VEDIVISION – A FAST BCD DIVISION ALGORITHM FACILITATED BY VEDIC MATHEMATICS

Diganta Sengupta[1] Mahamuda Sultana[2] and Atal Chaudhuri[3]

[1]Department of Applied Electronics and Instrumentation Engineering, Future Institute of Engineering and Management, Kolkata, India
sg.diganta@gmail.com
[2]Department of Computer Science and Engineering, Swami Vivekananda Institute of Science and Technology, Kolkata, India
sg.mahamuda3@gmail.com
[3]Department of Computer Science and Engineering, Jadavpur University, Kolkata, India
atalc23@gmail.com

ABSTRACT

Of the four elementary operations, division is the most time consuming and expensive operation in modern day processors. This paper uses the tricks based on Ancient Indian Vedic Mathematics System to achieve a generalized algorithm for BCD division in a much time efficient and optimised manner than the conventional algorithms in literature. It has also been observed that the algorithm in concern exhibits remarkable results when executed on traditional mid range processors with numbers having size up to 15 digits (50 bits). The present form of the algorithm can divide numbers having 38 digits (127 bits) which can be further enhanced by simple modifications.

KEYWORDS
Division Algorithm, Fast BCD Division, Vedic Division Algorithm.

1. INTRODUCTION

The Ancient Indian Vedic Mathematics comprises of sixteen Sutras and thirteen corollaries [1] [2]. The four elementary operations of a processor, the addition, subtraction, multiplication and the division have been extensively dealt with in the sixteen sutras of Vedic Mathematics. The work in this paper involves the *Nikhilam* and the *Paravartya* Sutra which deal with the division. The *Nikhilam* Sutra also deals with multiplication and some amount of work has been done using another sutra known as the *Urdhva Tiryakbhyam* Sutra [3] [4] [5] [6] [7] [8]. The novelty of the *Vedivision* lies in the fact that the procedure incorporates addition and negation operations, both of which are much faster than the traditional successive subtraction methods. The *Nikhilam* Sutra can be stated as follows:

1.1. The *Nikhilam* Sutra

1. This Sutra breaks up the dividend into two parts, one part resembling the Quotient and the other part resembling the Remainder. The number of digits in the Remainder part equals the number of digits in the divisor. For example, if the dividend and divisor are 2002002 and 89998 respectively, then 2002002 is broken up into two parts, 20 (part 1) and 02002 (part 2).
2. The next step in the Sutra adjusts the divisor by complimenting it using the procedure "*subtract all from 9 and the last from 10*" in which all the digits in the divisor are

DOI : 10.5121/ijcsit.2013.5405

subtracted from 9 barring the last significant digit which is subtracted from 10.Therefore, the divisor 89998 after adjustment becomes 10002.

3. Next, the first digit of the quotient part (part 1 of the dividend) is divided by the first digit of the actual divisor. This is the only step where division is unavoidable, but in this case also the maximum division is that of a single digit number by a single digit number. In our work a look-up table has been created which stores all the single digit division results in the form of quotient and remainder, and accessed on demand. The first digit of the quotient part '2'is divided by '8' (the first digit of the actual divisor) and the remainder '2' is noted down.
4. In this step, the remainder from the previous step is first written as the most significant digit of the quotient part and then it is multiplied by the adjusted divisor and placed below the dividend after shifting it one place right. This multiplication can be achieved by either the *Nikhilam* Sutra or the *Urdhva Tiryakbhyam* Sutra.

89998	)	2002002	
10002	)	20 | 02002	: Divisor adjustment
		2 | 0004	
		2**2** | …….	

5. In the above example, we observe that after the single digit multiplication, the digits in the quotient part (part 1) of the dividend have been added. Now the adjusted divisor is again multiplied by the new digit (marked by bold and underlined in example of step-4) to obtain another number and place it again by shifting it to the right by one position as shown below in Example-1:

89998	)	2002002	
10002	)	20 | 02002	: Divisor adjustment
		2 | 0004	: Result after Step 4
		| 20004	: Result after Step 5
		22 | 22046	

Example 1.

Here it can be seen that since the result after step 5 has entered fully into the remainder part (part 2); hence the algorithm is concluded by adding all the intermediate results. Therefore, after dividing 2002002 by 89998, we get the quotient as 22 and the remainder as 22046.

Observations.

It can be concluded that no subtraction procedure is performed in the entire division process. In the third step of the sutra, a single digit division has been done but that can be performed using a look-up table. The division process has been performed with multiplication process in subsequent steps and addition. Multiplication is a relatively faster and cheaper operation than division. Also the largest multiplication that may be required is multiplying 9 by 9.

