Implementation of Different Square Root Algorithms

M. Franke[1], A. Th. Schwarzbacher[2], M. Brutscheck[1,2], St. Becker[1]

[1] Department of Computer Science and Communications Systems,
Merseburg University of Applied Sciences, GERMANY
marco-franke@gmx.de
michael.brutscheck@hs-merseburg.de
steffen.becker@hs-merseburg.de

[2] School of Electronic and Communications Engineering,
Dublin Institute of Technology, IRELAND
andreas.schwarzbacher@dit.ie

Abstract

This work presents fixed point square root algorithms and their implementation. These algorithms are referred to as non-restoring and restoring algorithm. This paper compares two such algorithms with a traditional lookup table (LUT) implementation of the square root algorithm. The LUT and the square root algorithms have an input range of 0 to 1. Furthermore, the square root algorithms and the LUT were implemented using different bitwidths. This research compared the input bitwidths of 4 bit, 8 bit, 16 bit, 32 bit and 64 bit. The power consumption, the area consumption and the propagation delay are then investigated. It will be shown the different behaviour of the non-restoring square root algorithm, of the restoring square root algorithm and of the lookup table square root causes different implementations to be useful for different applications. The in this paper presented results will therefore, provide the hardware designer with guidelines in choosing the most appropriate implementation for a given task.

Keywords: Fix Point Square Root, Non-Restoring, Restoring Square Root Algorithm.

1 Introduction

The determination of the square root is an important mathematical operation in digital signal processing (DSP) System. Thus, square root algorithms are much more complex in its structure and its functions and therefore, are not easy to implement. This paper compares three different algorithms. All three algorithms work with fixed point values and an input range between 0 and 1. The aim of this work is to implement the different square root algorithm. Thereafter, the power consumption, the propagation delay and the area consumption are compared. Furthermore, the results are presented for different bitwidths of 4 bit, 8 bit, 16 bit, 32 bit and 64 bit. Both square root algorithms, the non-restoring square root algorithm and the restoring square root algorithm, have been described in theory by Israel Koren in 2004 [1]. The name restoring means that the tentative remainder is negative the partial remainder is restored and the tentative remainder is again shifted one digit to left to $4r_i$. The two algorithms work differently, but they both calculate the same radix and the same final remainder. The third algorithm is based on a lookup table.

1.1 The Non-Restoring Square Root Algorithm

The non-restoring algorithm operates with a two's complement representation. In the process an exact value with a remainder is calculated at each iteration. This remainder is used for the further calculation until the radix has half the bitwidth of the radicand. If it is assumed that the radicand is denoted by an 8 bit vector then the final remainder and the radix can be obtained after four iterations. This is possible since the square root of the denominator is also determined and thus the bitwidth is halved. An example can be seen in (1) for an 8 bit wide radicand and a 4 bit wide radix.

$$\sqrt{x/256} = \sqrt{x}/16 \tag{1}$$

The non-restoring algorithm is based on the recursive relationship $r_i=2r_{i-1}\pm 2Q_{i-1}-2^{-i}$, where r_i is the ith partial remainder, Q_i is the ith square root bit. The computation is split into four sub-computations. First, the radicand or the partial remainder r_i is shifted by one digit to the left to produce $2r_i$. The second step is the determination of q_{i+1} by checking the sign of the partial remainder. If $2r_i \geq 0$ then $q_{i+1}=1$ and the sign of $2Q_i$ is positive otherwise $q_{i+1}=-1$ and the sign of $2Q_i$ is negative. At the first calculation where r_0 is the radicand then $Q_0=0$. After this, Q_i is shifted one digit to the left to produce $2Q_i$ and then 2^{i+1} is subtracted to $Y=2Q_i-2^{i+1}$. At the final step the partial remainder is calculated by $r_i=2r_{i-1}\pm Y_{i-1}$ where r_i is the partial remainder at iteration i. This calculation is carried out until the remainder is zero or the radix reaches half the bitwidth. For the definition of the final remainder only the MSB up to half of the input bit width is useful.

