Journal of Systems Architecture 54 (2008) 957–966

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Modular array structure for non-restoring square root circuit

S. Samavi *, A. Sadrabadi, A. Fanian

Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran

ARTICLE INFO

Article history:
Received 19 June 2007
Received in revised form 31 March 2008
Accepted 1 April 2008
Available online 11 April 2008

Keywords:
Square root
Non-restoring algorithm
CMOS
Expandability
Standard cell
VLSI

ABSTRACT

Square root is an operation performed by the hardware in recent generations of processors. The hardware implementation of the square root operation is achieved by different means. One of the popular methods is the non-restoring algorithm. In this paper, the classical non-restoring array structure is improved in order to simplify the circuit. This reduction is done by eliminating a number of circuit elements without any loss in the precision of the square root or the remainder. For a 64-bit non-restoring circuit the area of the suggested circuit is about 44% smaller than that of a conventional non-restoring array circuit. Furthermore, in order to create an environment for modular design of the non-restoring square root circuit, a number of modules are suggested. Using these modules it is possible to construct any square root circuit with an arbitrary number of input bits. The suggested methodology results in an expandable design with reduced-area. Analytical and simulation results show that the delay of the proposed circuit, for a 64-bit radicand, is 80% less than that of a conventional non-restoring array circuit.

1. Introduction

In scientific computations next to the basic mathematical operations of addition, subtraction, multiplication, and division, square root is the most useful and vital operation. Numerical analysis, complex number computations, statistical analysis, computer graphics, and radar signal processing are among the fields where square root has relevance [1]. The early processors performed the square root operation by software means, which resulted in long delays for its completion. With the advancement of technology and possibility of integrating large circuits on a single chip and also increase in demand for higher computational speeds, hardware implementation of many of the operations that previously had been performed by software became more attractive.

There are different methods for computing the square root operation, which can be grouped into two distinct categories:

- Methods that compute the square root in a digit-by-digit manner, such as restoring, non-restoring [2,3], and SRT [4]. The precision of these methods, which compute only one digit of the square root at each step, is directly dependent on the number of iterations. Hardware implementation of these methods usually consists of an array of simple computational modules. Each row of the array performs one iteration of the algorithm.
- Estimation methods, such as Newton–Raphson [5], which are based on iterative solving of nonlinear equations. In these methods, the square root operation starts with an initial estimation of the square root or its inverse. This initial estimation is improved through a recursive operation, which finally generates the square root of the intended number [6–8].

Design of integrated circuits with large number of transistors is a lengthy and costly process. If these circuits consist of a number of simple and repeated blocks, then the design process is significantly simplified. Modularity, which is a quality factor for the design of digital circuits, is defined as the ratio of total number of transistors in a circuit to the number of transistors that are used in unique modules that are designed separately. This factor should be as high as possible. If one can design a large circuit with the aid of one or two small blocks, which are used repeatedly, then the modularity factor will be a large number [9].

Unlike many arithmetic circuits, such as adders and multipliers, the square root circuits are not modular and regular. For example in carry propagate adders a larger module can be constructed by concatenating modules. This is not true for the current designs of square root circuits with array structures. In the present paper, a modular design is suggested for a non-restoring square root circuit. In many modular designs, for the sake of regularity, some blocks contain parts that are not always useful. An example of such a design is the array multiplier where some of the full adders can be replaced by half adders, but the replacement would reduce the regularity of the design. In our proposed structure, while keeping regularity, all of the internal elements of the circuit blocks are necessary for the correct function of the circuit. Also, we eliminated non-essential circuit elements of the existing non-restoring square root circuit. Then, two different modules are designed for the suggested array formation. Using these modules, any non-restoring

* Corresponding author. Tel.: +98 311 391 5431; fax: +98 311 391 2451.
E-mail address: samavi96@cc.iut.ac.ir (S. Samavi).

1383-7621/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2008.04.004

square root circuit with any number of input bits can be designed. The simulation results are compared with those of a standard non-restoring circuit in order to show the superiority of the suggested design. Finally, we discuss hardware requirements and delay analysis of the design.

2. Non-restoring square root algorithm

The non-restoring square root algorithm is a digit-by-digit scheme where at each iteration of the algorithm only one digit of the square root is generated [10]. In an iterative loop square root digits are selected based on the sign of the partial remainder. If the partial remainder is negative the square root digit is selected as -1 otherwise 1 will be selected at that stage. This strategy is in contrast with the *restoring* algorithm where a negative partial remainder is not tolerated and must be restored to a positive value.

Suppose a $2n$-bit operand X is such that $\frac{1}{4} \leqslant X < 1$, hence, its square root Y will be such that $\frac{1}{2} \leqslant Y < 1$. Eq. (1) shows a recursive relation that generates the square root.

$$Y_i = Y_{i-1} + y_i \times 2^{-i} \tag{1}$$

where Y_{i-1} is the partial square root thus far and y_i is the square root digit at the ith iteration. When working with radix 2 numbers, the value of y_i appended to the right of a fractional Y_{i-1} at the ith position requires a scaling by 2^{-i}. The partial remainder at the ith stage is shown in the following equation:

$$R_i = X - Y_i^2 \tag{2}$$

With substitution for Y_i in Eq. (2) we get

$$R_i = X - (Y_{i-1} + y_i \times 2^{-i})^2 \tag{3}$$

$$R_i = (X - Y_{i-1}^2) - y_i 2^{-i}(2Y_{i-1} + y_i \times 2^{-i}) \tag{4}$$

$$R_i = R_{i-1} - y_i 2^{-i}(2Y_{i-1} + y_i \times 2^{-i}) \tag{5}$$

Before computing R_i we need to decide on the value of y_i based on the sign of R_{i-1} in the following manner:

$$y_i = \begin{cases} -1 & \text{if} \quad R_{i-1} < 0 \\ +1 & \text{if} \quad R_{i-1} \geqslant 0 \end{cases} \tag{6}$$

If the signed-digit square root Y were to be computed first and then converted to its 2's complement equivalent, Q, then the following algorithm could have been employed [11]

(a) Replace all -1 digits with 0's to get an n-bit number $Q = 0 \cdot q_1 q_2 \ldots q_n$ with $q_i \in \{0,1\}$.
(b) Complement q_1 and append a 1 bit to the LSB position to get the 2's complement quotient $Q = 0 \cdot \bar{q}_1 q_2 \ldots q_n 1$

Since we want to use regular logic circuits for implementation of the algorithm we cannot use signed-digits such as -1. These signed y_i digits are converted to q_i binary digits to make up Q, the binary square root. Since Y_0, which is the radicand, is always positive, Q is positive, and hence $q_0 = 0$. In the other steps if $y_i = +1$ then $q_{i-1} = 1$ and if $y_i = -1$ then $q_{i-1} = 0$. Therefore, the above mentioned algorithm can be modified to perform on the fly conversion from Y_{i-1} to Q_{i-1} in the following manner:

$$Q_{i-1} = \begin{cases} Y_{i-1} - 2^{-(i-1)} & \text{if} \quad y_i = -1 \\ Y_{i-1} & \text{if} \quad y_i = +1 \end{cases} \tag{7}$$

