

A Reconfigurable Co-Processor for Variable Long Precision Arithmetic Using Indian Algorithms

Ranjani Parthasarathi, Easwaran Raman, Karthik Sankaranarayanan, Lakshmi N Chakrapani

School of Computer Science and Engineering
Anna University, Chennai - 600 025, India.
rp@annauniv.edu

Abstract

This paper discusses a set of algorithms for performing Variable Long Precision Arithmetic (VLPA) and their implementation on a reconfigurable hardware. These algorithms are characterized by a high degree of similarity, scalability and massive parallelism. Due to these features, a reconfigurable target provides for a reduced design time, easy scalability and the ability to make a cost performance tradeoff. Moreover, similarity in the data path of various algorithms opens the possibility of using partial reconfiguration to reduce the silicon footprint. Performance figures obtained from the implementation compare favorably with that of the GNU multiprecision library. Analysis shows that performance can be enhanced by a deeper pipeline and scaling the width of the data path, with little increase in design complexity, indicating that these algorithms are a promising choice for implementing on a reconfigurable target.

1. Introduction

Floating point arithmetic is the conventional way for performing arithmetic operations on large numbers. The usage of floating point representation reduces the space required to store large numbers but results in a loss of accuracy. For applications that require a very high degree of accuracy, a class of algorithms called Variable Long Precision Arithmetic (VLPA) algorithms are used [1]. In these algorithms, the numbers are represented using a fixed-point representation. The applications of these algorithms are in the fields of elliptic curve cryptography, computational algebra, computational geometry etc [2]. Several algorithms exist for doing VLP arithmetic [1,3].

A set of algorithms from Indian arithmetic [4,5,6,7] is also suited for VLP arithmetic. They have the following desirable features.

- These algorithms satisfy the on-line property. Online algorithms are those, in which for every digit of input operand, one result digit is obtained, after an initial delay of ' δ ' digits [1].
- The basic operations involved in the Indian arithmetic algorithms are simple operations like digit-by-digit multiplication, two digits by single digit division and additions.
- The nature of the calculations in these algorithms results in less intermediate storage when compared to other VLPA algorithms.
- Further, these algorithms are amenable to parallelism.

A preliminary investigation on the efficiency of the software implementation of these algorithms for division and square root has shown promising results [5,6]. This work has also suggested certain modifications to the basic algorithms, to further improve efficiency. These modifications include the use of general techniques like normalizing the operands, use of the Signed Digit Number System (SDNS), and some which are specific to the algorithms. However, the full power of these modifications can only be realized with specific hardware support. This suggests the need for custom hardware. This paper explores this aspect by designing custom hardware and implementing it on a reconfigurable platform.

The paper is organized as follows. Section 2 provides an overview of the approach used. Section 3 presents the details of the algorithms. Section 4 discusses the design and implementation of the hardware. Section 5 analyses the performance of the proposed implementation in comparison with a standard software library for VLPA. Section 6 concludes the paper.

2. Overview

This paper explores four arithmetic algorithms. These algorithms are for performing multiplication, squaring, division and square root. The multiplication and division algorithms fall under the Urdhva Tiryak (UT) class of algorithms. The squaring and the square-root algorithms fall under the Dwandwa class. The basic operation in the UT class is the calculation of cross-product of two numbers of equal sizes. The cross-product is an operation similar to discrete convolution. The Dwandwa class of algorithms makes use of the Dwandwa operator on a single number. This is essentially a discrete convolution of a number with itself, but requiring half the number of steps as the cross-product (CP).

It may be noted that these algorithms have several common operations. Hence, a common data path, which can accommodate all the algorithms, can be designed. By using the corresponding control unit, the required operation can then be performed. If a reconfigurable platform is used, the control unit can be configured based on the required operation. This is one major reason for considering a reconfigurable architecture. Further, the use of a reconfigurable platform aids scalability of the design. The algorithms are applicable for any radix. Thus, by going for higher radices, the number of steps required for a given computation can be reduced. But this would require changing the basic units. This can be easily achieved in a reconfigurable setup.

