The Seven Wonders of Computer Arithmetic

Earl E. Swartzlander, Jr.
Department of Electrical and Computer Engineering
University of Texas at Austin, Austin, TX 78712

Abstract

There are many good concepts that have been developed that collectively enable the achievement of incredibly high levels of performance in current computers. In this paper, seven key concepts are identified.

Keywords: Computer arithmetic, two's complement numbers, fast addition, Booth multiplication, fast multiplication, SRT division, Newton-Raphson division, IEEE floating point format.

1. Introduction

Modern computers provide exceptionally high levels of computational performance. While much of the credit is due to the techniques of data manipulation (multi level cache memories, multiple execution threads, etc.) much of the credit belongs to the computational units and the arithmetic algorithms that they implement. This paper presents a candidate selection of the seven most significant developments of computer arithmetic. While few systems employ all seven, most current systems employ most of them.

2. Two's Complement Number System

Three binary number systems (sign/magnitude, one's complement and two's complement) were used for early computers. Actually each of these can be considered to be derivatives of corresponding decimal number systems, i.e., sign/magnitude, nine's complement and ten's complement, respectively. It has been suggested [1] that the ten's complement number system was used to implement subtraction on Pascal's stylus operated adding machines which were constructed starting in 1642.

In the two's complement fractional number system, the value of a number is the sum of n–1 positive binary fractional bits and a sign bit which has a weight of –1. The sign bit is 1 for negative numbers and 0 for positive numbers.

$$A = -a_{n-1} + \sum_{i=0}^{n-2} a_i \, 2^{i-n+1} \tag{1}$$

Table 1 shows the representation of the16 numbers that comprise a four bit fractional two's complement number system. Note that there is a single unique representation of 0 and that the system is not symmetric (i.e., there is a negative number [–1] for which there is no positive equivalent [since the largest positive number is 7/8]).

Table 1. Four-bit Two's Complement Fractions.

VALUE	REPRESENTATION	VALUE	REPRESENTATION
–1	1 0 0 0	0	0 0 0 0
–7/8	1 0 0 1	1/8	0 0 0 1
–3/4	1 0 1 0	1/4	0 0 1 0
–5/8	1 0 1 1	3/8	0 0 1 1
–1/2	1 1 0 0	1/2	0 1 0 0
–3/8	1 1 0 1	5/8	0 1 0 1
–1/4	1 1 1 0	3/4	0 1 1 0
–1/8	1 1 1 1	7/8	0 1 1 1

The two's complement system is attractive because addition and subtraction are simple to implement. Subtraction (or addition of numbers of unlike sign) is performed by adding the two's complement of the subtrahend (or the negative number) to the other number. Determining that overflow occurred (which only happens when adding numbers of like sign is done by comparing the carry into and the carry out from the most significant bit. If the carries match (i.e., both 0 or both 1) no overflow has occurred.

Two's complement numbers are negated by complementing all bits and adding a ONE to the least significant bit position. For example, to form –3/8:

+3/8		0 0 1 1		
invert all bits	=	1 1 0 0		
add 1		0 0 0 1		
		1 1 0 1	=	–3/8
Check:				
invert all bits	=	0 0 1 0		
add 1		0 0 0 1		
		0 0 1 1	=	+3/8

If a two's complement fractional number is reduced in size by truncation (i.e., deleting some number of least significant bits), the resulting number is less than or equal to the original number. This bias occurs because all low order bits (some of which are removed in the truncation process) have positive weight. Thus if many truncated numbers are summed, there may be a significant error due to the accumulation of sub LSB biases.

The two's complement number system is one of the wonders of computer arithmetic because it simplifies addition and subtraction (which are the most frequent arithmetic operations).

3. Carry Lookahead Addition

The baseline adder is a ripple carry adder which is implemented by concatenating n full adders. At the k-th bit position, bits a_k and b_k of the operands A and B and the carry signal from the preceding stage, c_k, are used to generate the k-th bit of the sum, s_k, and the carry to the next adder stage, c_{k+1}. This is called a ripple carry adder, since the carry signals "ripple" from the least significant bit position to the most significant. Since most full adders have 2 gate delays from the carry in to the carry out, an n-bit ripple carry adder requires slightly more than 2n gate delays to produce the most significant sum bit.

