A Novel Square Root Algorithm and its FPGA Simulation

A Novel Square Root Algorithm and its FPGA Simulation

ZHOU Zhongcheng[1], HU Jingchun[2]

[1]The department of Mechanical Engineering, Tsinghua University, Beijing, China

[2]The department of Mechanical Engineering, Tsinghua University, Beijing, China

hujinchun@tsinghua.edu.cn

Abstract. As a basic operation in elementary mathematics, square root operation is widely used in numerical calculation and digital signal processing. The square root operation is a nonlinear operation and cannot be solved directly on FPGA. Many previous square root algorithms have been proposed. This paper introduces several previous square root algorithms, then presents a novel square root algorithm which uses 16-bit integer input and 16-bit integer output. Using the idea of normalization, the square number is mapped to the range of 0-1, and the square root is approximated by the sum of numerical sequence. We discuss the proposed square root algorithm, error analysis. The proposed algorithm was programmed by the Verilog in Xilinx's Vivado 17.2 software environment, and the simulation was verified on the Virtex-7 series chip. Compared with other square root algorithms, the algorithm only uses shifter, adders and sub-tractor. It does not involve complex operations, consumes less resources, and it's suitable for implementation on FPGA.

1. Introduction

With the development of IC, the application fields of FPGA products have expanded from the communication technology to the fields of consumer electronics, automotive electronics and industrial control [1]. Operations such as addition, subtraction, and shift can be directly performed on FPGA. Square root operation is a nonlinear operation and it cannot be directly solved by an analytical method on FPGA. This paper introduces several previous square root algorithms, such as: Newton-Raphson method [2] [3], Restoring algorithm [4], Non-Restoring algorithm [5], Vedic Mathematics square algorithm [6] [7] [8] [9] and Vector Rotation Square algorithm [10]. Based on the requirements of FPGA for square algorithm, this paper proposes a new algorithm for FPGA square algorithm, using 16-bit integer input and 16-bit integer output.

Chapter 2 introduces several previous square algorithms; Chapter 3 proposes a new square algorithm; Chapter 4 is the error analysis of the proposed square root algorithm; Chapter 5 is the simulation of the proposed square root algorithm, and Chapter 6 is a summary of the proposed square algorithm.

2. Previous Work

2.1. Newton-Raphson method

Newton-Raphson method treats square root as solving nonlinear equation $f(x)=x^2-A=0$, A is radicand. This equation can be derived using Taylor's expansion and it's denoted

$$f(x)=\left(x_0^2-A\right)+2x_0\left(x-x_0\right)+\left(x-x_0\right)^2=0 \tag{2.1}$$

Omit the nonlinear part, the equation can be denoted

$$\left(x_0^2 - A\right) + 2x_0\left(x - x_0\right) = 0 \tag{2.2}$$

By using the above formula, we can list the iterative equation of Newton-Raphson method, and it can be denoted

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{A}{x_n}\right) \tag{2.3}$$

The x_n is the square root result of the nth iteration. According to the iterative equation, there is a division operation in each iteration. In order to avoid division, this method calculates the reciprocal of x_n and multiply A. The calculation result is also related to the selection of the initial value of the iteration. To improve the calculation accuracy, this method need to divide the interval of the squared number into several intervals. Each interval corresponds to an optimal initial square root value. In summary, the Newton-Raphson method is complex and not suitable for implementation on FPGA.

2.2. Restoring algorithm

In this algorithm, the radicand is denoted A=16b'$a_{15}a_{14}\ldots a_1a_0$, and square root result is denoted Q=8b'$q_7q_6\ldots q_1q_0$. The iterative initial value is denoted Q_0=8b'00000000. There are eight iterations from k=7 to k=0. In each iteration, the algorithm defines q_k=1 and calculate $Q_0\times Q_0$-A. If the result greater than 0, q_k=1 and go to the next iteration. Otherwise q_k=1 and go to the next iteration. The algorithm needs to modify the iteration results twice, and the calculation process needs multiplier, which will occupy more resources.