Drawback of the *Nikhilam* Sutra

The sutra provides best results when division requires large divisors. In cases where the divisor is a small number, this sutra provides ambiguous results. This drawback is accomplished by another sutra known as the *Paravartya* Sutra.

1.1. The *Paravartya* Sutra

The *Paravartya* Sutra is suitable for divisions including large as well as small divisors. The sutra is actually known as "*Paravartya Yojayet*" which means "Transpose and Apply". The *Paravartya* Sutra can be easily explained using the famous *Remainder Theorem* [9] as follows:

1. If E = Dividend, D = Divisor, Q = Quotient and R = Remainder and if the divisor is taken to be (x-p), then a relationship can be stated as follows:
$$E = D.Q + R, \quad \text{or} \quad E = Q.(x-p) + R.$$

2. Now, if 'x' is substituted by 'p' then the identity becomes E = R, thus the expression E automatically becomes the remainder as 'p' is achieved by equating x-p to zero. Hence, actually the sign of 'p' is reversed [1].

3. In *Paravartya* Sutra the digits of the divisor are first negated, i.e. if the divisor is 112, the new adjusted divisor becomes -1 -1 -2. Then the first digit is excluded and the remaining digits become the new divisor -1 -2.

4. Next the dividend part is broken up into two parts as in the *Nikhilam* Sutra and the operation is processed as shown in the Example-2.

```
1 1 2   )      1 2 3 4
 -1 -2  )      1 2 | 3 4    Broken up using Nikhilam Sutra.
               -1 | -2
                  | -1 -2
               ---------
               1 1 | 0 2
```

Example 2.

On close observation it can be seen that the first digit of the divisor, which was excluded initially, actually divides the first digit of the dividend as in the case of the *Nikhilam* Sutra, and the remainder obtained from that single digit division is used to multiply the adjusted divisor in the *Paravartya* Sutra and then proceeded with the *Nikhilam* Sutra to provide the quotient as 11 and the remainder as 2.

2. *VEDIVISION*, THE VEDIC DIVISION ALGORITHM

The proposed division algorithm in this paper is a combination of the earlier discussed two sutras, the *Nikhilam* Sutra and the *Paravartya* Sutra, with slight modification so as to obtain a generalized algorithm for all the possible divisors. Presently, numerous division algorithms are used depending upon system or application requirements such as the *Restore Type Division Algorithm, SRT Division Algorithm, and the Non Restore Type Division Algorithm* [10-12], the latter being the fastest and economic. It has been statistically proven further in our work that the proposed algorithm performs better with respect to the *Non Restore Type Division Algorithm* in terms of speed and memory requirement.

The proposed algorithm performs the calculations on the number of digits in the divisor and the dividend rather than on the number of bits representing them. In *Non Restore Type Division Algorithm*, the time estimate of the division is proportional to the number of bits. But in the *Vedic Division Algorithm*, the time requirement is based mainly on the number of normalizations (illustrated further) of the intermediate remainders. Hence, the algorithm exhibits remarkable results on divisions involving big numbers. The novelty of the algorithm lies in the fact that since the computation is done on digits rather than on the bits, very large numbers, having size up to 38 digits (127 bits), can be divided in the present form and if modified, it can divide even larger numbers.

2.1. Step – By – Step Algorithm description using an example

1. In the first step the divisor is adjusted using a combined logic of both the *Nikhilam* Sutra and the *Paravartya* Sutra. All the digits that are less than or equal to 5 are negated. For all those digits which have values more than 5, 10's compliment of the digit is taken and 1 is added to the next higher digit. If the divisor is 47483647, then a close observation reveals that all the consecutive digits are alternately less than and greater than 5. The divisor adjustment starts from the Least Significant Digit. As 7 > 5, hence 10's compliment of 7 is taken and 1 is added to the next higher digit '4',replacing 7 by 3 and 4 by 5. Now this 5 is adjusted by -5. Hence the adjustment of the divisor is shown below.

4	7	4	8	3	6	4	7	becomes
-5	3	-5	2	-4	4	-5	3	(Adjusted divisor)

2. Let us take an example in which the dividend is 99999 and the divisor is 456. Therefore, the divisor 456 after adjustment becomes -5 4 4. It may also be observed that the first digit of an adjusted divisor will always be a negative digit.

