A 32-Bit Signed/Unsigned Fixed Point Non-Restoring Square-Root Operation Using VHDL

Ms. M.U. Buradkar, Prof. P. P. Zode
Department of Electronics Engg.
Yeshwantrao Chavan College of Engineering, Nagpur, India

Abstract
After analyzing the advantages and disadvantages of all the general algorithms adopted in designing square root on FPGA chips with pipeline technology, a proposed algorithm based on digit by digit calculation method is discussed. The algorithm is realized on the ModelSim SE 6.3f development platform with VHDL language and the simulation results show that it is characterized by occupying less resource as well as processing is in a faster speed. Therefore it is an effective algorithm for implementing square root on commonly-used FPGA chips with pipeline technology. Square root operation deserves attention because of its frequent use in a number of applications. Square root operation basically is considered difficult to implement in hardware and it's a basic operation in computer graphics and scientific calculation applications. A pipelined architecture to implement 32-bit fixed-point signed/unsigned square root operation on an FPGA using a non-restoring pipelined algorithm that does not require floating-point hardware is discussed. The main principle of proposed method is two-bit shifting and subtract-multiplexing operations, in order to achieve a simpler implementation and faster calculation. The proposed algorithm is used to implement FPGA based signed and unsigned 32-bit square root successfully and it is efficient in hardware resource.

Keywords- Pipelining, fixed-point arithmetic, signed and unsigned square root, non-restoring algorithm, FPGA, digit recurrence calculation.

I. Introduction

Square root is one of the most useful and vital operation in computer graphics and scientific calculation applications such as digital signal processing and control and even multimedia application [1-6]. In addition to the basic arithmetic operations (+, -, * and ÷), the square root is also an essential operation frequently employed by signal and image processing applications. [1-3]. It is a classical problem in computational number theory and often encountered , which is hard task to get an exact result [7-8]
Fixed-point hardware with appropriate software support is often used to achieve low -cost implementation of algorithms. This alternative approach usually provides accuracy very close to that of floating-point hardware. Furthermore, the fixed-point accuracy heavily depends upon the operand values and selection of format for fixed-point operations. Small or extremely large values of operands may produce minor errors for complex operations due to the rounding and the truncation of least significant digits [9-11]. However, real-time portable applications emphasize time and power efficient solutions since minor errors in the answer do not often affect the quality of the results.

Fixed-point addition and subtraction operations are relatively easy to implement on an FPGA, while rounding and truncation errors are negligible. The multiplication, division and square-root operations require complex procedures towards accuracy. They often rely on special types of algorithm for embedded system realization. Out of these three operations, the square root is the most complex one [8] because it usually involves convergence or approximation algorithms, and thus becomes computationally intensive [4, 13].

Numerous techniques have been reported in the literature to implement the square root operation on FPGAs [8]. FPGA implementations provide tradeoffs between general purpose and ASIC solutions. They offer flexibility near that of a general purpose solution and performance closer to an ASIC since they support low-level hardware design. They provide dynamic reconfiguration capabilities for run-time architecture changes. They are cost effective for customized applications and are commonly used for prototyping before ASIC realization. However, end products containing FPGAs are now commonly employed to create time and power efficient designs for real-time portable applications.

A lot of square root algorithms has have been studied, developed and implemented, such as Rough estimation, Babylonian method, exponential identity, Taylor-series expansion algorithm, Newton-Raphson method, Sweeney Robertson Tocher redundant method (SRT redundant method), SRT non redundant method and sequential algorithm (digit-by-digit method) [1-9]. However, the early processors carry out the square root operation of the algorithms above by software means, which have long delays for its completion [6].

With the rapid advancement of technology which is possible to integrate large circuits on a single chip and also increase in demand for faster computational execution time, hardware implementation of square root operation became more attractive

[6]. Unfortunately because of the complexity of the square root algorithms, the square root calculation is not easy to implement on field programmable array (FPGA) technology [1, 3, 5, 10].

There are some algorithms of square root which are implemented on FPGA. They are generally grouped into two distinct categories. In first category is called estimation methods, such as Rough estimation and Newton-Raphson method (and also its derivations: CORDIC, DeLugish's and
Chen's), and in second category is called digit-by-digit method. Finally, it is necessary to classify further digit-by-digit method into two distinct classes: restoring and non-restoring algorithm. The restoring algorithm has a big limitation at restoring step in the regular flow. Primarily for this reason, although initially having led the way for all the other methods, it has declined in importance and nowadays it is no longer used [11]. Compared to the restoring algorithm, the non-restoring algorithm does not restore the remainder, which can be implemented with fewest hardware resource and the result is hardware simple implementation. It is most suitable for FPGA implementation and allows for IEEE standard rounding to be readily implemented [1-3, 6].