Now using Eq. (7) we can express Y_{i-1} in terms of Q_{i-1} and by plugging it back into Eq. (5) partial remainder at iteration i is obtained as

$$R_i = \begin{cases} R_{i-1} + (2Q_{i-1} + 3 \times 2^{-i})2^{-i} & \text{if} \quad R_{i-1} < 0 \\ R_{i-1} + (2Q_{i-1} + 2^{-i})2^{-i} & \text{if} \quad R_{i-1} \geqslant 0 \end{cases} \tag{8}$$

It should be noticed that $2Q_{i-1} + 3 \times 2^{-i}$ is equivalent to one position left-shift of Q_{i-1} and appending of 11 to the right of the shifted Q_{i-1}. Also, $2Q_{i-1} + 2^{-i}$ is interpreted as left shifted Q_{i-1} with a 01 appended to it at the least significant position. Furthermore, multiplication by 2^{-i} is done by shifting the operand i positions to the right. Hence, we rewrite Eq. (8) in the following form:

$$R_i = \begin{cases} R_{i-1} + 0.00\ldots0q_1q_2q_3\ldots q_{i-1}11 & \text{if} \quad R_{i-1} < 0 \\ R_{i-1} - 0.00\ldots0q_1q_2q_3\ldots q_{i-1}01 & \text{if} \quad R_{i-1} \geqslant 0 \end{cases} \tag{9}$$

In Eq. (9), the least significant position has a value of 2^{-2i}, hence the number of leading zeros before q_1 can be calculated. Fig. 1 shows the five main steps of the non-restoring algorithm for performing square root of real fixed point numbers [10,11]. In this algorithm X is a $2n$-bit operand and its square root is an n-bit number Q such that

$$X = Q^2 = 0.x_1x_2\ldots x_{2n}$$
$$Q = 0.q_1q_2\ldots q_n \approx \sqrt{X}$$

Steps 1 and 2 in Fig. 1 show the initialization of the algorithm based on Eq. (9), where $R_0 = 0.x_1x_2$ is always positive, and hence an initial subtraction is always performed. After R_1 is formed, from there on, steps 3–5 of the algorithm are iterated. At each iteration of these steps, two bits of the operand are augmented to the right of the previous step's partial remainder to form R_i. On the other hand, square root bits are right shifted and augmented with two constant digits of 01 or 11 to form a variable called F_i. Based on the value of R_i, a square root digit q_i is selected. The next partial remainder R_{i+1} is formed by adding either F_i or $-F_i$ to R_i. A classical hardware implementation of the non-restoring algorithm by an array structure is shown in Fig. 2 [12,13]. The main building blocks of the array circuit are blocks known as controlled add-subtract (CAS). The details of a CAS block are presented in Fig. 3, where a full adder (FA) can be seen with an XOR at one of the FAs inputs.

Each row of the circuit in Fig. 2 performs one iteration of the non-restoring square root algorithm, where either an addition or a subtraction is carried out. A row of CAS blocks can perform either an addition or a subtraction depending on the value of the control signal. Referring to Fig. 3, if the control signal P of a CAS is 0, then the sum of inputs D, X, and C_{in} appears at outputs C_{out} and R. If line P is set to 1, then the one's complement of D is added to X and C_{in}, which by using the arrangement of Fig. 2 this performs a 2's complement subtraction. For ease of reference, from here on, the classical non-restoring circuit is referred to as the *Classical-NR* design. Simulation results for a 16-bit *Classical-NR* are shown in Fig. 4, where the square root of $0.\text{C000}_\text{H}$ is found to be $0.\text{DD}_\text{H}$ with a remainder of 0.0137_H. These waveforms show the correct functionality of the circuit and its delay. The inherent ripple effects in the circuit produce some intermittent results before the circuit's outputs reach to a set of final results. In the following sections we propose improvements to the *Classical-NR* design. By comparing the waveforms of the proposed designs with those of Fig. 4 we can show the delay reduction as well as the correctness of the circuits. All circuits are tested with a wide range of numbers but the waveforms of only one example for each design are illustrated.

In Fig. 2, in any row, i, there are $2i$ number of CAS blocks. Furthermore, the connections between the CAS blocks are not identical. In each row, the D input of the first block, at the right end, is 1. The second block has its D input as the complement of the P input. Then the third block has its D input coming directly from the P input. For the rest of the blocks of a row, the D input is taken from the D output of a block in the previous row. These differences in the connections cause the structure to not be a modular one. The aim of a modular design in this context is to come up with a number of blocks that by laying them next to each other, without need for other circuit elements, such as logic gates, one can construct a

```
algorithm Non-restoring square root is
    input: 2n-bit operand X
    output: n-bit quotient Q,
        n-bit remainder R
    Step 1: $R_0=0.x_1x_2$ ; $F_0=0.01$; $i=1$ ;
    Step 2: $R_1=R_0-F_0$;
    Step 3: If ($R_i<0$)
        then ($q_i \leftarrow 0$ ;$R_i \leftarrow R_i . x_{2i+1}x_{2i+2}$ ;$F_i \leftarrow 0.0...0q_1q_2...q_i11$ ; $R_{i+1} \leftarrow R_i+F_i$ ;)
        Else ($q_i \leftarrow 1$ ;$R_i \leftarrow R_i . x_{2i+1}x_{2i+2}$ ;$F_i \leftarrow 0.0...0q_1q_2...q_i01$ ; $R_{i+1} \leftarrow R_i-F_i$ ;)
    Step 4: $i \leftarrow i+1$;
    Step 5: If ($i \le$ n) then (go to step 3)
            Else ($Q=0.q_1q_2q_3...q_n$);
End;
```

Fig. 1. Non-restoring square root algorithm.

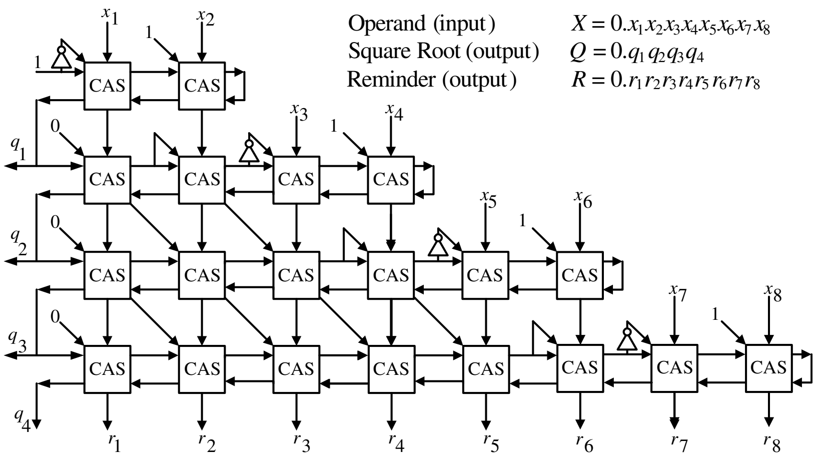

Operand (input) $X = 0.x_1x_2x_3x_4x_5x_6x_7x_8$
Square Root (output) $Q = 0.q_1q_2q_3q_4$
Reminder (output) $R = 0.r_1r_2r_3r_4r_5r_6r_7r_8$
CAS
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
q_1 q_2 q_3 q_4
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8
1
0

Fig. 2. 8-bit classical non-restoring square root circuit using CAS blocks.