If the final remainder r_i and the radix Q_i are not negative, the precise final remainder can be obtained without corrections. Otherwise if the final remainder r_i and the partial remainder r_{i-1} have a different sign a correction is essential. Here, the following cases are to be distinguished. If Y and r_{i-1} have the same sign, the correct final remainder is obtained by adding of r_i and Y ($r_{corr}=r_i+Y$) and the correct square root is calculated by subtracting 2^i from Q_i ($Q_{corr}=Q_i-2^i$). If Y and r_{i-1} have opposite signs, the correct final remainder is obtained by subtracting Y from r_i ($r_{corr}=r_i-Y$) and the correct square root is calculated by adding of Q_i and 2^i ($Q_{corr}=Q_i+2^i$). At the end of the calculation the final remainder is multiplied by 2^{-i} were i is the last iteration. This example is described in Figure 1 below.

Rule for q_i :	$q_i = \begin{cases} 1 & if\ 2r_{i-1} \geq 0 \\ -1 & if\ 2r_{i-1} < 0 \end{cases}$			
$r_0 = X$	$= 0.00,10110000$	$= 176/256$		
$2r_0$	$= 0.01,01100000$	$= 352/256$	*set*	$q_1 = 1,\quad Q_1 = 0,0 + 0,1$
$-\left(0 + 2^{-1}\right)$	$-0.00,10000000$	$= 128/256$		$\underline{\underline{Q_1 = 0,1}}$
r_1	$= 0.00,11100000$	$= 224/256$		
$2r_1$	$= 0.01,11000000$	$= 448/16$	*set*	$q_2 = 1, Q_2 = 0,10 + 0,01$
$-\left(2Q_1 + 2^{-2}\right)$	$-0.01,01000000$	$= 320/16$		$\underline{\underline{Q_2 = 0,11}}$
r_2	$= 0.00,10000000$	$= 128/256$		
$2r_2$	$= 0.01,00000000$	$= 256/16$	*set*	$q_3 = 1, Q_3 = 0,110 + 0,001$
$-\left(2Q_2 + 2^{-3}\right)$	$-0.01,10100000$	$= 416/16$		$\underline{\underline{Q_3 = 0,111}}$
r_3	$= 1.11,01100000$	$= -160/256$		
$2r_3$	$= 1.10,11000000$	$= -320/256$	*set*	$q_4 = -1, Q_4 = 0,1110 - 0,0001$
$+\left(2Q_3 - 2^{-4}\right)$	$-0.01,10110000$	$= 432/256$		$\underline{\underline{Q_4 = 0,1101}}$
r_4	$= 0.00,01110000$	$= 112/256$		
corrected final remainder :				
$\underline{\underline{r_{corr} = 0,0111 \cdot 2^{-4} = 0,00000111 = 7/256}}$				
square root :				
$\underline{\underline{Q}}$	$= 0,1101$	$= 13/16$		

Figure 1: Non-Restoring Square Root Algorithm

1.2 The Restoring Square Root Algorithm

The restoring algorithm also works using a two's complement representation. However, unlike to the non-restoring algorithm, the square root and the final remainder are obtained with *i* iterations if the radicand is *i* bit wide. First an enlargement with zeros is necessary, around the double bitwidth. This enlargement is essential because the square root is also determined from the denominator.

$$0,0101 = \sqrt{5/16} \;\rightarrow\; 0,01010000 = \sqrt{80/256} \tag{2}$$

At each iteration four calculations are essential. The radicand r_0 is shifted one digit to the left so that the product $2r_0$ is obtained. Then the value of q_i is determined by the sign of the tentative remainder. For the first iteration $q_0=0$. At the next iterations q_i is constrained of tentative remainder. If this remainder is negative $q_i=0$, otherwise $q_i=1$. Only if $q_i=1$, Q_i is shifted by one digit to the left, afterwards $2Q_i$ and 2^{i+1} are added to $Y=2Q_i+2^{i+1}$. If $q_i=1$ and $r_i \geq 0$, then, the sum $Y=2Q+2^{i+1}$ is subtracted of $2r_i$. The result is the tentative remainder. If $q_i=0$ the partial remainder is less then 0 and the previous remainder must be restored. The remainder is shifted again one digit to the left to get $4r_i$. Following $Y=2Q_i+2^{i+1}$ is subtracted of $4r_i$. Assuming that the radicand has a bitwidth of 4 bit then the correct square root is calculated in four iterations. However, the final remainder must be corrected. If the algorithm finishes in *i* iteration, the final remainder is multiplied by 2^i at the end of computation $(r_{corr}=r_i \cdot 2^i)$. An example is shown in Figure 2.