One factor to be considered in any use of reconfigurability is the time taken for reconfiguration, which will have an impact on the overall performance. But, for this set of algorithms, the re-use of many components across the algorithms suggests that the entire design need not be reconfigured every time. Using partial reconfiguration, the necessary logic alone may be swapped in, thus reducing the reconfiguration overhead.

3. Algorithms

A. Multiplication

The multiplication algorithm makes use of the cross-product (CP) operator. The CP operator performs a set of single digit multiplications and sums up the individual products. The algorithm to multiply two numbers A and B is given below.

The multiplier A[n] is of size 'n' words and the multiplicand B[m] is of size 'm' words, where A and B are given by

$$A[n] = \sum_{i=0}^{n-1} a_i * x^i \quad (1)$$

$$B[m] = \sum_{i=0}^{m-1} b_i * x^i \quad (2)$$

The product of A and B is given by equation 3.

$$\begin{aligned} P[n+m] &= A[n]*B[m] \\ &= \sum_{i=1}^n CP[0,0,i] * x^{i-1} + \\ &\quad \sum_{j=1}^{m-n} CP[0,j,n] * x^{n+j-1} + \\ &\quad \sum_{k=1}^{n-1} CP[k,k+m-n,n-k] * x^{m+k-1} \end{aligned} \quad (3)$$

$$\begin{aligned} \text{where } CP[n,m,q] &= \sum_{i=n}^{n+q-1} a_i * b_j \\ \text{where } j &= (m+n+q-i-1) \end{aligned} \quad (4)$$

In the algorithm, equation 4 gives the cross-product of two numbers. Equation 3 shows the cross-products being shifted based on their place value and added to give the product of A and B.

B. Squaring

The squaring algorithm makes use of the Dwandwa (D) operator. The algorithm is as follows:

The input to the algorithm is the number A[n] of size 'n' words given by equation 5. The output S of size 2n words is given by equation 6

$$A[n] = \sum_{i=0}^{n-1} a_i * x^i \quad (5)$$

$$\begin{aligned} S[m] &= \sum_{i=1}^m D[0,i] * x^{i-1} \\ &+ \sum_{j=1}^{m-1} D[j, m-j] * x^{m+j-1} \end{aligned} \quad (6)$$

where $m = 2n$ or $2n-1$ and D are given by equations 7a and 7b

$$D[i, 2n] = 2 * \sum_{j=0}^{n-1} a_{i+j} * a_{i+2n-1-j} \quad (7a)$$

$$D[i, 2n+1] = a_{i+n}^2 + 2 * \sum_{j=0}^{n-1} a_{i+j} * a_{i+2n-j} \quad (7b)$$

The equations 7a and 7b give the Dwandwa of a number with even and odd number of digits respectively. The equation 6 calculates the square of the number by shifting the Dwandwa values based on the place value and adding them.

C. Straight Division

The straight division algorithm uses a speculative approach to carry out the division process. This algorithm makes use of the CP operator. The algorithm is as follows:

