Hardware Starting Approximation For The Square Root Operation

Eric M. Schwarz

IBM Enterprise Systems
P.O. Box 950
Poughkeepsie, NY 12602

Michael J. Flynn*

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4055

Abstract

A novel method for obtaining high-precision approximations of high-order arithmetic operations is presented. These approximations provide a starting approximation for high-precision iterative algorithms. An accurate starting approximation translates into few iterations and a short overall latency. The proposed method uses a partial product array to describe an approximation and sums the array on an existing multiplier. By reusing a multiplier the amount of dedicated hardware is very small. For the square root operation, a 16-bit approximation costs less than 1000 dedicated logic gates to implement and has the latency of approximately one multiplication. This is 1/500 the size of an equivalent look-up table method and over twice as many bits of accuracy as an equivalent polynomial method. Thus, a high-precision approximation of the square root operation and many other high-order arithmetic operations such as reciprocal, division, logarithm, exponential, and trigonometric functions are possible at low-cost.

1 Introduction

There are many iterative algorithms for calculating the square root of an operand. One common algorithm for high-order arithmetic operations is the Newton-Raphson algorithm[1, 2, 3]. It is a root solving algorithm. A function is chosen which has a root equal to the desired operation. An example is the square root operation and the function:

$$f(X) = X^2 - A = 0,$$

where A is the input operand. The function's root is equal to $X = \sqrt{A}$ which is the square root operation. The Newton-Raphson algorithm and many other iterative algorithms start with an initial approximation. They then proceed iteratively to create a higher precision approximation. For the Newton-Raphson algorithm the iteration is described by:

$$x_{i+1} = 1/2 * (x_i + A/x_i),$$

where x_i denotes the i-th approximation to the root. The latency of an iteration is one division and one addition. There are several other algorithms for the square root which may have a shorter latency (since they do not require a division operation each iteration[4]) but for simplicity this algorithm is discussed. The proposed method benefits any algorithm which uses a starting approximation and is not restricted to the Newton-Raphson algorithm.

The Newton-Raphson algorithm converges quadratically since the absolute error of the $i+1$ iteration is described by the following:

$$\begin{aligned} \epsilon_{i+1} &= |x_{i+1} - \sqrt{A}| \\ \epsilon_{i+1} &= \epsilon_i^2/(2x_i) \\ \textit{If } 0.25 \leq A &< 1.0 \\ \textit{then } 0.5 \leq x_i &< 1 \\ \textit{and } \epsilon_{i+1} &< \epsilon_i^2. \end{aligned}$$

The error in each iteration's approximation is dependent on the iteration before it. Thus, the precision of any iteration's approximation is dependent on the starting approximation, and its precision determines the number of iterations. This directly affects the latency of the overall algorithm since a slightly better approximation might translate into one less iteration. An iteration for the Newton-Raphson algorithm requires one division and an addition. Thus, a reduction of one iteration has a substantial effect on the overall latency.

A high precision approximation can be very costly or slow. This study proposes a cost-efficient hardware implementation of an approximation to the square root operation. The proposed method costs less than 1000 logic gates, has the latency of one multiplication, and produces an approximation of at least 16 bits correct. The proposed method also applies to many other high-order arithmetic operations such as: reciprocal, division, log, exponential, and trigonometric operations. Thus, a method is presented for creating a high-precision approximation of many high-order arithmetic operations at low-cost.

This study first describes standard methods of creating starting approximations. Then the proposed method is discussed. The implementation of the proposed method is detailed which consists of reusing an existing multiplier to sum a partial product array describing an approximation. The derivation of the partial product array is then described. Results are given

[1] The work is supported by the IBM Resident Study Program and NSF Contract No. MIP88-22961.

1063-6889/93 $03.00 © 1993 IEEE

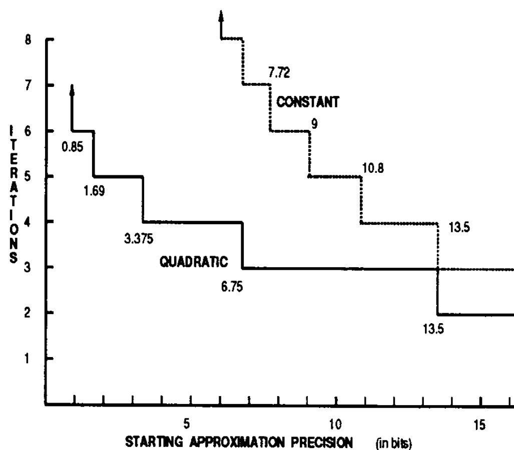

Figure 1: Iterations Versus Starting Approximation

			y_0	y_1	y_2	y_3	$\cdots$
		X	x_0	x_1	x_2	x_3	$\cdots$
				$\vdots$	$\vdots$	$\vdots$	$\cdots$
			y_0x_3	y_1x_3	y_2x_3	y_3x_3	$\cdots$
		y_0x_2	y_1x_2	y_2x_2	y_3x_2	$\cdots$	
	y_0x_1	y_1x_1	y_2x_1	y_3x_1	$\cdots$		
y_0x_0	y_1x_0	y_2x_0	y_3x_0	$\cdots$			
p_0	p_1	p_2	p_3	$\cdots$			

Figure 2: Multiplier's Binary Partial Product Array

for the proposed method as compared to standard methods. Also, differences between this method and that of other researchers are discussed and the contributions are summarized. Thus, this study proposes a low-cost hardware method of creating a high-precision starting approximation.

2 Starting Approximations

The accuracy of the starting approximation has a quantum effect on the number of iterations as shown in Figure 1. Both a quadratically converging algorithm, such as the Newton-Raphson algorithm, and a constantly converging algorithm are plotted for a 53 bit result ($\epsilon_i < 2^{-54}$). Not included in this plot is rounding error that is implementation dependent. The quadratically converging algorithm needs three iterations for an approximation between 6.75 and 13.5 bits, and two iterations for an approximation between 13.5 and 27 bits. There are more steps in the graph of the constantly converging algorithm. A reduction of one iteration has a significant impact on the overall latency.