Rule for q_i :	$q_i = \begin{cases} 1 & if\ 2r_{i-1} \geq 0 \\ 0 & if\ 2r_{i-1} < 0 \end{cases}$			
$X = 0,1011$	$= 11/16$	$= 0,10110000$	$= 176/256$	
$2r_0$	$= 01,0110$	$= 22/16$		
$-\left(0 + 2^{-1}\right)$	$-00,1000$	$= 8/16$		
r_1	$= 00,1110$	$= 14/16$	*set*	$q_1 = 1,\ Q_1 = 0,1$
$2r_1$	$= 01,1100$	$= 28/16$		
$-\left(2Q_1 + 2^{-2}\right)$	$-01,0100$	$= 20/16$		
r_2	$= 00,1000$	$= 8/16$	*set*	$q_2 = 1,\ Q_2 = 0,11$
$2r_2$	$= 01,0000$	$= 16/16$	*is smaller than*	$\left(2Q_2 + 2^{-3}\right)$
$-\left(2Q_2 + 2^{-3}\right)$	$-01,1010$	$= 26/16$		
$r_3 = 2r_2$	$= 01,0000$	$= 16/16$	*set*	$q_3 = 0,\ Q_3 = 0,110$
$2r_3$	$= 10,0000$	$= 32/16$		
$-\left(2Q_3 + 2^{-4}\right)$	$-01,1001$	$= 25/16$		
r_4	$= 00,0111$	$= 7/16$	*set*	$q_4 = 1,\ Q_4 = 0,1101$
correct final remainder is :				
$r_4 \cdot 2^{-4} = 0,0111 \cdot 0,0001$				
final remainder	$= 0,00000111$	$= 12/256$		
radix	$= 0,1101$	$= 13/16$		

Figure 2: Restoring Square Root Algorithm

1.3 The Lookup Table

Lookup tables have a simple data structure. LUTs work with known input values and in relation to them a known output value. The output values are precalculated and saved in a table. The computation of values is not necessary in the system. For every input value a corresponding output value is available. Lookup tables are useful for many applications, because they are very fast. However, they have also an important disadvantage. The size of the lookup table might become too large if many values are stored. For the determination of the square root only integer results are stored without remainder and it is illustrated in Equation (3).

$$\sqrt{80/256} = \sqrt{0,01010000} \qquad = 9/16 = 0,1001 \tag{3}$$

2 VLSI Implementation

The hardware design was written with the hardware description language VHDL according to the VHDL-87 standard [2]. For synthesis was used the Synopsys Design Compiler [3] without any design constrains. The implementation technology used for synthesis is the Europractice ES2 ECPD 0.7µm CMOS technology [4].

2.1 The Non-Restoring Square Root Algorithm

The non-restoring algorithm requires only a limited amount of arithmetic operations. Two multipliers, one subtractor, one decision block, one controller and one correction block appertain to the non-restoring algorithm. The complete processing sequence of the non-restoring algorithm is managed by the control block. The signals *Q* and *r* are handed to the multipliers. The multiplication block is based on a simple shirt structure, since only a multiplication by two is needed. The multiplication by two can be realised by shifting the input by one digit to the left. Both multipliers *2Q* and *2r* are designed using the same approach. The subtraction is implemented with the default VHDL subtractor. This is possible because this subtractor has the best properties for the non-restoring application. After subtraction, the decision module is dependent of the value *q*, which was determined and stored one step before. If value *q* is positive then *2r* and *Y* will be added, otherwise they are subtracted in the decision module. The result of the decision module is the radix. However, in some cases the result must be corrected. For these cases the correction radix module will modify the result and the correct radix will be distributed.