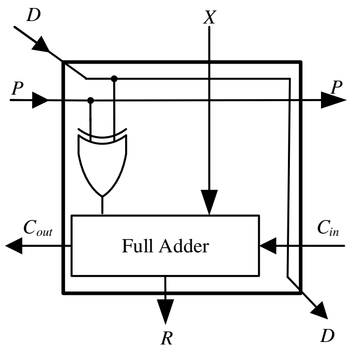


Fig. 3. Internal structure of a CAS block.

square root circuit, which can be expanded for any number of input bits.

Implementation of a CAS block at transistor level using CMOS technology can be found in Refs. [13,14]. Many different designs have been proposed for both full adder and XOR because these two are widely used in computational circuits. For the sake of comparison of different architectures that we propose in this paper, we consider a typical implementation of these circuits which is a 28-transistor full adder and a 10-transistor XOR [14]. These transistor counts just lay down a basis for comparison and will not affect the overall arguments. In other words, if we improve the *Classical-NR* circuit by reducing the number of CAS blocks we can claim that the overall area that the improved design requires on a chip has been reduced as compared to the *Classical-NR* circuit. This reduction in area would be irrespective of the transistor implementations of the CAS blocks.

3. Improved non-restoring circuit

In Fig. 2 to obtain the partial remainder at the *i*th step, a 2*i*-bit add/subtract circuit is used. The add/subtract circuit is constructed from 2*i* number of CAS cells. We want to show that some of these CAS cells are redundant and can be eliminated.

Lemma. *At the* ith *iteration of the non-restoring square root algorithm,* i *+ 2 bits are sufficient to represent the remainder.*

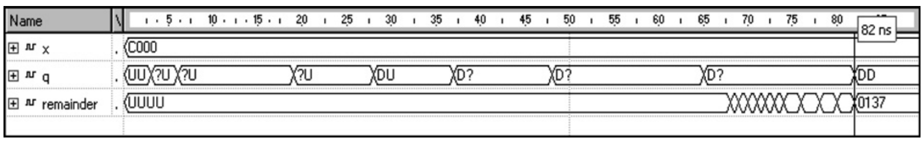


Fig. 4. Timing and functionality test for the classical non-restoring circuit (*Classical-NR*).

Proof. Assuming that $Q = q_1q_2 \ldots q_n$ is the n-bit square root of a $2n$-bit number, X, then the remainder, R, is

$$R = X - Q^2 \tag{10}$$

For Q to be the square root of X, then $(Q + 1)^2$ has to be greater than X, therefore

$$(Q+1)^2 = Q^2 + 2Q + 1 > X \tag{11}$$

Using (11) an upper limit for the remainder of the square root is obtained

$$R = X - Q^2 < 2Q + 1 \tag{12}$$

Therefore, the maximum value for the remainder is

$$R_{\max} = 2Q \tag{13}$$

This means that the remainder is at most two times the obtained square root. In the non-restoring algorithm, it is possible to have negative remainders. This happens when the computed square root is one unit more than the actual value. In this case, if R' is the negative remainder, the following shows its value as a function of the actual remainder, R, and the actual square root, Q

$$R' = X - (Q+1)^2 = R - (2Q+1) \tag{14}$$

Hence, the most negative value that the remainder of the square root operation can assume is

$$R_{\min} = -(2Q+1) \tag{15}$$

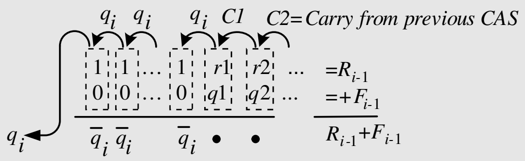
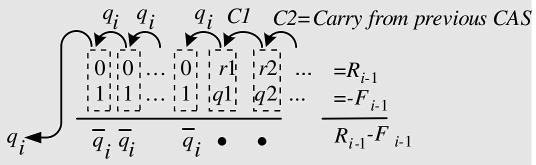
We know that Q is a positive n-bit number that is produced in n iterations. The two's complement representation of the remainder R requires at most $n + 2$ bits. Hence, in general, at the ith iteration, where an i-bit square root is available, the remainder can be represented by at most $i + 2$ bits. □

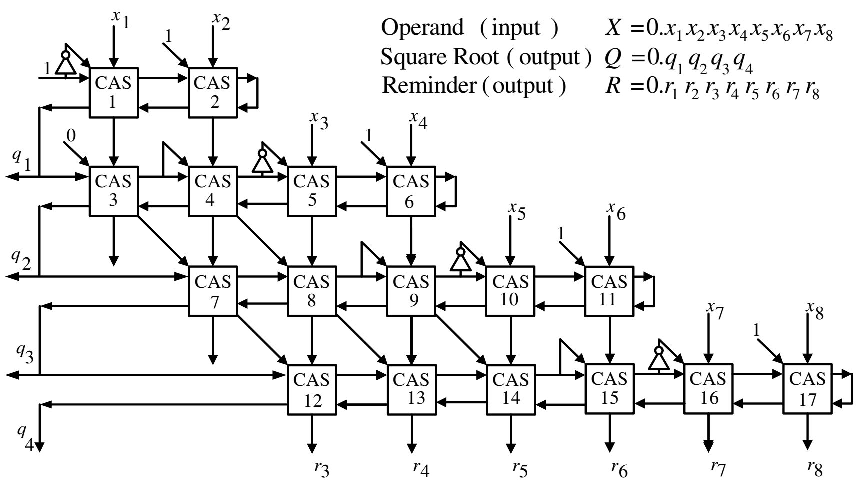
Based on the above lemma we notice that at the ith step, it is sufficient to use an add/subtract circuit with $i + 2$ bits. This implies that in the ith row of a *Classical-NR* circuit, rather than using $2i$, only $i + 2$ number of CAS blocks are required. Hence, $i - 2$ blocks in the ith row are redundant. In the following we show which CAS blocks are redundant.

In a *Classical-NR* circuit, such as the one in Fig. 2, at the ith row, q_i is produced, which is one bit of the square root. This q_i is actually the carry out of a 2's complement addition of $R_{i-1} \pm F_{i-1}$. The addition with F_{i-1} is performed if R_{i-1} were to be negative. If that is the case, then q_{i-1} is 0 and the sign of R_{i-1} is 1. The second case that can occur is a positive R_{i-1}, carry out of the row is 1, and the sign of R_{i-1} is 0. Table 1 shows the only two possible cases that can happen at the ith row.

It is shown in Table 1 that for either case of $R_{i-1} \pm F_{i-1}$, the produced carry at the ith row is not known, but this carry is propagated through $i - 2$ leftmost CASs of each row. Hence, the CAS blocks that only propagate a row's carry and produce identical outputs are redundant. By eliminating these redundant blocks, the structure of Fig. 5 is obtained. From here on, this circuit is referred to as the *Reduced-Area-NR* circuit. Simulation results for this type of structure, with 16-bit radicand, are shown in Fig. 6.