There is a tradeoff in the accuracy of an approximation and its hardware cost and latency. Three standard methods of approximation are: 1) look-up tables, 2) polynomials, and 3) rationals. A look-up table typically requires a large hardware cost which increases exponentially for each additional bit of precision. Commonly, a look-up table is implemented as a ROM or PLA. Its latency is small even though an off-chip delay may be required. The second method is the polynomial method. It requires very little hardware, since it is typically implemented in software and only its constants are stored in hardware. The polynomial method is slow to converge on an accurate approximation. A rational approximation is even slower but it can converge for a lower order equation than the polynomial method. It requires at least one division operation of latency, and thus, is considered too slow in comparison to the other methods. Additionally, there are combinations of these methods such as using a polynomial with coefficients stored in a look-up table [5, 6]. These methods may require a long latency, and have been traditionally applied to transcendental functions rather than square root. For simplicity the combination of methods is not compared. Thus, the standard method of creating a starting approximation for the square root operation is either to use a look-up table or a polynomial method.

3 Proposed Method

The proposed method is a non-standard hardware method of approximating a high-order arithmetic operation. The proposed method expresses its approximation in the form of a partial product array (PPA). This PPA is summed on an existing multiplier. The result is a low-cost high-precision approximation. The dedicated cost is much lower than an equivalent look-up table and the precision is much greater than a polynomial approximation of equivalent latency. The latency of the proposed method is approximately one multiplication latency and the dedicated hardware is less than 1000 gates. The precision of the proposed method is 16 bits in the worst case which requires one less iteration than most other approximation methods.

The simplest type of PPA is that of a direct binary multiplication as shown in Figure 2. Each binary element of one operand is multiplied by each binary element of the other operand. The array consists of many multiplications of two Boolean variables which can be implemented as logical AND gates. This is shown by the following:

$$P(X,Y) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (x_i \wedge y_j) * 2^{-(i+j)}$$

Each of the columns in the array is weighted by a different power of two. The sum of all these Boolean elements is equal to the product of the two input operands.

The PPA in Figure 2 can be generalized as shown in Figure 3. Each Boolean element (two way ANDing) has been replaced by a generalized Boolean element, $B_{(i,j)}$. This element is a function of an input operand(s) and can be any Boolean logic gate. A four

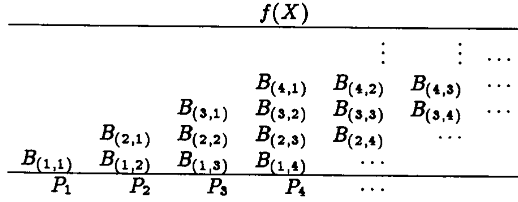

Figure 3: General Binary Partial Product Array

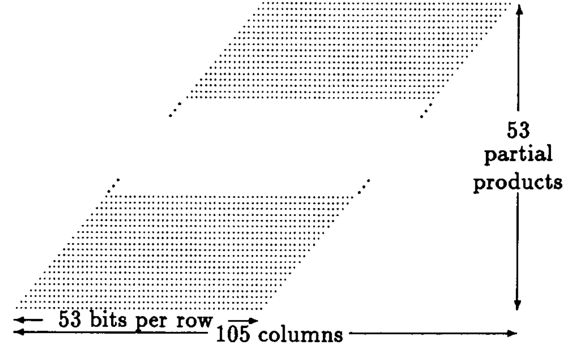

Figure 4: Direct Multiplier's Partial Product Array

way logical OR gate is shown below:

$$
\begin{aligned}
B_{(i,j)}(X) &= B_{(i,j)}(x_0, x_1, x_2, \cdots, x_n) \\
i.e.\ B_{(i,j)}(X) &= (x_0 \mid x_1 \mid x_4 \mid x_5) \\
B_{(i,j)}(X) &\in \{0, 1\}
\end{aligned}
$$

The Boolean elements are chosen such that their summation produces an approximation to the operation being considered:

$$f(X) \approx \sum_{i=1}^{M} \sum_{j=1}^{N} B_{(i,j)}(X) * 2^{-(i+j)+k}$$

where k is a constant which allows the approximation to be a shifted sum. If the shape of the general PPA describing an approximation is chosen to be the same as that of a particular multiplier, then it can be summed by that multiplier. Thus, the proposed method consists of describing an approximation to an operation with a generalized PPA and then using an existing multiplier to sum this PPA.

4 Implementation

The implementation of the proposed method consists of reusing the internal hardware of a multiplier to sum a PPA. A multiplier is used because it is capable of summing Boolean elements, and many arithmetic units have large fast multipliers. Floating-point multiplication is the second most frequently executed floating-point arithmetic operation. Therefore, it is common to implement a full direct multiplication in one iteration. Thus, typical implementations sum a large number of Boolean elements very quickly. This ability to sum Boolean elements is used by the proposed method to create a more accurate approximation than by using the multiplier's ability to perform a generic multiplication. This is shown by comparing the proposed method to a polynomial method with similar latency. Thus, a multiplier's hardware is used effectively for approximations by reusing only its Boolean element summing ability.

In this study, an IEEE 754 format [7] double-precision floating-point multiplier is considered for reuse. The operands of this multiplier are 53 bits. A direct multiplication is assumed which produces a large partial product array as shown in Figure 4. This array consists of 53 rows of 53 elements. Each element of the array is a Boolean element and is equal to the logical AND of two Boolean variables. There are a total of 2809 Boolean elements in the array. This allows for a very large generalized PPA to be mapped onto this PPA. Thus, a total of 53 rows are allowed which is the critical dimension of the PPA.