3. Next, the first digit of the dividend is divided by the magnitude of the first digit of the adjusted divisor to obtain the quotient. In the example, the first digit of the dividend (99999) is 9 which is divided by magnitude of the first digit of the adjusted divisor (-5 4 4) which is 5 to obtain the quotient 1. This is the only step where division is unavoidable but it has been accomplished with the aid of two look – up tables as shown in Table 1 and Table 2. The use of these tables saves the division time otherwise required for obtaining the quotient and the remainder.

Table 1. Look – up Table for Quotient

Q : Quotient Table										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	2	3	4	5	6	7	8	9
2	0	0	1	1	2	2	3	3	4	4
3	0	0	0	1	1	1	2	2	2	3
4	0	0	0	0	1	1	1	1	2	2
5	0	0	0	0	0	1	1	1	1	1

Table 2. Look – up Table for Remainder

R : Remainder Table										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	1	0	1	0	1	0	1
3	0	1	2	0	1	2	0	1	2	0
4	0	1	2	3	0	1	2	3	0	1
5	0	1	2	3	4	0	1	2	3	4

The second row and the first column in Table 1 and Table 2 are shown in bold and represent the array indices. The row indices have been started from 1. Column 1 represents the denominator values and Row 2 represents the numerator values. Suppose we want to divide 'a' by 'b'. The quotient is obtained from the cell Q[b,a] of Table 1

where the maximum value of b can be 5 and not equal to 0. The remainder is obtained from Table 2 in the same manner, that is, the value of cell R[b,a].

The new quotient is then multiplied by all the digits of the adjusted divisor and placed below the dividend at the exact positions and then added to get the new remainder as shown below:

```
-5 4 4  ) 9  9  9  9  9        ( 1 0 0 0 0
        ) -5  4  4      .
          4 13 13 9  9
```

4. This step normalizes the remainder. Normalization means replacing a multiple digit value by a single digit value at each position. The procedure is started from the Least Significant Number of the remainder and followed to the Most Significant Number. The Least Significant Digit of the multiple digit number is kept at the position and the rest is added to the next Higher Significant Digit.

   There are primarily three reasons for Normalization. They are as follows:

   - If there is more than 1 digit in a single position: Suppose at the two adjacent places, there are 2 and 23. Then the normalization procedure is as follows:

     ```
          2      23
     =  (2 + 2)   3
     =    4       3
     ```

     Thus, 2 23 after normalization becomes 4 3.
   - If there is a negative digit at any position: The we normalize as follows:

     ```
          2       -3
     =  (2 – 1)   (10 – 3)
     =    1         7
     ```

     Thus, 2 -3 becomes 1 7 after normalization.
   - If there are negative numbers at any position: In cases where there are negative numbers (multiple digit number), then normalization is done as follows:

     ```
          36      -17
     =  (36 – 1)  (10 – 17)
     =    35       -7
     ```

   Thus the normalized result is 35 -7 which further needs normalization by the first two procedures for obtaining the final result.

   If the most significant digit is negative, then normalization is not performed and the digit is kept as it is.

   The above mentioned procedure of normalization is also performed for normalization of the Quotient finally. The number of normalizations required for a single step determines the time estimation of the division procedure which has been elaborated further in the Performance Analysis part of this paper.

5. The next step checks for a ‘0’ at the Most Significant Digit of the normalized remainder. If it is not ‘0’, as in this case, then the normalized remainder is again divided by the adjusted divisor and the new quotient is added to the previous quotient. Else if it is ‘0’, then the previous procedures are repeated till completion. The whole procedure is shown in the output snippet in Figure 1 below:

```
Enter numerator: 99999

Enter denominator: 456

-5 4 4 )9 9 9 9 9 (
-5 4 4 )9 9 9 9 9 (1 0 0 0 0
   -5 4 4
    4 13 13 9 9
normalized remainder5 4 3 9 9
-5 4 4 )5 4 3 9 9 (2 0 0 0 0
   -5 4 4
-5 4 4 )8 7 9 9 (2 1 0 0 0
   -5 4 4
    3 11 13 9
normalized remainder4 2 3 9
-5 4 4 )42 3 9 (2 1 8 0 0
   -40 32 32
    2 35 41
normalized remainder5 9 1
-5 4 4 )5 9 1 (2 1 9 0 0
   -5 4 4
    13 5
normalized remainder1 3 5
-5 4 4 )
    13 5
normalized remainder1 3 5
no further division possible.checking the remainder
The final remainder   1 3 5
The normalized quotient      2 1 9
```

Figure 1.

It can be seen that the remainder normalization forms the main time consuming part of the algorithm. Also there is no standard procedure for predicting the number of normalizations. Hence, the time estimate does not depend on the size of the number. Even in divisions having higher number of normalizations, *Vedic Division Algorithm* performs better with respect to the conventional algorithms.