Inputs

Divisor A: $A[n] = \sum_{i=0}^{n-1} a_{i+1} * x^i$

Dividend B: $B[m] = \sum_{i=0}^{m-1} b_{i+1} * x^i$

Precision : $p \geq m-n+1$

Base : x

Output

Quotient Q: $Q[p] = \sum_{i=m-p+1}^m q_i * x^{i-n}$

Algorithm

S : $S_m S_{m-1} \dots S_{m-p+1}$

D : $d_m d_{m-1} \dots d_{m-p}$

$d_m = b_m$

for $i = m$ to $m-p+1$

$$q_i = d_i / a_n \text{ when } d_i < a_n * x$$

$$= x-1 \text{ otherwise}$$

$$s_i = d_i - q_i * a_n$$

$$d_{i-1} = s_i * x + b_{i-1} - \sum_{j=0}^g q_{i+j} * a_{n-j+1}$$

where $g = \min(n-1, m-i+1) - 1$

loop while $d_{i-1} < 0$

$r = 0$

loop while $q_{i+r} = 0$

$q_{i+r} = x-1$

$r = r + 1$

end while

$$d_{i-1} = d_{i-1} + a_n * x + \sum_{w=0}^r a_{n-w-1}$$

$q_{i+r} = q_{i+r} - 1$

end while

end for

In the division algorithm, during each iteration of the 'for' loop, one digit of the quotient (q_i) is obtained by dividing the Partial Dividend (d_i) with the most significant digit of the divisor (a_n). The remainder obtained in that iteration (s_i) is multiplied by the base (x) and added with the next digit of the dividend (b_{i-1}). From this value, the cross-product obtained in the next iteration ($q_{i+j} * a_{n-j+1}$) is subtracted to get the next Partial Dividend (d_{i-1}). If the Partial Dividend becomes negative, it indicates that the previous quotient digit was higher than what it ought to have been. Then, the previous quotient digit is decremented iteratively by the correction operation (the inner while-loop in the algorithm). This reduces the cross-product and hence increases the Partial Dividend iteratively. The correction operation is continued until the Partial Dividend becomes positive again.

D. Dwandwa Square Root

Inputs

Number $B[m+2] = \sum_{i=0}^{m+1} b_{i+1} * x^i$

where m is even ($m = 2*n$)

Precision $p \geq m/2 + 1$

Base = x

Output

Square Root $A[p] = \sum_{i=m-p+2}^{m+1} a_i * x^{i-n-1}$

Algorithm

S : $S_{m+1} S_m S_{m-1} \dots S_{m-p+2}$

P : $P_{m+1} P_m P_{m-1} \dots P_{m-p+1}$

$P_{m+1} = b_{m+2} * x + b_{m+1}$

$$\begin{aligned}
a_{m+1} &= \lfloor \sqrt{p_{m+1}} \rfloor \\
T &= 2 * a_{m+1} \\
s_{m+1} &= p_{m+1} - a_{m+1}^2 \\
p_m &= s_{m+1} * x + b_m
\end{aligned}$$

for i = m to m-p+2

$$\begin{aligned}
a_i &= p_i / T \quad \text{when } p_i < T * x \\
&= x-1 \quad \text{otherwise} \\
s_i &= p_i - a_i * T \\
p_{i-1} &= s_i * x + b_{i-1} - D[i, m-i+1]
\end{aligned}$$

where $D[i, m-i+1]$ is defined by equations 7a and 7b.

loop while $p_{i-1} < 0$

j = 0

loop while $a_{i+j} = 0$

$$a_{i+j} = x-1$$

$$j = j+1$$

end while

$$a_{i+j} = a_{i+j} - 1$$

$$p_{i-1} = p_{i-1} + T * x + 2 * \sum_{w=0}^j a_{m-w}$$

if $2 * j = m-i$ then

$$p_{i-1} = p_{i-1} - 1$$

end if

end while

end for

In the Dwandwa Square root algorithm, the Most Significant Digit (MSD) of the quotient (a_{m+1}) is given as an input to the algorithm. During every iteration of the 'for' loop, one digit of the square root (a_i) is obtained by dividing the partial dividend (p_i) with twice the value of a_{m+1} (T). The remainder obtained (s_i) is multiplied by the radix (x) and the next input digit (b_{i-1}) is added to this. From this value, the dwandwa of $a_m a_{m-1} a_{m-2} \dots a_i$ ($D[i, m-i+1]$) is subtracted to obtain the next partial dividend. If this becomes negative, the

correction process similar to that of straight division is applied.

4. Design and Implementation

The hardware implementation of the algorithms includes the modifications to the basic algorithms viz. normalization, use of SDNS and look-ahead [5,6]. The MSD of the divisor (a_{m+1} in case of Dwandwa Square root) is normalized [8] to be greater than half the base. The quotient digits produced are allowed to be negative too (use of SDNS)[9]. Finally, in addition to the CP and the Dwandwa, another product called look-ahead is used in determining the Partial Dividend. The look-ahead is the 'next-iteration' CP or Dwandwa except for one product term. So, the look-ahead calculation in an iteration is used for the CP or Dwandwa calculation in the following iteration. Use of look-ahead, together with CP or Dwandwa, gives a better estimate of the quotient digit. These modifications reduce the average number of corrections and hence improve the efficiency of the basic algorithms. The data path of the various units is given in fig1.