Multipliers using other formats or algorithms can also be reused. Booth multipliers are common and typically have an array of 27 rows [8, 9, 10]. The same techniques developed for direct IEEE format multipliers can be applied to other multipliers noting the shape and size of their PPA. Results are reported for both 53 and 27 row implementations.

Assuming a generalized PPA can be mapped onto the PPA of a multiplier, it can be summed on the multiplier with some minor adjustments. The adjusted dataflow of the multiplier is shown in Figure 5. To sum the PPA on this multiplier a multiplexor is added above the counter tree and another Boolean element creator is added to evaluate the auxiliary PPA. The multiplexor can simplify into an OR gate if the PPAs are guaranteed to evaluate to all zeros when their operation is not being performed. Thus, the adjustment is: 1) a Boolean element creator is added for each auxiliary PPA and 2) a multiplexor is added above the counter tree.

One issue currently being addressed is how to reduce the wiring complexity. In a standard cell design the added wiring may be acceptable since there are less than 500 wires that need to be added to an area which already has 2809 wires. This only affects the wiring prior to the first counter stage and is less than a 20% addition. For a custom design, the Boolean elements are created in close proximity to the first counter stage. To eliminate a significant number of wires common sub-expression are created in a preliminary stage as shown in Figure 6. Also, it may be possible to reuse some of the wires of the multiplier by placing the input operand into either the multiplicand or multiplier register. Thus, several techniques

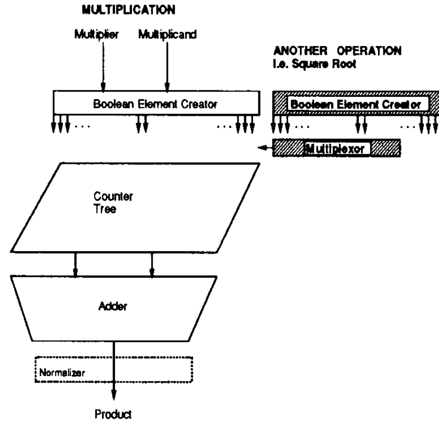

Figure 5: Multiplier Dataflow with Adjustments

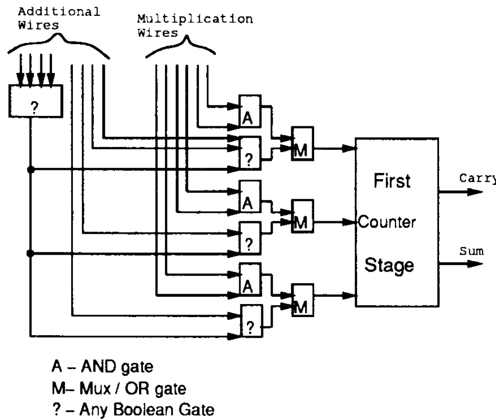

Figure 6: Custom Wiring of Proposed Method

can be used to reduce the wiring complexity.

Other issues are that the critical path is affected slightly and there is a cost in hardware. The hardware cost is approximately equal to two gates per element in the auxiliary PPA. One gate is to evaluate the element in the PPA and the other is for a one bit multiplexor. Also, this is an advantage since this is a very low hardware count as compared to an equivalent look-up table. Other advantages are fewer iterations for the same hardware cost, the approximations can be pipelined, and many approximations can be implemented on the same multiplier. The latency is also good because it is equal to the latency of one multiplication. Thus, the adjustments have been described and they have a low impact on the multiplier and provide a high-precision approximation. The next section describes how to create a PPA that provides a high-precision approximation.

5 Derivation of Proposed Method

The derivation of the PPA is based on a method by Renato Stefanelli [11] which has been enhanced by David Mandelbaum [12, 13, 14, 15] and the authors of this study [16, 17, 18, 19, 20]. The method is presented in two steps which are both described in algorithm form. Algorithm 1 derives a signed PPA which approximates an operation. A signed PPA has Boolean elements with known signs[1]. Algorithm 2 takes the signed PPA and transforms it into an unsigned PPA which has a shape similar to the multiplier's PPA. Thus, a PPA is derived which is summed on a multiplier and produces a high-precision approximation.

5.1 Algorithm 1

Algorithm 1 creates a high-precision approximation in the form of a signed PPA. This is accomplished in six steps:

1. Express operation as a multiplication (or series),
2. Expand multiplication (or series) into a PPA,
3. Back-solve PPA for unknown operand,
4. Form new PPA and reduce,
5. Determine size of approximation which fits the multiplier, and
6. Add error compensating elements and reduce.

These six steps are general steps which can be applied to many high-order arithmetic operations. For this study an example is shown of the square root operation.

The input operand of the square root operation is assumed to be normalized. The normalization is different than usual and is between $0.25 \leq A < 0.5$ and the exponent is assumed to be even. If the exponent is odd, the square root of two is multiplied by the result's magnitude. The new exponent is assumed to be calculated elsewhere in hardware and is calculated as follows:

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

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. This multiplication can have a short latency since the operands are not very wide (slightly over 16 bits). This added delay for half the operands is neglected and a latency of one multiplication is assumed for simplicity.

Algorithm 1 proceeds as follows:

5.1.1 Express operation as a multiplication:

For the square root operation this step is shown below:

$$\begin{aligned} a_0 &= 0, \quad a_1 = 0, \quad a_2 = 1 \\ A &= a_2 * 2^{-2} + a_3 * 2^{-3} + \cdots \\ 0.25 \leq & \; A < 0.5 \end{aligned}$$

[1] $B_{(i,j)}(X)$ is replaced by $s_{i,j} * B_{(i,j)}(X)$ where $s_{i,j}$ is an element of $\{-1, +1\}$.

$$
\begin{aligned}
Q &= q_1 * 2^{-1} + q_2 * 2^{-2} + \cdots \\
0.5 \leq\ Q &< 1/\sqrt{2} = 0.707\cdots \\
A &= Q * Q
\end{aligned} \tag{1}
$$

Equation 1 represents the square root operation as a multiplication.