Statistically it has been found that in division intensive environments, like cryptographic algorithms etc, the overall performance of the *Vedic Division Algorithm* is much faster when compared to the conventional *Non Restore Type Division Algorithm.*

2.2. The Algorithm

Initialization Part

1. The dividend and the divisor are held in two arrays – Array ’a’ and Array ‘b’ having index ‘m’ and ‘n’ respectively. Array ‘b’ finally stores the remainder. A temporary array ‘temp’ has been used to hold the quotient having index‘t’. Another array ‘c’ is used to hold the copy of the divisor. All the data has been stored in Little Endean Formats and initialized to 0.The length of Array ‘b’ is ‘n’ and Array ‘a’ is ‘m+1’, Array ’c’ is ‘m’ and ‘temp’ is ‘n’ respectively.
2. The divisor is adjusted so that no digit in the divisor is more than 5.
3. ‘m’ is updated if adjusting the divisor causes increase in length of the divisor.
4. Inverse each digit of the divisor (including the most significant digit). Steps 3 and 4 have been clubbed in Section 2.1.

Division Part

5. A new variable ‘j’ is taken for holding the value of the possible number of iterations and its limits are set from 1 to ‘n-m+1’.
6. Compare the most significant digits of divisor and dividend.
7. If MSD (Most Significant Digit) of divisor >MSD of dividend, club the most significant pair of digits of dividend. Update (decrement) n and t.
8. If clubbing results in decrease of the number of digits of dividend below that of divisor, break from the ‘For Loop using ‘j’’ and go to step 13.
9. Quo= MSD of dividend/ MSD of divisor. Calculate the remainder (dividend).
10. If MSD of dividend not divisible by that of divisor, then break from the “for loop”. A fresh iteration is being started. Go to step 13.
11. Decrement n and t, go to step 8.
12. The remainder is then normalized. Increment ‘n’ and‘t’ if normalization increments the number of digits in remainder.
13. If (n<m) or if (j>n-m+1) in the previous iteration, no further iterations are possible and 2nd condition means we have already tried to divide once more but failed, go to step 16.
14. Else division is possible, go to step 7.
15. Check the remainder. If the remainder is negative, the quotient is decremented once. Else if it is positive, it is the ‘normalized version of divisor’ and cannot be divided again. So check if original form of divisor (stored in array ‘c’) can divide again. This division can result into incrementing the quotient by 1.
16. The quotient is stored in array ‘temp’ with the leading and trailing 0’sand remainder in array ‘b’. Quotient can further be normalized.

2.3. The Flowchart

The flowchart of the *Vedic Division Algorithm* is shown in Figure 2.

