates needed is one to create a Boolean element and one to multiplex it, for each bit in the auxiliary operation's PPA. There is a slight increase in the latency of the multiplier but there is a substantial savings in hardware versus implementing separate counter trees for each auxiliary operation. The non-Booth multiplier has a very large PPA which can accommodate very large applications with up to 53 rows of elements.

3.2 Booth Multipliers

The Booth multiplier has a smaller PPA than a non-Booth. Several bits of one of the operands are scanned for each partial product row. Typically two bits are recoded into one partial product which decreases the number of rows by half. This is shown in Figure 5. There are 27 rows of 54 bits each neglecting the sign and hot-one encoding (for more details see [Mac61, Hwa79, WF82, VSH89, VSS91]). To directly use the PPA of this multiplier the auxiliary operation's PPA must have less than 28 rows. This is much smaller than the non-Booth PPA and therefore the precision of the approximation is smaller than for a non-Booth multiplier.

The dataflow of the Booth multiplier is shown in Figure 6. Since each partial product is created by scanning several bits, there are extra stages of hardware needed prior to the counter tree. The three stages are: l)Booth decode, 2)Booth multiplexing, and 3)inversion. The Booth decoder scans three bits with one bit as overlap per partial product. It determines whether to form two times, one times, or zero times the multiplicand, and also whether the row should be negative or positive. This involves in the worst case delay the latency of a 3 by 2 AND-OR gate (3 way ANDs followed by a 2 way OR gate). The second stage is multiplexing the different multiples of the multiplicand and requires a 2 by 2 AND-OR gate per bit. The third stage is an inversion (or one's complementing) if the partial product is negative. This is implemented as an exclusive-OR gate for each bit. Thus the latency of the Booth multiplier prior to the counter tree is longer than the non-Booth multiplier. This allows the possibility of adding a counter stage into the path of an auxiliary operation. If one 3/2 counter stage is added then the auxiliary operation can have a PPA of 40 rows maximum rather than 27. For small arrays with only a few number of large columns this would be good solution. A Dadda scheme [Dad65]of counters would reduce the hardware requirements by only placing counters in the columns that exceed the constraints of the multiplier's PPA. For larger arrays the hardware cost may be too great. Thus, two datapaths are shown in Figure 6 depending on whether counter stages are added prior to the counter tree or if the Boolean element creators feed directly into the multiplexor before the counter tree. Note that depending on the path lengths, there can be premultiplexing of the auxiliary operations prior to the multiplexor before the counter

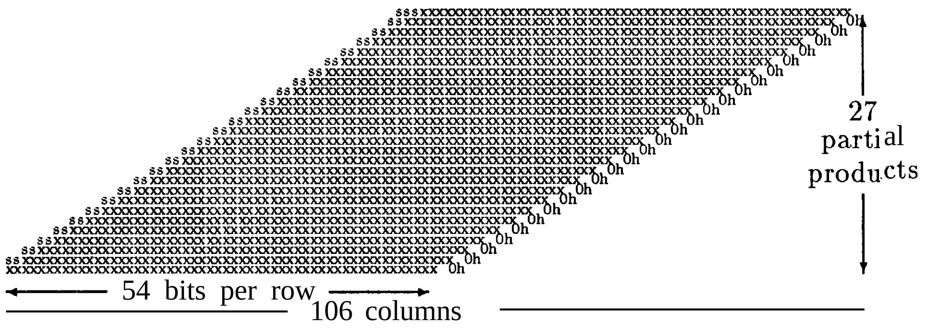


Figure 5: Array of a 53 bit Booth Multiplier

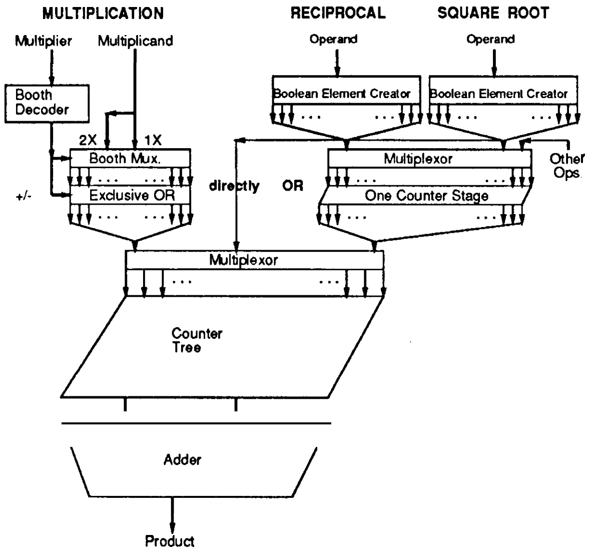


Figure 6: Implementation Using Booth Multiplier

tree. Thus, a Booth multiplier has a longer latency prior to counter tree which can aid in the design of the auxiliary operation, but the PPA is smaller than that of a non-Booth multiplier.

The basic method of adapting an auxiliary operation's PPA to that of a multiplier is to limit the PPA's size and shape to the multiplier's constraints. The non-Booth multiplier for double precision IEEE notation has 53 rows and the Booth multiplier has 27 rows. It might be possible to add more counters to the Booth multiplier without affecting the latency. This would increase the maximum number of rows to 40 for a Booth multiplier. The overall adjustment to either multiplier is to create the elements in each PPA with a separate Boolean element creator and then multiplex the different elements into the counter tree. Note that the auxiliary operations will then have the same properties as the multiplier. If the multiplier is pipelined then the approximations are pipelined. They also have the same latency as the multiplier. Thus, a multiplier easily can be adapted to sum the elements of another operation's PPA.

4 Results of Reciprocal Approximation

The general method of deriving a. PPA for a given arithmetic operation has been shown along with how to adapt a multiplier to sum this PPA. The derivation used an example of a five bit reciprocal approximation. Its PPA is very small; the maximum column height is 5 and the total number of elements is 13. A non-Booth multiplier is able to sum a PPA with a maximum column height of 53 and a total number of 2809 elements, and a Booth multiplier can sum 27 rows and a total of 1458 elements. Thus, a larger estimate of the reciprocal operation is easily possible.

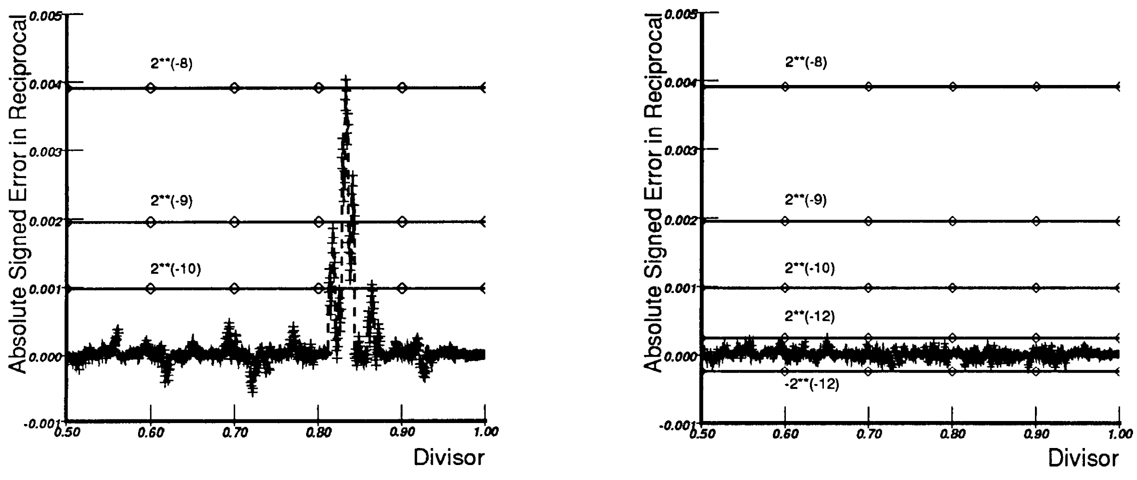
Deriving a PPA for the reciprocal operation which provides a reasonable approximation and fits on a multiplier, requires many sizings and simulations. This process can be divided into four steps:

1. Derivation of formulas,
2. Initial sizings,
3. Error compensation, and
4. Final array.

Step one involves deriving the initial formulations of the operation. This already has been shown for the reciprocal operation. Step two sizes several PPAs for different number of bit approximations and decides on the biggest PPA that will fit in a given multiplier. Step three adds error compensating elements to the array. And, step four presents the final array which can be summed by a given multiplier.

Many PPAs are designed in the process of choosing a PPA to implement. There are two parameters which can be changed in the design: the number of bits to approximate, and the number of compensating elements to add. Step two performs rough sizings without adding any compensating elements (algorithm 1, step 5) and without using expansions of worst case columns (algorithm 2, step 3). A 16 bit estimate of the reciprocal requires 49 rows and a 17 bit estimate requires 64 rows. Assuming a non-Booth multiplier is to be used in the implementation, 53 rows are available. For this study a 17 bit approximation of the reciprocal is chosen. Expansions of the worst case columns can bring the PPA to under 53 rows and this allows some space to add elements to compensate for error.

The next step is to add error compensating elements until the usable elements in the multiplier's PPA are filled. The first step in adding compensating elements is to determine the error. The 17 bit PPA from step two has an average of 14.58 bits correct and has a worst case of 7.91 bits correct. The absolute signed error is plotted in Figure 7.a. Notice that the worst case error is restricted to a small region between a divisor of 0.80 and 0.90. If this region didn't exist the worst case error would be better than 10 bits correct. Mandelbaum [Man90b] used a PLA to store the reciprocal estimate in this region. This study adds elements to the PPA to correct for this error. Many elements can be added to reduce the error. After this region is improved, the error of the new PPA is plotted and new compensating elements are added. This continues until there is no space in the PPA to add new elements due to the constraints of the multiplier's PPA. The total number of elements in the PPA without compensating elements is 343 and with compensating elements and adjusting to be unsigned is 484. The error of the array with compensating elements is 15.18 bits correct on average (20 bit simulation) and has a worst case of 12.003 bits correct. A plot of the error of the PPA with compensating elements is given in Figure 7.b. The axis is the same scale as Figure 7.a to give a direct comparison of the error. The worst case error has been reduced by over 4 bits and the



A: Without Error Compensation B: With Error Compensation

Figure 7: Absolute Signed Error of 17-bit Reciprocal Approximation

average error has improved slightly. A more interesting plot is given in Figure 8 which shows the minimum bits correct versus the divisor. This shows that many more compensating elements would need to be added to gain another bit or two of minimum bits correct. Thus, 12 bits correct appears to be the limit on the worst case error for a 53 by 53 bit partial product array implementation.

The last step is forming the final array. The shape of the array is shown in Figure 9. z's indicate Boolean variables and l's indicate the known constant one. The outline of a non-Booth multiplier's PPA is shown around the reciprocal's PPA. The reciprocal's PPA does not use the whole PPA. Less than 20% of the array is used. The array is restricted by the maximum number of rows. The worst case error (both relative and absolute) is approximately 2^{-12} which causes any new compensating elements to be added to column 13 and 14 which are completely full. Thus, no more significant compensating elements can be added to the array. The equations of each column of this PPA are given in Appendix A.1. Thus, a PPA has been created for the reciprocal operation which can fit on a non-Booth multiplier and has a reasonable accuracy of more than 12 bits correct (one integer bit and eleven fractional bits).

The same type of derivation can be done for a Booth multiplier implementation. The equations of the PPA are given in Appendix A.2. For the Booth multiplier implementation only 27 rows were used to give a PPA with 175 elements. The approximation produced by this PPA has 12.71 bits correct on average and 9.17 bits in the worst case. This approximation requires the same number of iterations as the non-Booth PPA for a quadratically converging algorithm. Since,

$$9 \Rightarrow 18 \Rightarrow 36 \Rightarrow 53 + bits$$

$$12 \Rightarrow 24 \Rightarrow 48 \Rightarrow 53 + bits.$$

Thus, either approximation can be used with a quadratically converging algorithm for division but the non-Booth PPA is preferred for a constantly converging algorithm. Note that the Booth PPA is smaller and can be superimposed on either multiplier if only a small precision approximation is needed. The following section details the double-precision algorithms for division using the non-Booth PPA but easily can be compared for the Booth PPA.

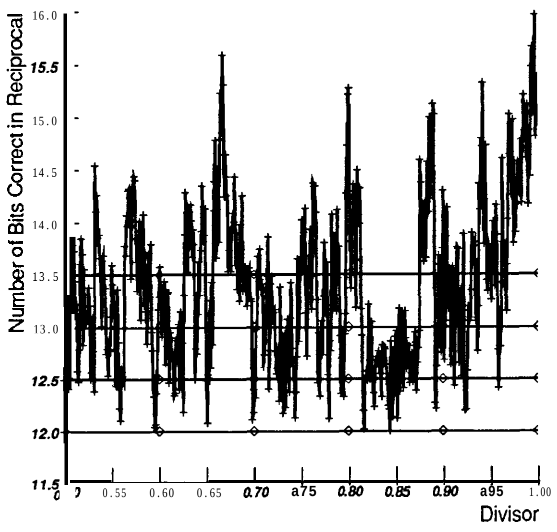


Figure 8: Minimum Bits Correct For 17-bit Reciprocal Approximation With Error Compensation

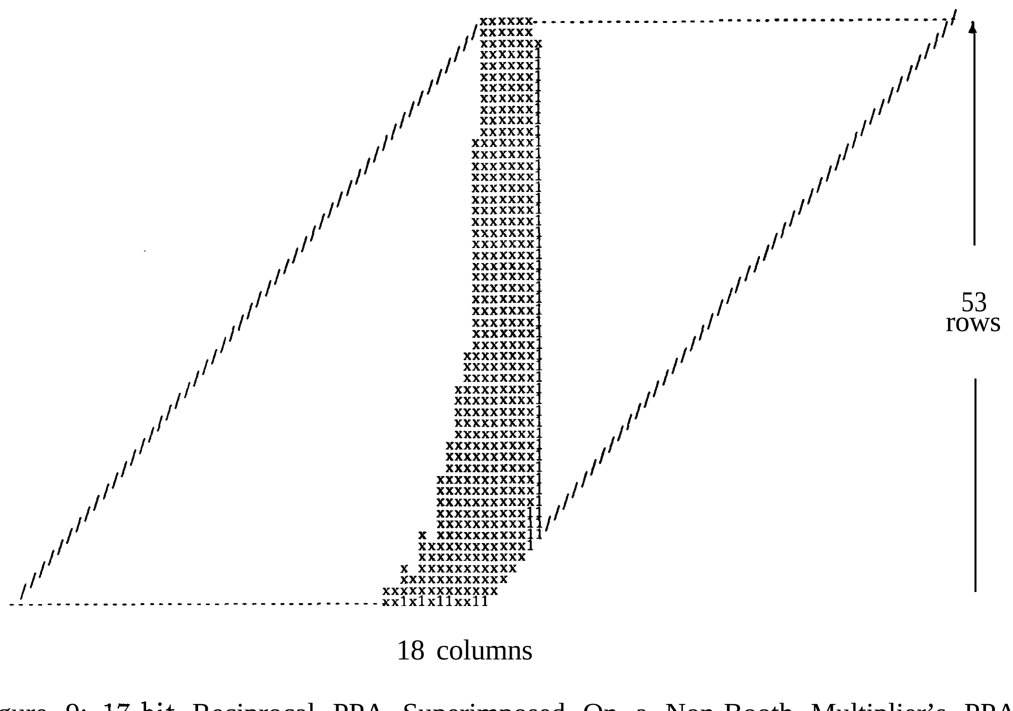


Figure 9: 17-bit Reciprocal PPA Superimposed On a Non-Booth Multiplier's PPA

5 Algorithms for High Precision Division Using a Reciprocal Approximation

The reciprocal estimate given by the PPA described in the previous section can be combined with several algorithms to perform a high precision division operation. This section highlights some of these algorithms and gives a rough comparison between the algorithms. Each algorithm will be presented, giving details of the steps and the latency. The latency to calculate the quotient and the remainder will be given. In some implementations only the quotient is needed and in others which require proper rounding the remainder is necessary. Also, internal recursion methods will be discussed but not compared. The following are the algorithms to be present cd and **a** dagger(i) denotes algorithms to be compared.

1. Newton-Raphson then multiply†,
2. Goldschmidt algorithm†,
3. Newton-Raphson then SRT†,
4. High-radix SRT†,
5. Modified high-radix SRT†,
6. Prescale SRT†, and
7. Internal recursion.

The reciprocal approximation is assumed to be implemented on a non-Booth multiplier and provides at least 12 bits of accuracy (one integer bit and eleven bits of fraction).

5.1 Newton-Raphson then multiply

The Newton-Raphson algorithm is commonly used if only the quotient is needed without proper rounding. It can be used with or without an initial approximation. The algorithm converges quadratically. It has been detailed in many places [Fly70, FS89, WF82]. Basically this algorithm computes a reciprocal to a high accuracy and then the result is multiplied by the dividend to produce the quotient. If a remainder is needed, the quotient is multiplied by the divisor and subtracted from dividend. The algorithm is shown below:

$$\begin{aligned}
\textit{For:}\quad Q &= N/D \\
\textit{initialize}: & \\
X_0 &\approx 1/D \\
\textit{iterate}: & \\
X_{i+1} &= X_i * (2 - D * X_i) \\
\textit{final}: & \\
Q &= X_{last} * N \\
R &= N - (Q * D)
\end{aligned}$$

An interesting implementation of this algorithm used a lookup table accurate to $2^{-14.42}$ which required a 32k by 16 bit ROM [FS89]. The 17-bit PPA of this study provides a 12 bit approximation and requires 3 iterations for a 53 bit result ($12 \Rightarrow 24 \Rightarrow 48 \Rightarrow 53+$). Each iteration requires 2 multiplications and a two's complement operation. If only the quotient is needed then one additional multiply is required but if the remainder is also needed then an additional two multiplications and a subtraction are needed. Thus, a total of 7 multiplications and 3 two's complement operations are needed for the quotient. To calculate both the quotient and remainder 8 multiplications, 3 two's complement, and 1 subtraction are required. Note that none of these operations can be performed in parallel.