5.1.2 Expand multiplication into a PPA:

The multiplication is then expanded into a PPA as shown below:

$$
\begin{array}{rcccccc}
 & & 0 & \cdot q_1 & q_2 & q_3 & q_4 \\
X & & 0 & \cdot q_1 & q_2 & q_3 & q_4 \\
\hline
 & & & & q_1q_4 & q_2q_4 & q_3q_4 \\
 & & & q_1q_3 & q_2q_3 & q_3q_3 & q_4q_3 \\
 & & q_1q_2 & q_2q_2 & q_3q_2 & q_4q_2 & \\
 & q_1q_1 & q_2q_1 & q_3q_1 & q_4q_1 & & \\
\hline
 & 1 & a_3 & a_4 & a_5 & \cdots &
\end{array}
$$

5.1.3 Back-solve the PPA:

Q is in a redundant notation and chosen such that the PPA does not have any carries propagating between columns. Each column forms a separate equation. This set of equations is solved for several digits of the unknown operand.

$$q_1 = 1; \ q_2 = a_3/2; \ q_3 = a_4/2 - a_3/8;$$

Any number of unknown operand digits can be back-solved, though their complexity increases exponentially for each less significant digit. This complexity translates into both a larger hardware cost and an increased difficulty to derive the formulations. Mathematica[21] is a software tool which is used to automate the derivation of the less significant digits.

5.1.4 Form a new PPA and reduce:

The equations of the unknown operand digits are placed into a PPA. Each equation forms a separate column since they are weighted by different powers of two. The following is the new PPA for three digits back-solved:

$$
\begin{array}{ccc}
q_1 & q_2 & q_3 \\
 & & \\
 & & -a_3/8 \\
1 & a_3/2 & a_4/2
\end{array}
$$

Since the fractions are powers of two, they are easily represented. For other operations which produce fractions that are not powers of two, a minimal redundant binary notation (with $-1, 0, +1$) is used [15, 19]. For PPAs with more elements, many Boolean and algebraic equivalencies are applied to reduce the PPA (see [22, 17]). This process is automated by a dedicated program written in C language which performs reductions to an element, a column, or the full array. The reduced array is shown below:

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

5.1.5 Determine size of approximation:

The previous step showed that back-solving three digits yields a PPA of one row. Its accuracy is increased by back-solving more digits of Q. This is a coarse adjustment of the precision. Since a direct multiplier with 53 rows is to be used for the implementation an approximation of higher precision is easily possible. The dedicated program for reducing PPAs is used to determine the size for different number of digits back-solved. After many sizings a PPA derived from back-solving 19 digits is chosen. The resulting signed PPA has a maximum of 59 rows but this only occurs in one column. The reshaping step of algorithm 2 brings this PPA under 53 rows. Thus, a signed PPA from back-solving 19 digits of the square root is chosen.

5.1.6 Add error compensating elements:

Back-solving 19 digits of the array causes the first 19 columns to have no carry-outs. But, there can be carries from less significant columns into the back-solved columns. This introduces error into the approximation. In the previous step, the precision is increased by back-solving as many digits as possible. In this step a fine adjustment of the precision is accomplished by adding error compensating elements directly to the PPA. Since the correction is internal, the effect on the size and latency is negligible. Also, the control circuitry is not affected by this type of correction. The error of the previous step's PPA is small on average but has a large worst case error. Since most implementations assume a fixed number of iterations the worst case dictates the latency. Thus, a method is presented for decreasing the worst case error which involves three steps: 1) plot the error, 2) determine correction terms, and 3) add them to the PPA and reduce.

The first substep is to plot the absolute signed error as shown in Figure 7. Error spikes can occur if each successive back-solved digit alternates in sign which results in an instability in the approximation. This sometimes occurs for a pattern of successive "01" in the input operand. This is the case for the spike in the figure which is for the input $0.416015625 = (0.011010101)_2$. The average error from 20 bit simulation is 19.24 bits correct and there is a worst case [2] of 14.00 bits correct. If the error in this region is ignored, the worst case is 16 bits correct ($< 2^{-17}$). A correction term is added to reduce this error. Adding 2^{-16} between $0.40625(0.011010_2)$ and $0.421875(0.011011_2)$, which corresponds to the Boolean element $a_3\overline{a_4}a_5\overline{a_6}$, reduces the worst case error. This element is added to the column of the PPA weighted by 2^{-16} and then the PPA is reduced. This process is a direct manual method of increasing the precision of the approximation.

[2] The worst case is calculated from exhaustive simulation of the model. The bound of values on the square root for each iteration step is subtracted from the model's calculated value to determine the limit on the worst case error. This simulation provides verification of the proposed method.

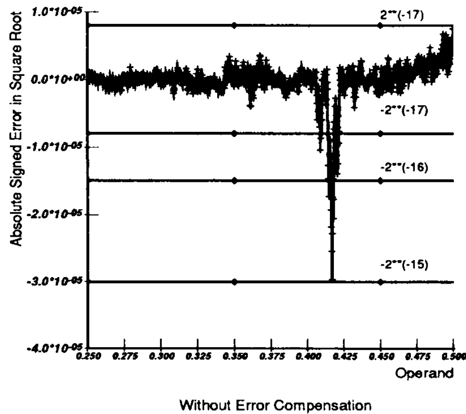

Figure 7: Absolute Signed Error of Approximation

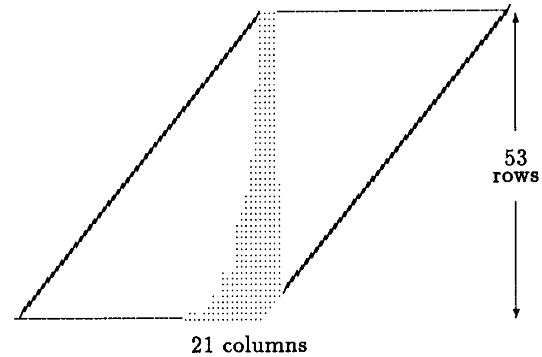

Figure 8: Square Root PPA Superimposed On Multiplier's PPA

Typically the error region is enlarged and replotted. Then more Boolean elements are determined which correct for the worst case error. For this PPA a total of 14 elements are added which improves the approximation to 19.44 bits correct on average and 16.08 bits worst case. Thus, there has been a significant improvement in the worst case error by adding a few elements to the PPA. These elements have been added only to non-critical columns. Any additional correction elements will affect the critical columns and therefore no further correction is applied.