The input vectors are stored in the register files A, B and C inside the FPGA. This is only for the ease of implementation. For the algorithm to work, it is enough to supply one digit at a time. For multiplication and squaring, the cross-product of B and C, or the Dwandwa of C is obtained in the register R1 after the required number of cycles. Whenever a cross-product or Dwandwa is obtained, it is added with the previous carry. The unit's digit of this sum is sent to the appropriate position in the vector D. The remaining digits are kept as carry to be added to the next cross-product.

In the division operation, the dividend is in A, the divisor in B and the quotient is obtained in C. For square root, the number is in A and the result is obtained in C. The register R1 stores the cross-product or the Dwandwa. The register R3 stores the partial dividend. For division, R4 stores the MSD of the dividend and for square root, R4 contains twice the MSD of the square root ($T = 2 * a_{m+1}$ as given in the algorithm). The correction unit labeled as CU is used in division and square root.

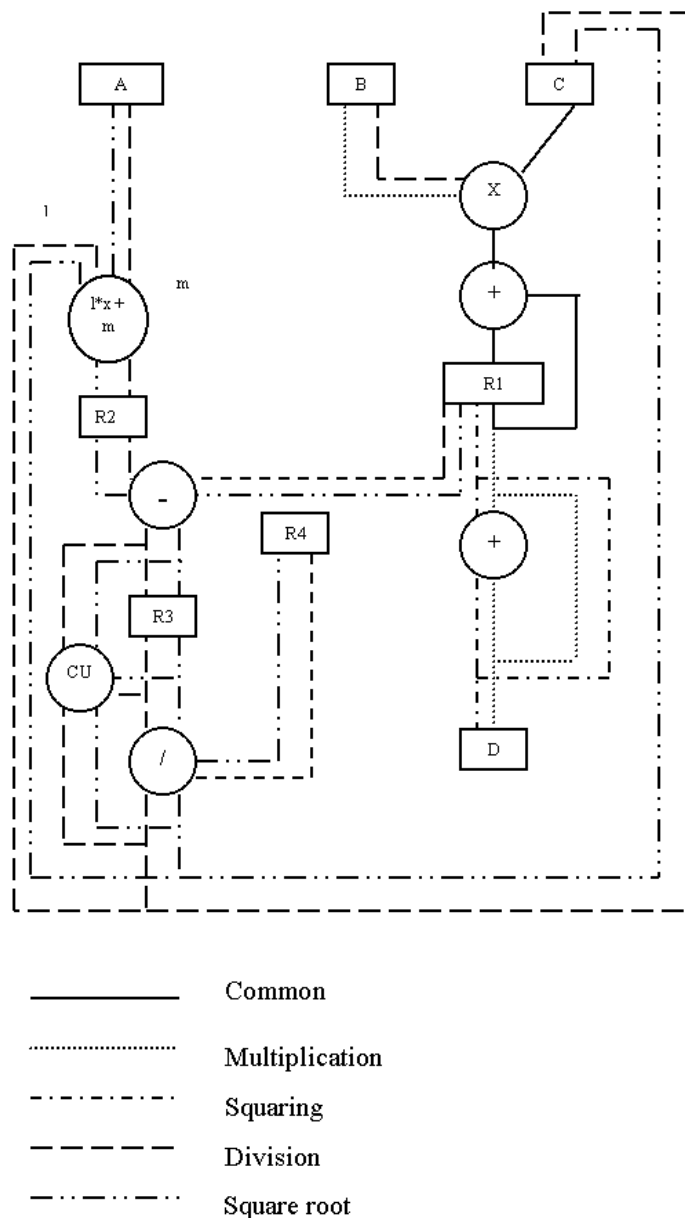


Figure 1
 Combined Data path for the VLPA Units

The hardware unit is designed as a reconfigurable coprocessor, which is controlled by a host processor. The co-processor has been implemented on a Xilinx 4010XL [10,11] FPGA present on an XESS 40xv board, which has a parallel port interface. The XESS 40xv board has been interfaced to the host via the parallel port. Since this implementation was adopted

for ease of validation, the communication costs have not been considered. However a tighter coupling which is ideal for such applications would benefit from negligible IO overhead. The host processor configures the coprocessor for the desired operation and performs the required preprocessing operations such as normalization and conversion to SDNS.