2.3. Non-Restoring algorithm

In this algorithm, the radicand is denoted A=16b'$a_{15}a_{14}\ldots a_1a_0$, and the square root result is denoted Q=8b'$q_7q_6\ldots q_1q_0$. The remainder is denoted R=9b'$r_8q_7\ldots r_1r_0$. The algorithm is given below:

```
Step 1. $q_7$=0, $r_8$=0 and $k$=7 to 0;
Step 2. If $r_{k+1}\geq0$, $r_{k+1}$= $r_{k+1}a_{2k+1}a_{2k}$- $q_{k+1}$01 else $r_{k+1}$= $r_{k+1}a_{2k+1}a_{2k}$ +$q_{k+1}$01;
Step 3. If $r_{k+1}\geq0$, $q_k$= $q_{k+1}$1(i.e. $q_k$) else $q_k$= $q_{k+1}$0(i.e. $q_k$);
Step 4. Repeat step 2 until $k$=0.
```

2.4. Vedic mathematics square algorithm

Vedic mathematics square algorithm is derived from ancient 16 sutra and Upa-Sutras. In this algorithm, the radicand is denoted A=16b'$a_{15}a_{14}\ldots a_1a_0$ and the square root is denoted Q=8b'$q_7q_6\ldots q_1q_0$. The binary number Vedic mathematics square algorithm is given below:

```
Step 1. $i$=15, $j$=7; $i$=$i$-2, $j$=$j$-1;
Step 2. If $a_ia_{i-1}$= 00, $q_i$ = 0 and go to step 3, otherwise go to step 4;
Step 3.  $i$=$i$-2, $j$=$j$-1, go to step 2;
Step 4. $q_i$=1, $R_i$ =$a_ia_{i-1}$ - 01;
Step 5. $S_i$ = {$R_i$, $d_{i-2}$};
Step 6.  If $T$<0, $q_j$ =0 and $S$=$T$+10, otherwise $q_j$ =1;
Step 7. $R$=$S$, $i$=$i$-1 and calculate Duplex, $S$= {$R$, $a_{i-1}$}- Duplex;
Step 8. If $S$<0, go to step 9, otherwise go to step 10;
Step 9. $j$=$j$+1, $i$=$i$+1;
Step 10. Calculate Duplex, $S$= {$R$, $a_{i-1}$}- Duplex;
Step 11. $T$=$S$-10, $j$=$j$-1;
Step 12. Go to step 6 until $j$<0.
```

In this algorithm, *Duplex* is the number itself for a 1-bit number. For a 2-bit number, *Duplex* is twice the product of both the bits. For a 3-bit number, *Duplex* is twice the product of outermost pair of bits plus the middle bit. For a 4-bit number, *Duplex* is twice the product of outermost pair of bits plus twice the product of next pair of bits. For example:

Duplex (0) = 0

$$\begin{aligned}&\textit{Duplex}\ (11) = 2\times1\times1=2 \text{ or } (10)_2\\&\textit{Duplex}\ (111) = 2\times1\times1+1=3 \text{ or } (11)_2\\&\textit{Duplex}\ (1101) = 2\times1\times0+2\times1\times1=2 \text{ or } (10)_2\end{aligned} \tag{2.4}$$

2.5. Unit Vector Rotation Square algorithm

The initial unit vector $M_0(1,0)$ coincides with the X-axis, rotate the unit vector $\Delta\theta$ counterclockwise i times. The coordinates of the unit vector $M_i\,(x_i, y_i)$ are denoted

$$\begin{aligned}&x_i = \cos(i\Delta\theta) = 2\cos^2\left(\frac{i}{2}\Delta\theta\right) - 1\\&y_i = \sin(i\Delta\theta)\end{aligned} \tag{2.5}$$

When the radicand is $0\leq A\leq1$, t=2A-1. Then this algorithm rotates the unit vector to make the its abscissa equal to t and it can be denoted t =cos($i\Delta\theta$). We can get the number of iterations i from the above iterations. Finally rotate the unit vector [$2^{-1}i$] times to get cos($i\Delta\theta$/2) ([] is a Round operation). The square root result is denoted $Q=A^{0.5}=\cos(i\Delta\theta/2)$. When the radicand is A>1, the square number can be mapped to the range of 0-1 using the idea of normalization. Use above algorithm to calculate cos($i\Delta\theta$/2), the square root result Q is obtained by inverse normalization. This algorithm needs to set a suitable rotation angle $\Delta\theta$ for different square results. Otherwise, it will lead to too many iterations or inaccurate calculation.