5.2 Goldschmidt algorithm

This algorithm was studied by Goldschmidt [Gol64] and implemented on the IBM 360 model 91 [AEGP67]. This method is an extension to prescaling [Kri70a] which will be detailed later. The basic method is to multiply the numerator, N, and denominator, D, by an approximation of the reciprocal of the denominator. The denominator approaches one and the numerator approaches the quotient. The algorithm is shown below:

$$\begin{aligned}
\textit{For:}\quad Q &= N/D \\
\textit{initialize}: & \\
X_0 &\approx 1/D \\
D_0 &= D * X_0 \\
N_0 &= N * X_0 \\
\textit{iterate}: & \\
X_{k+1} &= \overline{D_k^{tr}} \\
N_{k+1} &= N_k * X_{k+1} \\
D_{k+1} &= D_k * X_{k+1} \\
\textit{final}: & \\
Q &= N_{last} \\
R &= N - (Q * D)
\end{aligned}$$

An overline indicates a two's complement operation and a superscript tr indicates truncation. This algorithm also converges quadratically but has the advantage that some of the operations can be executed in parallel. For a 12 bit approximation of the reciprocal the quotient can be calculated in 7 multiplications and 3 two's complement operations and the remainder requires an additional multiplication and subtraction. If the multiplier has a throughput of one operation per cycle and a latency of two cycles, and the adder has a latency of one cycle, then the following is a timing diagram:

Cycles :	0	1	2	3	4	5	6	7	8	9
$X's$:	X_0									
$D's$:		D_0	D_0	D_1	D_1	D_2	D_2			
$N's$:			N_0	N_0	N_1	N_1	N_2	N_2	N_3	N_3

The diagram indicates which variables are being calculated in a given cycle. Note that a D and an N can be in the multiplier at the same time executing in parallel. This saves time over a Newton-Raphson scheme if the multiplier can be pipelined.

5.3 Newton-Raphson then SRT

Another interesting alternative is to use a hybrid of schemes. Matula [Mat91, BM91] has suggested combining a Newton-Raphson algorithm with a SRT [Rob58, Toc58] or high-radix non-restoring algorithm. This algorithm is used by Cyrix coprocessors. Be used two iterations of the Newton-Raphson algorithm to get a 17 bit approximation of the reciprocal with two guard bits ($4.75 \Rightarrow 9.5 \Rightarrow 17 + 2$) and then used a high-radix SRT algorithm to iterate 4 times. A similar algorithm has been implemented on the IBM RS/6000 [Mar90]. The major differences between the two implementations are that Matula optimized for small multipliers and the RS/6000 optimized for double precision multiply-add operations. The general algorithm is shown below:

$$\begin{aligned}
For:\ Q &= N/D \\
initialize:& \\
X_0 &\approx 1/D \\
N_0 &= N \\
Q_0 &= 0 \\
NR\ iterate:& \\
X_{i+1} &= X_i * (2 - D * X_i) \\
SRT\ iterate:& \\
q_{k+1} &= N_k * X_{last} \\
Q_{k+1} &= Q_k + q_{k+1} * 2^{-i\beta} \\
N_{k+1} &= (N_k - q_{k+1} * D) * 2^{\beta} \\
final:& \\
Q &= Q_{last} \\
R &= N_{last}
\end{aligned}$$

If this type of algorithm is used with an initial 12 bit estimate then there are two possible combinations. The first is to iterate twice with the Newton-Raphson and then once with the SRT algorithm, and the second is to iterate once on the Newton-Raphson algorithm and twice on the SRT algorithm. Newton-Raphson converges quadratically and the SRT converges constantly. The Newton-Raphson requires 2 multiplications and **a** two's complement operation each iteration, and the SRT requires a startup penalty of a multiply and an add, and a delay each iteration of two multiplications and an add. Thus, the first method requires 7 multiplications, 2 additions, and 2 two's complement for the quotient and the second method requires 7 multiplications, 3 additions, and 1 two's complement operation. Both require an additional multiply delay for the remainder to be calculated. An interesting aspect of this method is that the size of the multiplier required can be rectangular (i.e. W by L bits, where $W << L$).

5.4 High-radix SRT using reciprocal approximation

Another algorithm implemented recently is a high-radix SRT algorithm which uses a large reciprocal approximation. This algorithm has been implemented on the IBM ES9121[SV91]. A small multiplier also can be used for this method. The following is the algorithm:

$$For:\quad Q = N/D$$

$$\begin{aligned}
&\mathit{initialize}: \\
&\qquad x \approx 1/D \\
&\qquad R_0 = N \\
&\qquad Q_0 = 0 \\
&\mathit{iterate}: \\
&\qquad q_{k+1} = (X * R_k^{tr})^{tr} \\
&\qquad Q_{k+1} = Q_k + q_{k+1} \\
&\qquad R_{k+1} = R_k - q_{k+1} * D \\
&\mathit{final}: \\
&\qquad Q = Q_{last} \\
&\qquad R = R_{last}
\end{aligned}$$

Each iteration requires two multiplications and an addition of latency, and there is a startup delay of one multiplication and one addition. The accumulation of the quotient is assumed to be executed in parallel. For a 12 bit approximation of the reciprocal, four iterations are necessary. The latency of the quotient requires 9 multiplications and 5 additions and the remainder 10 multiplications and 5 additions. None of these multiplications can be executed in parallel.

5.5 Modified high-radix SRT using reciprocal approximation

An interesting alternative to the high-radix SRT algorithm is an algorithm suggested by the authors of this paper. The algorithm resembles a series expansion algorithm ($1/D = X * (1 + Y + Y^2 + Y^3 + \cdots)$) more than the SRT algorithm. Also it resembles the first steps of a Newton-Raphson algorithm [Mar90]. The algorithm is derived using the high-radix SRT equations assuming that the quotient estimate, q_{k+1}, is not truncated. The following is the derivation:

$$\begin{aligned}
\mathit{For}\ Q &= N/D \\
q_{k+1} &= X * R_k \ \mathit{instead\ of}\ (X * R_k^{tr})^{tr} \\
Q_{k+1} &= Q_k + q_{k+1} \\
Q_{k+1} &= Q_k + X * R_k \\
Q_{k+1} &= X * \sum_{i=0}^{k} R_i \\
Q_{k+1}/X &= \sum_{i=0}^{k} R_i \\
\mathit{Let}\ S_{k+1} &= Q_{k+1}/X \\
S_{k+1} &= S_k + R_k \\
\\
R_{k+1} &= R_k - q_{k+1} * D \\
R_{k+1} &= R_k - (R_k * X) * D \\
R_{k+1} &= R_k * (1 - X * D) \\
\mathit{Let}\ Y &= (1 - X * D) \\
R_{k+1} &= R_k * Y
\end{aligned}$$

The latency per iteration of two multiplications and one addition has been replaced by only one multiplication. The disadvantage of this algorithm is that all the multiplications involve very wide operands (53 plus several guard bits). The following is the modified high-radix SRT algorithm:

$$\begin{aligned}
&\textit{initialize}: \\
&\quad X \approx 1/D \\
&\quad Y = (1 - X * D) \\
&\quad S_0 = 0 \\
&\quad R_0 = N \\
&\textit{iterate}: \\
&\quad R_{k+1} = R_k * Y \\
&\quad S_{k+1} = S_k + R_k \\
&\textit{final}: \\
&\quad Q = S_{last} * X \\
&\quad R = N - Q * D
\end{aligned}$$

These modified formulations require a startup latency of a multiplication and an addition, a multiplication delay each iteration, and a multiplication at the end to produce the quotient. Another multiplication and addition are needed to compute the remainder. At first glance it appears that the remainder is equal to R_{last} but this quantity has too much error to be usable. For a 12 bit approximation of the reciprocal, the quotient can be calculated in 6 multiplications and 2 additions. This is comparable to the fastest of the previous algorithms but does require large multipliers throughout the whole process. Thus, this algorithm is an interesting alternative algorithm.

5.6 Prescale SRT

Prescaling operands for division to provide a simple to compute quotient estimate has been detailed by several authors: Svoboda [Svo63], Krishnamurthy[Kri70b], and Ercegovac[EL89, EL90]. An approximate reciprocal can be used to scale the dividend and divisor so that the new divisor is close to one. In this case the quotient can be estimated to be the most significant bits of the new dividend. This prescaling is combined into a SRT algorithm to get a high-radix algorithm with simple selection of the quotient estimate. The following is the algorithm:

$$\begin{aligned}
&\textit{For}: Q = N/D \\
&\textit{initialize}: \\
&\quad X_0 \approx 1/D \\
&\quad D_0 = D * X_0 \\
&\quad N_0 = N * X_0 \\
&\textit{iterate}: \\
&\quad q_{k+1} = (N_k)^{tr} \\
&\quad Q_{k+1} = Q_k + q_{k+1}
\end{aligned}$$

$$\begin{aligned} N_{k+1} &= N_k - q_{k+1} * D_0 \\ \textit{final}: & \\ Q &= Q_{last} \\ R &= N - Q * D \end{aligned}$$

Two multiplications are needed to prescale the operands and then a multiply and addition are needed each iteration. The quotient digit selected each iteration is just a truncation of the most significant digits of the partial remainder (or dividend, N_k). Using a 12 bit approximation to the reciprocal, the latency of a 53 bit quotient is 6 multiplications and 5 additions. The remainder requires another multiplication and addition. Thus this method requires less multiplications than the other methods but requires more additions.

5.7 Internal recursion

There are many other algorithms for high-radix division. In this section a brief description is given of research into high-precision algorithms specifically designed for the approximations created by the PPAs. Two methods are considered in this section: using extra look-up tables and iterating on the multiplier. Both these methods will be discussed prior to comparing all the previous suggested algorithms.

5.7.1 Extra look-up tables

The proposed method creates an approximation of the reciprocal with a small amount of dedicated hardware. There are some other interesting alternatives. One alternative is to use many small look-up tables to form the approximation. The method is based on expanding a Taylor series and using several terms rather than just one. The method has been studied by Farmwald[Far81] and Wong[WF91, WF92]. This is an interesting method of reducing the required size of the look-up tables.

Some ideas from this method appear to be applicable to the proposed method. That by adding a small look-up table the approximations can be enhanced. This is true in general but the look-up table can not just be several additional terms in the Taylor series. The look-up table would have to store the error of the approximations. This is equivalent to having additional error compensating elements. Adding more compensating elements is a fine adjustment rather than a coarse adjustment such as estimating more bits of the operation. Thus, this alternative is interesting to study if only a small amount of enhancement of the present formulations is necessary.

5.7.2 Iterating on the multiplier

The previous method suggests a way to have additional error compensating elements. This idea along with finding a way to iterate on the present estimate, can be implemented by recursing on the multiplier. This is called internal recursion of the Stefanelli algorithm. These two ideas: recursing to add more correction terms, or back-solving additional quotient digits each iteration are detailed.

Adding more compensating elements: The first idea of adding more correction terms easily can be done by feeding back the approximation into the carry save adder tree and then adding error compensating elements. A second PPA would need to be designed with one row reserved for the previous quotient estimate. This is an interesting method of extending the algorithm to have a larger PPA. This method not only allows simple additive compensating elements to be added but also allows multiplicative corrections. The first approximation can be treated as a variable or set of Boolean variables which can be multiplied by other Boolean elements. This allows many possible types of corrections to the present method's approximation. Currently, the added accuracy of this type of correction is not enough to justify the latency of another pass through the multiplier.

Back-solving more quotient digits: The other possibility is to combine the recursion into the approximation algorithm. This has been shown for a division approximation but not for a reciprocal approximation [SF91b]. A non-restoring division can have its quotient approximated by a PPA and have the next remainder directly determined from another PPA. The PPA for the next remainder forms a rectangular shape rather than a triangle. The remainder's PPA can be a completely filled rectangle of 30 rows by 53 columns. It does not map easily onto the parallelogram shape of a multiplier's PPA. Though, for small radices this PPA may fit in the center of the multiplier's array. If this PPA is implementable the latency per iteration is one multiplication. The disadvantage is that only small radices are possible and the algorithm converges constantly rather than quadratically. Thus, it is an interesting algorithm but presently not competitive with the other algorithms presented (see [SF91b] for more details).

Thus high-radix algorithms for division have been discussed showing several algorithms. Most of these were general algorithms for any method of approximation. The last methods focused on enhancing the specific approximations given by the PPAs derived in this paper. Currently these algorithms appear to not converge as fast as generic algorithms. They require an additional pass through the multiplier to add another PPA full of compensating elements. Thus, at this point in the study of these algorithms, generic high-radix algorithms for division are suggested for implementation, and will be compared.

5.8 Comparison

Algorithms for high-radix division using a reciprocal approximation have been presented and several are compared in Table 1. The table lists the algorithm's name, whether operations other than accumulating the quotient can bc executed in parallel, the latency of the quotient and remainder, and the size of the multiplications. The latency is divided into the number of multiplications, additions, and two's complement operations in the worst case path. Total cycles are given for two types of implementations. The first implementation assumes a two cycle multiply, a one cycle add, and a zero cycle two's complement. The second implementation assumes the same add latency but a short multiplication (such as less than 20 bits by 53 bits) takes two cycles and a long multiplication requires three cycles. The cycles of implementation 1 arc given first and then implementation **2.** The latency does not include the delay of the reciprocal approximation which is equivalent to the latency of the long multiplication. Also, all these algorithms assume the approximation is 12 bits.

The table shows that the Goldschmidt algorithm appears to be the fastest method. A major factor which influences the latency of actual implementations is the difference in the size of the multiplications. A small width multiplication can have a lower latency than a large width multi-

Algorithm	Parall-elism	Quotient Latency					Remainder Latency					Multiplier
		Mult	Add	Two's Comp	Cycles 1	Cycles 2	Mult	Add	Two's Comp	Cycles 1	Cycles 2	
Newton-Raphson/mult	No	7	0	3	14	20	8	1	3	17	24	Varies
Goldschmidt	Yes	7	0	3	9	13	8	1	3	12	17	Varies
2Newton-Raphson/1SRT	No	7	2	2	16	22	8	2	2	18	25	Varies
1Newton-Raphson/2SRT	No	7	3	1	17	23	8	3	1	19	26	Rect.
High-Radix SRT	No	9	5	0	23	23	10	5	1	25	25	Rect.
Modified High-Radix SRT	No	6	2	0	14	20	7	3	0	17	24	Large
Prescale SRT	No	6	5	0	17	17	7	6	0	20	21	Rect.

Table 1: Comparison of Algorithms for Division

plication. Implementation 2 captures some of the effects of multiplier latency differences. It can not completely capture a well adapted implementation, but it does give a rough comparison. The Goldschmidt algorithm gives the lowest number of cycles for either of the implementations because of its parallelism and varying size multiplications. Another interesting quantity to compare is the latency per iteration. For quadratically converging algorithms the Goldschmidt algorithm is again the fastest. Two multiplications are executed in parallel giving a delay per iteration of one multiplication. For the constantly converging algorithms the modified high-radix SRT algorithm is fastest since it requires only one multiplication per iteration. Thus, a variety of algorithms have been compared.

There are many algorithms which can use the reciprocal approximation presented in this paper for a high-precision division. All the algorithms are very fast and the fastest appears to be the Goldschmidt algorithm. Though this varies depending on the type of multiplier available. Thus, this study has derived a PPA which approximates the reciprocal operation and has shown its use in a high level algorithm for division.

6 Results of Square Root Approximation

Two different square root approximations are described in this section: the square root, and the reciprocal of the square root. These approximations can be used in a high-precision algorithms for the square root operation as will be shown in the next section. In this section the four steps of finding an appropriate PPA are discussed. The steps are: the derivation of formulas, initial sizings, error compensation, and the final array. Emphasis is given to the derivation and discussing properties of the final array.

6.1 Approximating Square Root

The derivation of the formulas for the square root operation are very similar to the derivation for the reciprocal operation. The operand is assumed to be normalized. Though the normalization is different: $0.25 \leq A < 0.5$ and the exponent is assumed to be even. The new exponent is assumed to be calculated elsewhere in hardware (equal to approximately half the original). If the exponent is odd, the square root of two must be multiplied by the result's magnitude.

$$\begin{aligned} \mathcal{A} &= A * 2^e \\ \mathcal{A}^{1/2} &= A^{1/2} * 2^{e/2} \quad even\ e \\ &= A^{1/2} * 2^{1/2} * 2^{(e-1)/2} \quad odd\ e \end{aligned}$$

Thus, this study solves directly for the square root of operands which are normalized within this range and have even exponents, but an additional multiplication by the square root of two is needed for odd exponents.

The derivation proceeds using algorithm 1.