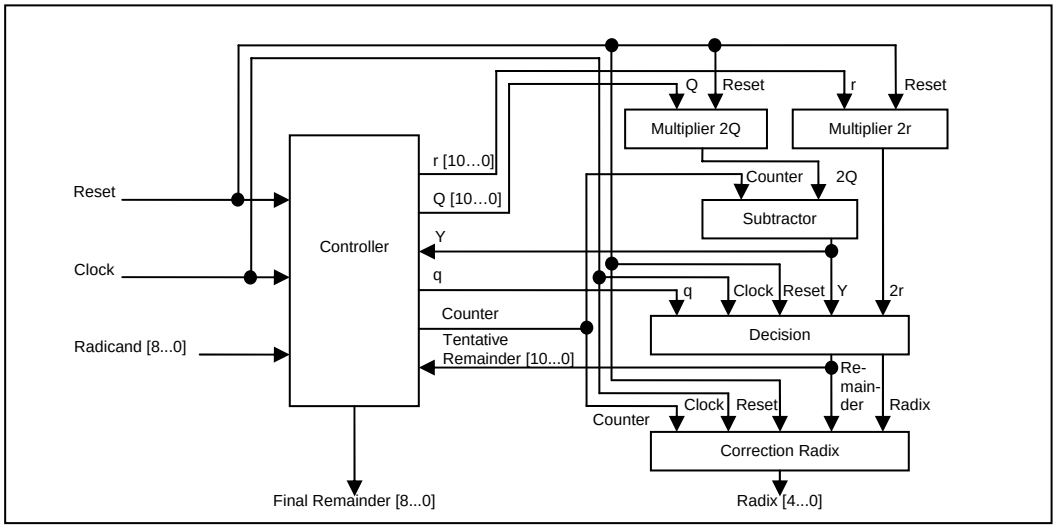

Figure 3: Block Diagram of the Implemented Non-Restoring Square Root Algorithm

As described, the correction is dependent of Y and the tentative remainder. The output of the square root algorithm is controlled by the counter and the final remainder. If the counter reaches the half bitwidth from the radicand or the final remainder is zero then the radix will be issued. The correction of the final remainder is controlled by the controller module. It is dependent on Q, Y and the tentative remainder. A block diagram of the implemented non-restoring square root algorithm is presented in Figure 3.

2.2 The Restoring Square Root Algorithm

The restoring algorithm differs only slightly from the non-restoring algorithm. Therefore, the implementation of the restoring algorithm is similar to the non-restoring algorithm. The entire process is divided into four tasks. Therefore, one controller, two multipliers, one adder, and one subtractor are needed. The controller is responsible for the main task of the process. The first step of the calculation is the multiplication of r_i by 2 whereas r_i is the partial remainder of the last calculation. Since the multiplication by two is a shifting to the left, the structure of the multipliers is very simple. In the next step Q_i is multiplied by 2. After this step, the adder is used for the addition of $2Q_i$ with 2^{-i+1}. This adder is designed with the default VHDL code. The last calculation is performed by subtracting $2Q_i+2^{-i+1}$ from $2r_i$, whereas the subtraction is implemented with the default VHDL subtractor. If this sum $2Q_i+2^{-i+1}$ is greater than $2r_i$, the next calculation is negative. The controller checks the sign of the tentative remainder. If the sign of the remainder is positive, the multiplier will provide the remainder r_{i+1} again through the controller. If the sign is negative $2r_i$ will be provided for the next iteration. Therefore, the subtract block has two outputs. The new remainder r_{i+1} will be used for the first output and the restore remainder $2r_i$ will be used for the second output. If the restoring algorithm has calculated the square root of the radicand, the final remainder is determined in the last step. The controller calculates the correct final remainder. The complete overview of all modules is presented in the following Figure 4.