The carry lookahead adder was developed by Weinberger and Smith in 1958 [2]. Here specialized logic computes the carries in parallel. The carry lookahead adder uses an adder module (similar to a full adder) for each bit position and lookahead modules, which compute carries for the adder modules. The adder modules form $g_k = a_k b_k$ which indicates that a carry is generated at that bit position and $p_k = a_k + b_k$ which indicates that a carry in to that position will be propagated to the output of that bit position.

$$c_{k+1} = g_k + p_k c_k \tag{2}$$

Extending the concept to subsequent stages:

$$\begin{aligned} c_{k+2} &= g_{k+1} + p_{k+1} c_{k+1} \\ &= g_{k+1} + p_{k+1}(g_k + p_k c_k) \\ &= g_{k+1} + p_{k+1} g_k + p_{k+1} p_k c_k \end{aligned} \tag{3}$$

$$\begin{aligned} c_{k+3} &= g_{k+2} + p_{k+2} c_{k+2} \\ &= g_{k+2} + p_{k+2}(g_{k+1} + p_{k+1} g_k + p_{k+1} p_k c_k) \\ &= g_{k+2} + p_{k+2} g_{k+1} + p_{k+2} p_{k+1} g_k + p_{k+2} p_{k+1} p_k c_k \end{aligned} \tag{4}$$

Although it would be possible to continue this process indefinitely, each additional stage increases the fan-in of the logic gates. Four inputs (as required to implement Equation (4)) is frequently the maximum number of inputs per gate for current technologies. To continue the process, block generate and block propagate signals are defined over four bit blocks (stages k to k+3), $g_{k\text{-}k+3}$ and $p_{k\text{-}k+3}$, respectively:

$$g_{k\text{-}k+3} = g_{k+3} + p_{k+3} g_{k+2} + p_{k+3} p_{k+2} g_{k+1} + p_{k+3} p_{k+2} p_{k+1} g_k \tag{5}$$

and

$$p_{k\text{-}k+3} = p_{k+3} p_{k+2} p_{k+1} p_k \tag{6}$$

Equation (2) can be expressed in terms of the four bit block generate and propagate signals:

$$c_{k+4} = g_{k\text{-}k+3} + p_{k\text{-}k+3} c_k \tag{7}$$

Thus the carry out from a 4-bit wide block can be computed in only four gate delays (the first to compute p_i and g_i for $i = k$ through $k+3$, the second to evaluate $p_{k\text{-}k+3}$, the second and third to evaluate $g_{k\text{-}k+3}$, and the third and fourth to evaluate c_{k+4} using Equation (7)).

A 16-bit carry lookahead adder is shown on Figure 1.

In general the delay of an n-bit carry lookahead adder using r bit lookahead blocks is

$$\text{DELAY}_{\text{CLA}} = 2 + 4\lceil \log_r n \rceil \tag{8}$$

The complexity of an n bit carry lookahead adder is about 1/3 greater than a ripple carry adder.

The carry lookahead adder is one of the wonders of computer arithmetic because it performs addition in time proportional to the logarithm of the word size with only a modest increase in complexity relative to the ripple carry adder.

4. Booth Multiplication

Prior to the development of the Booth algorithm multiplication of two's complement numbers required a correction if the multiplicand is negative, a different correction if the multiplier is negative, and both corrections if both the multiplicand and multiplier are negative [3]. Such data dependent corrections must be accommodated by allowing time for them even in the more common case of multiplying two positive numbers