3. Proposed algorithm

In this paper, a new square root algorithm is proposed. Using the idea of normalization, the square number is mapped to the range of 0-1. And the square root is approximated by the sum of numerical sequence. It is divided into the following three cases according to the size of the radicand:

When radicand A= 0, the square root can be directly obtained.

When the radicand is $0 < A < 1$, A is normalized to be itself. The square result can be denoted

$$Q = \sum d_k 2^{-k} \tag{3.1}$$

d_k=1 or d_k=-1. Iterative initial value x_0=0 and the nth iterative result updates are based on the first iteration. Each update is denoted $x_{k+1} = x_k + d_k \cdot 2^{-k}$. After the nth iteration, square root result is denoted

$$x_{k+1} = \sum_{i=1}^{k} d_i 2^{-i} \tag{3.2}$$

The square of x_{k+1} is given below:

$$\begin{aligned}c_{k+1} &= (x_{k+1})^2\\&= \left(x_k + d_k\cdot\frac{1}{2^k}\right)^2\\&= (x_k)^2 + d_k\cdot\frac{x_k}{2^{k-1}} + \frac{1}{2^{2k}}\\&= c_k + d_k\cdot\frac{x_k}{2^{k-1}} + \frac{1}{2^{2k}}\end{aligned} \tag{3.3}$$

Due to $2^{-(k-1)}>2^{-2k}$, if d_k =-1, $c_{k+1} < c_k$; If d_k =1, $c_{k+1}>c_k$. d_k should be chosen to make c_{k+1} closer to A than c_k.

$$d_n = \begin{cases}+1 & if \quad A > c_n\\ -1 & else(A \leq c_n)\end{cases} \tag{3.4}$$

The division in the above calculation can be performed on the FPGA by shifting, which can be written in the following form:

$$x_{k+1} = x_k + d_k\cdot(1 >> k) \tag{3.5}$$

$$c_{k+1} = c_k + d_k\cdot\left[x_k >> (k-1)\right] + \left[17'H10000 >> (k << 1)\right] \tag{3.6}$$

When radicand is A>1, the square number can be mapped to the range of 0-1 using the idea of normalization. The normalization consists of the following steps: Firstly, find suitable m to make $2^{2(m-1)} < A \leq 2^{2m}$ (m is positive integer), then $A=A \cdot 2^{-2m}$. After normalization, $2^{-1}<A\leq 1$, we can calculate x_k with the second case. Finally, the square root result can be denoted

$$Q = 2^m x_n = \sum_{k=1}^{n} d_k 2^{m-k} \tag{3.7}$$

The above operations can be expressed in the FPGA as

$$A = A >> 2m \tag{3.8}$$

$$Q = \left(x_n << \left(m-k\right)\right) \tag{3.9}$$

<< is shift function. This algorithm is also explained in the flow chart given figure 1.