Table 1
Illustration of redundancy in sign extension part of *Classical-NR* circuit

Carry out of row $i - 1$	Sign ($R_i - 1$)	Operation at ith row	Value of carry out of ith row
$q_{i-1} = 0$	1	$R_{i-1} + F_{i-1}$	
$q_{i-1} = 1$	0	$R_{i-1} - F_{i-1}$	



Operand (input) $X = 0.x_1x_2x_3x_4x_5x_6x_7x_8$
Square Root (output) $Q = 0.q_1q_2q_3q_4$
Reminder (output) $R = 0.r_1r_2r_3r_4r_5r_6r_7r_8$
CAS 1 … CAS 17
q_1 q_2 q_3 q_4
r_3 r_4 r_5 r_6 r_7 r_8

Fig. 5. Non-restoring circuit with redundant CAS blocks eliminated (*Reduced-Area-NR*).

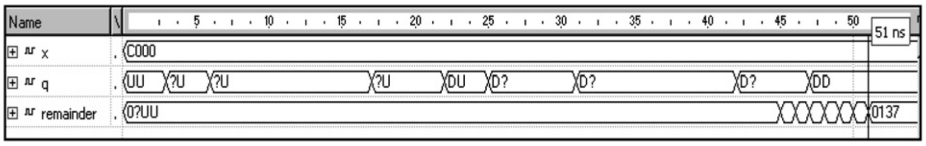


Fig. 6. Timing and functionality test for *Reduced-Area-NR* circuit.

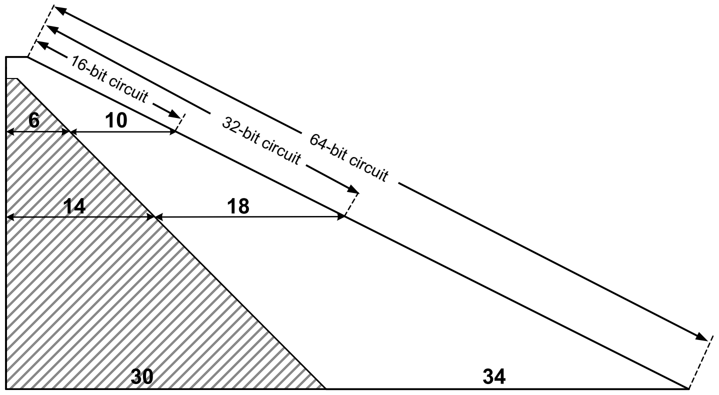


Fig. 7. Comparison between *Classical* and *Reduced-Area-NR* circuits in terms of required CASs.

In Fig. 6, a 16-bit radicand *X* with a value of C000$_H$ is fed in a *Reduced-Area-NR* circuit. The computed square root of DD$_H$ and remainder of 0137$_H$ are accurate and show the correct functionality of the design. Furthermore, we see that there are fewer ripples in the waveforms of Fig. 6. This is due to the presence of fewer CAS blocks in this circuit as compared to the circuit that produced waveforms of Fig. 4. Hence, the remainder output of the *Reduced-Area-NR* circuit is valid after 51 ns while the *Classical-NR* circuit produces a similar output in 82 ns.

Fig. 7 shows a comparison between the *Classical* and the *Reduced-Area-NR* circuits.

By referring to Fig. 2 we see that a *Classical-NR* circuit layout resembles a right trapezoid. The height of the trapezoid is the number of CAS rows, which in turn is the number of bits of the square root. The top base of the trapezoid is always 2 since there are always two CAS blocks in the first row of the circuit. Based on the algorithm of Fig. 1, we observe that the bottom base of the trapezoid has as many CAS blocks as the number of bits in the radicand. In Fig. 7 three examples for 16-bit, 32-bit, and 64-bit *Classical-NR* circuits are illustrated. For example, for a 64-bit *Classical-NR* circuit, the length of the bottom base is 64, since there are 64 CASs in its last row. Also, the height of the trapezoid in this case is 32, since there are 32 rows in this circuit. In Fig. 7 the redundant CAS blocks are shown as the hatched area. From the third row of any *Classical-NR* circuit we encounter the redundant CSA blocks. As the size of the circuit is increased to accommodate larger radicands, the number of CAS blocks, that are redundant and are eliminated in the *Reduced-Area-NR* circuit, increases. For example, in a 64-bit *Classical-NR*, just in the last row, 30 out of the 64 CAS blocks are redundant. The number of CAS blocks in a 2*n*-bit *Classical-NR* circuit is $\sum_{i=1}^{n} 2i$ while this number is reduced to $2 + \sum_{i=1}^{n}(2+i)$ for a *Reduced-Area-NR* circuit. For a *Classical-NR* circuit with 64-bit radicands there are 1056 CASs while there are only 591 CASs in a *Reduced-Area-NR* circuit. Therefore, for a 64-bit circuit, the proposed *Reduced-Area-NR* circuit results in a 44% reduction in the required hardware.

4. Modular design

In this section, we intend to take the *Reduced-Area-NR* circuit of the previous section and partition it into groups of blocks, called modules. This will consist of partitioning seven CAS blocks into an X module and partitioning 4 neighboring CAS blocks into a Y module. This strategy increases the modularity of the design. Therefore, by improving one module in terms delay or area, the whole circuit is improved. Each X module consists of seven CAS blocks from two consecutive rows. Fig. 8 shows the internal structure and connections of the X module. The X module is actually a portion of the circuit in Fig. 5. CAS$_9$ to CAS$_{11}$ of Fig. 5 correspond to CAS$_a$ to CAS$_c$ of the X module. Also, CAS$_{14}$ to CAS$_{17}$ of Fig. 5 correspond to CAS$_d$ to CAS$_g$ of Fig. 8. The circuit of Fig. 8 is an 8-bit Reduced-Area-NR circuit. In general, for any size circuit of this type, X modules can be formed at the far right of every two rows. To the left of each X module, the rest of the CAS blocks of a Reduced-Area-NR circuit have regular and direct connections. The inputs of CAS blocks in an X module are same as the ones shown in Fig. 5. Therefore, in Fig. 8 we see that the *D* inputs of CAS$_c$ and CAS$_g$ are set to 1. Also, the *D* inputs of CAS$_b$ and CAS$_f$ of Fig. 8 are the complement of the *P* inputs of those cells. And, the *D* inputs of CAS$_a$ and CAS$_e$ are directly taken from the *P* inputs of the blocks.