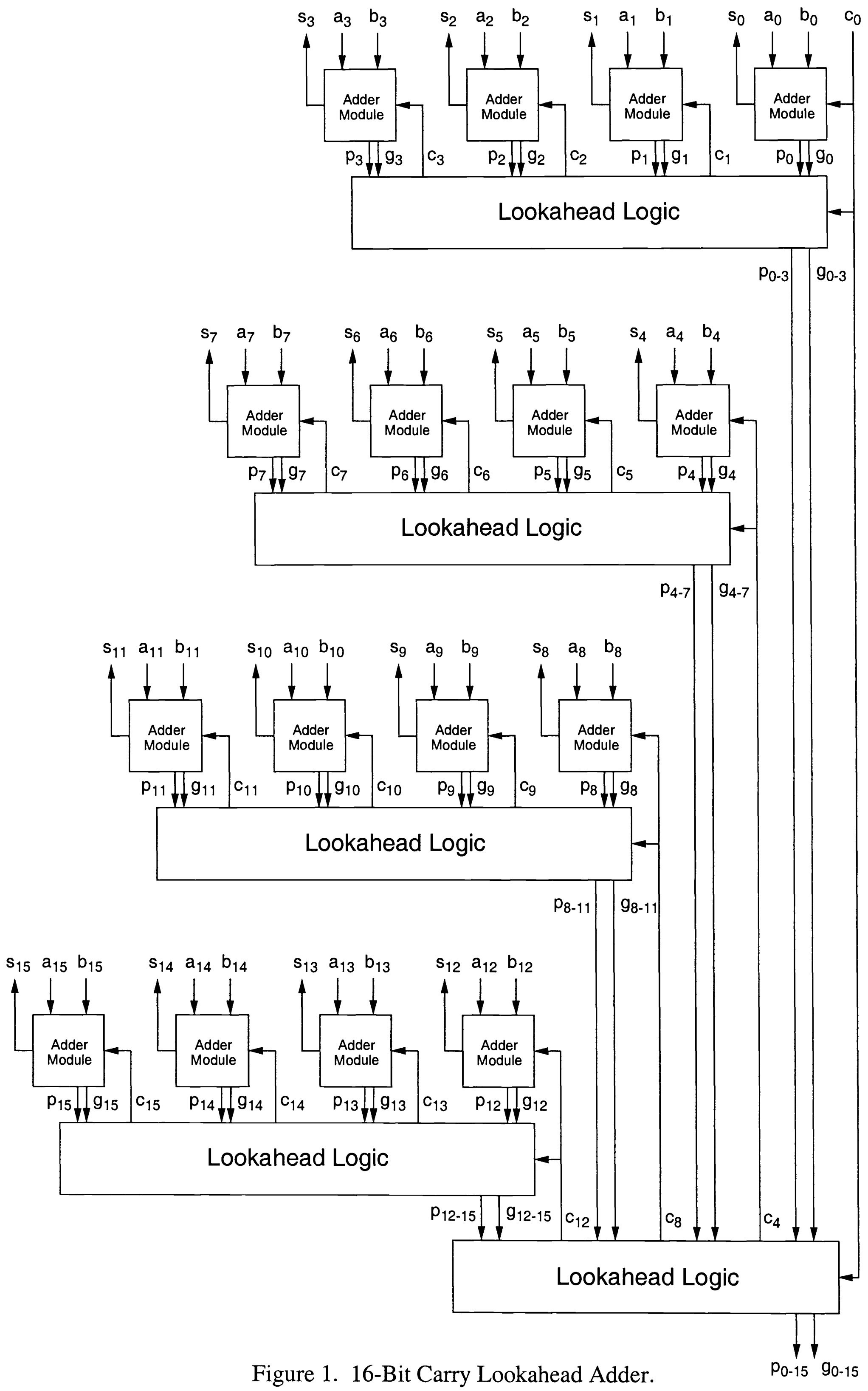
4.1 The Booth Multiplier

The Booth algorithm [4] eliminates the data dependent correction steps. To multiply A B, the product, P, is initially set to ZERO. Then, the bits of the multiplier, A, are examined in pairs of adjacent bits starting with the least significant bit (i.e., $a_0\, a_{-1}$) and assuming $a_{-1} = 0$. Depending on the two bits, actions are taken in accordance with Table 2:

Table 2. Booth Multiplication.

a_i	a_{i-1}	OPERATION	a_i	a_{i-1}	OPERATION
0	0	$P = P/2$	1	0	$P = (P - B)/2$
0	1	$P = (P + B)/2$	1	1	$P = P/2$

The division by two is not performed on the last stage (i.e., when $i = n-1$). All of the divide by two operations are simple arithmetic right shifts (i.e., the word is shifted right one position and the old sign bit is repeated for the new sign bit), and overflows in the addition process are ignored. The Booth multiplier requires n cycles to form the product of a pair of n bit numbers, where each cycle consists of an n bit addition and a shift, an n-bit subtraction and a shift, or a shift without any other arithmetic operation.



s3 a3 b3 s2 a2 b2 s1 a1 b1 s0 a0 b0 c0
Adder Module
p3 g3 c3 p2 g2 c2 p1 g1 c1 p0 g0
Lookahead Logic
p0-3 g0-3
s7 a7 b7 s6 a6 b6 s5 a5 b5 s4 a4 b4
p7 g7 c7 p6 g6 c6 p5 g5 c5 p4 g4
p4-7 g4-7
s11 a11 b11 s10 a10 b10 s9 a9 b9 s8 a8 b8
p11 g11 c11 p10 g10 c10 p9 g9 c9 p8 g8
p8-11 g8-11
s15 a15 b15 s14 a14 b14 s13 a13 b13 s12 a12 b12
p15 g15 c15 p14 g14 c14 p13 g13 c13 p12 g12
p12-15 g12-15 c12 c8 c4
p0-15 g0-15

Figure 1. 16-Bit Carry Lookahead Adder.