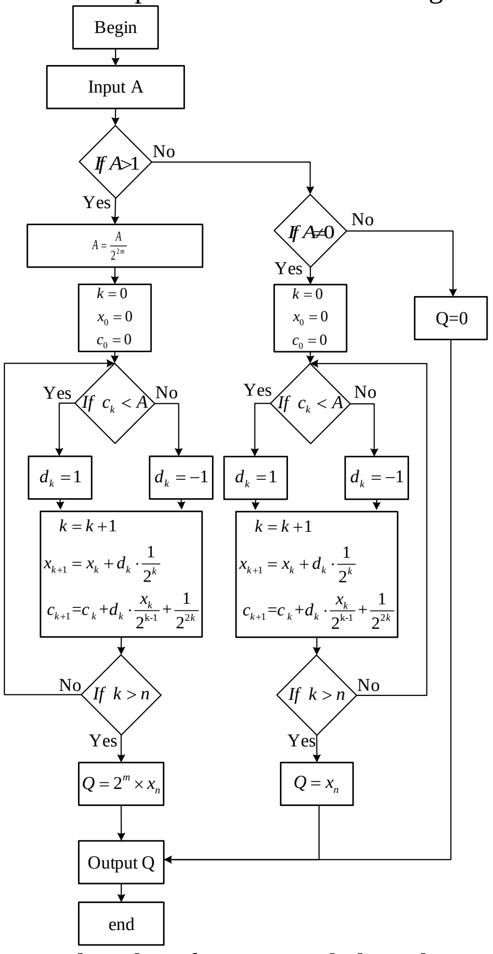

Figure 1. Flowchart for proposed algorithm

4. Error Analysis

When the radicand is $0<A\leq 1$, the square root result is denoted

$$Q = \sum_{i=1}^{k-1} d_i \cdot 2^{-i} \tag{4.1}$$

n is the number of iterations. It can be seen from the expression that Q is not equal to 1, so it is maximum from the square root result error at $A=1$. when A=1 (m is positive integer),

$$\lim_{n\to\infty}\left(\sum_{k=1}^{n} 2^{-k}\right) = 1 \tag{4.2}$$

The number of iterations cannot be infinite,

$$\sum_{i=1}^{k} d_i \cdot 2^{-i} < 1 \tag{4.3}$$

Errors can be expressed as

$$e = \max\left(\sum_{i=1}^{k} d_i 2^{i}\right) - 1 = -2^{-k} \tag{4.4}$$

This algorithm is also explained in the flow chart given figure 2.

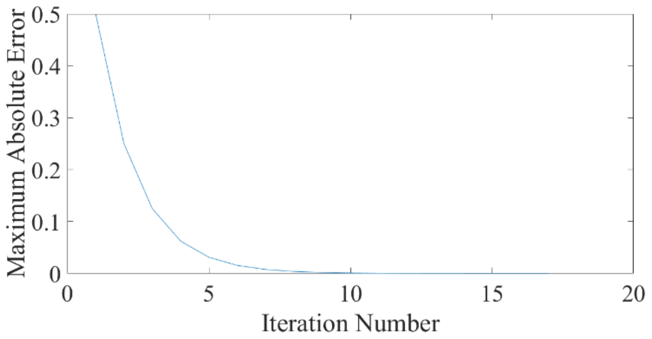

Figure 2. The Relation between maximum absolute error and iteration number

When the radicand is A>1, firstly find suitable m to make $2^{2(m-1)} < A \leq 2^{2m}$ (m is positive integer). The square root result is denoted

$$Q = \sum_{k=1}^{n} d_k 2^{m-k} \tag{4.5}$$

n is the number of iterations and m is the normalized coefficient. It can be seen from the expression that Q is not equal to 2^m, it is maximum from the square root result error at $A=2^{2m}$. When $A=2^{2m}$ (m is positive integer),

$$\lim_{n\to\infty}\left(\sum_{k=1}^{n} 2^{m-k}\right) = 2^m \tag{4.6}$$

The number of iterations cannot be infinite

$$\sum_{k=1}^{n} 2^{m-k} < 2^m \tag{4.7}$$

The maximum error can be expressed as

$$e = \left(\sum_{k=1}^{n} 2^{m-k}\right) - 2^m = -2^{m-k} \tag{4.8}$$

This example is illustrated below: When the radicand is 16≥A>4, the maximum error increases with the number of iterations is denoted e=-2^{2-k}. This algorithm is also explained in the flow chart given figure 3.