Figure 2 - Division Performance

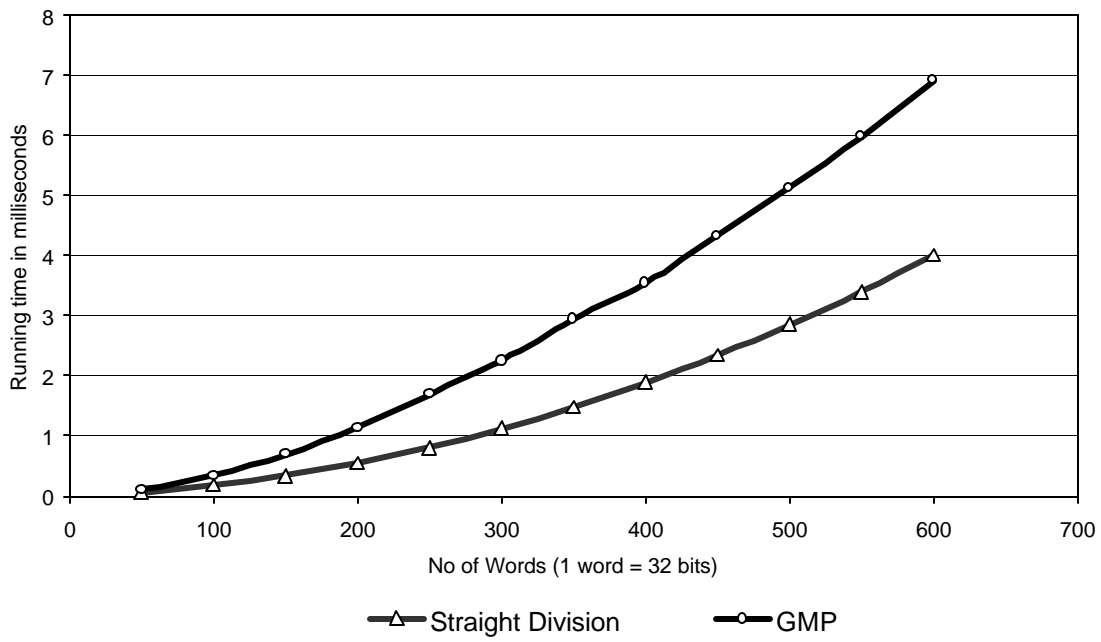
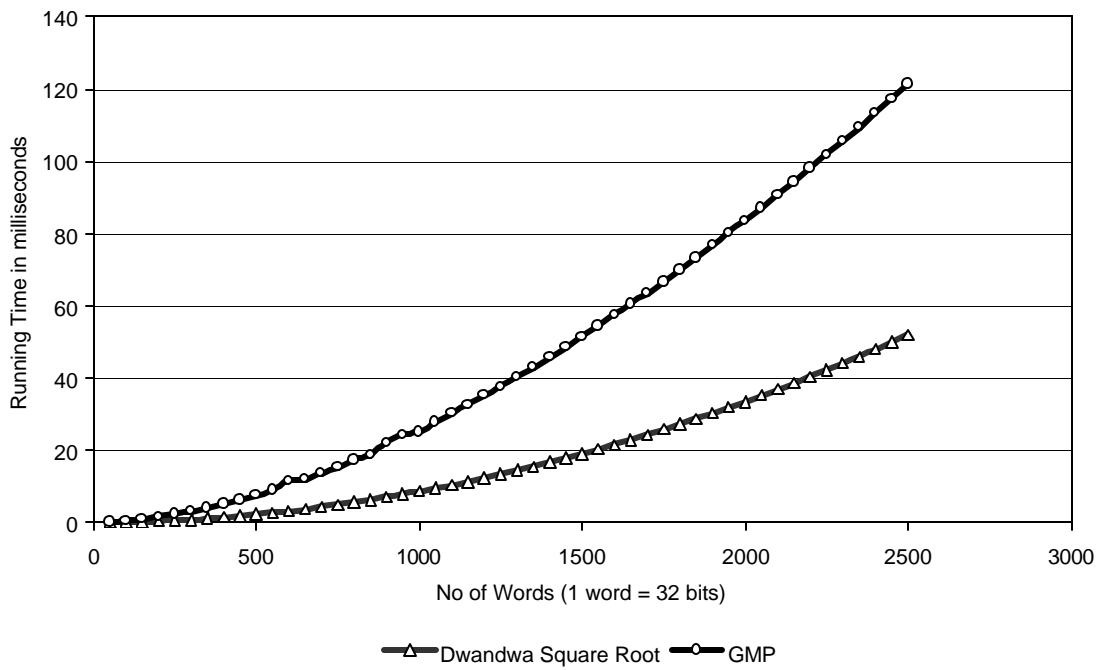