In general the delay of an n-bit Booth multiplier using a carry lookahead adder is

$$\text{DELAY}_{\text{Booth}} = n\,(2 + 4\lceil \log_r n \rceil) \qquad (9)$$

4.2 The Modified Booth Multiplier

The radix-4 Modified Booth Multiplier described by MacSorley [5] uses n/2 cycles where each cycle examines three adjacent bits, adds or subtracts 0, B or 2 B and shifts two bits to the right. Table 3 shows the operations as a function of the three bits a_{i+1}, a_i, and a_{i-1}. The radix-4 modified Booth multiplier takes half the number of cycles as the original Booth multiplier although the operations performed during a cycle are slightly more complex (since it is necessary to select one of five possible addends instead of one of three). Extensions to higher radices that examine more than three bits are possible, but generally not attractive because the addition/subtraction operations involve non-power of two multiples (such as 3, 5, etc.) of B which raises the complexity.

Table 3. Radix-4 Modified Booth Multiplication.

a_{i+1}	a_i	a_{i-1}	OPERATION	a_{i+1}	a_i	a_{i-1}	OPERATION
0	0	0	P = P/4	1	0	0	P = (P – 2 B)/4
0	0	1	P = (P + B)/4	1	0	1	P = (P – B)/4
0	1	0	P = (P + B)/4	1	1	0	P = (P – B)/4
0	1	1	P = (P + 2 B)/4	1	1	1	P = P/4

In general the delay of an n-bit radix-4 modified Booth multiplier using a carry lookahead adder is

$$\text{DELAY}_{\text{Mod Booth}} = n\,(2 + 4\lceil \log_r n \rceil)/2 \qquad (10)$$

The Booth multiplier and modified Booth multiplier are wonders of computer arithmetic because they performs two's complement multiplication without requiring data dependent correction steps. The modified Booth multiplier is especially attractive because it reduces the number of cycles by 50%.

5. Fast Multiplication

While the Booth and modified Booth multipliers achieved two's complement multiplication without corrections, they are require time proportional to n Log n to multiply two n bit numbers if the addition is done with a carry lookahead adder. A method for fast multiplication was developed by Wallace [6] and refined by Dadda [7].

With this method, a three step process is used to multiply two numbers: (1) the bit products are formed, (2) the bit product matrix is "reduced" to a two row matrix where the sum of the rows equals the sum of the bit products, and (3) the two numbers are summed with a fast adder to produce the product.

The first step requires a single gate delay as all of the n^2 bit products can be formed in parallel.

The second step is shown for a 6 by 6 Dadda multiplier on Figure 2. Although the matrix shown is for unsigned operands, simple corrections (replacing some AND gates with NAND gates) accommodate two's complement operands [8]. An input 6 by 6 matrix of dots (each dot represents a bit product) is shown as Matrix 0. Columns having more than four dots (or that will grow to more than four dots due to carries) are reduced by the use of half adders (each half adder takes in two dots and outputs one in the same column and one in the next more significant column) and full adders (each full adder takes in three dots from a column and outputs one in the same column and one in the next more significant column) so that no column in Matrix 1 will have more than four dots. Half adders are shown by a "crossed" line in the succeeding matrix and full adders are shown by a line in the succeeding matrix. In each case the right most dot of the pair that are connected by a line is in the column from which the inputs were taken in the preceding matrix for the adder. In the succeeding steps reduction to Matrix 2 with no more than three dots per column, and finally Matrix 3 with no more than two dots per column is performed.

The height of the matrices is determined by working back from the final (two row) matrix and limiting the height of each matrix to the largest integer that is no more than 1.5 times the height of its successor. Each matrix is produced from its predecessor in one adder delay. Since the number of matrices is logarithmically related to the number of rows in matrix 0 which is equal to the number of bits in the words to be multiplied, the delay of the matrix reduction process is proportional to log n.

The third step is to sum the two rows of the final matrix with a fast adder such as a carry lookahead adder. This requires time proportional to log n.