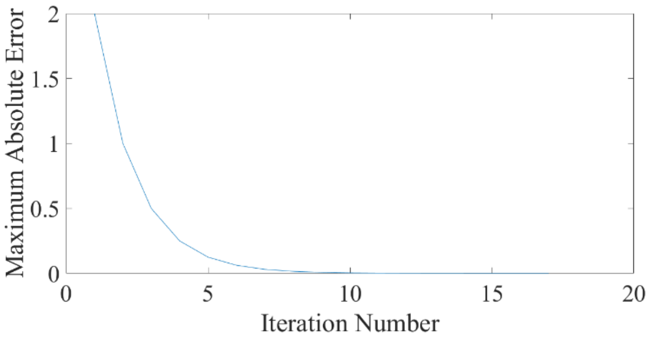

Figure 3. The Relation between maximum absolute error and iteration number

In summary, the square root accuracy of the algorithm is related to the number of iteration. By modifying the number of iteration, any given square root accuracy can be achieved theoretically.

5. Simulation and results

The proposed algorithm was programmed by the Verilog in Xilinx's Vivado 17.2 software environment, and the simulation was verified on the Virtex-7 family xc7vx485t device with speed -1 and package FFG1157 FPGA. The proposed algorithm uses 16-bit integer input and 16-bit integer output. The relation between Occupied Resources and Iteration Number is shown in figure 4. As can be seen from the figure, the occupancy of LUTs and Registers increases with the number of iterations. This algorithm uses more resources to get more accurate square root results.

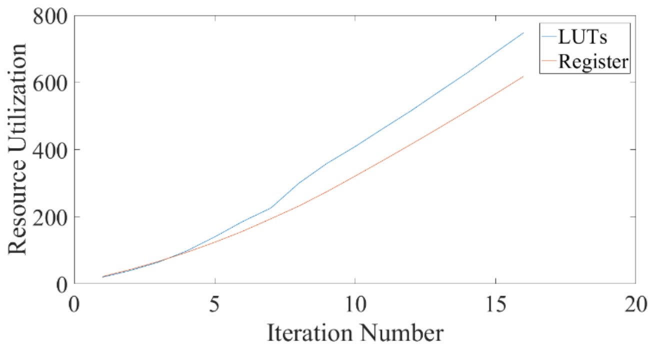

Figure 4. The relation between Occupied Resources and Iteration Number

When the radicand is 0<*A*≤1, the simulation output for 16-bit square root design is shown in the figure 5 and table 1.

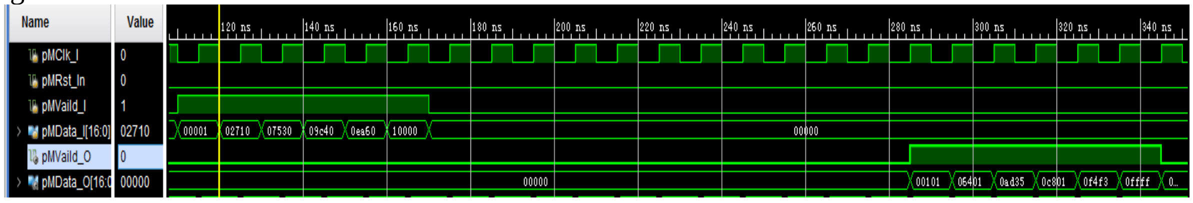

Figure 5. The simulation output when the radicand is 0<*A*<1

Table 1. The simulation output when the radicand is 0<*A*<1

Radicand		Square Root Result		Iteration Number	Maximum Error
Hex	Dec	Hex	Dec		
00001	1.52588e-05	00101	0.0039215	16	-1.52588e-05
02710	0.1525879	06401	0.3906403	16	-1.52588e-05
07530	0.4577637	0ad35	0.6765899	16	-7.61911e-06
09c40	0.6103516	0c801	0.7812653	16	-1.52588e-05
0ea60	0.9155273	0f4f3	0.9568329	16	-9.54967e-07
10000	1	0ffff	0.9999847	16	+1.52588e-05

When the radicand is 1<A≤16, the simulation output for 16-bit square root design is shown in the figure 6 and table 2.