A 4-bit square root circuit is actually the same as the first two rows of Fig. 5. Hence, to implement a 4-bit circuit only one X module is needed. We see that an X module has 3 CAS blocks in its first row and to perform a 4-bit square root we only need two CAS blocks in the first row. This means that we do not need CAS$_a$. We need to set PI1 = 1 to perform the first subtraction that is required by the algorithm. Furthermore, by setting $A_0 = 1$ we force CAS$_a$ of Fig. 8 to only propagate its input carry to its output and play no other role. Same as in Fig. 5, where the carry out of the first row is fed to the *P* input of the second row, CO1 has to be connected to PI2. Now that the radicand is a 4-bit number it is fed through A_1 to A_4 and the square root is taken at CO1 and CO2 outputs. In this way, $0 \cdot A_1A_2A_3A_4$ is the input, $Q = 0 \cdot q_1q_2$ is the computed square root, and the remainder is available at R_1 to R_4. To get the

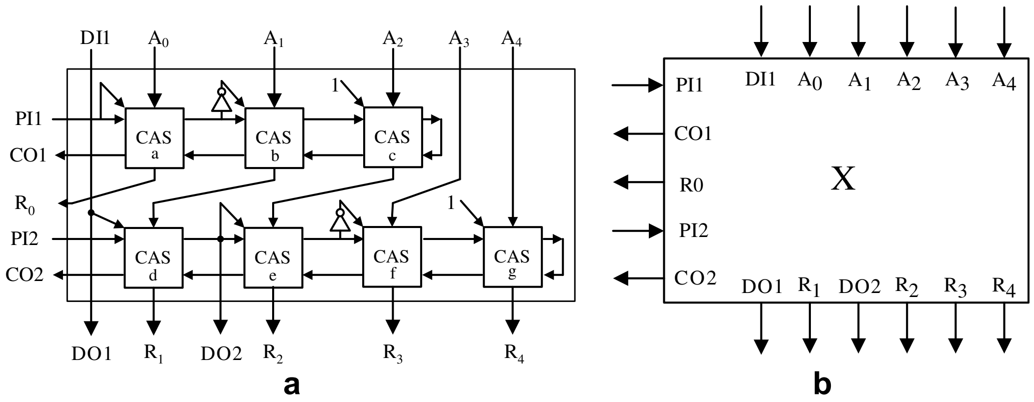


Fig. 8. The X module, (a) internal structure, and (b) block diagram.

first bit of the square root, q_1, using the algorithm of Fig. 1, a two's complement addition between A_1A_2 and -01 should be performed. The 2-bit partial remainder from the first row is $r_{11}r_{12}$ and the carry output is q_1 which appears at CO1. To get the second bit of the square root, q_2, it is necessary to do $(r_{11}r_{12}A_3A_4 + (-0q_101))$ if $q_1 = 1$ and to do $(r_{11}r_{12}A_3A_4 + 0q_111)$ for $q_1 = 0$. This 4-bit addition is performed by CAS$_d$ to CAS$_g$. Therefore, it can be seen that one X module can perform a 4-bit square root operation.

Now that we can carve out an X module, for example from the 3rd and 4th rows of Fig. 5, we see that four CAS blocks are left in these two rows.. These four CAS blocks are grouped together to form a Y module. Fig. 9 shows the details of a Y module. CAS$_7$, CAS$_8$, CAS$_{12}$, and CAS$_{13}$ of Fig. 5 correspond to CAS$_h$, CAS$_i$, CAS$_j$, and CAS$_k$ of a Y module. The D input of each CAS in a Y module is independent of the P input, unless intentionally tied together. It can be seen that the circuit of Fig. 5 can be partitioned into two X and one Y modules.

Fig. 10 shows the connections between the X and Y modules in order to construct a 16-bit square root circuit. As mentioned before, using the X and Y modules, the square root of any $4n$-bit operand can be calculated. To do so, n number of X modules and $n(n-1)/2$ number of Y modules are arranged in n rows. In the first row, one X module is present. In the second row one X module to the right, and one Y module at the left exist. In the following rows, the number of Y modules is increased by one, while only one X module is present at the far right of each row. In other words, at the ith row there are $i-1$ number of Y modules and one X module. Fig. 11 shows the floor plan and the general location of each module in a $4n$-bit square circuit. The inputs are applied to the A_1 through A_4 inputs of the X modules.

When we use the X and Y modules to construct a square root circuit, we refer to the design as the *Modular-NR* circuit. The simulation results for a 16-bit*Modular-NR* are presented in Fig. 12.

To see the effect of partitioning in a *Reduced-Area-NR* circuit the waveforms of Fig. 12 are presented where a radicand of $A_1A_2\ldots A_{16} = 0.\text{C000}_\text{H}$ is applied to the circuit of Fig. 10. The square root of $0.q_1q_2q_3\ldots q_8 = 0\cdot\text{DD}_\text{H}$ is produced. For this specific example the output settles down after 53 ns, as compared to the 51 ns of Fig. 6. The reason for this extra delay is the one extra CAS in the first X module of the circuit. Except for this difference the rest of the circuit is identical to a *Reduced-Area-NR* design.

5. Delay analysis and hardware complexity

In order to compute the hardware complexity of *Modular-NR* design the number of transistors of the X and Y modules must be found out. An X module consists of seven CAS cells and two NOT gates. Each CAS consists of one full adder with 28 transistors and one XOR which has 10 transistors. We also consider a typical CMOS NOT gate with 2 transistors. In an X module there are seven CAS blocks and two NOT gates. Each CAS consists of 38 transistors and the two NOT gates require 4 transistors. Therefore, the number of transistors of an X module with seven CAS blocks and two NOT gates is

$$T_X = 7 \times 38 + 4 = 270 \tag{16}$$

The Y module consists of only four CAS cells. Hence, its number of transistors is equal to

$$T_Y = 4 \times 38 = 152 \tag{17}$$

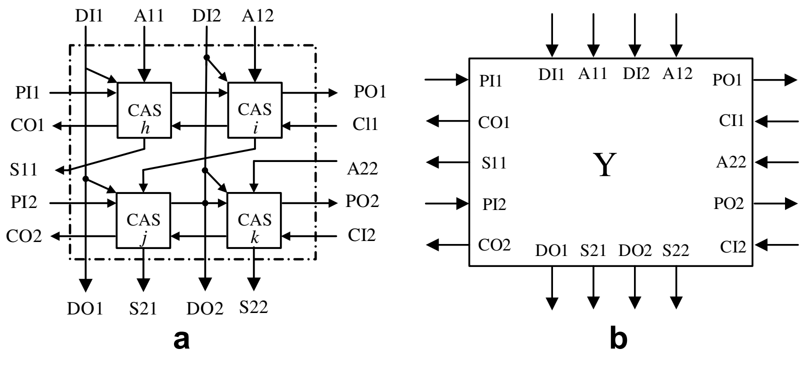


Fig. 9. The Y module, (a) internal structure, and (b) block diagram.