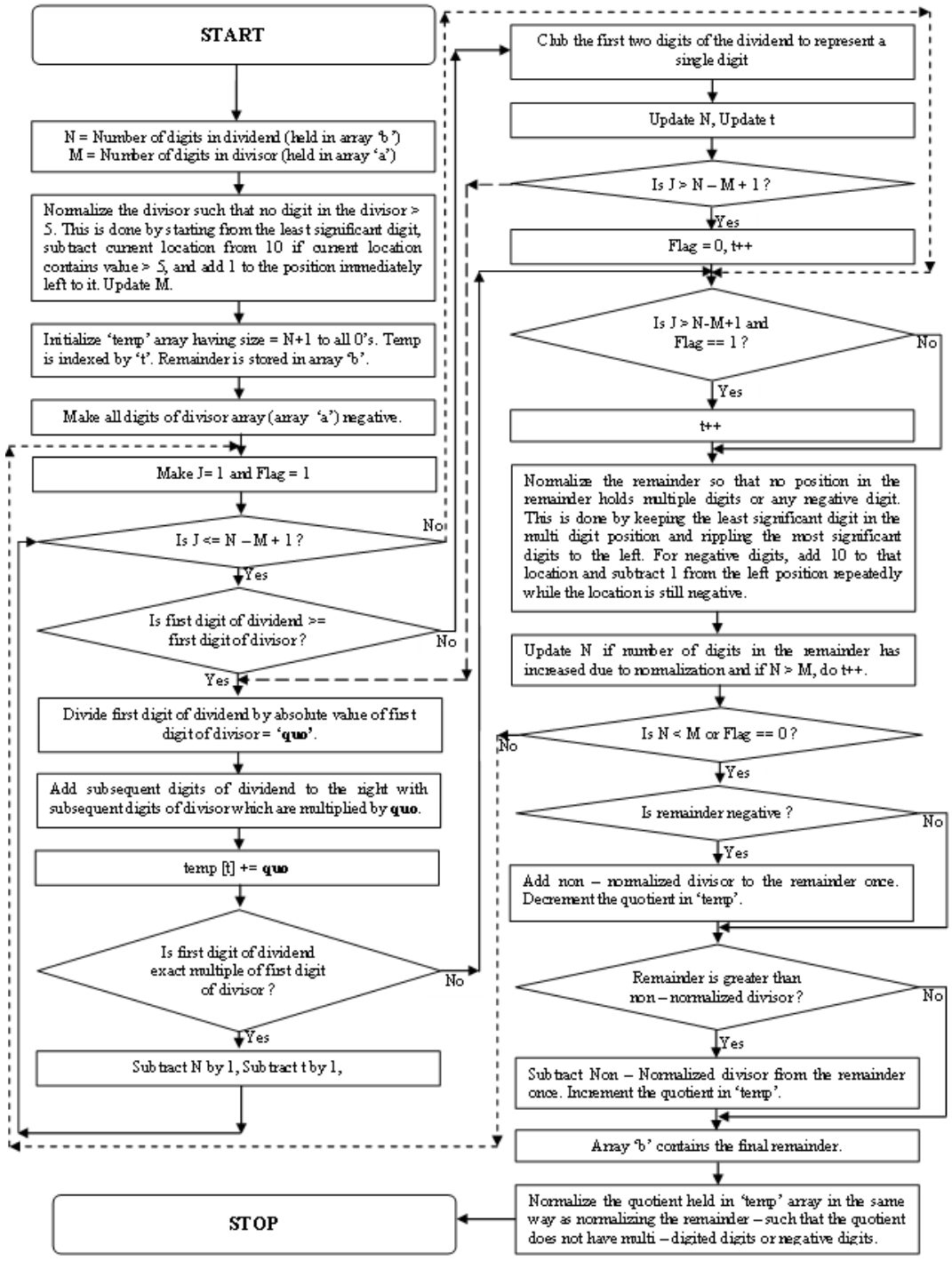
START
N = Number of digits in dividend (held in array 'b')
M = Number of digits in divisor (held in array 'a')
Normalize the divisor such that no digit in the divisor > 5. This is done by starting from the least significant digit, subtract current location from 10 if current location contains value > 5, and add 1 to the position immediately left to it. Update M.
Initialize 'temp' array having size = N+1 to all 0's. Temp is indexed by 't'. Remainder is stored in array 'b'.
Make all digits of divisor array (array 'a') negative.
Make J= 1 and Flag = 1
Is J <= N – M + 1 ?
No
Yes
Is first digit of dividend >= first digit of divisor?
No
Yes
Divide first digit of dividend by absolute value of first digit of divisor = 'quo'.
Add subsequent digits of dividend to the right with subsequent digits of divisor which are multiplied by quo.
temp [t] += quo
Is first digit of dividend exact multiple of first digit of divisor ?
No
Yes
Subtract N by 1, Subtract t by 1,
Club the first two digits of the dividend to represent a single digit
Update N, Update t
Is J > N – M + 1 ?
Yes
Flag = 0, t++
Is J > N-M+1 and Flag == 1 ?
No
Yes
t++
Normalize the remainder so that no position in the remainder holds multiple digits or any negative digit. This is done by keeping the least significant digit in the multi digit position and rippling the most significant digits to the left. For negative digits, add 10 to that location and subtract 1 from the left position repeatedly while the location is still negative.
Update N if number of digits in the remainder has increased due to normalization and if N > M, do t++.
Is N < M or Flag == 0 ?
No
Yes
Is remainder negative ?
No
Yes
Add non – normalized divisor to the remainder once. Decrement the quotient in 'temp'.
Remainder is greater than non – normalized divisor ?
No
Yes
Subtract Non – Normalized divisor from the remainder once. Increment the quotient in 'temp'.
Array 'b' contains the final remainder.
Normalize the quotient held in 'temp' array in the same way as normalizing the remainder – such that the quotient does not have multi – digited digits or negative digits.
STOP

Figure 2

2.4. Division examples illustrating the Best Case and the Worst Case

Best Case

Let the dividend be 80217727 and the divisor be 20202. The output snippet of the division is shown in Figure 3. It can be observed that remainder is normalized only in the last step. Hence, this type of division can be termed as a Best Case.

```
Enter numerator: 80217727

Enter denominator: 20202

-2 0 -2 0 -2 )8 0 2 1 7 7 2 7 (
-2 0 -2 0 -2 )8 0 2 1 7 7 2 7 (4 0 0 0 0 0 0 0
   -8 0 -8 0 -8
-2 0 -2 0 -2 )-6 1 -1 7 2 7 (4 0 -3 0 0 0 0 0
   6 0 6 0 6
-2 0 -2 0 -2 )
    15 7 8 7
normalized remainder1 5 7 8 7
no further division possible.checking the remainder
The final remainder   1 5 7 8 7
The normalized quotient      3 9 7 0
```

Figure 3

Worst Case

The example shown in Figure 1, Section 2.1, can be said to be a Worst Case as in almost all the steps, remainder normalization has been required.