There are many strategies or architectures conducted to implement the non-restoring digit-by-digit square root algorithm in FPGA hardware. Yamin and Wanming [1-2, 9] have introduced a non-restoring algorithm with fully pipelined and iterative version that requires neither multipliers nor multiplexors. They introduced the carry save adder (CSA) and carry propagate adder (CPA) as basic building blocks. The other architecture proposed is fully combinational architecture. However, the FPGA is very suitable for adoption of the fully pipelined architecture because of the characteristics of its structure. Hence, the very little or even needless extra cost, if the pipeline technology is implemented in FPGA.

This paper proposes a new strategy to implement modified non restoring algorithm based on FPGA which adopt fully pipelined architecture. In the proposed strategy is introduced a new basic building block called subtract-multiplex (SM), and also it needs fewer pipeline stages.

II. SIGNED NUMBER REPRESENTATIONS

Signed-digit representation of numbers indicates that digits can be prefixed with a − (minus) sign to indicate that they are negative.

Signed-digit representation can be used in low-level software and hardware to accomplish fast addition of integers because it can eliminate carries. In computing, signed number representations are required to encode negative numbers in binary number systems. In mathematics, negative numbers in any base are represented by prefixing them with a − sign. However, in computer hardware, numbers are represented in bit vectors only without extra symbols. The four best-known methods of extending the binary numeral system to represent signed numbers are: sign-and-magnitude, ones' complement, two's complement.[12]

A. Sign-and-magnitude method

In the first approach, the problem of representing a number's sign can be to allocate one **sign bit** to represent the sign: set that bit (often the most significant bit) to 0 for a positive number, and set to 1 for a negative number. The remaining bits in the number indicate the magnitude (or absolute value). Hence in a byte with only 7 bits (apart from the sign bit), the magnitude can range from 0000000 (0) to 1111111 (127). Thus you can represent numbers from -127_{10} to $+127_{10}$ once you add the sign bit (the eighth bit). A consequence of this representation is that there are two ways to represent zero, 00000000 (0) and 10000000 (−0). Decimal −43 encoded in an eight-bit byte this way is 10101011. [12]

This approach is directly comparable to the common way of showing a sign (placing a "+" or "−" next to the number's magnitude). Some early binary computers (e.g. IBM 7090) used this representation, perhaps because of its natural relation to common usage. Sign-and-magnitude is the most common way of representing the significand in floating point values.

B. Ones' complement

The **ones' complement** of a binary number is defined as the value obtained by inverting all the bits in the binary representation of the number (swapping 0's for 1's and vice-versa). The ones' complement of the number then behaves like the negative of the original number in most arithmetic operations. However, unlike two's complement, these numbers could not have widespread use because of issues like negative zero, end-around borrow, etc.

A **ones' complement system** or **ones' complement arithmetic** is a system in which negative numbers are represented by the arithmetic negative of the value. In such a system, a number is negated (converted from positive to negative or vice versa) by computing its ones' complement. An N-bit ones' complement numeral system can only represent integers in the range $-(2^{N-1}-1)$ to $2^{N-1}-1$. A conventional eight-bit byte is -127_{10} to $+127_{10}$ with zero being either 00000000 (+0) or 11111111 (−0). 8-bit one's complement is as given in table 1. [12-14]

C. Two's complement

The **two's complement** of a binary number is defined as the value obtained by subtracting the number from a large power of two (specifically, from 2^N for an N-bit two's complement). The two's complement of the number then behaves like the negative of the original number in most arithmetic, and it can coexist with positive numbers in a natural way.[15]

A two's-complement system, or two's-complement arithmetic, is a system in which negative numbers are represented by the two's complement of the absolute value; this system is the most common method of representing signed integers on computers. In such a system, a number is negated (converted from positive to negative or vice versa) by computing its ones' complement (i.e. its bitwise negation) and adding one. An N-bit two's-complement numeral system

8 bit ones' complement

Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	+0	0
00000001	1	1
...	...	...
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
...	...	...
11111101	−2	253
11111110	−1	254
11111111	−0	255

Table.1. 8-bit One's Complement

can represent every integer in the range -2^{N-1} to $2^{N-1}-1$. 8-bit two's complement is as given in table 2.