Thus the total delay for the fast multiplier is the sum of a constant and two terms that are proportional to log n giving a total delay that is proportional to the logarithm of the word size.

The modified Booth algorithm can be used as a first step to reduce the height of the initial matrix by 50%. This approach accommodates two's complement operands.

The fast multiplier is a wonder of computer arithmetic since it performs multiplication in time proportional to the logarithm of the data word size..

6. SRT Division

Division is often implemented with a recurrence process that is similar to paper and pencil decimal division. The basic equation is:

$$p_{k+1} = r\, p_k - q_{n-k-1}\, D \qquad (11)$$

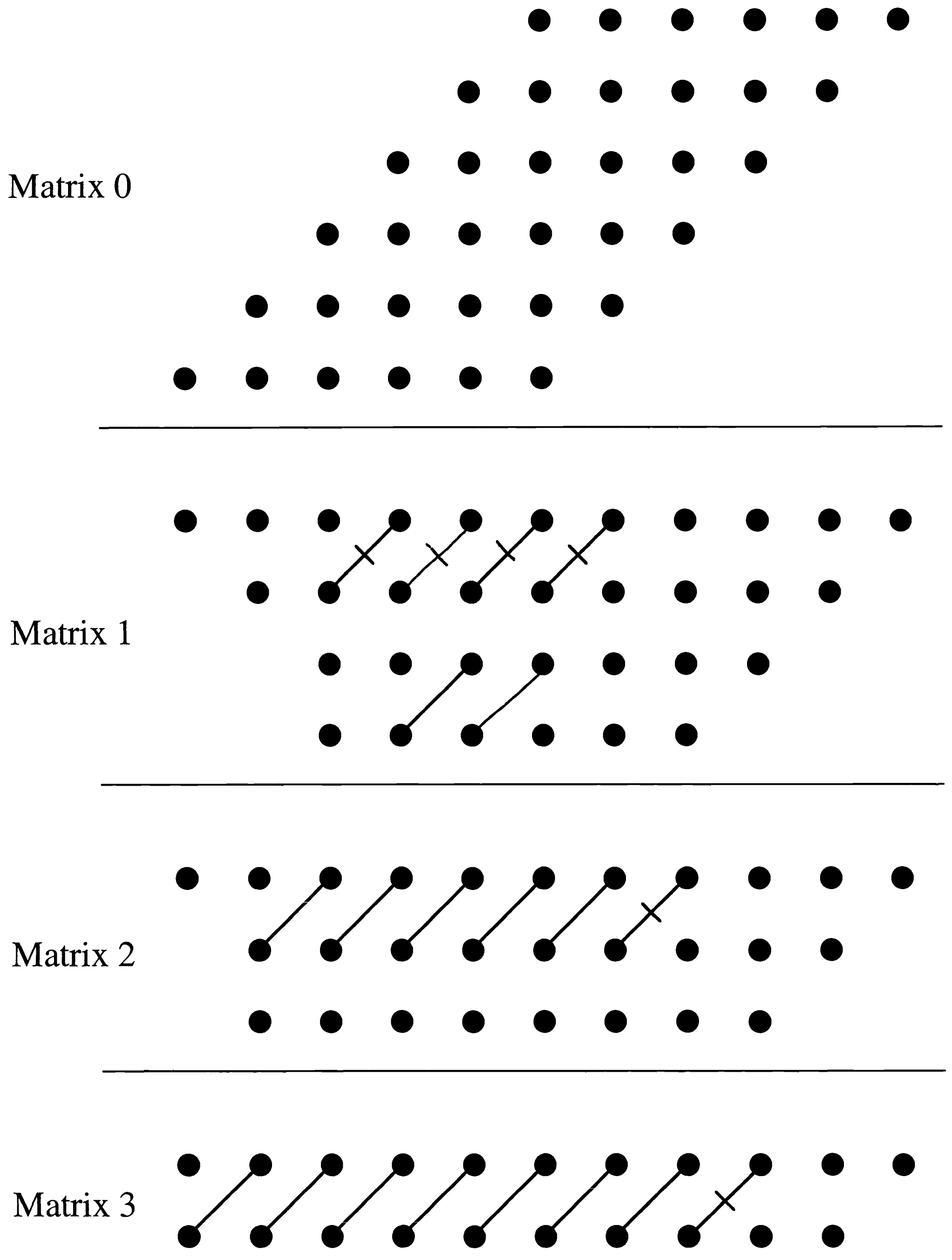


Figure 2. Unsigned 6 by 6 Dadda Multiplier.