Algorithm 1 has two main variables to increase precision: 1) the number of digits back-solved, and 2) the number of compensating elements to add. The first variable performs a coarse adjustment of the error. The maximum number digits are chosen which forms a PPA that fits within the multiplier's PPA constraints (53 rows maximum). The second variable performs a fine precision adjustment of the worst case error. The regions of large error are corrected by directly adding compensating Boolean elements to the PPA. This step is performed until no new elements can be added which correct for error and fit within the constraints of the multiplier's PPA. Thus, a signed PPA is created which provides a high-precision approximation of an operation. The next step is to transform it into an unsigned PPA by algorithm 2.

5.2 Algorithm 2

Algorithm 2 transforms a signed PPA into an unsigned PPA. This algorithm consists of three steps: 1) complement negative variables and subtract one, 2) sum all constants and add result to PPA, and 3) perform minor shape adjustments of PPA. These steps are very simple as compared to algorithm 1 and are only discussed briefly. Note that these steps are applied by the same program used in algorithm 1 to reduce the PPA. Step one replaces negative elements with their complement and a negative one. This step is valid since $-a_i = \overline{a_i} - 1$. Thus all the negative variables are replaced by their positive complement, and the only negative elements left are constants. Step two sums all the constants and adds the result back to the PPA. If the result is negative the two's complement is added to the PPA. Thus, all the negative elements are eliminated and at most one row is added. This one row is the result of step two and consists only of constants. Step three performs minor reshaping of the array and is discussed in more detail.

There are two methods of adjusting the PPA so that it can be superimposed on a multiplier's PPA. The first is to shift the whole array into a different portion of the multiplier's array. The auxiliary PPA is smaller than the multiplier's PPA, typically less than 20% of the size. Therefore it is sometimes necessary to sum the auxiliary PPA by superimposing it on the middle portion of the multiplier's PPA since this is where the multiplier's PPA has the most rows. Either the normalizer's shift amount is adjusted which only requires a few logic gates, or the result is shifted after the product is formed. This is due to the result being in the middle bits of the product register. The second form of adjustment is to shift one element to a less significant column and to replicate it by the appropriate amount. If a column had 54 rows and only 53 were allowed, one element is shifted to the next less significant column and replicated twice due to the equivalency: $a = (a+a)/2$. Thus, there are two methods of adjusting the shape, the first method performs a coarse shape adjustment and the second alters one element at a time.

The resulting PPA for back-solving 19 digits of the square root is shown in Figure 8. The array requires 398 elements which is less than 15% of the size of the direct multiplier's PPA. The amount of dedicated hardware is equal to approximately two times the total number of elements which is 800 gates. The PPA provides 19.44 bits correct on average and a worst case of 16.08 bits correct. The formulations of each column are provided in the Appendix. Thus, a PPA has been created which provides a high-precision approximation of the square root.

Operation		Proposed		Look-up Table		Size
	Min. Bits	Max. Rows	Eq. Gates	Shape	Eq. Gates	Ratio
Sqrt.	16.08	53	800	$2^{16}x14$	400t	500:1
Sqrt.	13.07	26	470	$2^{13}x11$	39t	83:1
$1/Sqrt.$	13.52	53	1100	$2^{13}x12$	43t	39:1
$1/Sqrt.$	11.16	27	610	$2^{11}x10$	9t	15:1
Recipr.	12.00	53	1000	$2^{13}x11$	39t	39:1
Recipr.	9.17	27	400	$2^{10}x8$	3.6t	9:1

Table 1: Size of Proposed Method Versus Equivalent Look-up Table

Operation		Proposed		Poly.	Accuracy
	Max. Rows	Total Ele.	Min. Bits	Min. Bits	Ratio
Sqrt.	53	398	16.08	7.06	2.28:1
Sqrt.	26	232	13.07	7.06	1.85:1
$1/Sqrt.$	53	534	13.52	5.03	2.69:1
$1/Sqrt.$	27	304	11.16	5.03	2.22:1
Recipr.	53	484	12.00	4.08	2.94:1
Recipr.	27	175	9.17	4.08	2.25:1

Table 2: Accuracy of Proposed Method Versus First Order Polynomial

6 Comparison

The proposed method has been shown for the square root operation. Also, it can be applied to many other operations, a few are shown in Table 1. Sizings of the square root, reciprocal of the square root, and the reciprocal operations are given in this table. The minimum bits correct, maximum number of rows, and equivalent gate counts are detailed for each operation. Also, given is the shape and equivalent gate count($t = 1000$) of a look-up table with the same precision. [3] The size of the proposed method compares favorably to the look-up table since it is between 9 times and 500 times smaller.

Table 2 shows that the proposed method also compares favorably to a first order polynomial (see [23, 24, 25] for coefficients). A first order polynomial is chosen since it has a similar latency, a multiplication and an addition versus an average of 1.5 multiplications for the proposed method. The proposed method provides between 1.85 and 2.94 times the number of bits correct as compared to a similar latency polynomial method. Thus, the proposed method is much more accurate than a polynomial method of equivalent latency.