Figure 3 - Square root Performance



After pre-processing, the operands are sent to the coprocessor. For ease of implementation, the operands are stored in the memory configured in the FPGA. The coprocessor performs the configured operation, and stores the results back in memory, and indicates completion to the host. The host can then read out the results

The implementation has been carried out for different values of the radix. A functional verification of the approach was carried out using 4-bit operands, with 1 sign bit, resulting in a radix 8 implementation. These designs fit into the XL4010 device, which has 400 configurable logic blocks (CLBs). The units for multiplication and squaring, being of lower space complexity, radix 128 implementations have been carried out and they also fit into the said device. However, radix 128 division and square-root units require greater number of CLBs. Hence they require a higher capacity device for implementation.

5. Performance estimation

To obtain a comparative estimate of the performance of these designs, a software implementation using GNU-multi-precision library [2] has been carried out for division and square root. In order to have a fair comparison, a 32-bit version of the units has been considered, as the GNU-mp implementation uses 32-bit operations. For obtaining an estimate of the time taken by the hardware implementation, the number of cycles taken by each unit is derived from the Finite State Machine, which controls the corresponding unit. The clock frequency of the two units has been obtained based on the timing simulation run on the Xilinx Foundation express [11] and is found to be 31 MHz for square root and 25.5 MHz for division. Considering the spatial complexity, a Virtex device [10,11] has been chosen as the target device. After mapping the design into a Virtex device, it has been found that the division unit occupies 3285 CLB slices (about 83K gates) and the square root unit occupies 2739 CLB slices (about 66K gates). The GNU-MP based implementation has been run on a Sun Ultra 10 workstation with sparcv9 333MHz processor and 256 MB RAM. The comparative results are shown in fig(s) 2 and 3.

It can be seen that the hardware implementation performs better than the GMP implementation. As the number of digits increases, the speedup for

division is found to vary between 2 to 1.6 whereas for square root it varies from 3.8 to 2.8. This decrease in speedup is because GMP implementation uses Karatsuba multiplication [2,8], which has an asymptotic time complexity of $O(n^{\log 3})$, whereas the Indian algorithms have a complexity of $O(n^2)$. However since these algorithms have sufficient scope for further parallelism, the performance of the hardware can be further improved by exploiting parallelism and aggressive pipelining. This would increase the speedup obtained by a significant value. However, in doing so, the effect of pipelining on the complexity of the design, in terms of number of cycles, should not be neglected. Interestingly, the pipelining factor contributes to an increase in the linear term in the complexity and to a decrease in the coefficient of n^2 term. So, the increase in clock frequency obtained due to pipelining, will outweigh the pipelining overhead (extra cycles due to pipelining), thus leading to a substantial speed-up.

6. Summary and Future work

VLPA finds application in the areas of elliptic curve cryptography, computational algebra, computational geometry etc [2]. This paper outlines a set of VLPA algorithms and their implementation in hardware. Performance figures obtained from such an implementation compare favorably with GNU Multiprecision library. Analysis has shown that further performance gains can be obtained by increasing the width of the data path, increasing the number and the pipeline stages of the execution units. These enhancements would have only a minimal impact on the complexity and the clocks per operation of the hardware. Further, utilizing partial reconfigurability would reduce the silicon footprint of the hardware. These features make the algorithms a promising choice for implementing VLPA in a reconfigurable target. Future work in this direction can explore the implementation of algorithmic enhancements like least remainder division [5] and the effectiveness of partial reconfigurability.