Where p_k is the k-th partial dividend (initially p_0 is the dividend), r is the radix (usually a power of 2), q_i is the i-th quotient digit and D is the Divisor. Restoring and non-restoring division as used on early computers is based on this process.

6.1 Binary SRT Division

The SRT division is a digit recurrent algorithm was developed independently by D. Sweeney of IBM, J. E. Robertson [9] and T. D. Tocher [10] in approximately 1958. Binary (i.e., radix = 2) SRT division allows the quotient digits to be selected from {±1, 0}. The divisor is restricted to the range $.5 \leq D < 1$. The digit selection and resulting partial remainder are given for the k-th iteration by the relations in Table 4:

Table 4. Binary SRT Division.

P_k	q_{n-k-1}	P_{k+1}
If $P_k \geq .5$	$q_{n-k-1} = 1$	$P_{k+1} = 2\,P_k - D$
If $-.5 < P_k < .5$	$q_{n-k-1} = 0$	$P_{k+1} = 2\,P_k$
If $P_k \geq .5$	$q_{n-k-1} = -1$	$P_{k+1} = 2\,P_k + D$

Computing an n bit quotient with binary SRT division involves n evaluations of Equation (11). Since each evaluation of Equation (11) requires one subtraction (implemented via addition in two's complement), the quotient (in the {±1, 0} digit set) is obtained with n additions. A single additional n-bit addition is required to convert the result to a two's complement number.

6.2 Radix-4 SRT Division

Radix-4 SRT division is similar to binary SRT division. Radix-4 SRT division algorithm uses either a minimally redundant digit set of {±2, ±1, 0} or the maximally redundant digit sets of {±3, ±2, ±1, 0} The operation of the algorithm is similar to the binary SRT algorithm except that p_k and D are applied to a look up table or a Programmable Logic Array (PLA) to determine the quotient digit. See the text by Ercegovac and Lang [11] for details of the quotient selection process. Computing an n bit quotient with radix-4 SRT division involves n/2 evaluations of Equation (11). Since each evaluation of Equation (11) requires one subtraction (implemented via addition in two's complement), the quotient (in the {±1, 0} digit set) is obtained with n/2 additions. A single additional n-bit addition is required to convert the result to a two's complement number.

The SRT divider is a wonder of computer arithmetic since it is widely used to perform division in current systems.

7. Newton-Raphson Division

A second division technique uses a form of Newton-Raphson iteration to develop a quadratically convergent approximation to the reciprocal of the divisor which is then multiplied by the dividend to produce the quotient. In systems which include a fast multiplier, this process is often faster than conventional division [12].

The Newton-Raphson division algorithm to compute $Q = \frac{N}{D}$ consists of three basic steps:

(1) Calculate a starting estimate of the reciprocal, $R_{(0)}$. If the divisor, D, is normalized (i.e., $1/2 \leq D < 1$), then $R_{(0)} = 3 - 2\,D$ exactly computes $\frac{1}{D}$ at D = .5 and D = 1 and exhibits maximum error (of approximately 0.17) at $D = \sqrt{1/2}$. Adjusting $R_{(0)}$ downward to by half the maximum error gives:

$$R_{(0)} = 2.915 - 2\,D \tag{12}$$

This produces an initial estimate that is within about 0.087 of the correct value for all points in the interval $1/2 \leq D < 1$.

(2) Compute more accurate estimates of the reciprocal by the following iterative procedure:

$$R_{(i+1)} = R_{(i)}\,(2 - D\,R_{(i)}) \quad \text{for: } i = 0, 1, \ldots, k \tag{13}$$

(3) Compute the quotient by multiplying the dividend, N, times the estimate of the reciprocal of the divisor.

$$Q = N\,R_{(k)}, \tag{14}$$

With this algorithm, the error decreases quadratically, so that the number of correct bits in each approximation in step 2 is roughly twice the number of correct bits on the previous iteration. Thus, from a $3\frac{1}{2}$-bit initial approximation, two iterations produce a reciprocal estimate accurate to 14 bits, four iterations produce a reciprocal estimate accurate to 56 bits, etc.

The efficiency of this process is dependent on the availability of a fast multiplier, since each iteration of Equation (13) requires two multiplications and a subtraction. The complete process for the initial estimate, three iterations, and the final quotient determination requires four subtraction operations and seven multiplication operations to produce a 16-bit quotient. This is faster than a conventional non-restoring divider if multiplication is roughly as fast as addition, which is often true for systems which include hardware multipliers.