The proposed method has shown its advantages over both standard types of approximation methods. Also, it is an improvement over non-standard methods from which it is based. Stefanelli [11] created formulations for the reciprocal and division operations and they had linear latency. Each quotient digit of his formulations is dependent on all the more significant quotient digits. His implementation requires a dedicated implementation in the form of a cellular array. Mandelbaum [12, 13, 14, 15] enhanced Stefanelli's method by removing the recursion of the formulations. Instead of each quotient digit being dependent on other quotient digits, substitutions are made and the dependency is removed. His method has log latency since each quotient digit is calculated in parallel. Mandelbaum also requires a dedicated implementation but in the form of a counter tree. He applied his method to several other operations such as square root, log, and exponential. The proposed method differs from both of these researchers in that it reuses a multiplier, each Boolean element is allowed to be more complex, and the precision is enhanced. The implementation of the proposed method reuses an existing multiplier and only a small amount of hardware is dedicated. This is the only method known to reuse the internals of a floating-point multiplier for a non-multiply operation (only other previous uses are for fixed point multiplication [10] and for a multiply-add [26]). Also, each Boolean element of the PPA is allowed to be true or complement and use any type of Boolean gate such as a logical OR gate. The precision has been enhanced by adding error compensating elements and allowing larger PPAs to be implemented due to the smaller dedicated cost of reusing a multiplier. Thus, the proposed method has transformed these high-cost low-precision methods into a low-cost high-precision method.

[3] For simplicity an equivalent precision model is used rather than an equivalent number of iterations since the number of iterations is dependent on whether the high-level algorithm is quadratic or constantly converging.

7 Conclusion

A method has been presented to create a partial product array (PPA) that sums to an approximation of an arithmetic operation. The summation is performed by an existing multiplier to produce a low-cost approximation. Many iterative algorithms use a starting approximation. A higher precision approximation translates into fewer iterations and a shorter overall latency. The proposed method provides a 16 bit approximation which only requires two iterations for a quadratically converging algorithm such as the Newton-Raphson. Thus, the proposed method contributes to the reduction in latency by lowering the number of iterations. The proposed method requires 1/9 to 1/500 the size of an equivalent precision look-up table and provides two to three times the accuracy of an equivalent polynomial approximation. Thus, the proposed method provides a low-cost high-precision approximation to many high-order arithmetic operations such as the square root operation.

References

[1] C. T. Fike, *Computer Evaluation of Mathematical Functions*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1968.

[2] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, John Wiley & Sons, New York, 1979.

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

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

[5] R. C. Agarwal, F. G. Gustavson, J. McComb, and S. Schmidt, "Engineering and scientific subroutine library release 3 for IBM ES/3090 vector multiprocessors," *IBM Systems Journal*, 28(2):345–350, 1989.

[6] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Trans. on Math. Software*, 17(1):26–45, March 1991.

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

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

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

[10] 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.

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

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

[13] D. M. Mandelbaum, "Some results on a SRT type division scheme," *IEEE Trans. Comput.*, 42(1):102–106, Jan. 1993.

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

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

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

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

[18] 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*, Oct. 1992.

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

[20] E. M. Schwarz, "High-radix algorithms for high-order arithmetic operations," PhD thesis, Dept. Elec. Eng., Stanford Univ., available as Tech. Report CSL-TR-93-559, pp. 210, Jan. 1993.

[21] Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, New York, 1988.

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

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

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

[25] N. Anderson, "Minimum relative error approximations for $1/t$," *Numerische Mathematik*, 54(2):117–124, 1988.

[26] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC system/6000 floating-point execution unit," *IBM Journal of Research and Development*, 34(1):59–70, Jan. 1990.

Appendix: Square Root Formulations

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

$$
\begin{aligned}
q[1] &= 0; \quad q[2]=1; \quad q[3]=1+a_3; \quad q[4]=1+a_4; \\
q[5] &= a_5; \quad q[6]=\overline{a_3}+a_6; \quad q[7]=1+(\overline{a_3}|\overline{a_4})+a_7; \\
q[8] &= 1+\overline{a_4}+a_3\overline{a_5}+(\overline{a_4}|\overline{a_5})+a_8; \\
q[9] &= (\overline{a_3}|\overline{a_4})+(\overline{a_3}|\overline{a_6})+(\overline{a_4}|\overline{a_6})+a_9; \\
q[10] &= 1+\overline{a_3}+(a_4|\overline{a_5})+(\overline{a_3}|a_4|\overline{a_5})+(\overline{a_3}|a_4|\overline{a_6}) \\
&\quad +(\overline{a_5}|\overline{a_6})+(\overline{a_3}|\overline{a_7})+(\overline{a_4}|\overline{a_7})+a_{10}; \\
q[11] &= \overline{a_6}+\overline{a_3}a_4a_6+(\overline{a_3}|\overline{a_5}|a_6)+(\overline{a_3}|\overline{a_7})+(\overline{a_5}|\overline{a_7}) \\
&\quad +(\overline{a_3}|\overline{a_8})+a_4\overline{a_8}+a_{11}; \\
q[12] &= a_4a_5a_6+(\overline{a_3}|\overline{a_4}|a_5|a_6)+a_3a_4a_7+a_6\overline{a_7} \\
&\quad +(\overline{a_3}|\overline{a_8})+(\overline{a_5}|\overline{a_8})+(\overline{a_3}|\overline{a_9})+(\overline{a_4}|\overline{a_9})+a_{12}; \\
q[13] &= 1+(a_3|\overline{a_4}|\overline{a_6})+a_5a_6+a_3\overline{a_4}\,\overline{a_5}a_6+a_4a_7 \\
&\quad +a_3a_4a_7+a_3a_5a_7+(\overline{a_4}|\overline{a_5}|a_7)+a_3a_6a_7 \\
&\quad +a_3a_4a_8+a_3a_5a_8+(a_3|\overline{a_6}|\overline{a_8})+(\overline{a_3}|\overline{a_9}) \\
&\quad +(\overline{a_5}|\overline{a_9})+(\overline{a_3}|\overline{a_{10}})+(\overline{a_4}|\overline{a_{10}})+a_{13}; \\
q[14] &= 1+a_3\overline{a_4}a_6+(a_3|\overline{a_7})+(a_3|\overline{a_4}|a_7)+a_5a_7 \\
&\quad +a_3\overline{a_4}a_5a_7+a_4a_6a_7+a_4a_8+a_3a_4a_8+a_4a_5a_8 \\
&\quad +(\overline{a_3}|\overline{a_6}|\overline{a_8})+(\overline{a_7}|\overline{a_8})+a_3a_4a_9+(\overline{a_6}|\overline{a_9}) \\
&\quad +a_3\overline{a_{10}}+a_5\overline{a_{10}}+(\overline{a_3}|\overline{a_{11}})+(\overline{a_4}|\overline{a_{11}})+a_{14}; \\
q[15] &= 1+a_3a_5+a_3a_4\overline{a_6}a_7+\overline{a_4}a_5a_6a_7+a_5a_8
\end{aligned}
$$