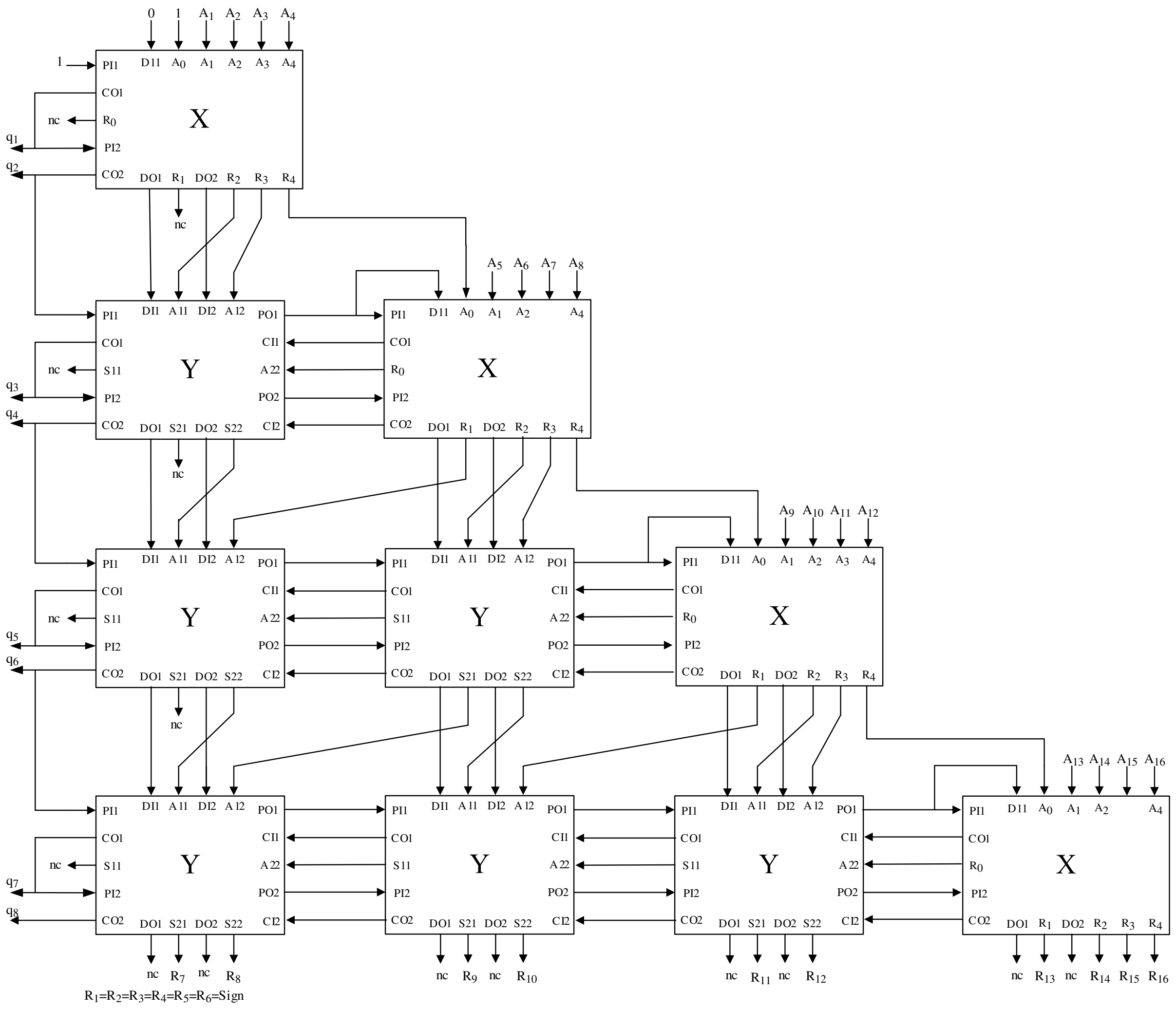


Fig. 10. A 16-bit square root circuit using X and Y modules.

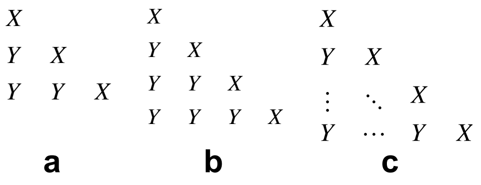


Fig. 11. Floor plan of a square root circuit using X and Y modules, (a) 12-bit, (b) 16-bit, and (c) any 4n-bit circuit.

It was shown in the previous section that for a 4n-bit square root circuit there are n number of X and $n(n-1)/2$ number of Y modules. Hence, the total number of transistors for a 4n-bit square root circuit is

$$T_{\text{total}} = 270n + 152n(n-1)/2 \tag{18}$$

This means that a 16-bit *Modular-NR* circuit uses 1992 transistors. A 16-bit *Classical-NR* circuit has 8 rows with a total of 72 CASs and 7 NOT gates which requires $72 \times 38+7 \times 2 = 2750$ transistors. Therefore *Modular-NR* circuit causes a 27.5% reduction in the number of utilized transistors. This reduction in the required transistors become more pronounced as the circuit becomes larger to accommodate larger radicands.

The total delay of the circuit can be computed based on the delay of a CAS cell. Let us assume that the delay of a CAS is Δ. To compute each digit of the square root one addition or subtraction operation is required and the carry output of the last CAS cell has to be obtained. Since this carry output determines the function of the next row, it is not possible to start the next stage before finishing the operation of the current row. Therefore, in computing the total delay, the delay of each row has to be considered. Accumulation of delays of every row gives the total delay. In the ith row an $(i+2)$-bit add/subtract operation is performed, which has a delay

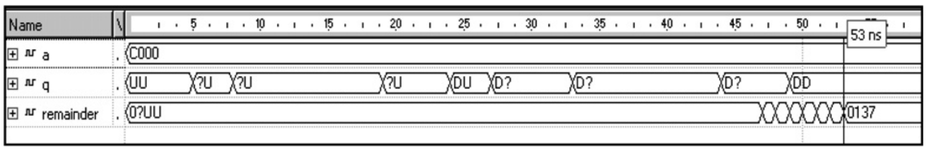


Fig. 12. Timing and functionality test for a 16-bit modular expandable circuit (*Modular-NR*).

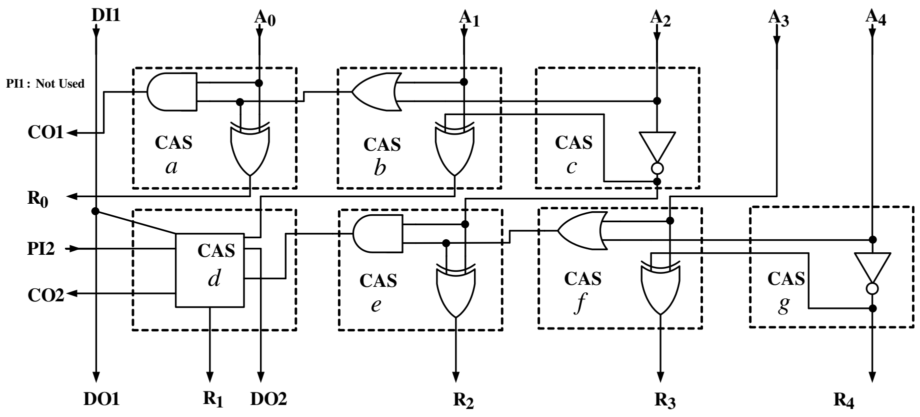


Fig. 13. Logic diagram of a simplified X module.

of $(i+2)\Delta$. Hence, a 4*n*-bit *Modular-NR* design, which generates *n*-bit square roots, has the following delay:

$$D = 3\Delta + 4\Delta + \ldots + (n+2)\Delta = \left[\sum_{i=1}^{n}(2+i)\right]\Delta = \frac{n}{2}(n+5)\Delta \tag{19}$$

As mentioned before, at the top of the modular structure, the first X module in a *Modular-NR* circuit has 3 CASs in its first row while only 2 CASs are required. Hence, 3Δ in Eq. (19) is justified. The rest of terms in Eq. (19) follow Fig. 5, where at every row there is one more CAS as compared to its previous row. For a 32-bit *Modular-NR* square root circuit the total delay is 152Δ which is 44% better than the 272Δ of the *Classical-NR* design.