1. Express square root as a multiplication:

$$\begin{aligned} a_0 &= 0,\ a_1 = 0,\ a_2 = 1 \\ A &= a_2 * 2^{-2} + a_3 * 2^{-3} + \cdots \\ 0.25 \leq A &< 0.5 \\ Q &= q_1 * 2^{-1} + q_2 * 2^{-2} + \cdots \\ 0.5 \leq Q &< 1/\sqrt{2} = 0.707\cdots \\ A &= Q^2 \end{aligned}$$

2. Expand multiplication into a PPA:

$$\begin{array}{cccccc} & 0 & .q_1 & q2 & q_3 & q\text{-}i \\ x & 0 & .q_1 & q2 & q3 & q_4 \\ \hline & & & q_1q_4 & q_2q_4 & q_3q_4 \\ & & q_1q_3 & q_2q_3 & q_3q_3 & q_4q_3 \\ & 4142 & q_2q_2 & q_3q_2 & q_4q_2 & \\ 4141 & q_2q_1 & q_3q_1 & q_4q_1 & & \\ \hline 1 & a_3 & a_4 & a_5 & . & . \ . \end{array}$$

3. Back-solve the PPA:

$$q_1 = 1;\ q_2 = a_3/2;\ q3 = a_4/2 - a_3/8;$$

4. Form a new PPA and reduce:

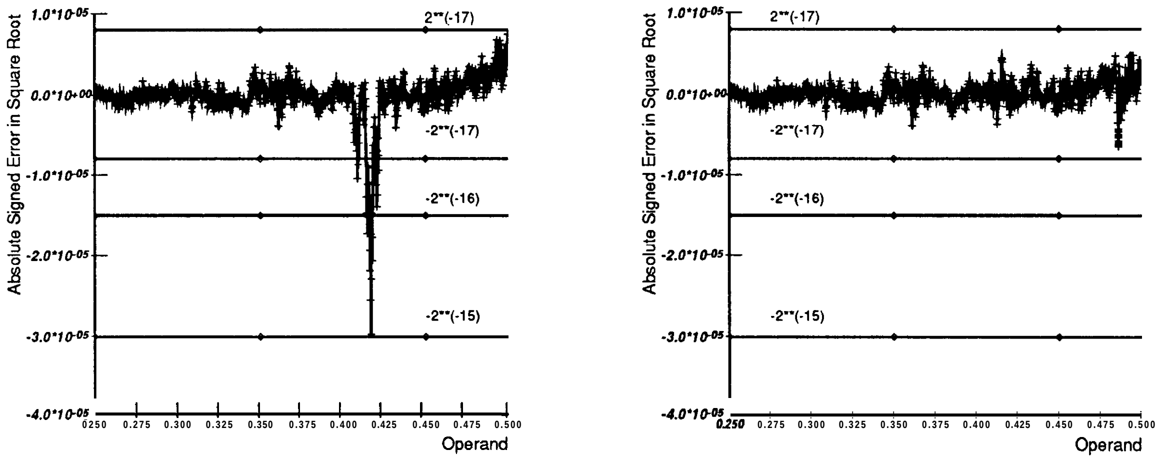
   Note that there are several fractional terms in this derivation. Since the fractions are powers of two, they can easily be represented. For other operations which produce fractions that are not powers of two, a minimal redundant binary notation (with $-1, 0, +1$) is used. The fraction is rounded to a given number of bits (usually to the end of the PPA) and represented in this notation.

$$\begin{array}{cccccc} q_1 & q_2 & q_3 & & & \\ & & & & & \\ 1 & 0 & a_3 & a_4 & 0 & -a_3 \end{array}$$

5. Add error compensating elements:

   This step is skipped until some sizings for different number of bits approximated are compared.

Prior to adding error compensating elements several sizings are done. A 19 bit approximation is chosen because its PPA exceeds 53 rows by a small amount. This approximation provides an average of 19.24 bits correct and a worst **case** of 14.00 bits correct. This assumes that number of bits correct is equal to negative log base two of the absolute error minus one. The minus one term is included since there are no significant integer bits. The square root can be represented as



A: Without Error Compensation

B: Without Error Compensation

Figure 10: Absolute Signed Error of 19-bit Square Root Approximation

$Q = 0.1xxx\cdots$. The PPA has 59 rows, 318 total elements, and is truncated to give a twenty bit wide array. The absolute signed error of this PPA is plotted in Figure 10.a.

After 14 error compensating elements have been added the approximation improves to 19.44 bits on average and 16.08 bits worst case. The absolute signed error is plotted in Figure 10.b. There has been a significant decrease in the worst case error. The array is full because it is limited to a maximum of 53 rows. To achieve another bit correct in the worst case would require many more elements. Thus, a point has been reached to accept this final PPA. The signed PPA is then adapted to a non-Booth multiplier by using algorithm 2. The result is shown in Figure 11 and the formulations for every column are given in Appendix B.

Thus, a PPA describing a square root approximation has been created which has 16 bits correct ($0.1xx\ldots$ with absolute error less than 2^{-17}) in the worst case and an average of 19.4 bits correct. The estimate is 22 bits wide and the PPA directly maps onto a non-Booth multiplier.

6.2 Approximating Reciprocal of Square Root

Another useful operation to approximate is the reciprocal of the square root. Division can be eliminated in quadratically converging, high-precision square root algorithms if an approximation to the reciprocal of the square root is available. Thus, a PPA will be derived for this operation and then high-level algorithms are compared using both types of approximations.

The four step process is repeated for the reciprocal of the square root operation. The derivation and final array are discussed in detail. There are two major differences in this derivation versus the derivation of the square root operation. The normalization is different and the binary operands are represented as polynomials. The following is the derivation using algorithm 1 with some minor modifications for polynomial operands (see [MM91, SF92a, SF92b] for further details).

Normalization is assumed to be between $0.5 \leq A < 1.0$ with A having an even exponent. An additional multiplication is necessary if the exponent is odd. Also the root has the range of $1.0 < Q \leq \sqrt{2} = 1.414\cdots$.

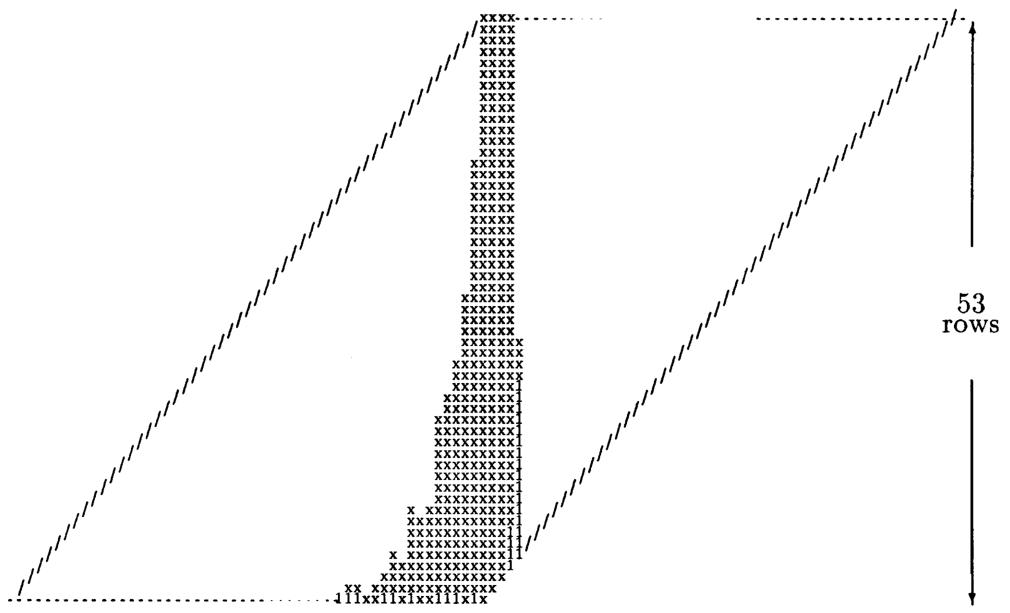


Figure 11: 19-bit Square Root PPA Superimposed On a Non-Booth Multiplier's PPA

1. Express reciprocal of square root as a multiplication.

   The binary variables are transformed into polynomials by the following:

$$\begin{aligned}
A &= a_0 * 2^0 + a_1 * 2^{-1} + a_2 * 2^{-2} + \cdots \\
A(x) &= a_0 * x^0 + a_1 * x^1 + a_2 * x^2 + \cdots \\
Q &= q_0 * 2^0 + q_1 * 2^{-1} + q_2 * 2^{-2} + \cdots \\
Q(x) &= q_0 * x^0 + q_1 * x^1 + q_2 * x^2 + \cdots \\
Q(x)^2 &= q_0^2 * x^0 + 2q_0q_1 * x^1 + (q_1^2 + 2q_0q_2) * x^2 + \cdots
\end{aligned}$$

$$\begin{aligned}
Q(x) &= 1.0/\sqrt{A(x)} \\
A(x) * Q(x)^2 &= 1.0 \\
A(x) * Q(x)^2 &= 0.111\ldots \\
A(x) * Q(x)^2 &= 0 * x^0 + 1 * x^1 + 1 * x^2 + \cdots \\
A(x) * Q(x)^2 &= a_0q_0^2 + (a_1q_0^2 + 2a_0q_0q_1) * x^1 + (a_2q_0^2 + 2a_1q_0q_1 + a_0q_1^2 + 2a_0q_0q_2) * x^2 + \cdots \\
A(x) * Q(x)^2 &= q_0^2 * x^1 + (a_2q_0 + 2q_0q_1) * x^2 + (a_3q_0^2 + 2a_2q_0q_1 + q_1^2 + 2q_0q_2) * x^3 + \cdots
\end{aligned}$$

$$\begin{aligned}
0 * x^0 + 1 * x^1 + 1 * x^2 + 1 * x^3 + \cdots = \\
q_0^2 * x^1 + (a_2q_0 + 2q_0q_1) * x^2 + (a_3q_0^2 + 2a_2q_0q_1 + q_1^2 + 2q_0q_2) * x^3 + \cdots
\end{aligned}$$

   Thus, the last equation expresses the multiplication.

2. Expand multiplication into **a** PPA:

This step can be done but results in a very complex PPA. Instead the equivalent is done which is to set corresponding coefficients of the polynomial of x equal to each other. This is the same as choosing the unknown operand to not cause any carries between columns (or between powers of the polynomial). The following are the resulting equations:

$$\begin{aligned}
1 &= q_0^2 \\
1 &= a_2 q_0^2 \text{ t } 2q_0q_1 \\
1 &= a_3 q_0^2 \text{ t } 2a_2q_0q_1 \text{ t } q_1^2 \text{ t } 2q_0q_2 \\
1 &= a_4 q_0^2 \text{ t } 2a_3q_0q_1 \text{ t } a_2q_1^2 \text{ t } 2a_2q_0q_2 \text{ t } 2q_1q_2 \text{ t } 2q_0q_3
\end{aligned}$$

3. Back-solve the PPA:

$$\begin{aligned}
q_0 &= 1 \\
41 &= 1/2 - a_2/2 \\
q_2 &= 3/8 \text{ t } (a_2)/8 - (a_3)/2 \\
43 &= 5/16 - (5a_2)/16 - (a_3)/4 \text{ t } (3a_2a_3)/4 - (a_4)/2
\end{aligned}$$

4. Form a new PPA and reduce:

$$\begin{array}{ccccccc}
40 & q_1 & q_2 & q_3 & & & \\
 & & & & & a_2a_3 & \\
 & & & & & -a_3 & \\
 & & & & & -a_2 & \\
 & & & & -a_4 & a_2 & \\
 & & -a_2 & & a_2a_3 & 1 & -a_2 \\
1 & 0 & 1 & -a_3 & 1 & 1 & 1
\end{array}$$

Reduces to :

$$\begin{array}{ccccccc}
 & & & & -a_4 & & \\
1 & 0 & \overline{a_2} & \overline{a_3} & a_2a_3 & -\overline{a_2}a_3 & \overline{a_2}
\end{array}$$

5. Add error compensating elements:

This step is skipped until sizings are done to determine the number of bits to approximate.

PPAs were sized for many different number of bits approximated and a 17 bit approximation was chosen. Its PPA has an average of 16.45 bits correct (one integer bit since Q = $1.xxx \ldots$) and a worst case of 9.88 bits correct. The PPA has a maximum of 50 rows and a total of 470 elements. Error compensating elements are then added to this PPA. The final array after many iterations of adding compensating elements has an average of 16.97 bits correct and a worst case of 13.52 bits correct. This PPA has a maximum of 53 rows and a total of 534 elements. The PPA can be placed directly on top of the multiplier's PPA. The array is not shown but the formulations are given in Appendix C.

Thus, a PPA can be derived for the reciprocal of the square root operation which has at least 13.5 bits correct. This array can be summed easily on a non-Booth multiplier. Thus, both types of approximations for the square root have been presented and their formulations are given in Appendix B and C. The square root approximation gives a higher accuracy of 16 bits correct versus 13.5 bits correct for the reciprocal of the square root.

7 Algorithms for High Precision Square Root

The approximations detailed in the preceding section easily can be included in high-precision algorithms for the square root operation. The square root approximation provides at least 16 bits correct and the reciprocal of the square root at least 13.5 bits correct.

The approximation for the reciprocal of the square root operation is on the border line for the number of iterations of a quadratically converging algorithm. There is a possibility that in actually implementing these algorithms that rounding and truncating errors may be introduced. For this study it is assumed that these do not affect the number of iterations. For each implementation of these algorithms an indepth error analysis should be performed to determine if these additional error terms affect the number of iterations. For this discussion it is assumed that a quadratically converging algorithm can increase the 13.5 bits to 53 bits in two iterations (13.5 $\Rightarrow$ 27 $\Rightarrow$ 54).

Five algorithms are compared in this section. The algorithms are named by using letters (using names given in [RGK72]). The algorithms are all based on Newton-Raphson's algorithm or on a Goldschmidt type iteration. The following are there names: NR (Newton-Raphson), F, N, R, and G. The following sections detail each algorithm and then a comparison is given between algorithms.

7.1 Algorithm NR

The Newton-Raphson algorithm for the square root operation can be described by the following:

$$\begin{aligned}
&\textit{initialize}: && \\
&B_0 &\approx\;& \sqrt{N} \\
&\textit{iterate}: && \\
&B_{k+1} &=\;& 1/2 * (B_k + N/B_k) \\
&\textit{final}: && \\
&\sqrt{N} &=\;& B\textit{last} \\
&\epsilon_{k+1} &=\;& \epsilon_k^2/(2B_k) \quad \textit{absolute error}
\end{aligned}$$

B approximates the square root operation. A division and a subtraction are needed each iteration. For a 16 bit estimate, 2 iterations are needed for this quadratically converging algorithm. Thus, a total of 2 divisions and 2 additions are needed. The first division has a 53 bit dividend, a 16+ bit divisor, and needs at least 32 bits of quotient. The second division has a 53 bit dividend, a 32-t bit divisor, and needs at least 53 bits of quotient. For a quadratically converging division algorithm with a 12 bit approximation, 2 iterations are required for the first division and 3 iterations for the second. There are many ways to implement these divisions which have been detailed in a previous section. A total of 2 divisions and 2 subtractions are needed for a 53 bit square root.

7.2 Algorithm F

Algorithm F is derived from the NR algorithm. The following is its derivation:

$$\begin{aligned}
B_{k+1} &= 1/2 * (B_k \text{ t } N/B_k) \\
B_{k+1} &= B_k/2 + N/(2B_k)
\end{aligned}$$

$$
\begin{aligned}
B_{k+1} &= B_k/2 + 1/(2B_k) * (N) \\
B_{k+1} &= B_k - B_k/2 + 1/(2B_k) * (N) \\
B_{k+1} &= B_k - (2B_k)/(2B_k) * B_k/2 + 1/(2B_k) * (N) \\
B_{k+1} &= B_k - (B_k^2)/(2B_k) + 1/(2B_k) * (N) \\
B_{k+1} &= B_k + 1/(2B_k) * (N - B_k^2) \\
B_{k+1} &= B_k + f_p(B_k) * (N - B_k^2)
\end{aligned}
$$

The operation $f_p(x)$ is the reciprocal of $2x$ to p bits of accuracy. Typically p is chosen as 8 bits but using the reciprocal estimate of this study it can be 12 bits. The convergence of this algorithm is constant and appears to be equal to the lesser of p or the size of the approximation.

Thus, the algorithm is restated by the following:

$$
\begin{aligned}
f_p(x) &\approx 1/(2x) \\
&\textit{initialize :} \\
B_0 &\approx \sqrt{N} \\
&\textit{iterate :} \\
B_{k+1} &= B_k + f_p(B_k) * (N - B_k^2) \\
&\textit{final :} \\
\sqrt{N} &= B_{last} \\
\epsilon_{k+1} &= -2\epsilon_k p_k B_k + \epsilon_k^2\left(\frac{1}{2B_k} + p_k\right)
\end{aligned}
$$

Something similar to this algorithm is used by the IBM RS/6000[Mar90] with approximations of S bits. This algorithm requires a reciprocal estimate, two multiplications, and two additions each iteration. The reciprocal approximation can be executed in parallel with the other operations and it is assumed that it does not add to the latency. Thus, at a constant convergence of 12 bits per iteration after an initial approximation of 16 bits, a total of 4 iterations are necessary for a 53 bit result. This requires a total of 8 multiplications and S additions.

7.3 Algorithm N

Algorithm N is a Newton-Raphson algorithm for the reciprocal of the square root operation. The algorithm is described by the following:

$$
\begin{aligned}
&\textit{initialize :} \\
R_0 &\approx 1/\sqrt{N} \\
&\textit{iterate :} \\
R_{k+1} &= R_k/2 * (3 - N * R_k^2) \\
&\textit{final :} \\
\sqrt{N} &= N * R_{last} \\
\epsilon_{k+1} &= -\epsilon_k^2 * \frac{3\sqrt{N}}{2} - \epsilon_k^3 * \frac{N}{2}
\end{aligned}
$$

Two iterations are necessary for a 13.5 bit approximation since this algorithm converges quadratically. Each iteration requires 3 multiplications and a three's complement operation. The three's complement operation is assumed to not add to the latency [RGK72]. Thus, a total of 7 multiplications are needed for a 53 bit result. Most of the multiplications are double-precision.

7.4 Algorithm R

Algorithm R uses both approximations and converges quadratically for every two iterations. The algorithm is described below:

$$\begin{aligned} &\textit{initialize}: \\ B_0 &\approx \sqrt{N} \\ R_0 &\approx 1/\sqrt{N} \\ &\textit{iterate}: \\ R_{k+1} &= R_k * (2 - B_k * R_k) \\ B_{k+1} &= 1/2 * (B_k + N * R_{k+1}) \\ &\textit{final}: \\ \sqrt{N} &= Blast \end{aligned}$$

This algorithm requires 3 iterations for a 13.5 bit approximation. A total of 3 multiplications, a two's complement operation, and one addition are required each iteration. Thus, a total of 9 multiplications, 3 two's complements, and 3 additions are required.