3. PERFORMANCE ANALYSIS OF THE *VEDIC DIVISION ALGORITHM*

For performance analysis, the *Vedic Division Algorithm* was executed on a 32 bit Operating System having Intel Pentium Dual CPU E2180 @2.00 GHz and 0.99 GB DDR2 RAM. At each execution, the algorithm was iterated 100,000 times with the same dividend and the divisor and the average time was noted, as the time taken for a single execution was so minuscule that it could not be detected. We think that it is the most rational method for time estimation. The comparison was made with respect to the *Non Restore Type Division Algorithm*, which was also iterated for an equal number of times for the same set of dividends and divisors, and the average execution time for this algorithm was also noted. The comparison was also made with respect to the *Restore Type Division Algorithm*, but since the performance of *Non Restore Type Division Algorithm* is much better than the *Restore Type Division Algorithm*, we tabulated the result with the *Non Restore Type Division Algorithm*. Each set of dividends and divisors ware executed by both the algorithms for seven times and the minimum value of time required of the seven executions were taken for analysis. This was done due to factors influencing the time analysis of the execution such as operating system time scheduling. The analysis details were noted down as shown in Table 3.

Table 3. Execution Time Analysis of Vedivision and Non Restore Division Algorithm

Dividend	Digits	Divisor	Digits	Vedic Division	Non Restore Division
				Time in µs	Time in µs
91	2	16	2	0.150	0.800
255	3	127	3	0.150	1.710
948	3	182	3	0.310	2.340
1,982	4	27	2	0.310	2.810
3,728	4	94	2	0.160	3.270
7,682	4	48	2	0.460	3.890
9,382	4	49	2	0.320	3.900
47,386	5	28	2	0.620	5.610
131,071	6	7,295	4	0.460	5.780
461,938	6	682	3	0.930	7.020
493,827	6	14	2	0.790	7.170
739,481	6	95	2	0.470	8.110
1,048,576	7	2,249	4	0.460	8.260
3,729,618	7	30,901	5	0.320	9.200
10,388,608	8	240	3	0.630	10.780
29,384,791	8	11	2	0.370	12.160
55,555,555	8	41	2	0.780	12.960
80,217,727	8	20,202	5	0.310	13.570
482,937,164	9	456	3	1.250	15.740
736,582,914	9	1,782	4	1.240	17.150
1073741824	10	262125	6	1.860	16.860
3147483648	10	7485629	7	0.780	17.790
4294967295	10	2147483647	10	0.610	17.770
19372864582	11	4286	4	0.930	26.350
965274638525	12	8258	4	1.240	31.050
9561346784625	13	764318	6	2.780	39.910
45615935785265	14	646464	6	3.120	40.080
56455825519553	14	61945	5	3.900	41.640
693582471951753	15	84265	5	2.490	48.760
731984265735195	15	289357	6	2.490	49.290

The dividends and the divisors were taken in a random manner and the results were tabulated. It can be seen that in the last row in Table 3, the dividend is a 15 digit number or in terms of bits, it is a 50 bit number. In this range the *Non Restore Type Division Algorithm* starts giving ambiguous results but the *Vedic Division Algorithm* performs satisfactorily. Also the *Vedic Division Algorithm* can compute numbers having 38 digits, 127 bit numbers, accurately in the present form, if modified it can divide even larger numbers. The tabulated data in Columns 5 and 6 in Table 3 have been analyzed further in Figure 4.