6. Improving the X modules

The X module shown in Fig. 8 consists of seven CAS cells. Due to its unique structure and special connections between its internal cells, it is possible to come up with a simpler structure for the X module. The improved structure will have fewer transistors and consequently, smaller delay. Inputs (X, D, P, C_{in}) and outputs (R, C_{out}) of a CAS are labeled in Fig. 2. The following Boolean equations define the relations between the inputs and outputs of a CAS:

$$\begin{aligned} C_{\text{out}} &= (X + C_{\text{in}})(D \oplus P) + X \cdot C_{\text{in}} \\ R &= X \oplus (D \oplus P) \oplus C_{\text{in}} \end{aligned} \tag{20}$$

Let us examine CAS$_c$ and CAS$_g$ of Fig. 8. In both mentioned CAS blocks, the *P* outputs are connected to C_{in} inputs and the *D* inputs are always 1. Therefore, by setting $D = 1$ and $C_{in} = P$ in Eq. (20), the following is achieved:

$$\begin{aligned} C_{\text{out}} &= (X + P)(1 \oplus P) + X \cdot P = X \\ R &= X \oplus (1 \oplus P) \oplus P = \overline{X} \end{aligned} \tag{21}$$

Thus, CAS$_c$ and CAS$_g$ can be replaced by two NOT gates. In CAS$_b$ and CAS$_f$ of Fig. 8, the *D* input is connected to *P* through a NOT gate. By setting $D = \overline{P}$ in Eq. (20) the Boolean expression relating inputs and outputs of these cells will be of the following form:

$$\begin{aligned} C_{\text{out}} &= (X + C_{\text{in}})(\overline{P} \oplus P) + X \cdot C_{\text{in}} = X + C_{\text{in}} \\ R &= X \oplus (\overline{P} \oplus P) \oplus C_{\text{in}} = \overline{(X \oplus C_{\text{in}})} \end{aligned} \tag{22}$$

The above relations imply that each of CAS$_b$ and CAS$_f$ requires an OR and an XNOR gate. Now considering CAS$_a$ and CAS$_e$, it is observed that the *P* input of each cell is connected to the *D* input of that cell. Hence, *D* in Eq. (20) is set equal to *P* resulting in the following Boolean expressions:

$$\begin{aligned} C_{\text{out}} &= (R + C_{\text{in}})(P \oplus P) + R \cdot C_{\text{in}} = R.C_{\text{in}} \\ R &= R \oplus (P \oplus P) \oplus C_{\text{in}} = R \oplus C_{\text{in}} \end{aligned} \tag{23}$$

An AND gate and an XOR are required to construct either CAS$_a$ or CAS$_e$. Therefore, using the above modifications a simplified X module is designed, as shown in Fig. 13.

By replacing the X modules, the overall delay and transistor count of a 4*n*-bit square root circuits are improved. An AND or an OR gate can be constructed by 4 transistors in the CMOS technology. Keeping in mind that the improved X module has two NOT, four XOR, two AND, two OR gates, and one CAS cell, the number of its transistors can be computed as

$$T'_X = 2 \times 2 + 4 \times 10 + 4 \times 4 + 28 = 88 \tag{24}$$

It is obvious that the number of transistors for the modified X module compared to the previous design, which had seven CASs and 270 transistors, is 70% less. Using this improved X module for a 4*n*-bit square root circuit, the total number of transistors will be

$$T_{\text{total}} = 88n + 76n(n-1) \tag{25}$$

Modular circuits that are built by the X module of Fig. 13 will be referred as the *Simple-X-Module* designs. The proposed simplification has more dire effects on smaller circuits, where the X modules are the larger portion of the whole circuit. For a 32-bit circuit, built by simplified X modules, the number of transistors will be 4960. This number is derived from Eq. (25) by plugging $n = 8$. If the same circuit is built by X modules of Fig. 8, the required transistor count will be 6416, which is 29% more transistors than the *Simple-X-Module* design. Simulation results for a 16-bit *Simple-X-Module* design are shown in Fig. 14.

Waveforms of Fig. 14 show an example of the function of the circuit built from *Simple-X-Modules*. The radicand C000$_H$, shown in Fig. 14, is the same value that is used to test other designs. Due to simpler internal structure of its X modules, the simulated circuit produces its remainder, for this specific example, in 40 ns. This circuit is much faster than other circuits studied in this paper.

7. Simulation results

In the previous sections, using the ModelSim SE 5.7f VHDL simulator, using large number of sample inputs, the correct behavior of the circuits were verified. In this type of simulation, a number of input data are defined and after running the simulator, output signals are generated. Arbitrary data patterns as test samples were applied to the inputs of the circuits and correct square roots and remainders were observed at the outputs. The delays of the circuits were studied in the simulation by assigning a 1 ns delay to each gate.

In this section we look at the results obtained by implementing all of the designs using the Xilinx Foundation ISE version 6.1i

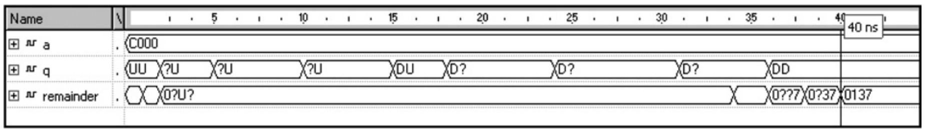


Fig. 14. Timing and functionality test for the modular 16-bit *Simple-X-Module* design.

development tools. The implementations were aimed for XCV100E-8 FPGA from the Xilinx Virtex E series. This was done to create a platform for area consumption comparison between the designs.

After the correct operation of the circuit was verified, the synthesis phase can be applied. The gates and signal paths' delays are considered in this phase and the design becomes ready for FPGA implementation. This is actually a timing simulation of the design. Since the path that a signal may go through from an input port all the way to an output depends on the initial value of that signal, the circuit delay may vary for different sample data. Hence, the worst-case delay should be considered for each circuit. This means that the worst-case delay should elapse before one can be certain that the correct output is generated. Prior to this time, the output signals may have many transitional changes due to inherent ripple effects that exist in-between blocks.

The main assets of an FPGA are "*look up tables*" (LUTs) and "*configurable logic blocks*" (CLBs). The number of LUTs and CLBs that are in an FPGA depend on the type of that specific IC. The FPGA that is referred to in this paper has 1200 CLBs and 2400 LUTs. In Table 2 the results obtained from implementation of different designs on the mentioned FPGA are listed. The number of employed CLBs and LUTs is a good indication of the size of the implemented circuit. We can then compare two different designs by comparing the used assets of each design. Table 2 shows that the 64-bit *Classical-NR* design occupies 85% of the CLBs and 74% of the LUTs of the FPGA. The other three designs are close to each other in terms of the FPGA assets that they require for implementation.

The circuit delays for two of the designs are presented as two curves in Fig. 15. We can see that the X module was significantly improved in terms of the employed hardware. This brought by a significant reduction in the delay of this module. It was shown that to accommodate for a 4-bit radicand only one X module is enough. This means that by using a *Simple-X-Module* the whole 4-bit circuit is affected. For an 8-bit radicand two X modules and one Y module are needed. By using *Simple-X-module* a big portion of the circuit is then modified. As the size of the radicand increases bigger portions of the circuits will be comprised of the Y modules. Therefore, the improved part (X modules) would cover smaller portions of the circuits. Hence, an 8-bit *Simple-X-Module* has a speedup of 1.8 as compared to a corresponding *Modular-NR* circuit. We see that the speedup is reduced to 1.07 when a 64-bit *Simple-X-Module* is compared with a corresponding *Modular-NR* circuit. On the other hand, when we compare the *Classical-NR* design with the *Simple-X-Module*, as shown in Fig. 15, the reduction in delay has two sources. In Fig. 15 we see that for bigger radicands the reduction in delay is greater and that is mainly due to the eliminated CAS blocks. A second parameter, with lesser effect, that has helped the higher speed of the *Simple-X-Module* design is the employment of the improved X modules.