$$
\begin{aligned}
&+a_3 a_4 a_5 \overline{a_8} + a_4 a_6 a_8 + a_3 a_7 a_8 + a_4 a_9 + a_3 a_4 a_9 \\
&+a_3 a_5 a_9 + a_4 a_5 a_9 + a_3 a_6 a_9 + (\overline{a_7}|\overline{a_9}) \\
&+a_3 a_4 a_{10} + a_3 a_5 a_{10} + (\overline{a_6}|\overline{a_{10}}) + (\overline{a_3}|\overline{a_{11}}) \\
&+(\overline{a_5}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_{12}}) + (\overline{a_4}|\overline{a_{12}}) + a_{15};
\end{aligned}
$$

$$
\begin{aligned}
q[16] \quad = \quad & a_4 a_5 + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}) + \overline{a_3} a_6 a_7 + (\overline{a_3}|\overline{a_5}|\overline{a_6}|\overline{a_7}) \\
&+(\overline{a_4}|\overline{a_5}|\overline{a_6}|a_7) + a_4 a_8 + a_3 a_5 \overline{a_8} + (a_3|a_6|\overline{a_8}) \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_8}) + \overline{a_3} a_5 a_6 a_8 + (\overline{a_4}|\overline{a_7}|a_8) \\
&+a_3 \overline{a_4} a_5 \overline{a_6} a_7 \overline{a_8} + a_3 \overline{a_4} a_5 a_9 + \overline{a_3} a_4 a_6 a_9 \\
&+a_3 a_7 a_9 + (a_3|\overline{a_8}|\overline{a_9}) + a_4 a_{10} + a_4 a_5 a_{10} \\
&+a_3 a_4 \overline{a_5} a_{10} + (\overline{a_3}|\overline{a_6}|a_{10}) + (\overline{a_7}|\overline{a_{10}}) \\
&+a_3 a_4 a_{11} + (\overline{a_6}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_{12}}) + (\overline{a_5}|\overline{a_{12}}) \\
&+(\overline{a_3}|\overline{a_{13}}) + (\overline{a_4}|\overline{a_{13}}) + a_{16};
\end{aligned}
$$

$$
\begin{aligned}
q[17] \quad = \quad & 1 + a_3 a_5 a_6 + a_3 \overline{a_4} a_8 + (\overline{a_5}|\overline{a_8}) + (\overline{a_3}|\overline{a_5}|\overline{a_8}) \\
&+a_3 a_4 a_5 \overline{a_8} + a_3 a_4 a_5 a_6 a_8 + (\overline{a_5}|\overline{a_7}|a_8) \\
&+(a_3|\overline{a_6}|a_7|\overline{a_8}) + a_3 a_4 \overline{a_6} a_7 \overline{a_8} + a_3 \overline{a_4} a_5 \overline{a_6} a_7 a_8 \\
&+a_3 \overline{a_4} a_5 \overline{a_6}\, \overline{a_7}\, \overline{a_8} + \overline{a_3} a_4 a_9 + a_5 a_9 \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_6}|a_9) + a_5 a_6 a_9 + a_4 a_7 a_9 \\
&+a_3 a_4 a_5 a_7 a_9 + a_3 \overline{a_8} a_9 + a_3 \overline{a_4} a_5 \overline{a_6} a_7 \overline{a_8} a_9 \\
&+a_4 a_6 a_{10} + a_3 a_4 a_5 a_6 a_{10} + a_3 a_7 a_{10} \\
&+(\overline{a_8}|\overline{a_{10}}) + a_4 a_{11} + a_3 a_4 a_{11} + a_3 a_5 a_{11} \\
&+(\overline{a_4}|\overline{a_5}|a_{11}) + a_3 a_6 a_{11} + (\overline{a_7}|\overline{a_{11}}) \\
&+a_3 a_4 a_{12} + a_3 a_5 a_{12} + a_4 a_5 a_{12} + a_6 \overline{a_{12}} \\
&+a_3 \overline{a_{13}} + (\overline{a_5}|\overline{a_{13}}) + (\overline{a_3}|\overline{a_{14}}) + (\overline{a_4}|\overline{a_{14}}) \\
&+(\overline{a_3}|\overline{a_{15}}) + a_{17};
\end{aligned}
$$