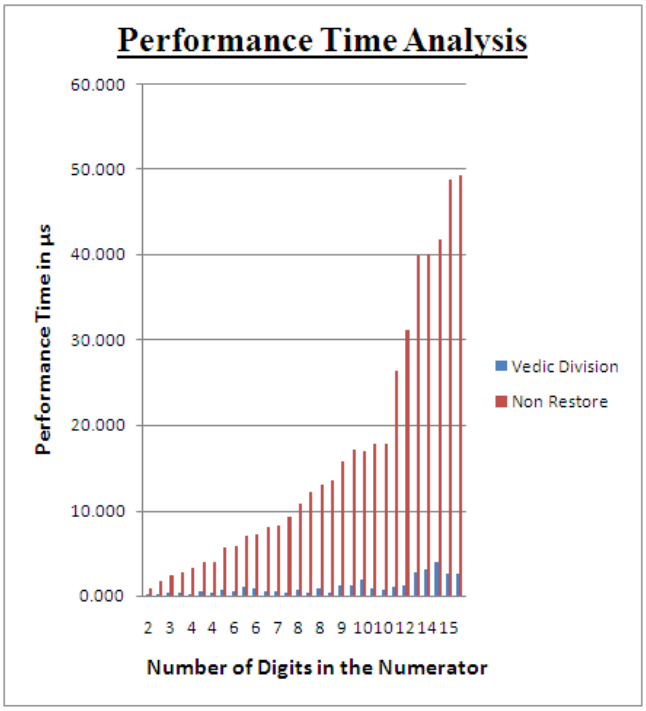

Figure 4

It can be well observed in Figure 4 that with increase in value in the dividend, the performance time in case of Non Restore Type Division Algorithm increases drastically compared to the *Vedic Division Algorithm*. This is because the *Non Restore Type Division Algorithm* computes on the number of bits, hence with increasing number of bits, the execution time increases. It can also be seen that there are a few glitches while computing with the large sized numbers in the *Vedic Division Algorithm*, but those are due to the random selection of dividends and the divisors, otherwise it can be stated that the computation time required by the *Vedic Division Algorithm* is almost constant irrespective of the size of the dividend. The only factor affecting the speed of execution is the number of normalizations required in a particular division.

Further in Table 4, a comparison of the performance time with the value of the quotient has been made. Normally, more the value of the quotient more is the number of divisions required. Hence, it can be assumed that with increasing size of quotient, the performance time should increase. But, in case of Vedic Division Algorithm, we observe that the time estimation is not a function of the size of the quotient or number of divisions as it solely depends on the number of normalizations required.

Table 4. Quotient Size Vs Time Analysis

Value of Quotient	Digits in Quotient	Time taken by Vedivision (μs)
2	1	0.610
5	1	0.150
5	1	0.310
17	2	0.460
39	2	0.160
73	2	0.310
120	3	0.320
160	3	0.460
191	3	0.320
420	3	0.780
466	3	0.460
677	3	0.930
1692	4	0.620
3970	4	0.310
4096	4	1.860
7784	4	0.470
35273	5	0.790
43285	5	0.630
413346	6	1.240
1059072	7	1.250
1355013	7	0.780
2671344	7	0.370
4520033	7	0.930
12509644	8	2.780
70562221	8	3.120
116889638	9	1.240
911386318	9	3.900
2529692614	10	2.490
8230967447	10	2.490

The results in Table 4 have been graphically represented in Figure 5. There it can be observed that with increasing value of the quotient, the performance time may be lower or may be higher. For example, from Table 4, it can be observed that for a quotient value of 70562221 the performance time is 3.120 μs whereas for 116889638, it is 1.240 μs. Thus the time is less in case of a bigger quotient.

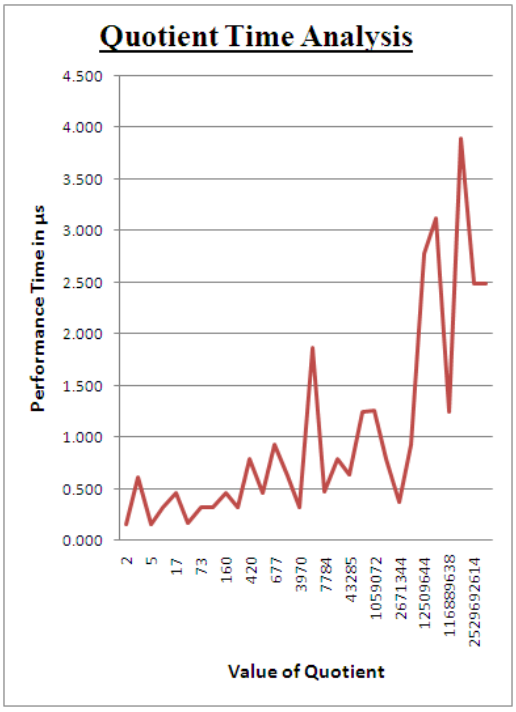

Figure 5

4. CONCLUSIONS

Vedivision, or the "Vedic Division Algorithm", has exhibited remarkable results with respect to conventional division algorithms in terms of fast BCD division, thereby once again proving the famous proverb that '*Old is Gold*'. Also it has been observed that the execution time does not depend on the size of the dividend or the divisor, but on the number of remainder normalizations required. Further VLSI implementation of the algorithm remains to be tested.