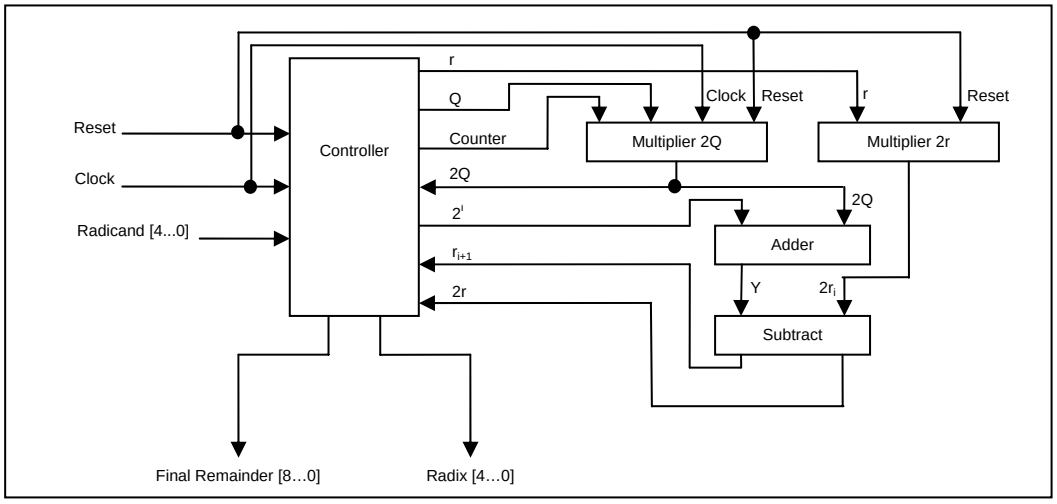

Figure 4: Block Diagram of the Implemented Restoring Square Root Algorithm

2.3 The Lookup Table

This section describes the implementation of the lookup table. The LUT is a direct implementation of precalculated values which are stored in a table. No arithmetic operations are carried out. At the implementation an array will be created where all values will be stored. The dimension of the values stored is particularly large. With an input bitwidth of 4 bit 16 values must be stored, and with an input bitwidth of 8 bit 256 values must be stored. The example illustrates that the duplication of the bitwidth leads to quadratic number of values stored. The implementation of 32 bit and 64 bitwidth has presented that the memory of the synopsis design compiler is too small for the amount of values. Therefore, the input values with a bitwidth of 32 and 64 bit were interpolated to compare the LUT with the other algorithms used.

3 Simulation Results

This chapter summarises the results of the different square root algorithms and the lookup table implemented. The non-restoring and the restoring algorithm were implemented with a bitwidth of 4 bit, 8 bit, 16 bit, 32 bit and 64 bit. The lookup table was implemented with a bitwidth of 4 bit, 8 bit and 16 bit. The results of 32 bit and 64 bit were interpolated for comparison with all bitwidths used. In the following chapter the results are presented for the power consumption, the area consumption and the propagation delay.

3.1 Power Consumptions

The square root algorithms and the lookup table implemented were tested towards their power efficiency. The mode of operation of LUTs and square root algorithms is different at several bitwidths and therefore, the both algorithms and the LUT were investigated with different bitwidths. The power consumption is described by the active capacitance of the transistors, by the supply voltage and by the switching frequency and can be calculated as follows:

$$P = C_L V_{dd}^2 f_{clk} \tag{4}$$

The power consumption arises only when a gate changes its state. Therefore, a test program, PowerCount was used, that creates different random test vectors and measures the active capacitance [5]. For the tests in Figure 5 the following values were set a clock frequency of 10 MHz was chosen and the power supply was set to 5 Volt. Figure 5 shows the power consumption of the different algorithms implemented.