7.5 Algorithm G

Algorithm G is a very interesting algorithm which parallels the Goldschmidt algorithm for division. The following is its algorithm:

$$\begin{aligned} &\textit{initialize}: \\ r_0 &\approx 1/\sqrt{N} \\ B_0 &= N \\ X_0 &= N \\ &\textit{iterate}: \\ SQr_k &= r_k * r_k \\ B_{k+1} &= B_k * r_k \\ X_{k+1} &= X_k * SQr_k \\ r_{k+1} &= 1 + 0.5 * \overline{X^f_{k+1}} \\ &\textit{final}: \\ \sqrt{N} &= B_{last} \end{aligned}$$

X^f indicates the fractional part of X. Each iteration requires 3 multiplications. Additionally, there needs to be hardware to halve the fractional part of the two's complement of X and put a one in the integer part of the result ($r_{k+1} = 1 + 0.5 * \overline{X^f_{k+1}}$). The latency is very small if the result can be

Algorithm	Operation	Square Root Latency					Convergence
		Mult	Add	Div	Cycles		
					1	2	
NR	$SQRT$	0	2	2	22	29	Quad.
F	$SQRT$	8	8	0	24	27	Constant
N	$1/SQRT$	7	0	0	14	20	Quad.
R	BOTH	9	3	0	21	28	1/2 Quad.
G	$1/SQRT$	7	[illegible]	0	10	15	Quad.

Table 2: Comparison of Algorithms for Square Root

approximated (one's complement versus two's complement). Though approximating may introduce an error term which could increase the number of iterations. Assuming that only 2 iterations are necessary and that this complex operation requires no added latency, then the following is its timing diagram:

Cycles :	0	1	2	3	4	5	6	7	8	9	10
$r's$:	r_0				r_1				r_2		
$SQr's$:		SQr_1	SQr_1			SQr_2	SQr_2				
$B's$:			B_1	B_1			B_2	B_2		B_3	B_3
$X's$:				X_1	X_1			X_2	X_2		

This timing chart indicates that for two cycle pipelined multiplications, that the square root can be calculated in 10 cycles. The total delay is equivalent to 5 multiplications, though a total of 7 multiplications are required.

7.6 Comparison

These square root algorithms are compared in Table 2. This table lists the name of the algorithm, the type of operation approximated, and the latency of the square root operation. The latency is assumed without rounding. Latency is listed in terms of the number of multiplications, adds, and divisions and also in cycles for the two implementations. Implementation 1 assumes a one cycle add, two cycle pipelined multiply, and zero cycles for a two's complement (which are not listed in the table). The second implementation assumes a difference between a small multiply (two cycles latency) and a large multiply (three cycles latency) which **is** defined to be greater than a 20 bit by 53 bit multiply. The table shows that algorithm G which is adapted to use a Goldschmidt type multiplier appears to be the fastest with algorithm N also requiring the same number of operations. Thus, depending on the implementation of the multiplier, either of these algorithms may be best.

Thus, the square root and its reciprocal have been approximated with PPAs. The square root yields an approximation of 16 bits and the reciprocal of the square root yields a worst case of 13.5 bits. It appears from the table that the reciprocal of the square root is a more useful function. Either approximation is more accurate than the reciprocal operation's approximation. Due to this difference in accuracy, the square root implementation is almost as fast as a division operation. Thus, this study has derived an approximation for the square root operation which can be used in high performance implementations. The cost of the approximation is low due the use of existing hardware. Thus, a high-radix algorithm for the square root operation has been detailed which is low-cost.

Operation		Proposed Method			Look-up Table		Polynomial
		Ave.	Min.	Total			Accuracy
	Rows	Bits	Bits	Elements	Shape	Size	in Bits
Reciprocal	53	15.18	12.00	484	$2^{13}x11$	11kB	4.08
Reciprocal	27	12.71	9.17	175	$2^{10}x8$	1kB	4.08
Square Root	53	19.44	16.08	398	$2^{16}x14$	112kB	7.06
Reciprocal of Square Root	53	16.97	13.52	534	$2^{13}x12$	12kB	5.03

Table 3: Comparison of Approximations by the Proposed Method, Look-up Tables, and Polynomials

8 Comparison of Approximations

There are many methods for approximating functions. This study has presented a method which has the latency of a multiplication and provides high precision. There are several other methods such as polynomial approximations and look-up tables. There are several types of polynomial approximations but the most common with a small worst case error is Chebyshev polynomials. First degree polynomials are used for comparison and derived using the following studies [Fik66, Har68, And88]. Look-up tables are also a common method. The size of a look-up table for a given precision of a function is determined by:

$$|f(x) - f(x - 2^{-n})| \leq \epsilon_0.$$

where n is the number of bits of index into the look-up table and ϵ_0 is the worst case error (p.195 of [WF82]). n is determined for each operation and then reduced by the number of bits that are constant. Thus, the required table size is easy to determine. Look-up table methods of equivalent accuracy, and polynomials of equivalent latency are compared to the proposed method.

Table 3 summarizes a comparison between the proposed method, look-up tables, and first order polynomial approximations. The look-up tables of equivalent accuracy require between 1 kilo-byte(kB) of memory and 112 kB. The size of the proposed method can be approximated by multiplying the number of total elements by 2 and this is the number of gates required. For instance the square root approximation requires approximately 800 gates which is much smaller than a 112 kB ROM. The benefit of the proposed method is less apparent for lower accuracies (i.e. small reciprocal estimate requires 350 gates versus 1 kB ROM). In comparison to a Chebyshev polynomial the proposed method does much better. The Chebyshev polynomial has an accuracy of 7.06 bits correct for the square root operation but the proposed method has an accuracy of 16.08 bits correct. Thus, the proposed method compares favorably to look-up tables and polynomial approximations.

9 Conclusion

A three step method for creating high-radix algorithms for high-order operations with low-cost has been presented. The three steps are:

1. The derivation of a PPA using algorithm 1 and 2,
2. Adaptation to a multiplier, and
3. Inclusion in an overall iterative algorithm.

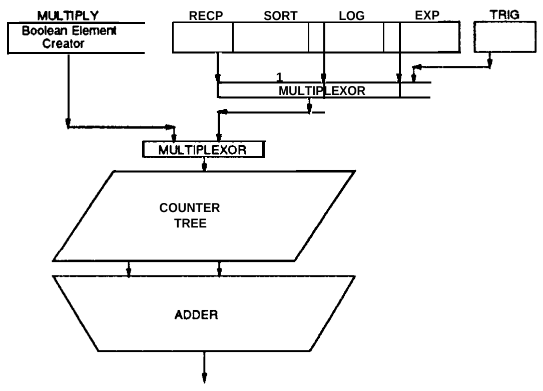


Figure 12: Overview of Reusing Multiplier

The first two steps apply to many high-order arithmetic operations and third step is specific to an operation. The benefits of this method have been shown by applying it to the division and square root operations. The PPAs that are created provide a high precision approximation which can be used in high-radix algorithms. Also, the implementations are low-cost. The savings over creating separate counters trees for the added operation is very large and the savings is great over a comparable look-up table. Thus, high-radix algorithms are possible at low-cost by using the approximation method presented in this study.

There are many contributions of this study. The Stefanelli algorithm [Ste72] appeared to be limited to only low radices because of the large dedicated counter trees required. This study has transformed this costly low-radix algorithm and made it into a low-cost high-radix algorithm. The low-cost has been achieved by reusing the internals of a multiplier. In addition, the algorithm creates small PPAs by using many equivalencies to reduce the array size. The array elements are made slightly more complex since they can be negative, positive, true, and complemented. The precision is also enhanced over previous methods by using error compensating elements. Thus, the final PPA is small and accurate. This PPA can be used by many high-precision algorithms. This study introduced a modified high-radix SRT algorithm which compares favorably to all the constantly converging algorithms. Additionally, novel internal recursive methods for creating a higher precision approximation were discussed. Thus, there are many contributions of this study, but the most important is that it has made the Stefanelli method competitive at high-radices and implementable at low-cost.

To summarize this study an algorithm has been presented for the derivation and implementation of high-precision approximations of high-order arithmetic operations. This method very efficiently uses hardware to provide a low-cost high-precision approximation. There are no ROM tables or off-chip delays required. The algorithm applies to the reciprocal, division, square root, log, exponential, and trigonometric operations, but is not limited to them. All these operations can be implemented reusing the same multiplier as shown in Figure 12. The internal multiplier hardware is now considered a basic primitive or basic building block which can have other operations mapped onto it. By reusing this large amount of hardware, there is **a** large savings in cost. Thus, a method has been introduced for creating high-radix implementations of high-order arithmetic operations at low-cost.

Acknowledgement

The authors would like to thank Gary Bewick for some initial suggestions on multiplier reuse. Also, the authors would like to acknowledge helpful discussions with Robert Risch and Fred Gustavson on error analysis and high-precision algorithms for square root and other high-order arithmetic operations.

A Reciprocal Formulations

A.1 Non-Booth Multiplier Implementation

The number of rows is 53 and total number of elements is 484.

$$
\begin{aligned}
q[0] &= 0 \\
q[1] &= \overline{d_2} + \overline{d_3} \\
q[2] & \;\; d_2 d_3 + \overline{d_4} \\
q[3] &= 1 + d_3 + (\overline{d_2}|d_4) + \overline{d_5} \\
q[4] &= \overline{d_2} d_3 d_4 + d_2 \overline{d_3} d_5 + \overline{d_6} \\
q[5] &= 1 + d_4 + (\overline{d_3}|d_5) + (d_2|d_3|d_5) + d_4 d_5 + (\overline{d_2}|d_6) + \overline{d_7} \\
q[6] &= d_3 d_4 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}) + \overline{d_2} d_3 d_6 + d_4 d_6 + d_2 d_7 + \overline{d_8} \\
q[7] &= \mathbf{1} + \overline{d_2} d_3 + \overline{d_4} d_5 + d_3 \overline{d_4} d_5 + d_2 d_3 d_4 \overline{d_5} + (\overline{d_2}|\overline{d_4}|\overline{d_6}) + d_3 \overline{d_4} d_6 \\
&\quad + d_5 d_6 + \overline{d_2} d_3 d_7 + d_4 d_7 + (\overline{d_2}|d_8) + \overline{d_9} \\
q[8] &= 1 + (\overline{d_2}|d_3|\overline{d_4}) + d_2 d_3 d_5 + (\overline{d_3}|\overline{d_4}|\overline{d_5}) + (d_3|d_4|d_6) \\
&\quad + (d_2|d_5|d_6) + (d_3|d_5|d_6) + d_3 \overline{d_4} d_7 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_7}) \\
&\quad + d_5 d_7 + (\overline{d_2}|d_3|\overline{d_4}|d_5|\overline{d_6}|d_7) + \overline{d_2} d_3 d_8 + (\overline{d_4}|d_8) + d_2 d_9 + \overline{d_{10}} \\
q[9] &= (\overline{d_3}|d_5) + \overline{d_3} d_6 + d_4 d_6 + (d_2|\overline{d_3}|\overline{d_4}|\overline{d_6}) + (\overline{d_2}|d_3|\overline{d_5}|\overline{d_6}) \\
&\quad + (d_3|d_4|d_5|d_6) + (d_3|d_5|d_7) + d_2 \overline{d_4} d_5 \overline{d_7} + d_2 d_3 d_4 d_5 \overline{d_7} \\
&\quad + d_6 d_7 + d_2 d_6 \overline{d_7} + (\overline{d_2}|d_3|\overline{d_4}|d_5|d_6|d_7) + (\overline{d_2}|d_3|\overline{d_4}|d_5|\overline{d_6}|\overline{d_7}) \\
&\quad + \&\& + (d_2|d_4|d_8) + d_5 d_8 + \overline{d_2} d_3 d_9 + d_4 d_9 + (\overline{d_2}|d_{10}) + \overline{d_{11}} \\
q[10] &= (d_2|d_3|d_6) + \overline{d_2} d_4 \overline{d_6} + (\overline{d_2}|\overline{d_3}|\overline{d_4}|d_6) + (d_2|d_5|d_6) \\
&\quad + \boldsymbol{@\&)} + d_2 \overline{d_4} d_7 + (d_3|d_4|d_7) + d_2 d_3 d_5 d_7 + d_4 d_5 \overline{d_7} \\
&\quad + d_3 d_4 d_5 d_7 + (d_2|\overline{d_3}|\overline{d_6}|\overline{d_7}) + (\overline{d_2}|d_3|\overline{d_4}|d_5|d_6|\overline{d_7}) \\
&\quad + (\overline{d_3}|\overline{d_4}|\overline{d_8}) + d_2 \overline{d_5} d_8 + (d_2|\overline{d_3}|\overline{d_5}|\overline{d_8}) + d_6 d_8 + d_2 \overline{d_3} d_4 \overline{d_5} d_6 \overline{d_7}\, d_8 \\
&\quad + \boldsymbol{\&\&} + d_5 d_9 + \overline{d_2} d_3 d_{10} + d_4 d_{10} + d_2 d_{11} + \overline{d_{12}} \\
\boldsymbol{\&\ l\ ]} &= 1 + \overline{d_2} d_3 + d_2\, d_4 + (d_3|\overline{d_5}) + d_3 d_5 d_6 + d_2 d_4 d_5 d_6 + \overline{d_3} d_7 + (d_4|d_7) \\
&\quad + d_2 d_3 \overline{d_4} d_7 + (d_5|d_7) + d_2 \overline{d_5} d_7 + d_2 d_3 d_5 d_7 + d_2 d_6 d_7 + (d_3|\overline{d_4}|\overline{d_6}|\overline{d_7}) \\
&\quad + d_2 d_5 d_6 d_7 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}|\overline{d_6}|d_7) + (d_4|d_8) + (d_2|\overline{d_3}|\overline{d_4}|\overline{d_8}) \\
&\quad + (d_3|\overline{d_4}|\overline{d_5}|\overline{d_8}) + (\overline{d_3}|\overline{d_6}|\overline{d_8}) + (\overline{d_2}|d_4|\overline{d_6}|\overline{d_8}) \\
&\quad + d_2 \overline{d_3} d_4 \overline{d_5}\, \overline{d_6}\, \overline{d_8} + \overline{d_2}\, \overline{d_3} d_7 d_8 + (\overline{d_2}|d_3|\overline{d_4}|d_5|\overline{d_6}|\overline{d_7}|\overline{d_8}) \\
&\quad + (d_3|d_4|d_9) + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_9}) + (\overline{d_3}|\overline{d_5}|\overline{d_9}) \\
&\quad + d_2 d_4\, d_5 d_9 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}|\overline{d_9}) + \overline{d_2} d_6 d_9 + d_2 d_3 d_6 d_9 \\
&\quad + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}|\overline{d_6}|\overline{d_8}|d_9) + (d_2|d_3|d_4|d_5|d_6|d_7|d_8|d_9) \\
&\quad + (d_2|d_4|d_{10}) + d_3 \overline{d_4} d_{10} + \overline{d_2} d_5 d_{10} + d_2 d_3 d_5 d_{10} + \overline{d_2} d_3 d_{11}
\end{aligned}
$$

$+\overline{d_2}d_4d_{11} + d_2d_3d_4d_{11} + d_2d_{12} + \overline{d_{13}}$

$q[12] \;=\; 1 + (d_4|d_5) + d_2d_6 + d_2d_3d_5d_6 + d_3\overline{d_4}d_5d_6 + (\overline{d_2}|\overline{d_3}|\overline{d_4}|\overline{d_7})$
$+(\overline{d_3}|d_5|\overline{d_7}) + (\overline{d_4}|\overline{d_5}|\overline{d_7}) + (\overline{d_2}|\overline{d_4}|\overline{d_5}|\overline{d_7})$
$+(\overline{d_3}|\overline{d_4}|d_5|\overline{d_7}) + (\overline{d_2}|\overline{d_3}|\overline{d_6}|\overline{d_7}) + (\overline{d_2}|\overline{d_4}|d_6|\overline{d_7})$
$+(\overline{d_5}|\overline{d_6}|\overline{d_7}) + d_2d_3d_4d_5d_6d_7 + d_2d_4d_8 + (d_3|d_4|d_8)$
+(&|&|&|&) $+ (\overline{d_2}|d_4|\overline{d_5}|\overline{d_8}) + (\overline{d_3}|\overline{d_4}|\overline{d_5}|\overline{d_8})$
+&&&& $+ (d_4|d_6|d_8) + (\overline{d_3}|d_7|\overline{d_8}) + \overline{d_2}d_3d_4d_5\overline{d_6}\,\overline{d_7}\,\overline{d_8}$
$+d_2d_3\overline{d_4}d_5d_6\overline{d_7}d_8 + (\overline{d_2}|d_3|d_4|d_5|\overline{d_6}|d_7|\overline{d_8}) + (d_2|d_3|d_4|\overline{d_5}|\overline{d_6}|\overline{d_7}|\overline{d_8})$
$+(\overline{d_2}|\overline{d_3}|d_4|d_5|\overline{d_6}|d_7|d_8) + (d_4|d_9) + (\overline{d_2}|\overline{d_3}|\overline{d_4}|\overline{d_9})$
$+d_2d_3d_5d_9 + (d_4|d_5|d_9) + (d_3|d_6|d_9) + d_7d_9 + d_2\overline{d_3}d_4\overline{d_5}\,\overline{d_6}\,\overline{d_7}\,\overline{d_9}$
$+(\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}|\overline{d_6}|\overline{d_8}|\overline{d_9}) + (d_2|\overline{d_3}|\overline{d_4}|d_5|d_6|d_7|d_8|\overline{d_9})$
$+(d_2|\overline{d_3}|\overline{d_4}|d_5|d_6|d_7|\overline{d_8}|d_9) + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}|\overline{d_6}|\overline{d_7}|d_8|d_9)$
$+(d_2|\overline{d_3}|\overline{d_4}|d_5|d_6|\overline{d_7}|\overline{d_8}|\overline{d_9}) + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}|d_6|d_7|\overline{d_8}|d_9)$
+ & d o $+ d_2d_3d_4d_{10} + (\overline{d_3}|\overline{d_5}|\overline{d_{10}}) + d_2\overline{d_3}d_4\overline{d_5}d_{10} + d_6d_{10}$
$+\overline{d_2}d_3d_4d_5d_6\overline{d_7}d_9d_{10} + \overline{d_2}d_3d_4d_5\overline{d_6}\,\overline{d_7}\,\overline{d_8}d_9d_{10} + d_3\overline{d_4}d_{11}$
$+d_5d_{11} + \overline{d_2}d_3d_{12} + d_4d_{12} + d_2d_{13} + \overline{d_{14}}$