ACKNOWLEDGEMENTS

We would like to thank the Department of Computer Science and Engineering, Jadavpur University, for providing us with all the facilities demanded by the research work. We would also like to extend our sincere thanks to Ms. Anisha Majumder and Mr. Sourav Sikdar for their valuable assistance.

REFERENCES

[1] Jagadguru Swami Sri Bharath, KrsnaTirathji, *Vedic Mathematics or Sixteen Simple Sutras From The Vedas*, Motilal Banarsidas , Varanasi(India),1986.

[2] Swami Bharati Krishna Tirtha's Vedic mathematics [Online]. Available: http://en.wikipedia.org/wiki/Vedic mathematics.

[3] HimanshuThapliyal, R.V Kamala and M.B Srinivas "RSA Encryption/Decryption in Wireless Networks Using an Efficient High Speed Multiplier", Proceedings of IEEE International Conference On Personal Wireless Communications (ICPWC-2005) , New Delhi, pp-417-420, Jan 2005.

[4] HimanshuThapliyal and M.B Srinivas, "High Speed Efficient Hierarchical Overlay Multiplier Architecture Based on Ancient Indian Vedic Mathematics", Proceedings of International Conference on Signal Processing, ICSP 2004, Turkey, Dec 2004.
[5] HimanshuThapliyal and M.B Srinivas, A High Speed and Efficient Method of Elliptic CurveEncryption Using Ancient Indian Vedic Mathematics.
[6] ManoranjanPradhan, Rutuparna Panda and Sushanta Kumar Sahu, "Speed Comparison of 16x16 Vedic Multipliers", International Journal of Computer Applications (0975 – 8887), Volume 21– No.6, May 2011
[7] Harpreet S. Dhillon and A.Mitra , "A Digital Multiplier Architecture using UrdhvaTiryakbhyam Sutra of Vedic Mathematics ",Department of Electronics and Communication Engineering,Indian Institute of Technology, Guwahati 781 039, India.
[8] Purushottam D. ChidgupkarMangesh T. Karad, The Implementation of Vedic Algorithms in DigitalSignal Processing, Global J. of Engg. Educ., Vol.8, No.2 © 2004 UICEE, Published in Australia.
[9] Shuangching Chen and ShugangWei,"A High-Speed Realization of Chinese Remainder Theorem", Proceedings of the 2007 WSEAS Int. Conference on Circuits, Systems, Signal and Telecommunications, Gold Coast, Australia, January 17-19, 2007
[10] Stuart F. Oberman, and Michael J. Flynn, "Division Algorithms and Implementations", IEEE Transactions on Computers, vol. 46, no. 8, August 1997.
[11] J. O'Leary, M. Leeser, J. Hickey and M. Aagaard, "Non-Restoring Integer Square Root: A Case Study in Design by Principled Optimization", LNCS.
[12] Website: csclab.murraystate.edu/bob.pilgrim/405/Computing%20Machinery%20Ch06.pdf.

Authors

Diganta Sengupta

He is affiliated with Future Institute of Engineering and Management, Kolkata, India, in the capacity of Assistant Professor in the department of Applied Electronics and Instrumentation Engineering. He has obtained his Master's degree from Jadavpur University. His areas of research interest include Vedic Mathematics, Reversible Logic, and Cryptography.

Mahamuda Sultana

She is presently affiliated with Swami Vivekananda Institute of Science and Technology, Kolkata, India, in the capacity of Assistant Professor in the department of Computer Science and Engineering. She has obtained her Master's degree from Jadavpur University and her research interests include Vedic Mathematics, Reversible Logic, and Cryptography.

Atal Chaudhuri

Prof. (Dr.) Atal Chaudhuri received his Master of Electronics and Telecommunication Degree with Computer Science specialization in the year 1982 and Doctorate of Philosophy in Engineering from Jadavpur University in 1989. He has worked in the capacity of R&D Engineer and Project Engineer in various research projects in India and abroad. Dr. Atal Chaudhuri is now a Professor in the Department of Computer Science & Engineering of Jadavpur University. He has more than 100 publications at both National and International level. He has served as an expert for All India Council of Technical Education, NAAC, University Grant Commission, India and NBA team for accreditation of various institutions for last 12 years. He is the life member of both Computer Society of India and Institute of Engineers, also the Vice chairman of Computer Division of Institute of Engineers. He is Senior Member of IEEE.