The Newton-Raphson divider is a wonder of computer arithmetic since it achieves quadratic convergence, which is much faster for large date word sizes than digit recurrent methods.

8. IEEE Floating Point Format

A floating point number, A, consists of a significand (or mantissa), S_a, and an exponent, E_a. The value of a number, A, is given by the equation:

$$A = S_a\,r^{E_a} \tag{15}$$

Where: r is the radix (or base) of the number system. Use of the binary radix (i.e., r = 2) gives maximum accuracy, but may require more frequent normalization than higher radices. Prior to the widespread adoption of the IEEE standard format [13], there were a wide variety of different floating point formats with varying radices, exponent sizes, significand sizes and normalization schemes.

7.1 IEEE Single Precision Format

The IEEE Std. 754 single precision (32-bit) floating point format which is widely implemented, has an 8-bit excess 127 integer exponent which ranges in value between –127 to 128. The two extreme values serve as flags for special cases. The special cases are shown in Table 5. The significand consists of one sign bit and a 24-bit magnitude mixed number (the binary point is to the right of the most significant bit and is always a ONE except for denormalized numbers). Since the most significant bit of the significand is always one it does not need to be stored.

Table 5. IEEE Single Precision Floating Point Numbers.

EXPONENT	SIGNIFICAND	MEANING
128	0	INFINITY
128	Non Zero	Not a Number (NAN)
-126 to 127	Anything	Normalized Number
-127	0	Zero
-127	Non Zero	Denormalized Number

7.2 IEEE Double Precision Format

The IEEE Std. 754 double precision (64-bit) floating point format which is widely implemented, has an 11-bit excess 1023 integer exponent which ranges in value between –1023 to 1024. The two extreme values serve as flags for special cases. The special cases are shown in Table 5. The significand consists of one sign bit and a 53-bit magnitude mixed number (the binary point is to the right of the most significant bit and is always a ONE except for denormalized numbers). Since the most significant bit of the significand is always one it does not need to be stored.

Table 6. IEEE Double Precision Floating Point Numbers.

EXPONENT	SIGNIFICAND	MEANING
1024	0	INFINITY
1024	Non Zero	Not a Number (NAN)
-1022 to 1023	Anything	Normalized Number
-1023	0	Zero
-1023	Non Zero	Denormalized Number

The IEEE floating point is a wonder of computer arithmetic since it has been widely accepted as a standard for computers ranging from the smallest microprocessors to the largest supercomputers.

9. Conclusions

Seven key developments in computer arithmetic have been described. They have been selected because they are widely used and have made a significant impact in improving the performance of computers of all types.

References

1. G. Ifrah, *The Universal History of Numbers, III: The Computer and the Information Revolution,* London, The Harvill Press, 2000, P. 122.
2. A. Weinberger and J. L. Smith, "A Logic for High-Speed Addition," *National Bureau of Standards Circular 591*, 1958, pp. 3-12.
3. R. F. Shaw, "Arithmetic Operations in a Binary Computer," Review of Scientific Instruments, vol. 21, 1950, pp. 687-693.
4. A. D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, 1951, pp. 236-240.
5. O. L. MacSorley, "High-Speed Arithmetic in Binary Computers," *IRE Proceedings*, vol. 49, 1961, pp. 67-91.
6. C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, 1964, pp. 14-17.
7. L. Dadda, "Some Schemes for Parallel Multipliers," *Alta. Frequenza*, vol. 34, May 1965, pp. 346-356.
8. C. R. Baugh and B. A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computers*, vol. C-22, 1973, pp. 1045-1047.
9. J. E. Robertson, "A New Class of Digital Division Methods,"*IRE Transactions on Electronic Computers*, vol. EC-7, 1958, pp. 218-222.
10. K. D. Tocher, "Techniques of Multiplication and Division for Automatic Binary Computers," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 11, 1958, pp. 364-384.
11. M. D. Ercegovac and T. Lang, *Division and Square Root: Digit Recurrence Algorithms and Implementations,* Boston: Kluwer Academic Publishers, 1994.
12. M. J. Flynn, "On Division by Functional Iteration," *IEEE Transactions on Computers*, vol. C-19, 1970, pp. 702-706.
13. *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std. 754-1985, New York: IEEE, 1985.