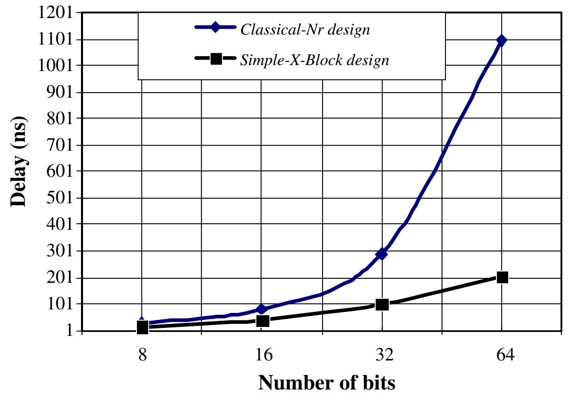


Fig. 15. Delay comparison between *Classical-NR* and *Simple-X-Module* designs.

Table 2
Comparison of the four designs in terms of area and delay

Design	# of input bits	Number of CLBs	Number of LUTs	Delay (ns)
Classical-NR	8	16	29	28
Classical-NR	16	62	107	82
Classical-NR	32	252	439	290
Classical-NR	64	1023 (85%)	1779 (74%)	1096
Reduced-Area-NR	8	11	19	25
Reduced-Area-NR	16	41	72	51
Reduced-Area-NR	32	158	274	113
Reduced-Area-NR	64	616 (51%)	1072 (44%)	227
Modular-NR	8	12	20	27
Modular-NR	16	42	73	53
Modular-NR	32	156	272	115
Modular-NR	64	617 (51%)	1073 (44%)	229
Simple-X-Module	8	12	21	15
Simple-X-Module	16	42	73	40
Simple-X-Module	32	162	282	100
Simple-X-Module	64	622 (51%)	1081 (45%)	214

8. Conclusion

Fixed point square root operation is an important function for graphical routines and nowadays many of the processors implement it. In this paper we examined the non-restoring square root routine in order to improve the classical design in terms of area and delay. The classical array structure was first improved by reducing the number of CAS blocks that are used in these circuits. Furthermore, a modular design was proposed in this paper where by using two distinct modules (X and Y) a circuit could be easily built. Expanding a circuit with the proposed modular design was shown to be feasible. Since our implementation platform was FPGA we did not have leverage in optimizing the internal connections and routings between the CLBs. It is reported that ASIC implementation of digital circuits could result in speedups of about 4 as compared to FPGA implementations [15]. In case that X and Y modules are designed as standard VLSI cells, the delay of the circuits will certainly be less than those mentioned in Table 2. Irrespective of the implementation platform, the relative speedups and area advantages of the proposed designs, as compared to the *Classical-NR* circuits, remain intact.

References

[1] V.K. Jain, L. Lin, Nonlinear DSP coprocessor cells-one and two cycle chips, Proceedings of the IEEE International Symposium on Circuits and Systems 2 (1998) 264–267.

[2] P. Kornerup, Digit selection for SRT division and square root, IEEE Transactions on Computers 54 (3) (2005).
[3] Y. Li, W. Chu, Parallel-array implementations of a non-restoring square root algorithm, in: Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1997, pp. 690–695.
[4] P. Kornerup, Digit selection for SRT division and square root, Transactions on Computers 54 (3) (2005) 294–303.
[5] K. Wires, M. Schulte, Reciprocal and reciprocal square root units with operand modification and multiplication, Journal of VLSI Signal Processing Systems 42 (3) (2006).
[6] A. Thakkar, A. Ejnioui, Computational sciences: pipelining of double precision floating point division and square root operations, in: Proceedings of the 44th Annual Southeast Regional Conference ACM-SE 44, March 2006.
[7] G. Cappuccino, G. Cocorullo, P. Corsonello, S. Perri, High speed self-timed pipelined datapath for square rooting, IEE Proceedings: Circuits, Devices and Systems 146 (1) (1999) 16–22.
[8] M. Ito, N. Takagi, S. Yajima, Efficient initial approximation and fast converging methods for division and square root, in: Proceedings of the 12th Symposium on Computer Arithmetic, 1995, pp. 2–9.
[9] L. Wang, M. Schulte, A decimal floating-point divider using Newton–Raphson iteration, The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology 49 (1) (2007) 3–18.
[10] W. Chu, Y. Li, Cost/performance tradeoff of n-select square root implementations, in: Fifth Australasian Computer Architecture Conference, 1999, pp. 9–16.
[11] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press, 2000.
[12] R. Zimmermann, Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication, in: Proceedings of the 14th IEEE Symposium on Computer Arithmetic, 1999, pp. 158–167.
[13] Y. Li, W. Chu, A new non-restoring square root algorithm and its VLSI implementations, in: Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1996, pp. 538–544.
[14] M. Michael Via, VLSI Design, CRC Press, New York, 2000.
[15] I. Kuon, J. Rose, Measuring the gap between FPGAs and ASICs, in: Proceedings of the 14th International Symposium on Field Programmable Gate Arrays, 2006, pp. 21–30.

Shadrokh Samavi received the B.S. degrees in industrial technology and electrical engineering from the California State University, Fresno, in 1980 and 1982, respectively, the M.S. degree from the University of Memphis, Memphis, TN, in 1985, and the Ph.D. degree in electrical engineering from Mississippi State University, Mississippi State, in 1989. In 1993, he joined the Electrical and Computer Engineering Department, Isfahan University of Technology, Isfahan, Iran, where he was a Professor. During the 2002/2003 academic year, he was a Visiting Professor with the Electrical and Computer Engineering Department, McMaster University, Hamilton, ON, Canada. His current research interests are implementation and optimization of image-processing algorithms and area-performance tradeoffs in computational circuits. Dr. Samavi is a Registered Professional Engineer (P.E.), USA, and is a member of Eta Kappa Nu, Tau Beta Pi, IEEE and the National Association of Industrial Technologists (NAIT).

Alireza Rezai Sadrabadi received his B.S. degree in electrical engineering from the Tehran Polytechnic in 1997, and M.S. degree from the Isfahan University of Technology, Isfahan, Iran, in 1999. Mr. Sadrabadi's research focuses are on embedded hardware–software design of arithmetic algorithms for signal processing systems-on-a-chip. His current research efforts are concentrated on CAD tools for design of high-performance signal processing architectures.

Ali Fanian received the B.S. and M.S. degrees in computer engineering (Hardware and Computer Systems Architecture) in 1999 and 2001, respectively from Isfahan University of Technology (IUT), Isfahan, Iran. He is currently a Ph.D. candidate at IUT. Different aspects of computer architecture and network security are Mr. Fanian's research interests. He currently works on ad hoc networks, wireless network security and hardware design.