D. Making the two's complement of a number

In two's complement, positive numbers are represented as binary numbers whose most significant bit $a_{m-1} = 0$. Negative numbers are represented with the most-significant bit $a_{m-1} = 1$, making use of the left-most bit's negative weight. All radix complement number systems use a fixed-width encoding. Every number x encoded in such a system has a fixed width so the most-significant digit can be examined.

E. How to create algorithmically for x the two's complement binary value?

(The algorithm is mainly due to $\sum_{i=-n}^{m-2} 2^i = 2^{m-1} - 2^{-n}$)

1. Express the binary value $b_{m-1}, \ldots, b_{-n}$ for $|x|$ (important: express all coefficients b_i)

2. If $x < 0$ (the case $x > 0$ is already finished by 1. since $x = |x|$ and $b_i = a_i$ for all i, especially $b_{m-1} = a_{m-1} = 0$)

2a. Complement each b_i (i.e. replace b_i by $1 - b_i$ for $i = m-1, \ldots, -n$)

2b. Add the coefficient 1 (this 1 represents 2^{-n}) to the latest binary representation. The addition leads to the final representation for x which is $a_{m-1}, a_{m-2}, \ldots, a_1, a_0, a_{-1}, a_{-2}, \ldots a_{-n}$

Bits	Unsigned value	2's complement value
00000000	0	0
00000001	1	1
00000010	2	2
01111110	126	126
01111111	127	127
10000000	128	−128
10000001	129	−127
10000010	130	−126
11111110	254	−2
11111111	255	−1

Table. 2. 8-bit two's-complements

III. DIGIT-BY-DIGIT CALCULATION METHOD

In digit-by-digit calculation method, each digit of the square root is found in a sequence where it only one digit of the square root is generated at each iteration [2, 6]. It has several advantages, such as: every digit of the root found is
known to be correct and it will not has to be changed later; if the square root has to expand, it will terminate after the last digit is found; and the algorithm works for any number base.

In general, this method can be divided in two classes, i.e. restoring and non-restoring digit-by-digit algorithm [6]. In restoring algorithm, the procedure is composed by taking the square root obtained so far, appending 01 to it and subtracting it, properly shifted, from the current remainder. The 0 in 01 corresponds to multiplying by 2; the 1 is a new guess bit. The new root bit developed is truly 1, if the resulting remainder is positive, and vice versa is 0, which the remainder must be restored by adding the quantity just subtracted. It is different, in non-restoring algorithm does not restore the subtraction if the result was negative. Instead, it appends a 11 to the root developed so far and on the next iteration it performs an addition. If the addition causes an overflow, then on the next iteration you go back to the subtraction mode. Figure 1 (a) and (b) gives an example to take the binary square root of 01011101 (equivalent with 93 decimal) for restoring and non-restoring algorithm respectively.

```
   1  0  0  1
 ____________
/01 01 11 01
 -1
 -----
 00 01  <--- positive: first bit is a 1
 -1 01
 -----
 11 00  <--- negative: 2nd bit is a 0
 +1 01  <--- restore the wrong guess
 --------
 00 01 11
   -10 01
 --------
 11 11 10  <--- negative: 3rd bit is a zero
   +10 01  <--- restore the wrong guess
 ----------
    01 11 01
    -1 00 01
 ----------
     0 11 00  <--- positive: 4th bit is a 1
```

```
   1  0  0  1
 ___________
/01 01 11 01
-1
___
00 01  <---  positive: first bit is a 1
-1 01
____
11 00  <---  negative: 2nd bit is a 0
+1 01  <---  restore the wrong guess
_____
00 01 11
  -10 01
________
11 11 10  <---  negative: 3rd bit is a zero
  +10 01  <---  restore the wrong guess
__________
   01 11 01
   -1 00 01
   ________
    0 11 00  <---  positive: 4th bit is a 1
```

(a)

```
 1  0  0  1
 ___________
/01 01 11 01
 -1
 ___
 00 01  <--- positive: first bit is a 1
 -1 01  <--- Developed root is "1"; appended 01; subtract
 _____
 11 00 11  <--- negative: 2nd bit is a 0
   +10 11  <--- Developed root is "10"; append 11 and add
 ________
 11 11 10 01  <--- negative: 3rd bit is a 0
     1 00 11  <--- Developed root is "100"; append 11 and add
 ___________
1 00 00 11 00  <--- Overflow: 4th bit is a 1
```

(b)

Figure. 1. The example of digit-by-digit calculation to solve square root:

(a) restoring algorithm; (b) non-restoring algorithm

IV. PROPOSED NON-RESTORING SQUARE-ROOT ALGORITHM FOR SIGNED AND UNSIGNED NUMBER

A little different than conventional non-restoring digit-by-digit algorithm in Figure 1 (b), a modification as shown on Figure 2 can be conducted to give simpler implementation and faster calculation. In this modification, it only uses subtract operation and append 01, while add operation and append 11 is not used.