$q[13] \;=\; (\overline{d_2}|d_4|\overline{d_5}) + (\overline{d_2}|\overline{d_3}|d_4|\overline{d_5}) + (\overline{d_3}|d_6) + (d_4|d_6)$
$+(d_3|d_4|d_6) + d_2d_3d_5d_6 + (d_4|d_5|d_6) + (d_2|d_3|d_7)$
$+d_5d_7 + (d_2|d_3|d_4|d_5|d_7) + d_4\overline{d_6}d_7 + d_3\overline{d_4}d_6d_7 + (\overline{d_2}|\overline{d_5}|\overline{d_6}|\overline{d_7})$
$+(\overline{d_2}|\overline{d_3}|d_4|\overline{d_5}|d_6|\overline{d_7}) + d_2d_8 + (d_2|d_5|d_8)$
$+(d_3|d_5|d_8) + \overline{d_3}d_4\overline{d_5}d_8 + (\overline{d_2}|\overline{d_4}|\overline{d_6}|\overline{d_8}) + (\overline{d_5}|\overline{d_6}|\overline{d_8})$
$+d_2d_3\overline{d_7}d_8 + (\overline{d_4}|\overline{d_7}|\overline{d_8}) + \overline{d_2}d_3d_4d_5\overline{d_6}d_7d_8 + d_2\overline{d_3}d_4\overline{d_5}d_6d_7\overline{d_8}$
$+(d_2|d_3|d_9) + (d_4|d_9) + (d_3|d_4|d_9) + (\overline{d_2}|\overline{d_4}|\overline{d_5}|\overline{d_9})$
$+d_2d_3d_6\overline{d_9} + (\overline{d_4}|\overline{d_6}|\overline{d_9}) + (\overline{d_2}|\overline{d_7}|\overline{d_9}) + (\overline{d_3}|\overline{d_7}|\overline{d_9})$
$+(\overline{d_2}|d_3|d_4|d_5|\overline{d_6}|d_7|d_9) + \overline{d_2}d_8d_9 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_6}|d_7|d_8|d_9)$
$+d_2\overline{d_3}d_4d_5d_6\overline{d_7}d_8d_9 + (d_2|d_3|d_4|d_5|d_6|d_7|d_8|d_9)$
$+(\overline{d_2}|d_3|\overline{d_4}|d_5|d_6|\overline{d_7}|d_8|\overline{d_9}) + (\overline{d_2}|d_3|\overline{d_4}|d_5|\overline{d_6}|d_7|d_8|\overline{d_9})$
$+(d_2|d_4|d_{10}) + (\overline{d_2}|\overline{d_3}|\overline{d_5}|\overline{d_{10}}) + (\overline{d_4}|\overline{d_5}|\overline{d_{10}})$
$+d_2d_6\overline{d_{10}} + (\overline{d_3}|\overline{d_6}|\overline{d_{10}}) + d_2\overline{d_3}d_4d_5d_6\overline{d_{10}} + \overline{d_2}d_7d_{10}$
i-d:! $d_3d_4d_5d_9d_{10} + d_2d_3d_4d_5d_6d_9d_{10} + \overline{d_2}d_3d_4d_5\overline{d_6}\,\overline{d_7}\,\overline{d_8}\,\overline{d_9}d_{10}$
$+d_2d_3d_4\overline{d_{11}} + (\overline{d_2}|\overline{d_5}|\overline{d_{11}}) + (\overline{d_2}|d_3|\overline{d_4}|d_5|d_{11})$

$q[14] \;=\; d_2d_3 + \overline{d_4} + (\overline{d_2}|\overline{d_4}|\overline{d_6}) + (\overline{d_2}|\overline{d_3}|\overline{d_4}|\overline{d_6}) + (d_2|d_5|d_6)$
$+d_3d_4d_5\overline{d_6} + (d_3|d_4|d_7) + (\overline{d_2}|\overline{d_3}|\overline{d_5}|\overline{d_7}) + (\overline{d_2}|\overline{d_4}|\overline{d_5}|\overline{d_7})$
$+(d_6|d_7) + (d_2|d_4|d_6|d_7) + (d_3|d_8) + (\overline{d_2}|\overline{d_4}|\overline{d_5}|\overline{d_8})$
$+(d_2|d_6|d_8) + (\overline{d_2}|\overline{d_3}|\overline{d_6}|\overline{d_8}) + d_2d_7d_8 + (\overline{d_2}|d_4|\overline{d_9})$
$+(d_5|d_9) + (d_2|d_5|d_9) + (d_2|d_3|d_5|d_9) + d_2d_6d_9$
$+\overline{d_2}d_3d_4\overline{d_5}\,\overline{d_6}\,\overline{d_7}d_9 + (d_2|d_3|d_4|d_5|d_6|d_7|\overline{d_8}|d_9)$

$$
\begin{aligned}
&+(\overline{d_2}|\overline{d_3}|\overline{d_4}|\overline{d_{10}}) + d_2d_5d_{10} + (d_2|d_3|\overline{d_4}|\overline{d_5}|\overline{d_{10}}) \\
&+\overline{d_2}d_3d_4d_5\overline{d_6}\,\overline{d_7}\,\overline{d_8}d_9\overline{d_{10}} + d_2d_{11} + (d_2|d_4|d_{11}) + (d_3|d_5|d_{11}) \\
&+(\overline{d_3}|\overline{d_5}|\overline{d_{11}}) + \overline{d_2}d_6d_{11} + \overline{d_2}d_6d_{11} + \overline{d_2}d_3d_4\overline{d_5}\,\overline{d_6}\,\overline{d_7}d_{10}d_{11} \\
&+(d_2|\overline{d_3}|\overline{d_4}|d_5|d_6|d_7|d_{10}|d_{11}) + d_2d_{12} + d_2d_4\overline{d_{12}} + d_3\overline{d_4}d_{12} \\
&+(d_3|d_4|d_{12}) + \overline{d_2}d_5d_{12} + \overline{d_2}d_5d_{12} + d_3d_{13} + \overline{d_2}d_3d_{13} \\
&+\overline{d_2}d_3d_{13} + \overline{d_2}d_4d_{13} + \overline{d_2}d_4d_{13} + d_3d_{14} + d_3d_{14} + d_2\overline{d_3}d_{14} \\
&+d_2\overline{d_3}d_{14} + (d_2\,|\overline{d_{15}})
\end{aligned}
$$

$$
\begin{aligned}
q[15] \;=\;\; & d_3d_4d_5 + d_2d_3d_4\overline{d_5} + (\overline{d_3}|\overline{d_4}|\overline{d_6}) + (\overline{d_2}|\overline{d_3}|\overline{d_4}|\overline{d_6}) \\
&+(\overline{d_3}|\overline{d_5}|\overline{d_6}) + (\overline{d_2}|d_3|d_5|\overline{d_6}) + (\overline{d_4}|\overline{d_5}|\overline{d_6}) + (\overline{d_2}|\overline{d_4}|\overline{d_5}|\overline{d_6}) \\
&+(\overline{d_2}|d_3|\overline{d_7}) + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_7}) + (d_3|d_5|d_7) + (\overline{d_2}|\overline{d_3}|\overline{d_5}|\overline{d_7}) \\
&+d_4d_5d_7 + (\overline{d_2}|\overline{d_6}|\overline{d_7}) + d_3d_6d_7 + (\overline{d_4}|\overline{d_6}|\overline{d_7}) + (d_5|d_6|d_7) \\
&+(d_3|d_4|d_8) + (d_2|d_3|d_4|d_8) + \overline{d_2}d_5\overline{d_8} + d_3d_5\,d_8 \\
&+(d_4|d_5|d_8) + (d_3|d_6|d_8) + (d_4|d_6|d_8) + d_5d_6d_8 \\
&+\boldsymbol{(\&l\&1\&)} + (\overline{d_3}|\overline{d_7}|\overline{d_8}) + d_4d_7d_8 + (\overline{d_3}|\overline{d_4}|\overline{d_9}) \\
&+d_3d_5d_9 + (d_4|d_5|d_9) + d_2\,\overline{d_6d_9} + (d_3|d_6|d_9) + (d_4|d_6|d_9) \\
&+(d_2|d_7|d_9) + d_3d_7d_9 + (d_2|d_8|d_9) + (\overline{d_3}|d_4|\overline{d_{10}}) \\
&+\boldsymbol{(\&I\&l\&o)} + (d_3|d_5|d_{10}) + (\overline{d_4}|d_5|\overline{d_{10}}) + d_3d_6d_{10} \\
&+(d_2|d_7|d_{10}) + (d_2|d_3|d_{11}) + (d_2|d_4|d_{11}) + (d_2|d_5|d_{11}) \\
&+d_3d_5d_{11} + (d_2|d_6|d_{11}) + d_2d_3d_{12} + d_2d_4\overline{d_{12}}
\end{aligned}
$$

$$
\begin{aligned}
q[16] \;=\;\; & (d_2|d_3|d_4) + (d_2|d_5) + (d_3|d_5) + d_4d_5 + \overline{d_3}d_6 + d_2d_3d_6 \\
&+d_2d_4d_6 + d_3d_7 + (\overline{d_2}|d_4|\overline{d_7}) + (d_6|d_7) + (d_2|d_8) + \overline{d_2}d_3d_8 \\
&+\text{@\&\&)} + (d_5|d_8) + (d_2|d_3|d_9) + d_4d_9 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_9}) \\
&+(d_5|d_9) + (d_3|d_{10}) + d_2\,d_6d_{10} + (d_2|d_6|d_{10}) + d_2d_{11} \\
&+\&l + \overline{d_3}d_4d_{11} + (\overline{d_3}|\overline{d_4}|\overline{d_{11}}) + d_3d_{12} + (d_4|d_{12}) \\
&+(\&\,|d_4|d_{12}) + d_2d_4\overline{d_{12}} + \overline{d_3}d_{13} + d_2d_3d_{13} + d_2d_3d_{13} + (d_2\,|d_4|d_{13}) \\
&+(d_2|d_4|d_{13}) + d_5d_{13} + d_5d_{13} + d_{14} + d_2d_{14} + (d_2|d_3|d_{14}) \\
&+(\&\,|d_3|d_{14}) + d_4d_{14} + d_4d_{14} + \overline{d_2}d_{15} + d_3d_{15} + d_3d_{15} + d_{16} \\
&+(\overline{d_2}|d_{16}) + (\overline{d_2}|d_{16}) + \overline{d_{17}}
\end{aligned}
$$

$$
\begin{aligned}
q[17] \;=\;\; & 1+1+1+1+\overline{d_6}+\overline{d_6}+\overline{d_6}+\overline{d_6}+\overline{d_6}+\overline{d_6}+\overline{d_6}+\overline{d_6}+d_8d_{10} \\
&+d_8d_{10} + d_8d_{10} + d_8d_{10} + d_7d_{11} + d_7d_{11} + d_7d_{11} + d_7d_{11} + d_3d_4d_{12} \\
&+d_3d_4d_{12} + d_3d_4d_{12} + d_3d_4d_{12} + (d_2|d_5|d_{12}) + (d_2|d_5|d_{12}) \\
&+(\&l\&l\&a) + (d_2|d_5|d_{12}) + d_6d_{12} + d_6d_{12} + d_6d_{12} + d_6d_{12} \\
&+\overline{d_{15}}+\overline{d_{15}}+\overline{d_{15}}+\overline{d_{15}}+\overline{d_{15}}+\overline{d_{15}}+\overline{d_{15}}+\overline{d_{15}}+\overline{d_{16}}+\overline{d_{16}} \\
&+\overline{d_{16}}+\overline{d_{16}}+\overline{d_{16}}+\overline{d_{16}}+\overline{d_{16}}+\overline{d_{16}} \\
q[18] \;=\;\; & 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \\
&+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \\
&+\&\&\overline{A}\&\;\, d_6\;d_7\;d_8
\end{aligned}
$$

A.2 Booth Multiplier Implementation

The number of rows is 27 and total number of elements is 175.

$$
\begin{aligned}
q[0] &= \mathbf{0}\\
q[1] &= \overline{d_2} + \overline{d_3}\\
q[2] &= d_2d_3 + \overline{d_4}\\
q[3] &= 1 + d_3 + (\overline{d_2}|d_4) + \overline{d_5}\\
q[4] &= 1 + \overline{d_2}d_3d_4 + d_2\overline{d_3}d_5 + \overline{d_6}\\
q[5] &= d_4 + d_2d_3d_4 + (\overline{d_3}|d_5) + (d_2|d_3|d_5) + d_4d_5 + (\overline{d_2}|d_6) + \overline{d_7}
\end{aligned}
$$

$$
\begin{aligned}
q[6] &= \overline{d_2}d_3d_4 + (\overline{d_2}|\overline{d_4}|\overline{d_5}) + \overline{d_2}d_3d_6 + d_4d_6 + d_2d_7 + \overline{d_8}\\
q[7] &= \&\& + (d_2|d_4) + \overline{d_4}d_5 + d_3\overline{d_4}d_5 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_6})\\
&\quad +d_5d_6 + \overline{d_2}d_3d_7 + d_4d_7 + (\overline{d_2}|d_8) + \overline{d_9}\\
q[8] &= 1 + \overline{d_4} + d_2d_3d_5 + d_2d_3d_4d_5 + \overline{d_4}d_6 + d_3\overline{d_4}d_6 + d_2d_5\overline{d_6} + (\overline{d_3}|\overline{d_5}|\overline{d_6})\\
&\quad +(\overline{d_2}|\overline{d_3}|\overline{d_4}|d_5|\overline{d_6}) + (\overline{d_2}|\overline{d_4}|\overline{d_7}) + d_3\overline{d_4}d_7 + \overline{d_2}d_5d_7\\
&\quad +d_2d_3d_5d_7 + \overline{d_2}d_3d_8 + \overline{d_2}d_4d_8 + d_2d_3d_4d_8 + d_2\overline{d_3}d_4\overline{d_5}\, d_7d_8 + d_2d_9\\
&\quad +\&o\\
q[9] &= (\overline{d_3}|\overline{d_4}) + (\overline{d_3}|d_5) + (\overline{d_2}|d_4|\overline{d_5}) + d_2\, d_6 + d_3d_6 + d_2d_3\overline{d_4}d_6\\
&\quad +d_2d_3d_5d_6 + (d_4|d_5|d_6) + d_2d_7 + d_2d_3d_4d_7 + (\overline{d_3}|\overline{d_5}|\overline{d_7})\\
&\quad +d_6d_7 + d_2\overline{d_3}d_4d_5d_6\overline{d_7} + d_3\overline{d_4}d_8 + d_5d_8 + d_2\overline{d_3}d_4\overline{d_5}\,\overline{d_7}\,\overline{d_8}\\
&\quad +d_2\overline{d_3}d_4\overline{d_5}d_7d_8 + \overline{d_2}d_3d_9 + d_4d_9 + d_2d_{10} + \overline{d_{11}}
\end{aligned}
$$

$$
\begin{aligned}
q[10] &= (d_2|d_3) + (d_3|d_4|d_5) + \overline{d_3}d_4d_6 + (\overline{d_2}|\overline{d_4}|\overline{d_5}|\overline{d_6})\\
&\quad +(\overline{d_2}|d_3|\overline{d_4}|d_5|\overline{d_6}) + d_2\, d_4d_7 + (\overline{d_2}|\overline{d_3}|\overline{d_5}|\overline{d_7})\\
&\quad +(d_4|d_5|d_7) + (d_2|d_6|d_7) + (d_3|d_6|d_7) + d_2\overline{d_3}d_4d_5d_6d_7\\
&\quad +(\&l\&Id\&8) + d_2\overline{d_5}d_8 + (\overline{d_3}|\overline{d_5}|\overline{d_8}) + \overline{d_2}d_6d_8\\
&\quad +(d_2|d_3|d_4|d_5|d_6|d_7|d_8) + (d_2|d_4|d_9) + (\overline{d_3}|\overline{d_4}|\overline{d_9})\\
&\quad +\overline{d_2}d_5d_9 + (\overline{d_2}|d_3|\overline{d_4}|\overline{d_5}|\overline{d_6}|d_8|\overline{d_9}) + \overline{d_2}d_3d_{10}\\
&\quad +\overline{d_2}d_4d_{10} + d_3d_{11} + d_2\overline{d_3}d_{11} + \overline{d_{12}}\\
q[11] &= \overline{d_2}d_3 + (d_2|d_3|d_4) + (d_3|\overline{d_5}) + d_2\overline{d_3}d_4d_5 + (d_2|d_4|d_6)\\
&\quad +(d_5|d_6) + (\overline{d_2}|\overline{d_5}|\overline{d_6}) + (\overline{d_2}|\overline{d_3}|\overline{d_5}|\overline{d_6}) + d_2\, d_7\\
&\quad +(d_2|d_3|d_4|d_7) + (d_2|d_5|d_7) + d_4\overline{d_8} + (\overline{d_2}|d_4|\overline{d_8})\\
&\quad +d_2\overline{d_3}d_4\overline{d_5}d_7\overline{d_8} + d_2d_9 + d_3d_9 + d_2\overline{d_3}d_4\overline{d_5}\, d_9 + (d_2|\overline{d_3}|\overline{d_4}|\overline{d_5}|d_6|d_7|d_8|\overline{d_9})\\
&\quad +\&\&o + d_2d_{12} + \overline{d_{13}}\\
q[12] &= (d_2|d_4|d_5) + (d_2|\overline{d_3}|\overline{d_4}|\overline{d_5}) + (\overline{d_2}|d_3|\overline{d_6}) + \overline{d_3}d_4d_6\\
&\quad +(d_3|d_5|d_6) + (d_4|d_5|d_6) + (\overline{d_3}|d_4|\overline{d_7}) + (d_2|d_5|d_7)\\
&\quad +\overline{d_3}d_5d_7 + (\overline{d_4}|d_5|\overline{d_7}) + (d_2|d_6|d_7) + d_3d_6d_7 + (d_2|d_3|d_8)\\
&\quad +\&.\&\& + d_3d_4\overline{d_8} + \overline{d_2}d_5\overline{d_8} + d_3d_5d_8 + (d_2|d_6|d_8) + d_7d_8\\
&\quad +d_2d_3d_9 + (d_2|d_4|d_9) + d_3d_4d_9 + (d_2|d_5|d_9) + d_6d_9 + d_2d_3d_{10}\\
&\quad +(d_2|d_4|d_{10}) + d_5d_{10}\\
q[13] &= (d_2|d_3|d_4) + d_2\, d_5 + d_2d_3\overline{d_5} + (d_3|d_6) + (\overline{d_2}|d_3|\overline{d_6})\\
&\quad +\&l\&) + (d_3|d_7) + (d_4|d_7) + d_2d_8 + \overline{d_3}d_8 + (d_4|d_8)\\
&\quad +d_9 + d_3d_9 + \overline{d_3}d_{10} + d_{11} + d_2d_{11} + (d_2|d_3|d_{11}) + (d_2|d_3|d_{11})\\
&\quad +d_4d_{11} + d_4d_{11} + \overline{d_2}d_{12} + d_3d_{12} + d_3d_{12} + d_{13} + d_2d_{13} + d_2d_{13} + \overline{d_{14}}
\end{aligned}
$$