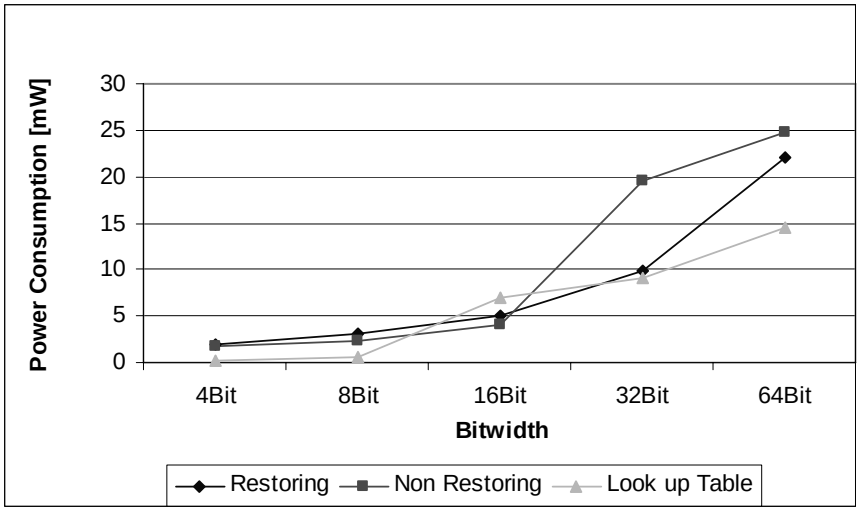

Figure 5: The Power Consumption

When comparing both algorithms and the lookup table, it can be seen that the non-restoring requires the most power. This can be explained with the higher number of arithmetic operations at the non-restoring algorithm. The lookup table requires the least power consumption, because it only accesses stored values. With a larger bitwidth it can be seen, that the power consumption of the algorithms and the lookup table changes. If the bitwidth will be increased the computation of the square root will be more complex. Therefore, the power consumption of the non-restoring and the restoring increases with the bitwidth whereas the power consumption of the lookup table remains nearly constant. The non-restoring algorithm has larger power consumption than the restoring algorithm. This can be explained with the additional correction at the non-restoring algorithm.

3.2 Area Consumption

One objective of any hardware design is to achieve a small silicon area. Therefore, the area of the different implantations is explored next. The design of the hardware structure can be controlled by the user or it can be automated by the compiler. If no design constrains are set by the user, the compiler searches the best trade-off between area and timing. The area consumption of the square root algorithms and the lookup table is presented in Figure 6.

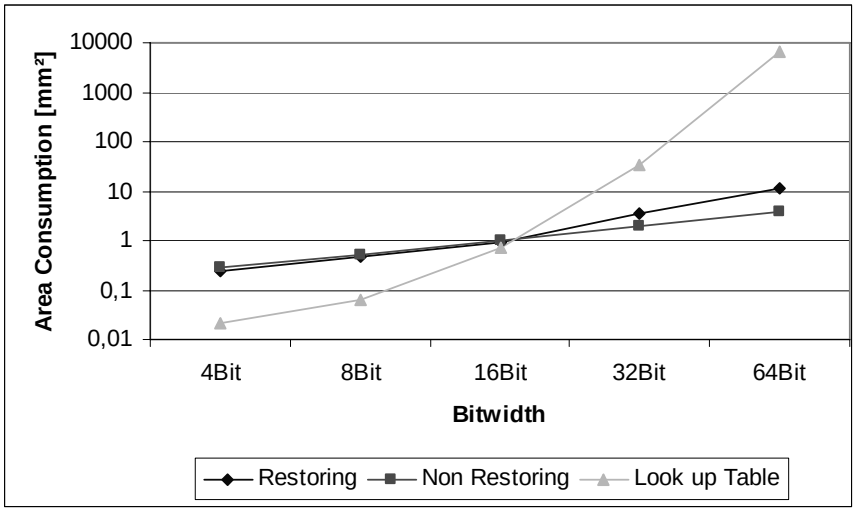

Figure 6: The Area Consumption

The algorithms implemented and the lookup table have different area requirements. Since, the stored values are still small for the lookup table at 4 bit and 8 bit, the areas consumption is smaller than that of the calculated algorithms. The non-restoring and restoring algorithms always require the same arithmetic operations and consequently the same arithmetic modules will be needed at different bitwidths. Only the operations steps get larger. If the input vector gets twice to long, then the restoring algorithm and the non-restoring algorithm need twice as much steps for the calculation of the square root. For the lookup table, the number of the stored values increases squarely. Therefore, the lookup table has initially a small area requirement at 4 bit and 8 bit. However, the implementation of the LUT for 16 bit leads to an increased silicon area. As mentioned above, the implementation of the lookup tables with a bitwidth of 32 and 64 bit is not possible because the calculation exceeds the limit of the RAM available in the synthesis server.