Acknowledgements

The authors would like to express their gratitude to Xilinx Inc., and Xess corp. for providing the foundation series software and the design boards through their university support programs. We would also like to thank the reviewers for providing valuable feedback.

References

- [1] K.S. Trivedi and M.D.Ercegovac, “On-line algorithms for division and multiplication”. IEEE Trans. on Computers C 26 (1977), 681--687. 14
- [2] www.swox.com/gmp - GNU MP home page
- [3] A.F. Tenca and M.D. Ercegovac. “A Variable Long-Precision Arithmetic Unit Design for Reconfigurable Coprocessor Architectures”. In IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998.
- [4] JSS Bharathi Krishna Tirthaji Maharaj - “Vedic Mathematics”, Motilal Banarasi Das, Varanasi, 1986.
- [5] Ranjani Parthasarathi, Ashok Jhunjhunwala - “Modified Straight Division: A Computer implementation of multi-precision division”, Microprocessing and Microprogramming, V41 (1995) 193-209.
- [6] Ranjani Parthasarathi, Ashok Jhunjhunwala - “Multi-precision square root using the Dwandwa square root algorithm”, Journal of Systems Architecture, 44 (1997) 143-158.
- [7] Ashok Jhunjhunwala - “Indian Mathematics - an Introduction”, Wiley Eastern Limited, Madras, 1995.
- [8] D.E.Knuth - “The Art of Compute Programming”, Vol.2 (Semi-numerical algorithms) Addison Wesley, 1980.
- [9] Xilinx data book, Xilinx Inc., 1999
- [10] www.xilinx.com – Xilinx Inc., web site.
- [11] Kai Hwang, “Computer Arithmetic - Principles, Architecture and Design”, John Wiley & Sons Inc. 1979.

Appendix

A.1 Straight division - example

Divide 31689 by 71 - basic algorithm, no corrections and no look ahead.

Let CP be cross product, GD the gross dividend ($s_i * x + b_{i-1}$ in the algorithm), PD the partial dividend, Q the quotient and R the remainder. The abridged divisor is 7 and S is 1.

$\begin{array}{r} 1 \\ 7 \end{array}$	$\begin{array}{r} 31689.000 \\ \underline{3} \\ 4 \\ 32 \\ \underline{ 34} \\ 44 \\ 3244 \\ \underline{ 342} \\ 446 \\ 324423 \\ \underline{ 3422} \\ 446.3 \\ 32442317 \\ \underline{ 34223} \\ 446.32 \end{array}$	<p>Divide 31 by 7 .R = 3 Q = 4. GD= 36.</p> <p>Subtract CP of 4 and 1 from GD=36 to PD =32. Divide 32 by 7 obtain R=4 and Q=4. Next GD=48</p> <p>Subtract CP of 4 and 1 from GD=48 to get PD =44. Divide 32 by 7 obtain R=2 and Q=6. Next GD=29</p> <p>Subtract CP of 6 and 1 from GD=29 to get PD =23. Divide 23 by 7 obtain R=2 and Q=3. Next GD=20</p> <p>Subtract CP of 3 and 1 from GD=20 to get PD =17. Divide 17 by 7 obtain R=3 and Q=2. Next GD=30</p>
---------------------------------------	--	---

The quotient when 31689 divided by 71 is 446.32

A.2 Square root – example

Determine the Square root of 1947 without SDNS and look ahead

Let $GD (s_i * x + b_{i-1})$ denote the gross dividend, PD the partial dividend, Q the quotient and R the remainder. Let LA denote the look ahead and $D(X)$ denote the Dwandwa of X.

8	19 4 7 0 0 0	Integral portion of Square root of 19 is 4. Divisor = $2*4 = 8$
	3	
	4	
	34	
8	19 4 7 0 0 0	GD = 34, PD = 34 Q = 4 R = 2
	3 2	
	4 4	
	34 11	
8	19 4 7 0 0 0	GD = 27 PD = $27 - D(4) = 11$ Q = 1 R = 3
	3 2 3	
	4 4 1	
	34 11 22	
8	19 4 7 0 0 0	GD = 30 PD = $30 - D(41) = 22$ Q = 2 R = 6
	3 2 3 6	
	4 4. 1 2	

Square root of 1947 is 44.12