B Square Root Formulations

The number of rows is 53 and total number of elements is 398.

$$\begin{aligned}
q[0] &= 0\\
q[1] &= 0\\
q[2] &= 1\\
q[3] &= 1 + a_3\\
q[4] &= 1 + a_4\\
q[5] &= a_5\\
q[6] &= \overline{a_3} + a_6\\
q[7] &= 1 + (\overline{a_3}|\overline{a_4}) + a_7\\
q[8] &= 1 + \overline{a_4} + a_3\overline{a_5} + (\overline{a_4}|\overline{a_5}) + a_8\\
q[9] &= (\overline{a_3}|\overline{a_4}) + (\overline{a_3}|\overline{a_6}) + (\overline{a_4}|\overline{a_6}) + a_9\\
q[10] &= 1 + \overline{a_3} + (a_4|\overline{a_5}) + (\overline{a_3}|a_4|\overline{a_5}) + (\overline{a_3}|a_4|\overline{a_6}) + (\overline{a_5}|\overline{a_6})\\
&\quad +(\overline{a_3}|\overline{a_7}) + (\overline{a_4}|\overline{a_7}) + a_{10}\\
q[11] &= \overline{a_6} + \overline{a_3}a_4a_6 + (\overline{a_3}|\overline{a_5}|a_6) + (\overline{a_3}|\overline{a_7}) + (\overline{a_5}|\overline{a_7}) + (\overline{a_3}|\overline{a_8})\\
&\quad +a_4\overline{a_8} + a_{11}\\
q[12] &= a_4a_5a_6 + (\overline{a_3}|\overline{a_4}|a_5|a_6) + a_3a_4a_7 + a_6\overline{a_7} + (\overline{a_3}|\overline{a_8}) + (\overline{a_5}|\overline{a_8})\\
&\quad +(\overline{a_3}|\overline{a_9}) + (\overline{a_4}|\overline{a_9}) + a_{12}\\
q[13] &= 1 + (a_3|\overline{a_4}|\overline{a_6}) + a_5a_6 + a_3\overline{a_4a_5}a_6 + a_4a_7 + a_3a_4a_7 + a_3a_5a_7\\
&\quad +(\overline{a_4}|\overline{a_5}|a_7) + a_3a_6a_7 + a_3a_4a_8 + a_3a_5a_8 + (a_3|\overline{a_6}|\overline{a_8}) + (\overline{a_3}|\overline{a_9})\\
&\quad +(\overline{a_5}|\overline{a_9}) + (\overline{a_3}|\overline{a_{10}}) + (\overline{a_4}|\overline{a_{10}}) + a_{13}\\
q[14] &= 1 + a_3\overline{a_4}a_6 + (a_3|\overline{a_7}) + (a_3|\overline{a_4}|a_7) + a_5a_7 + a_3\overline{a_4}a_5a_7 + a_4a_6a_7\\
&\quad +a_4a_8 + a_3a_4a_8 + a_4a_5a_8 + (\overline{a_3}|\overline{a_6}|\overline{a_8}) + (\overline{a_7}|\overline{a_8}) + a_3a_4a_9\\
&\quad +(\overline{a_6}|\overline{a_9}) + a_3\overline{a_{10}} + a_5\overline{a_{10}} + (\overline{a_3}|\overline{a_{11}}) + (\overline{a_4}|\overline{a_{11}}) + a_{14}
\end{aligned}$$

$$\begin{aligned}
q[15] &= 1 + a_3a_5 + a_3a_4\overline{a_6}a_7 + \overline{a_4}a_5a_6a_7 + a_5a_8 + a_3a_4a_5\overline{a_8} + a_4a_6a_8 + a_3a_7a_8\\
&\quad +a_4a_9 + a_3a_4a_9 + a_3a_5a_9 + a_4a_5a_9 + a_3a_6a_9 + (\overline{a_7}|\overline{a_9}) + a_3a_4a_{10}\\
&\quad +a_3a_5a_{10} + (\overline{a_6}|\overline{a_{10}}) + (\overline{a_3}|\overline{a_{11}}) + (\overline{a_5}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_{12}})\\
&\quad +(\overline{a_4}|\overline{a_{12}}) + a_{15}\\
q[16] &= a_4a_5 + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}) + \overline{a_3}a_6a_7 + (\overline{a_3}|\overline{a_5}|\overline{a_6}|\overline{a_7})\\
&\quad +(\overline{a_4}|\overline{a_5}|\overline{a_6}|a_7) + a_4a_8 + a_3a_5\overline{a_8} + (a_3|a_6|\overline{a_8}) + (\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_8})\\
&\quad +\overline{a_3}a_5a_6a_8 + (\overline{a_4}|\overline{a_7}|a_8) + a_3\overline{a_4}a_5\overline{a_6}a_7\overline{a_8} + a_3\overline{a_4}a_5a_9\\
&\quad +\overline{a_3}a_4a_6a_9 + a_3a_7a_9 + (a_3|\overline{a_8}|\overline{a_9}) + a_4a_{10} + a_4a_5a_{10} + a_3a_4\overline{a_5}a_{10}\\
&\quad +(\overline{a_3}|\overline{a_6}|a_{10}) + (\overline{a_7}|\overline{a_{10}}) + a_3a_4a_{11} + (\overline{a_6}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_{12}})\\
&\quad +(\overline{a_5}|\overline{a_{12}}) + (\overline{a_3}|\overline{a_{13}}) + (\overline{a_4}|\overline{a_{13}}) + a_{16}
\end{aligned}$$

$$\begin{aligned}
q[17] &= 1 + a_3a_5a_6 + a_3\overline{a_4}a_8 + (\overline{a_5}|\overline{a_8}) + (\overline{a_3}|\overline{a_5}|\overline{a_8}) + a_3a_4a_5\overline{a_8}\\
&\quad +a_3a_4a_5a_6a_8 + (\overline{a_5}|\overline{a_7}|a_8) + (a_3|\overline{a_6}|a_7|\overline{a_8}) + a_3a_4\overline{a_6}a_7\overline{a_8}\\
&\quad +a_3\overline{a_4}a_5\overline{a_6}a_7a_8 + a_3\overline{a_4}a_5\overline{a_6a_7a_8} + \overline{a_3}a_4a_9 + a_5a_9 + (\overline{a_3}|\overline{a_4}|\overline{a_6}|a_9)\\
&\quad +a_5a_6a_9 + a_4a_7a_9 + a_3a_4a_5a_7a_9 + a_3\overline{a_8}a_9 + a_3\overline{a_4}a_5\overline{a_6}a_7\overline{a_8}a_9 + a_4a_6a_{10}\\
&\quad +a_3a_4a_5a_6a_{10} + a_3a_7a_{10} + (\overline{a_8}|\overline{a_{10}}) + a_4a_{11} + a_3a_4a_{11} + a_3a_5a_{11}
\end{aligned}$$

$$
\begin{aligned}
&+(\overline{a_4}|\overline{a_5}|a_{11}) + a_3a_6a_{11} + (\overline{a_7}|\overline{a_{11}}) + a_3a_4a_{12} + a_3a_5a_{12}\\
&+a_4a_5a_{12} + a_6\overline{a_{12}} + a_3\overline{a_{13}} + (\overline{a_5}|\overline{a_{13}}) + (\overline{a_3}|\overline{a_{14}}) + (\overline{a_4}|\overline{a_{14}})\\
&+(\overline{a_3}|\overline{a_{15}}) + a_{17}
\end{aligned}
$$

$$
\begin{aligned}
q[18] \quad = \quad & a_5a_6a_7 + (\overline{a_3}|a_5|\overline{a_6}|\overline{a_7}) + (\overline{a_4}|\overline{a_5}|a_6|\overline{a_7}) + \overline{a_3}a_4a_8\\
&+(\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_8}) + (\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_8}) + (\overline{a_3}|\overline{a_4}|\overline{a_7}|\overline{a_8})\\
&+(\overline{a_4}|\overline{a_5}|\overline{a_7}|\overline{a_8}) + (\overline{a_3}|a_6|\overline{a_7}|\overline{a_8}) + (\overline{a_4}|\overline{a_6}|\overline{a_7}|\overline{a_8})\\
&+(\overline{a_5}|a_9) + (a6\ |\overline{a_9}) + (\overline{a_3}|\overline{a_6}|\overline{a_9}) + (\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_9})\\
&+a_3a_4\overline{a_7}a_9 + \overline{a_3}a_5a_7a_9 + (\overline{a_4}|\overline{a_5}|\overline{a_7}|\overline{a_9}) + a_6a_7a_9 + a_5a_8a_9\\
&+a_4\overline{a_5}a_8a_9 + (\overline{a_3}|a_4|\overline{a_5}|a_6|a_7|a_8|a_9) + a_3\overline{a_4}a_5a_6a_7\overline{a_8}a_9\\
&+\overline{a_3}a_4a_{10} + a_5a_{10} + a_3\overline{a_5}a_{10} + a_3a_4a_5a_{10} + (\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_{10}})\\
&+\overline{a_3}a_5a_6a_{10} + (\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_{10}}) + a_4a_7a_{10} + a_5a_7a_{10} + a_3a_8a_{10}\\
&+(\overline{a_9}\ |\overline{a_{10}}) + a_3\overline{a_4}a_5\overline{a_6}a_7a_8a_9\overline{a_{10}} + (\overline{a_3}|a_4|\overline{a_5}|a_6|\overline{a_7}|\overline{a_8}|a_9|\overline{a_{10}})\\
&+(\overline{a_3}|a_4|\overline{a_5}|a_6|\overline{a_7}|a_8|\overline{a_9}|\overline{a_{10}}) + (\overline{a_3}|\overline{a_5}|a_{11})\\
&+a_4a_6a_{11} + a_5a_6a_{11} + (\overline{a_3}|\overline{a_7}|a_{11}) + (a_4|\overline{a_8}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_{10}}|\overline{a_{11}})\\
&+(\overline{a_3}|\overline{a_4}|a_{12}) + a_4\overline{a_5}a_{12} + a_3a_6a_{12} + (\overline{a_7}|\overline{a_{12}}) + a_3a_4a_{13}\\
&+(a_4|\overline{a_6}|\overline{a_{13}}) + (\overline{a_4}|a_6|a_{13}) + (\overline{a_3}|a_4|\overline{a_{14}}) + (\overline{a_5}|\overline{a_{14}}) + (\overline{a_4}|\overline{a_{15}}) + a_{18}
\end{aligned}
$$

$$
\begin{aligned}
q[19] \quad = \quad & (a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}) + (\overline{a_3}|a_4|\overline{a_5}|\overline{a_7}) + a_3a_4\overline{a_6}a_8 + a_4a_7a_8\\
&+(\overline{a_5}\ |\overline{a_7}|a_8) + a_3a_5\overline{a_7}a_8 + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_7}|\overline{a_8})\\
&+\overline{a_6}a_7a_8 + (\overline{a_3}|\overline{a_6}|\overline{a_7}|\overline{a_8}) + a_5a_6\overline{a_7}a_8 + (\overline{a_3}|\overline{a_5}|\overline{a_6}|\overline{a_9})\\
&+(\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_9}) + a_3\overline{a_7}a_9 + (\overline{a_3}|\overline{a_4}|\overline{a_7}|\overline{a_9})\\
&+(\overline{a_3}|\overline{a_5}|\overline{a_7}|\overline{a_9}) + a_4a_6a_7\overline{a_9} + (\overline{a_3}|\overline{a_4}|\overline{a_8}|\overline{a_9})\\
&+(\overline{a_4}|\overline{a_5}|\overline{a_8}|a_9) + \overline{a_3}a_6a_8a_9 + \overline{a_3}a_6a_{10} + a_3a_4\overline{a_6}a_{10}\\
&+(\overline{a_3}|\overline{a_5}|\overline{a_6}|\overline{a_{10}}) + (\overline{a_3}|\overline{a_4}|\overline{a_7}|\overline{a_{10}}) + (\overline{a_4}|\overline{a_5}|\overline{a_7}|\overline{a_{10}})\\
&+\overline{a_3}a_6a_7a_{10} + a_4a_8a_{10} + \overline{a_3}a_5a_8a_{10} + a_4a_9a_{10} + a_3\overline{a_4}a_9a_{10}\\
&+a_3a_{11} + a_5a_{11} + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_{11}})\\
&+(\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_{11}}) + a_4a_7a_{11} + \overline{a_3}a_5a_7a_{11} + a_4\overline{a_8}a_{11}\\
&+a_3\overline{a_4}a_8a_{11} + (\overline{a_9}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}|a_{10}|\overline{a_{11}})\\
&+(\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_{10}}|a_{11}) + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_{12}})\\
&+a_4a_6a_{12} + \overline{a_3}a_5a_6a_{12} + a_4a_7a_{12} + a_3\overline{a_4}a_7a_{12} + (a_4|\overline{a_8}|\overline{a_{12}})\\
&+a_3\overline{a_4}a_5\overline{a_8}a_{11}\overline{a_{12}} + (\overline{a_3}|a_4|\overline{a_5}|a_8|a_{11}|\overline{a_{12}}) + (a_3|\overline{a_4}|a_6|\overline{a_{13}})\\
&+a_4a_5a_{14} + a_3\overline{a_4}a_5a_{14}
\end{aligned}
$$

$$
\begin{aligned}
q[20] \quad = \quad & a_3a_6\overline{a_7} + (\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_7}) + (\overline{a_3}|\overline{a_4}|\overline{a_8}) + (\overline{a_3}|a_4|\overline{a_5}|\overline{a_8})\\
&+(\overline{a_3}|\overline{a_6}|\overline{a_8}) + (a_4|\overline{a_6}|\overline{a_8}) + (a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_8})\\
&+(\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_7}|\overline{a_8}) + (\overline{a_3}|\overline{a_5}|\overline{a_9}) + a_4a_5a_9 + (\overline{a_3}|\overline{a_6}|\overline{a_9})\\
&+(a_4|\overline{a_6}|\overline{a_9}) + (\overline{a_5}|\overline{a_6}|a_9) + \overline{a_3}a_7a_9 + a_4a_7a_9 + (\overline{a_5}|\overline{a_7}|a_9)\\
&+(\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_7}|\overline{a_9}) + a_6a_7\overline{a_9} + (\overline{a_5}|\overline{a_8}|\overline{a_9})\\
&+a_7a_8a_9 + (a\,5\ |\overline{a_{10}}) + (a_5\ |\overline{a_6}|\overline{a_{10}}) + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_{10}})\\
&+(\overline{a_5}|\overline{a_7}|\overline{a_{10}}) + a_6a_8a_{10} + a_5a_9a_{10} + (\overline{a_3}|\overline{a_4}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_5}|\overline{a_{11}})\\
&+(\overline{a_3}|\overline{a_6}|\overline{a_{11}}) + (\overline{a_5}|\overline{a_6}|\overline{a_{11}}) + a_6a_7a_{11} + a_5a_8a_{11} + a_4a_9a_{11}\\
&+(a_3|\overline{a_{10}}|\overline{a_{11}}) + (\overline{a_3}|a_4|a_{12}) + \overline{a_4}a_5a_{12} + a_5a_7a_{12} + a_4\overline{a_8}a_{12}\\
&+(a_3|\overline{a_9}|\overline{a_{12}}) + a_5a_6a_{13} + a_4a_7a_{13} + (a_3|\overline{a_8}|\overline{a_{13}}) + \overline{a_3}a_4a_{14}
\end{aligned}
$$

$$\begin{aligned}
&+a_4a_6a_{14}+a_7\overline{a_{14}}+a_3a_7a_{14}+a_3a_4a_{15}+a_4a_5a_{15}+(a_3|\overline{a_6}|\overline{a_{15}})\\
&+(\overline{a_3}|a_5|\overline{a_{16}})+(a_3|\overline{a_4}|\overline{a_{17}})
\end{aligned}$$

$$\begin{aligned}
q[21] &= a_{19}+a_{19}+a_{19}+a_{19}+(\overline{a_3}|\overline{a_{17}})+(\overline{a_3}|\overline{a_{17}})+(\overline{a_3}|\overline{a_{17}})\\
&+(\overline{a_3}|\overline{a_{17}})+(\overline{a_4}|\overline{a_{16}})+(\overline{a_4}|\overline{a_{16}})+(\overline{a_4}|\overline{a_{16}})+(\overline{a_4}|\overline{a_{16}})\\
&+(\overline{a_3}|\overline{a_{16}})+(\overline{a_3}|\overline{a_{16}})+(\overline{a_3}|\overline{a_{16}})+(\overline{a_3}|\overline{a_{16}})+(\overline{a_5}|\overline{a_{15}})\\
&+(\overline{a_5}|\overline{a_{15}})+(\overline{a_5}|\overline{a_{15}})+(\overline{a_5}|\overline{a_{15}})+a_3a_{15}+a_3a_{15}+a_3a_{15}\\
&+a_3a_{15}+(\overline{a_6}|\overline{a_{14}})+(\overline{a_6}|\overline{a_{14}})+(\overline{a_6}|\overline{a_{14}})+(\overline{a_6}|\overline{a_{14}})\\
&+(\overline{a_7}|\overline{a_{13}})+(\overline{a_7}|\overline{a_{13}})+(\overline{a_7}|\overline{a_{13}})+(\overline{a_7}|\overline{a_{13}})+a_3a_6a_{13}\\
&+a_3a_6a_{13}+a_3a_6a_{13}+a_3a_6a_{13}+a_4a_5a_{13}+a_4a_5a_{13}+a_4a_5a_{13}+a_4a_5a_{13}\\
&+a_3a_5a_{13}+a_3a_5a_{13}+a_3a_5a_{13}+a_3a_5a_{13}+1+1+1+1+(\overline{a_5}|\overline{a_{16}})+(\overline{a_5}|\overline{a_{16}})\\
q[22] &= 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+a_{20}+a_{20}+a_{20}+a_{20}
\end{aligned}$$