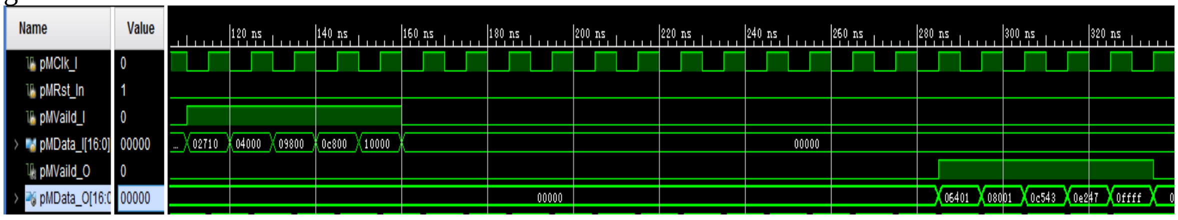

Figure 6. The simulation output when the radicand is 1<A<16

Table 2. The simulation output when the radicand is 1<A<16

Radicand		Square Root Result		Iteration	Maximum
Hex	Dec	Hex	Dec	Number	Error
02710	2.4414063	06401	1.5625610	16	-6.10352e-05
04000	4.0000000	08001	2.0000610	16	-6.10352e-05
09800	9.5000000	0c543	3.0822144	16	-7.35398e-06
0c800	12.500000	0e247	3.5355835	16	-4.95902e-05
10000	16.000000	0ffff	3.9999389	16	+6.10352e-05

The simulation results show that the algorithm satisfies the theoretical analysis of the square error. According to the requirement of accuracy of prescription and the utilization of resources under different conditions, different iterations are selected to achieve the balance of prescription speed and resources.

6. Conclusion

This paper proposes a new arithmetic method, elaborates the principle of square root algorithm. By the principle of the algorithm, this method only uses shifter, adders and sub-tractor without involving other complex operations which is suitable for implementation on FPGA. Then the error Analysis shows the relationship between the maximum error and the iteration number in this method. This method can achieve arbitrary accuracy by changing the number of iterations. Finally, the simulation in Vivado software shows that this method satisfies the theoretical error analysis and it can be used on FPGA.

References

[1] Samavi S, Sadrabadi A, Fanian A. Modular array structure for non-restoring square root circuit. Journal of Systems Architecture 2008; 54(10):957-966.
[2] Ramamoorthy C V , Goodman J R , Kim K H . Some properties of iterative square-rooting methods using high-speed multiplication[C]// 1972.
[3] Kabuo H , Taniguchi T , Miyoshi A , et al. Accurate rounding scheme for the Newton-Raphson method using redundant binary representation[J]. IEEE Transactions on Computers, 1994, 43(1):43-51.
[4] Li Y , Chu W . Implementation of single precision floating point square root on FPGAs[C]// IEEE Symposium on Field-programmable Custom Computing Machines. IEEE, 1997.
[5] Sajid I , Ahmed M M , Ziavras S G . Novel Pipelined Architecture for Efficient Evaluation of the Square Root Using a Modified Non-Restoring Algorithm[J]. Journal of Signal Processing Systems, 2012, 67(2):157-166.
[6] Tirthaji B K . Vedic Mathematics Sixteen Simple Mathematical Formulae from the Vedas[J]. Motilal Books, 2014.
[7] Rudagi J M , Ambli V , Munavalli V , et al. Design and implementation of efficient multiplier using Vedic Mathematics[C]// International Conference on Advances in Recent Technologies in Communication & Computing. IET, 2012.
[8] Kachhwal P , Rout B C . Novel square root algorithm and its FPGA implementation[C]// International Conference on Signal Propagation & Computer Technology. IEEE, 2014.
[9] Sharma R , Kaur M , Singh G . Design and FPGA implementation of optimized 32-bit Vedic multiplier and square architectures[C]// International Conference on Industrial Instrumentation & Control. IEEE, 2015.
[10] Zhong Hua, Sun Songlin, Jing Xiaojun. Algorithm for Solving Square Root Using Vector Rotation and Its FPGA Implementation[J]. Electronic Products, 2009, 16(8): 24-26.