3.3 Propagation Delay

The propagation delay is the time a signal requires from one functional block of a system to another. In this case the authors are providing the worst case delay of the input of the circuit until the output has settled.

As can be seen in Figure 7 the lookup table has the lowest propagation delay, because it needs no arithmetic operation. Since the input vector only causes a memory write operation, the propagation delay is short. This timing behaviour is not considerably changed with an enlargement of the bitwidth. As can be seen the non-restoring algorithm needs the longest to perform the calculation, because it has the most arithmetic modules and additional the results have to be corrected. Increasing the bitwidth, the iterations required are increased and thus the signal propagation delay gets longer.

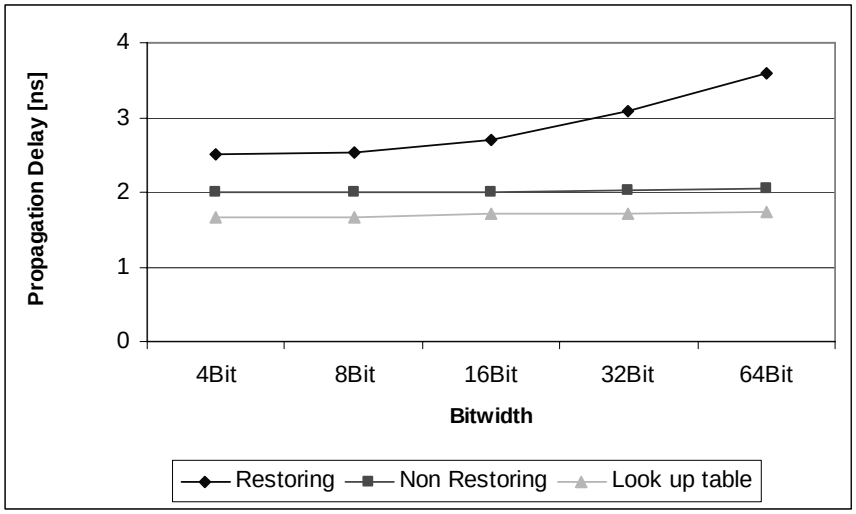

Figure 7: The Propagation Delay

The restoring algorithm requires less propagation delay as it is arithmetically less intensive and thus the result is faster computed. The propagation delay of the restoring algorithm increases continuously with the bitwidth.

4 Conclusions

This paper has investigated the implementation of three different square root algorithms into CMOS. Firstly, the mode of operation of the non-restoring, restoring and the lookup table was described. There the computation steps were investigated and explained. Furthermore, it was demonstrated that all three square root computations work with fixed point values between 0 and 1. During the process the mathematical complexity of the different algorithms was shown. Secondly, the implementation of the several algorithms into hardware was described. Various implementations were realised with different bitwidths in high-level VLSI design and using the hardware description language VHDL. The different bitwidths were implemented to compare the algorithms and the LUT in relation to the power consumption, the area requirements and the propagation delay. These properties were illustrated to compare the advantages and disadvantages of each algorithm. In the process it was shown, that the non-restoring, the restoring and the lookup table have different trade-offs. Therefore, with the diagrams presented in this paper the hardware designer is now are to pick the best possible implementation for a given problem.

5 References

[1] Koren, I. (Spring 2004). Sequential algorithms for multiplication and division. *Digital Computer Arithmetic ECE666.*

[2] Ashenden, P. J. (2002). *The Designer's Guide to VHDL.* Second Edition, Morgan Kaufmann Publishers.

[3] Synopsys, www.Synopsys.com

[4] European Silicon Structures (July 1995). *ES2 ECP D07 Library Databook.*

[5] A.Th. Schwarzbacher, P.A. Comiskey and J.B. Foley, *"Powercount: measuring the power at the VHDL netlist level,"* Electronic Devices and Systems Conference, Bruno, Czech Republic, pp. 70-73, June 1998.