C Reciprocal of Square Root Formulations

The number of rows is 53 and total number of elements is 534.

$$\begin{aligned}
q[0] &= 0\\
q[1] &= 1\\
q[2] &= 1+\overline{a_2}\\
q[3] &= \overline{a_3}\\
q[4] &= a_2a_3+\overline{a_4}\\
q[5] &= \overline{a_3}+a_2a_4+\overline{a_5}\\
q[6] &= 1+(a_3|\overline{a_4})+(\overline{a_2}|\overline{a_3}|\overline{a_4})+a_2a_5+a_3a_5+\overline{a_6}\\
q[7] &= a_3+a_3a_4+(\overline{a_2}|\overline{a_3}|\overline{a_5})+(a_4|\overline{a_5})+(\overline{a_2}|a_6)+(a_3|\overline{a_6})\\
&+\overline{a_7}\\
q[8] &= 1+(a_2|\overline{a_5})+(\overline{a_3}|\overline{a_5})+a_2a_4\overline{a_5}+(\overline{a_2}|\overline{a_3}|\overline{a_6})+a_4a_6\\
&+(a_2|\overline{a_7})+a_3a_7+\overline{a_8}\\
q[9] &= (a_2|\overline{a_3}|\overline{a_4})+(a_2|\overline{a_3}|\overline{a_4}|\overline{a_5})+(\overline{a_2}|a_6)+\overline{a_3}a_6+(\overline{a_2}|\overline{a_4}|\overline{a_6})\\
&+a_5a_6+(\overline{a_2}|\overline{a_3}|\overline{a_7})+(\overline{a_4}|a_7)+(a_2|\overline{a_8})+a_3a_8+\overline{a_9}\\
q[10] &= (a_2|\overline{a_3})+(a_2|\overline{a_3}|\overline{a_4})+a_2a_3a_5+\overline{a_4}a_5+a_3a_4\overline{a_5}+(\overline{a_3}|\overline{a_4}|\overline{a_6})\\
&+(\overline{a_2}|\overline{a_5}|\overline{a_6})+(\overline{a_3}|\overline{a_5}|\overline{a_6})+\overline{a_3}a_7+a_2\overline{a_4}a_7+(a_2|\overline{a_3}|\overline{a_4}|\overline{a_7})\\
&+a_5a_7+(\overline{a_2}|\overline{a_3}|\overline{a_8})+(\overline{a_4}|a_8)+(a_2|\overline{a_9})+a_3a_9+\overline{a_{10}}\\
q[11] &= (\overline{a_2}|\overline{a_5})+a_2a_4\overline{a_5}+(\overline{a_4}|\overline{a_6})+\overline{a_2}a_3a_4\overline{a_6}+a_2a_3a_5a_6\\
&+(a_3|\overline{a_4}|\overline{a_5}|\overline{a_6})+(\overline{a_2}|a_3|\overline{a_4}|a_5|\overline{a_6})+(\overline{a_3}|\overline{a_5}|\overline{a_7})\\
&+(\overline{a_2}|a_4|\overline{a_5}|\overline{a_7})+a_2a_3a_4a_5\overline{a_7}+\overline{a_2}a_6a_7+\overline{a_3}a_8+a_2\overline{a_4}a_8\\
&+a_5a_8+(\overline{a_2}|\overline{a_3}|\overline{a_9})+a_4a_9+(a_2|\overline{a_{10}})+a_3a_{10}+\overline{a_{11}}\\
q[12] &= 1+a_2+a_2a_3a_6+a_2a_4a_6+(\overline{a_2}|\overline{a_5}|\overline{a_6})+a_3\overline{a_4}a_5a_6+(\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_5}|a_6)\\
&+(\overline{a_2}|\overline{a_4}|\overline{a_7})+(a_3|\overline{a_4}|\overline{a_7})+a_2a_3\overline{a_4}a_7+a_2a_3a_5a_7+(a_3|\overline{a_4}|\overline{a_5}|\overline{a_7})\\
&+a_3a_6\overline{a_7}+(\overline{a_3}|\overline{a_4}|\overline{a_8})+(\overline{a_2}|\overline{a_5}|\overline{a_8})+(\overline{a_3}|\overline{a_5}|\overline{a_8})\\
&+a_6a_8+(\overline{a_2}|a_3|\overline{a_4}|a_5|\overline{a_6}|a_7|\overline{a_8})+(\overline{a_3}|\overline{a_9})+a_2\overline{a_4}a_9\\
&+(\overline{a_5}|a_9)+(\overline{a_2}|\overline{a_3}|\overline{a_{10}})+a_4a_{10}+(a_2|\overline{a_{11}})+a_3a_{11}+\overline{a_{12}}
\end{aligned}$$

$$
\begin{aligned}
q[13] \;=\;& (\overline{a_2}|\overline{a_5}) + a_2a_3\overline{a_5} + a_4\overline{a_5} + a_3a_4a_5 + (\overline{a_2}|a_3|\overline{a_4}|a_5)\\
&+\overline{a_6} + (\overline{a_5}|\overline{a_6}) + a_2a_3a_4\overline{a_5}a_6 + (a_3|\overline{a_7}) + a_3a_5a_7 + (\overline{a_2}|\overline{a_4}|\overline{a_5}|\overline{a_7})\\
&+a_2a_6a_7 + (\overline{a_2}|\overline{a_3}|a_6|\overline{a_7}) + (a_3|\overline{a_4}|\overline{a_6}|\overline{a_7}) + (\overline{a_2}|\overline{a_5}|a_6|\overline{a_7})\\
&+(\overline{a_2}|a_3|\overline{a_4}|a_5|a_6|a_7) + (\overline{a_2}|a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|a_7)\\
&+a_4\overline{a_8} + a_2a_3a_5a_8 + (a_3|\overline{a_4}|\overline{a_5}|\overline{a_8}) + (\overline{a_3}|\overline{a_6}|\overline{a_8})\\
&+(\overline{a_2}|a_4|\overline{a_6}|\overline{a_8}) + (\overline{a_2}|a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_8}) + (a_7|\overline{a_8})\\
&+(\overline{a_2}|a_3|\overline{a_7}|\overline{a_8}) + a_2\overline{a_3}a_4\overline{a_5}a_6a_7\overline{a_8} + a_9 + a_2a_4\overline{a_9} + (a_2|\overline{a_3}|\overline{a_4}|\overline{a_9})\\
&+(\overline{a_3}|\overline{a_5}|\overline{a_9}) + (\overline{a_2}|a_4|\overline{a_5}|\overline{a_9}) + \overline{a_2}a_6a_9 + a_2a_3a_6a_9\\
&+(\overline{a_2}|a_3|\overline{a_4}|a_5|a_7|a_9) + a_2\overline{a_3}a_4\overline{a_5}a_6a_7a_9 + (a_2|\overline{a_3}|\overline{a_4}|a_5|a_6|a_7|\overline{a_8}|\overline{a_9})\\
&+(\overline{a_3}|\overline{a_4}|\overline{a_{10}}) + a_5a_{10} + a_2\overline{a_5}a_{10} + a_2a_3a_5a_{10} + a_3\overline{a_{11}}\\
&+\overline{a_2}a_4a_{11} + (\overline{a_2}|\overline{a_3}|a_4|\overline{a_{11}}) + (a_2|\overline{a_{12}}) + \overline{a_{13}}
\end{aligned}
$$

$$
\begin{aligned}
q[14] \;=\;& a_2a_3\overline{a_4} + (a_2|\overline{a_4}|\overline{a_5}) + a_2a_5a_6 + a_3a_4a_5a_6 + (a_2|\overline{a_7}) + a_4a_7\\
&+a_2a_4\overline{a_7} + a_2a_3a_4a_7 + (\overline{a_3}|\overline{a_5}|\overline{a_7}) + (\overline{a_2}|\overline{a_3}|a_5|\overline{a_7})\\
&+a_3a_6a_7 + (\overline{a_5}|\overline{a_6}|\overline{a_7}) + \overline{a_2}\overline{a_3}a_4a_5a_6a_7 + (a_2|\overline{a_4}|\overline{a_8})\\
&+(a_3\ |\overline{a_4}|a_8) + a_2a_3a_4a_8 + a_2\overline{a_5}a_8 + a_3a_5a_8 + a_2\overline{a_3}a_4\overline{a_5a_8}\\
&+(\overline{a_4}|\overline{a_6}|\overline{a_8}) + a_3\overline{a_7}a_8 + a_2\overline{a_3}a_4\overline{a_5}a_6\overline{a_7}a_8 + \overline{a_2}a_3a_4a_5\overline{a_6a_7a_8}\\
&+(\overline{a_2}|a_3|a_4|a_5|\overline{a_6}|a_7|\overline{a_8}) + a_2a_3\overline{a_4}a_5a_6\overline{a_7}a_8 + a_2a_3a_9\\
&+\overline{a_4}a_9 + (\overline{a_4}|\overline{a_5}|\overline{a_9}) + a_3\overline{a_6}a_9 + a_2\overline{a_3}a_4a_5a_6a_9 + a_7a_9 + a_2\overline{a_3}a_4\overline{\overline{a_5a_6a_7a_8a_9}}\\
&+a_2\overline{a_3}a_4\overline{a_5}\overline{a_6}a_7\overline{a_8}a_9 + (a_2|a_3|a_4|\overline{a_5}|\overline{a_6}|\overline{a_7}|\overline{a_8}|\overline{a_9})\\
&+(a_2|\overline{a_3}|\overline{a_4}|a_5|a_6|a_7|a_8|\overline{a_9}) + (a_2|\overline{a_3}|\overline{a_4}|a_5|a_6|a_7|\overline{a_8}|a_9)\\
&+(\overline{a_2}|a_3|\overline{a_4}|\overline{a_5}|a_6|a_7|\overline{a_8}|a_9) + \overline{a_3}a_{10} + (\overline{a_2}|\overline{a_4}|\overline{a_{10}})\\
&+(\overline{a_3}|\overline{a_5}|\overline{a_{10}}) + a_6a_{10} + (\overline{a_2}|a_3|\overline{a_4}|a_5|a_6|\overline{a_{10}}) + a_2\overline{a_3}a_4\overline{a_5}a_6a_7\overline{a_{10}}\\
&+(\overline{a_2}|a_3|\overline{a_4}|a_5|\overline{a_6}|a_7|\overline{a_{10}}) + a_2a_3\overline{a_4}a_5a_6\overline{a_7a_{10}}\\
&+a_2a_{11} + a_3\mathrm{au}_{11} + a_5a_{11} + a_2\overline{a_3}a_4\overline{a_5}a_{11} + \overline{a_2}a_3a_{12} + a_4a_{12}\\
&+(a_2|\overline{a_{13}}) + a_3a_{13}
\end{aligned}
$$

$$
\begin{aligned}
q[15] \;=\;& (\overline{a_2}|\overline{a_5}) + (\overline{a_2}|\overline{a_3}|a_4|\overline{a_5}) + (\overline{a_3}|a_4|\overline{a_6}) + a_2a_4a_5a_6\\
&+a_3\overline{a_4}a_7 + (a_3|\overline{a_5}|\overline{a_7}) + (\overline{a_2}|\overline{a_3}|\overline{a_4}|a_5|\overline{a_7}) + a_2a_3a_6a_7\\
&+a_4a_6\overline{a_7} + a_2a_4a_6a_7 + a_2a_3a_4a_8 + a_2a_3a_5a_8 + (a_2|\overline{a_4}|\overline{a_5}|\overline{a_8})\\
&+a_2a_3a_6a_8 + \overline{a_5}a_6a_8 + a_4\overline{a_7}a_8 + (\overline{a_2}|\overline{a_3}|a_4|\overline{a_5}|a_6|\overline{a_7}|\overline{a_8})\\
&+(\overline{a_2}|\overline{a_3}|a_4|\overline{a_5}|\overline{a_6}|a_7|\overline{a_8}) + (\overline{a_2}|a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_7}|a_8)\\
&+a_2\overline{a_3}a_4a_5a_6a_7a_8 + (\overline{a_2}|a_3|\overline{a_4}|a_5|a_6|a_7|\overline{a_8}) + (\overline{a_2}|\overline{a_3}|a_4|\overline{a_9})\\
&+a_2a_3a_5a_9 + (\overline{a_4}|\overline{a_6}|\overline{a_9}) + (\overline{a_2}|\overline{a_3}|a_4|\overline{a_5}|a_6|a_9)\\
&+(\overline{a_3}|\overline{a_7}|\overline{a_9}) + a_8a_9 + a_2\overline{a_8}a_9 + (a_2|\overline{a_3}|\overline{a_4}|a_5|a_6|\overline{a_7}|\overline{a_8}|\overline{a_9})\\
&+(\overline{a_2}|a_3|\overline{a_4}|a_5|a_6|\overline{a_7}|\overline{a_8}|a_9) + \overline{a_3}a_{10} + (\overline{a_4}|\overline{a_{10}})\\
&+a_2a_3a_4a_{10} + a_4a_5\overline{a_{10}} + (\overline{a_3}|\overline{a_6}|\overline{a_{10}}) + a_7a_{10} + a_2\overline{a_7}a_{10}\\
&+a_2a_3\overline{a_4}a_5a_6\overline{a_7}a_{10} + (a_2|a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_7}|\overline{a_{10}})\\
&+\overline{a_2}a_3a_4a_5a_6\overline{a_7}a_{10} + (\overline{a_2}|a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|a_8|\overline{a_{10}})\\
&+a_2\overline{a_3}a_4a_5a_6a_8\overline{a_{10}} + a_{11} + a_3\overline{a_5}a_{11} + \overline{a_2}a_6a_{11} + (a_2|\overline{a_3}|\overline{a_4}|\overline{a_5}|a_6|\overline{a_7}|a_{10}|a_{11})\\
&+a_3\overline{a_4}a_{12} + a_5a_{12} + a_2\overline{a_5}a_{12} + (\overline{a_2}|a_3|\overline{a_4}|a_5|\overline{a_{12}}) + \overline{a_2}a_4a_{13}\\
&+(a_3|\overline{a_{14}})
\end{aligned}
$$

$$\begin{aligned}
q[16] = 1 &+ (\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_6}) + a_3a_5\overline{a_6} + a_4a_5a_6 + (\overline{a_4}|\overline{a_7}) + (a_2|\overline{a_4}|\overline{a_5}|\overline{a_7}) \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_7}) + \overline{a_5}a_6a_7 + a_2a_5\overline{a_6}a_7 + a_3a_4\overline{a_8} + (\overline{a_2}|\overline{a_5}|\overline{a_8}) \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_8}) + a_6\overline{a_8} + a_3a_6a_8 + (\overline{a_2}|\overline{a_4}|\overline{a_6}|\overline{a_8}) \\
&+a_2a_7a_8 + a_3\overline{a_7}a_8 + (\overline{a_2}|\overline{a_3}|a_7|a_8) + \overline{a_3}a_4a_9 + (\overline{a_3}|a_5|\overline{a_9}) \\
&+a_2a_4\overline{a_5}a_9 + a_2\overline{a_3}a_6a_9 + a_2a_7\overline{a_9} + (\overline{a_2}|\overline{a_4}|\overline{a_{10}}) + a_3a_4a_{10} \\
&+a_2\overline{a_3}a_5a_{10} + a_2a_6\overline{a_{10}} + (a_2|a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|a_7|\overline{a_{10}}) \\
&+a_8a_{10} + \overline{a_3}a_{11} + (\overline{a_4}|\overline{a_{11}}) + a_2\overline{a_3}a_4a_{11} + (\overline{a_2}|\overline{a_5}|\overline{a_{11}}) \\
&+a_7a_{11} + a_{12} + (\overline{a_2}|\overline{a_4}|\overline{a_{12}}) + a_6a_{12} + \overline{a_3}a_{13} + a_2\overline{a_3}a_{13} \\
&+a_5a_{13} + \overline{a_{14}} + \overline{a_{14}} + (a_2|\overline{a_{14}}) + a_2\overline{a_3}a_{14} + a_2\overline{a_3}a_{14} + (a_4|\overline{a_{14}}) \\
&+(a_2|\overline{a_{15}}) + (a_3|\overline{a_{15}}) + (a_2|\overline{a_{16}})
\end{aligned}$$