$$
\begin{aligned}
q[18] \quad = \quad & a_5 a_6 a_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_4 a_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) + (a_6|\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_3 a_4 \overline{a_7} a_9 + \overline{a_3} a_5 a_7 a_9 \\
&+(\overline{a_4}|\overline{a_5}|\overline{a_7}|\overline{a_9}) + a_6 a_7 a_9 + a_5 a_8 a_9 + a_4 \overline{a_5} a_8 a_9 \\
&+(\overline{a_3}|a_4|\overline{a_5}|a_6|a_7|a_8|a_9) + a_3 \overline{a_4} a_5 a_6 a_7 \overline{a_8} a_9 \\
&+\overline{a_3} a_4 a_{10} + a_5 a_{10} + a_3 \overline{a_5} a_{10} + a_3 a_4 a_5 a_{10} \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_{10}}) + \overline{a_3} a_5 a_6 a_{10} + (\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_{10}}) \\
&+a_4 a_7 a_{10} + a_5 a_7 a_{10} + a_3 a_8 a_{10} + (\overline{a_9}|\overline{a_{10}}) \\
&+a_3 \overline{a_4} a_5 \overline{a_6} a_7 a_8 a_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_4 a_6 a_{11} + a_5 a_6 a_{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_3 a_6 a_{12} \\
&+(\overline{a_7}|\overline{a_{12}}) + a_3 a_4 a_{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_3 a_4 \overline{a_6} a_8 \\
&+a_4 a_7 a_8 + (\overline{a_5}|\overline{a_7}|a_8) + a_3 a_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_7 a_8 + (\overline{a_3}|\overline{a_6}|\overline{a_7}|\overline{a_8}) \\
&+a_5 a_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_4 a_6 a_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_6 a_8 a_9 + \overline{a_3} a_6 a_{10} + a_3 a_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_6 a_7 a_{10} + a_3 a_{11} \\
&+a_4 a_8 a_{10} + \overline{a_3} a_5 a_8 a_{10} + a_4 a_9 a_{10} + a_5 a_{11} \\
&+a_3 \overline{a_4} a_9 a_{10} + (\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_4 a_7 a_{11} + \overline{a_3} a_5 a_7 a_{11} + a_4 \overline{a_8} a_{11} \\
&+a_3 \overline{a_4} a_8 a_{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_4 a_6 a_{12} + \overline{a_3} a_5 a_6 a_{12} + a_4 a_7 a_{12} + a_3 \overline{a_4} a_7 a_{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_4 a_5 a_{14} + a_3 \overline{a_4} a_5 a_{14};
\end{aligned}
$$

$$
\begin{aligned}
q[20] \quad = \quad & a_3 a_6 \overline{a_7} + (\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_7}) + (\overline{a_3}|\overline{a_4}|\overline{a_8}) \\
&+(\overline{a_3}|\overline{a_6}|\overline{a_8}) + (a_4|\overline{a_6}|\overline{a_8}) + (a_3|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_8}) \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_6}|\overline{a_7}|\overline{a_8}) + (\overline{a_3}|\overline{a_5}|\overline{a_9}) + a_4 a_5 a_9 \\
&+(\overline{a_3}|\overline{a_6}|\overline{a_9}) + (a_4|\overline{a_6}|\overline{a_9}) + (\overline{a_5}|\overline{a_6}|a_9) \\
&+a_4 a_7 a_9 + (\overline{a_5}|\overline{a_7}|a_9) + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_7}|\overline{a_9}) \\
&+a_6 a_7 \overline{a_9} + (\overline{a_5}|\overline{a_8}|\overline{a_9}) + a_7 a_8 a_9 + (a_5|\overline{a_{10}}) \\
&+(a_5|\overline{a_6}|\overline{a_{10}}) + (\overline{a_3}|\overline{a_4}|\overline{a_5}|\overline{a_6}|\overline{a_{10}}) \\
&+(\overline{a_5}|\overline{a_7}|\overline{a_{10}}) + a_6 a_8 a_{10} + a_5 a_9 a_{10} \\
&+(\overline{a_3}|\overline{a_4}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_5}|\overline{a_{11}}) + (\overline{a_3}|\overline{a_6}|\overline{a_{11}}) \\
&+(\overline{a_5}|\overline{a_6}|\overline{a_{11}}) + a_6 a_7 a_{11} + a_5 a_8 a_{11} \\
&+a_4 a_9 a_{11} + (a_3|\overline{a_{10}}|\overline{a_{11}}) + (\overline{a_3}|a_4|a_{12}) \\
&+\overline{a_4} a_5 a_{12} + a_5 a_7 a_{12} + a_4 \overline{a_8} a_{12} + (a_3|\overline{a_9}|\overline{a_{12}}) \\
&+a_5 a_6 a_{13} + a_4 a_7 a_{13} + (a_3|\overline{a_8}|\overline{a_{13}}) + \overline{a_3} a_4 a_{14} \\
&+a_4 a_6 a_{14} + a_7 \overline{a_{14}} + a_3 a_7 a_{14} + a_3 a_4 a_{15} \\
&+a_4 a_5 a_{15} + (a_3|\overline{a_6}|\overline{a_{15}}) + (\overline{a_3}|a_5|\overline{a_{16}}) \\
&+(a_3|\overline{a_4}|\overline{a_{17}}) + (\overline{a_3}|a_4|\overline{a_5}|\overline{a_8}) + \overline{a_3} a_7 a_9;
\end{aligned}
$$

$$
\begin{aligned}
q[21] \quad = \quad & 4[1] + 4[a_3 a_5 a_{13}] + 4[a_4 a_5 a_{13}] + 4[a_3 a_6 a_{13}] \\
&+4[(\overline{a_7}|\overline{a_{13}})] + 4[(\overline{a_6}|\overline{a_{14}})] + 4[a_3 a_{15}] \\
&+4[(\overline{a_5}|\overline{a_{15}})] + 4[(\overline{a_3}|\overline{a_{16}})] + 4[(\overline{a_4}|\overline{a_{16}})] \\
&+2[(\overline{a_5}|\overline{a_{16}})] + 4[(\overline{a_3}|\overline{a_{17}})] + 4[a_{19}]; \\
q[22] \quad = \quad & 16[1] + 4[a_{20}];
\end{aligned}
$$

Where $N[a_i]$ denotes a_i summed N times with a_i being a Boolean element.