$$\begin{aligned}
q[17] = \; & a_2\overline{a_4} + (\overline{a_4}|a_5) + a_2a_3a_4\overline{a_5} + a_2\overline{a_5}a_6 + a_4a_5a_6 + (\overline{a_3}|\overline{a_4}|a_5|\overline{a_6}) \\
&+(\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}) + a_2a_4a_7 + \overline{a_5}a_7 + a_2a_3a_5a_7 + (a_2|\overline{a_4}|\overline{a_5}|\overline{a_7}) \\
&+a_3a_4a_5a_7 + a_2a_3a_4a_5a_7 + a_3\overline{a_6}a_7 + a_2a_3a_6a_7 + (\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_7}) \\
&+\overline{a_2}a_5a_6a_7 + a_5\overline{a_8} + (\overline{a_3}|a_5|a_8) + a_2a_3a_5a_8 + a_3a_4a_5\overline{a_8} + a_2\overline{a_6}a_8 \\
&+a_4\overline{a_6}a_8 + a_2a_4\overline{a_6}a_8 + \overline{a_2}a_3a_7a_8 + (a_3|\overline{a_9}) + (\overline{a_2}|\overline{a_4}|\overline{a_9}) \\
&+(\overline{a_2}|\overline{a_3}|a_4|\overline{a_9}) + (\overline{a_2}|\overline{a_5}|\overline{a_9}) + (\overline{a_4}|\overline{a_5}|\overline{a_9}) \\
&+a_2a_4a_5\overline{a_9} + (\overline{a_3}|\overline{a_6}|\overline{a_9}) + a_2a_3a_6\overline{a_9} + (\overline{a_2}|\overline{a_7}|\overline{a_9}) \\
&+(\overline{a_2}|\overline{a_{10}}) + (a_4|\overline{a_{10}}) + (\overline{a_2}|\overline{a_4}|\overline{a_{10}}) + (\overline{a_3}|\overline{a_5}|\overline{a_{10}}) \\
&+(\overline{a_2}|\overline{a_3}|\overline{a_5}|\overline{a_{10}}) + (\overline{a_2}|\overline{a_6}|\overline{a_{10}}) + \overline{a_3}a_4a_{11} + (\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_{11}}) \\
&+(\overline{a_2}|\overline{a_5}|\overline{a_{11}}) + a_{12} + a_2a_3a_{12} + a_2\overline{a_4}a_{12} + (\overline{a_2}|\overline{a_3}|\overline{a_{13}}) \\
&+a_{14} + \overline{a_{15}} + \overline{a_{17}}
\end{aligned}$$

$$\begin{aligned}
q[18] = \; & 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 + (\overline{a_2}|a_3|\overline{a_5}) + a_3a_4a_5 \\
&+(\overline{a_2}|\overline{a_3}|\overline{a_5}|\overline{a_6}) + (\overline{a_2}|\overline{a_3}|\overline{a_7}) + (\overline{a_3}|\overline{a_4}|\overline{a_7}) \\
&+(a_2|a_5|\overline{a_7}) + a_2a_4\overline{a_6}a_7 + a_5a_6\overline{a_7} + a_2\overline{a_4}a_5a_8 + a_2a_3\overline{a_6}a_8 \\
&+a_4a_6\overline{a_8} + \overline{a_3}a_7a_8 + (a_2|\overline{a_4}|\overline{a_9}) + (\overline{a_3}|\overline{a_4}|\overline{a_9}) + a_2a_3\overline{a_5}a_9 \\
&+(\overline{a_4}|\overline{a_5}|\overline{a_9}) + \overline{a_3}a_6a_9 + (a_2|\overline{a_3}|a_{10}) + a_2a_3a_4\overline{a_{10}} + \overline{a_3}a_5a_{10} \\
&+(\overline{a_8}|\overline{a_{10}}) + a_2a_{11} + (\overline{a_3}|\overline{a_4}|\overline{a_{11}}) + (\overline{a_7}|\overline{a_{11}}) + (\overline{a_6}|\overline{a_{12}}) \\
&+\overline{a_{13}} + (\overline{a_2}|\overline{a_{13}}) + a_5\overline{a_{13}} + (\overline{a_2}|\overline{a_{14}}) + (\overline{a_4}|\overline{a_{14}}) + \overline{a_{15}} \\
&+(\overline{a_3}|\overline{a_{15}}) + \overline{a_{16}} + (\overline{a_2}|\overline{a_{16}})
\end{aligned}$$

$$\begin{aligned}
q[19] = \; & 1 + 1 + a_2\overline{a_6} + a_2a_3a_5\overline{a_6} + (\overline{a_2}|\overline{a_4}|\overline{a_5}|a_6) + (\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}) \\
&+a_2a_3a_4a_7 + a_2a_3a_5a_7 + a_2a_3a_6a_7 + (\overline{a_4}|\overline{a_6}|\overline{a_7}) + (a_5|\overline{a_6}|\overline{a_7}) \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_8}) + a_2\overline{a_3}a_4a_8 + a_2a_3a_5a_8 + (\overline{a_4}|\overline{a_5}|\overline{a_8}) \\
&+a_2\overline{a_6}a_8 + (\overline{a_3}|\overline{a_6}|a_8) + (a_5|\overline{a_6}|\overline{a_8}) + a_2a_7a_8 + a_4a_7a_8 \\
&+(\overline{a_2}|\overline{a_3}|a_4|\overline{a_9}) + (\overline{a_2}|\overline{a_5}|\overline{a_9}) + (\overline{a_3}|a_5|\overline{a_9}) + (a_4|\overline{a_5}|\overline{a_9}) \\
&+a_2a_6a_9 + a_4a_6a_9 + a_3a_7a_9 + (a_2|\overline{a_8}|\overline{a_9}) + a_{10} + a_2a_4a_{10} + (\overline{a_3}|\overline{a_4}|a_{10}) \\
&+a_2a_5a_{10} + a_4a_5a_{10} + (\overline{a_3}|a_6|\overline{a_{10}}) + \overline{a_2}a_7\overline{a_{10}} + \overline{a_{11}} + \overline{a_2}a_3\overline{a_{11}} \\
&+(a_2|\overline{a_4}|\overline{a_{11}}) + a_3a_5a_{11} + (a_2|\overline{a_6}|\overline{a_{11}}) + a_2a_{12} + a_3a_4a_{12} \\
&+(a_2|\overline{a_5}|\overline{a_{12}}) + a_3a_{13} + (a_2|\overline{a_4}|\overline{a_{13}}) + (a_2|\overline{a_3}|\overline{a_{14}})
\end{aligned}$$

$$q[20] = (\overline{a_4}|a_6) + (\overline{a_3}|\overline{a_4}|a_6) + a_3a_7 + a_2a_3a_7 + (\overline{a_2}|\overline{a_4}|\overline{a_5}|\overline{a_7})$$

$$
\begin{aligned}
& +(\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_7}) + (\overline{a_2}|\overline{a_4}|\overline{a_6}|\overline{a_7}) + a_3a_4a_6a_7 \\
& +(a_2\,|\overline{a_5}|\overline{a_6}|\overline{a_7}) + (\overline{a_2}|\overline{a_4}|a_8) + \overline{a_5}a_8 + (\overline{a_2}|\overline{a_4}|\overline{a_5}|\overline{a_8}) \\
& +(\overline{a_3}|\overline{a_4}|\overline{a_5}|a_8) + (\overline{a_3}|\overline{a_6}|a_8) + a_2a_3\overline{a_6}a_8 + (a_4|\overline{a_6}|\overline{a_8}) \\
& +(\overline{a_2}|\overline{a_4}|\overline{a_6}|a_8) + a_7a_8 + a_2a_7a_8 + a_3a_7a_8 + a_2a_3a_7a_8 + (\overline{a_3}|\overline{a_5}|a_9) \\
& +(\overline{a_2}|\overline{a_3}|\overline{a_5}|\overline{a_9}) + (a_2|\overline{a_4}|\overline{a_5}|\overline{a_9}) + (a_6|\overline{a_9}) + a_2a_6a_9 \\
& +a_3a_6a_9 + a_2a_3a_6a_9 + a_2a_7a_9 + (a_2|\overline{a_{10}}) + \overline{a_2}a_3a_4a_{10} + a_5a_{10} + a_2a_5a_{10} \\
& +a_3a_5a_{10} + a_2a_3a_5a_{10} + a_6a_{10} + a_2a_6a_{10} + (\overline{a_2}|a_3|\overline{a_{11}}) + a_2a_4a_{11} \\
& +a_3a_4a_{11} + a_2a_3a_4a_{11} + a_2a_5a_{11} + a_2a_4a_{12} + a_{13} + a_2a_3a_{13} + \overline{a_2}a_{15} \\
q[21] \;=\; & (\overline{a_2}|\overline{a_4}|a_5) + a_3a_4a_6 + (a_3|\overline{a_5}|\overline{a_6}) + (\overline{a_2}|a_4|\overline{a_5}|\overline{a_6}) \\
& +a_4a_7 + \overline{a_2}a_3a_4a_7 + a_2a_5a_7 + (a_3|\overline{a_4}|\overline{a_5}|a_7) + (\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_7}) \\
& +\overline{a_6}a_7 + (\overline{a_2}|\overline{a_6}|\overline{a_7}) + (\overline{a_3}|\overline{a_6}|\overline{a_7}) + (\overline{a_2}|\overline{a_3}|\overline{a_6}|a_7) \\
& +\overline{a_2}a_4a_6a_7 + (a_2|\overline{a_3}|\overline{a_8}) + (a_3|\overline{a_4}|\overline{a_8}) + (\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_8}) \\
& +(a_2\,|\overline{a_3}|\overline{a_5}|\overline{a_8}) + \overline{a_2}a_4a_5a_8 + a_2\overline{a_6}a_8 + (\overline{a_2}|\overline{a_3}|\overline{a_6}|\overline{a_8}) \\
& +a_2a_4a_9 + a_2a_3a_4a_9 + (\overline{a_2}\,|\overline{a_5}|\overline{a_9}) + a_2a_3\overline{a_5}a_9 + a_7a_9 + a_2a_7a_9 \\
& +(\overline{a_3}|a_{10}) + \overline{a_4}a_{10} + (\overline{a_2}|\overline{a_3}|\overline{a_4}|\overline{a_{10}}) + (a_2|\overline{a_6}|\overline{a_{10}}) \\
& +(\overline{a_3}|\overline{a_{11}}) + (a_5|\overline{a_{11}}) + a_2a_5a_{11} + (\overline{a_3}|\overline{a_{12}}) + (\overline{a_2}|\overline{a_3}|\overline{a_{12}}) \\
& +(a_4|\overline{a_{12}}) + (\overline{a_2}|a_4|\overline{a_{12}}) + (a_2|\overline{a_3}|\overline{a_{13}}) + (a_2|\overline{a_{14}})
\end{aligned}
$$

References

[AEGP67] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. "The IBM system/360 model 91: floating-point execution unit," *IBM Journal of Research and Development,* 11(1):34–53, Jan. 1967.

[And88] N. Anderson. "Minimum relative error approximations for l/t," *Numerische Mathematik*, 54(2):117–124, 1988.

[BM91] W. B. Briggs and D. W. Matula. "Method and apparatus for performing division using a rectangular aspect ratio multiplier," *U.S. Putent No. 5,046,038*, Sept. 3 1991.

[Boo51] A. D. Booth. "A signed multiplication technique," *Quarterly J. Mech. Appl. Math.*, 4:236–240, 1951.

[Dad65] L. Dadda. "Some schemes for parallel multipliers," *Alta Frequenza*, 34:349–356, May 1965.

[EL89] M. D. Ercegovac and T. Lang. "Fast radix-2 division with quotient-digit prediction," *Journal of VLSI Signal Processing,* 1(3):169–180, Nov. 1989.

[EL90] M. D. Ercegovac and T. Lang. "Simple radix-4 division with operands scaling," *IEEE Trans. Comput.,* 39(9):1204–1208, Sept. 1990.

[Far81] P. M. Farmwald. "On the design of high performance digital arithmetic units," PhD thesis, Dept. Comput. SC., Stanford Univ., 1981.

[Fik66] C. T. Fike. "Starting approximations for square root calculation on IBM system/360," *Comm. ACM,* 9(4):297–299, Apr. 1966.

[Fly70] M. J. Flynn. "On division by functional iteration," *IEEE Trans. Comput., C-19(8):702-*706, Aug. 1970.

[FS89] D. L. Fowler and J. E. Smith. "An accurate, high speed implementation of division by reciprocal approximation," In *Proc. of Ninth Symp. on Comput. Arith.*, pp. 60-67, Santa Monica, CA, Sept. 1989.

[Gol64] R. E. Goldschmidt. "Applications of division by convergence," Master's thesis, M.I.T., June 1964.

[Har68] J. F. Hart et al.. *Computer Approximations*, ch. 6, John Wiley & Sons, New York, 1968.

[Hwa79] K. Hwang. *Computer Arithmetic: Principles, Architecture and Design*, ch. 5-6. John Wiley & Sons, New York, 1979.

[IEEE85] "IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985," The Institute of Electrical and Electronic Engineers, Inc., New York, Aug. 1985.

[Kri70a] E. V. Krishnamurthy. "On optimal iterative schemes for high-speed division," *IEEE Trans. Comput.*, C-19(3):227-231, Mar. 1970.

[Kri70b] E. V. Krishnamurthy. "On range- transformation techniques for division," *IEEE Trans. Comput.*, C-19(2):157--160, Feb. 1970.

[Mac61] 0. L. MacSorley. "High-speed arithmetic in binary computers," *Proc. IRE*, 99:67–91, Jan. 1961.

[Man90a] D. M. Mandelbaum. "A systematic method for division with high average bit skipping," *IEEE Trans. Comput.*, 39(1):127–130, Jan. 1990.

[Man90b] D. M. Mandelbaum. "Some results on a SRT type division scheme," private written communication, submitted for publication, Nov. 1990.

[Man91] D. M. Mandelbaum. "A method for calculation of the square root using combinatorial logic," submitted to *Journal of VLSI Signal Processing*, Mar. 1991.

[Mar90] P. Markenstein. "Computation of elementary functions on the IBM RISC system/6000 processor," *IBM Journal of Research and Development*, 34(1):111–119, Jan. 1990.

[Mat91] D. W. Matula. "Design of a highly parallel IEEE standard floating point arithmetic unit," In *extended abstract from Symp. on Combinatorial Opt.* Sci. *and Tech. (COST),* RUTCOR/DIMACS, Apr. 1991.

[MM91] D. M. Mandelbaum and S. G. Mandelbaum. "Fast generation of logarithms using partitions and symmetric functions," submitted to *IEEE Trans. Comput.*, Oct. 1991.

[Pez71] S. D. Pezaris. "A 40-11s 17-bit by 17-bit array multiplier," *IEEE Trans. Comput.*, C-20:442-447, Apr. 1971.

[RGK72] C. V. Ramamoorthy, J. R. Goodman, and K. II. Kim. "Some properties of iterative square-rooting methods using high-speed multiplication ." *IEEE Trans. Comput.*, C-21(8):837–847, Aug. 1972.

[Rob58] J. E. Robertson. "A new class of digital division methods," *IEEE Trans. Comput.*, C-7:218-222, Sept. 1958.

[Sch89] W. G. Schneeweiss. *Boolean Functions with Engineering Applications and Computer Programs,* ch. 7. Springer-Verlag, New York, 1989.

[SF91a] E. M. Schwarz and M. J. Flynn. "Cost-efficient high-radix division," *Journal of VLSI Signal Processing,* 3(4):293–305, Oct. 1991.

[SF91b] E. M. Schwarz and M. J. Flynn. "Parallel high-radix non-restoring division," submitted to *IEEE Trans. Comput.,* revised June 1992, submitted Oct. 1991.

[SF92a] E. M. Schwarz and M. J. Flynn. "Approximating the sine function with combinational logic," In *Proc. of 26th Asilomar Conf. on Signals, Systems, and Computers,* to be published Oct. 1992.

[SF92b] E. M. Schwarz and M. J. Flynn. "Direct combinatorial methods for approximating trigonometric functions," Technical Report CSL-TR-92-525, Stanford Univ., May 1992.

[SF92c] E. M. Schwarz and M. J. Flynn. "Using a floating-point multiplier to sum signed Boolean elements," Technical Report CSL-TR-92-540, Stanford Univ., Aug. 1992.

[Ste72] R. Stefanelli. "A suggestion for a high-speed parallel binary divider," *IEEE Trans. Comput.,* C-21(1):42–55, Jan. 1972.

[SV91] T. J. Slegel and R. J. Veracca. "Design and performance of the IBM enterprise system/9000 type 9121 vector facility," *IBM J. Res. Develop.,* 35(3):367–381, May 1991.

[Svo63] A. Svoboda. "An algorithm for division," *Information Processing Machines,* Prague, Czechoslovakia, 9:25–32, 1963.

[Toc58] K. D. Tocher. "Techniques of multiplication and division for automatic binary computers," *Quarterly J. Mech. Appl. Math.,* 11:364–384, 1958.

[VSH89] S. Vassiliadis, E. M. Schwarz, and D. J. Hanrahan. "A general proof for overlapped multiple-bit scanning multiplications," *IEEE Trans. Comput.,* 38(2):172–183, Feb. 1989.

[VSS91] S. Vassiliadis, E. M. Schwarz, and B. M. Sung. "Hard-wired multipliers with encoded partial products," *IEEE Trans. Comput.,* 40(11):1181–1197, Nov. 1991.

[WF82] S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital Systems Designers,* ch. 4. CBS College Publishing, New York, 1982.

[WF91] D. C. Wong and M. J. Flynn. "Fast division using accurate quotient approximations to reduce the number of iterations," *In Proc. of the Tenth Symp. on Comput. Arith.,* pp. 191-201, Grenoble, France, June 1991.

[WF92] D. C. Wong and M. J. Flynn. "Fast division using accurate quotient approximations to reduce the number of iterations," *IEEE Trans. Comput.,* 41(8):981–995, Aug. 1992.