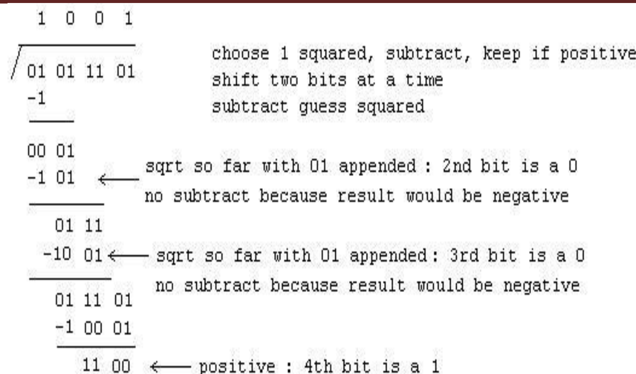


Figure 2. The example of using modified non-restoring digit-by-digit calculation algorithm to solve square root

A simple hardware implementation of the proposed non-restoring digit-by-digit algorithm for unsigned 8-bit square root by an array structure is shown in Figure 4. The radicand is P (P7,P6,P5,P4,P3,P2,P1,P0), U (U3,U2,U1,U0) as quotient and R (R5,R4,R3,R2,R1,R0) as remainder. It can be shown that the implementation needs 4 stage pipelines. The basic building blocks of the array are blocks called as controlled *subtract-multiplex* (SM). Figure 5 present the details of a SM. Input of the building block is x,y,b and u, and as an output is bo(borrow) and d result). If u=0, then d<=x-y-b else d<=x.

Step I. Start

Step II. Initialization of radicand (the n-bit number will be squared root), quotient (the result of squared root), and remainder. To calculate square root of a 2n bit number, it needs n stage pipelines to implement the proposed algorithm.

Step III. Divide the radicand into groups of two digits starting from MSB

Step IV. Beginning on the left (most significant bit), select the first group of one or two digit (If n is odd then the first groups is one digit, and vice versa)

Step V. Choose 1 squared, and then subtract.

First developed root is “1” if the result of subtract is positive, and vice versa is “0”

Step VI. Shift two bits, subtract guess squared with append 01. Nth-bit squared is “1”, if the result of subtract is positive, and .Because of subtract operation is done else Nth-bit squared is “0”, and not subtract.

Step VII. Go to step 5 until end group of two digits.

.

Step VIII. Follow step 0 through 6 for signed n-bit Number. If sign is negative, its negative number with MSB=1 else positive with MSB=0. Convert the negative number to equivalent 2’s Complement representation.

Step IX. The square root for signed number is computed as ***ni*** , where ***n= number of bits*** and ***i= imaginary output variable***

If MSB = ‘0’ then
imag = ‘0’
else
imag= ‘1’

Step X. End

Fig 3. Modified Algorithm Principle for signed/unsigned number

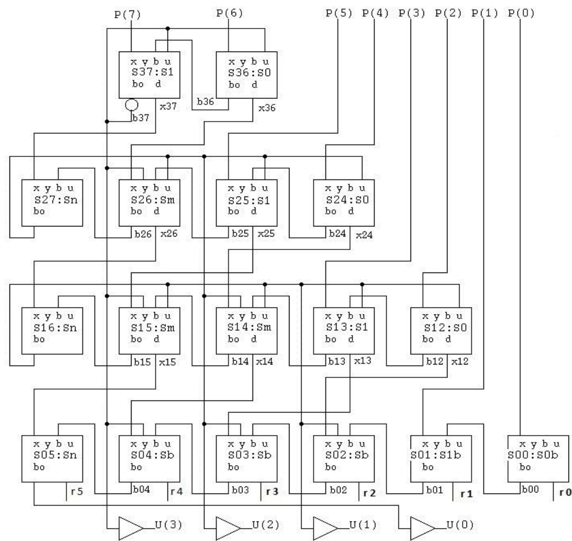


Fig 4. Simple hardware implementation of the non -restoring digit-by-digit algorithm for unsigned 8-bit square root

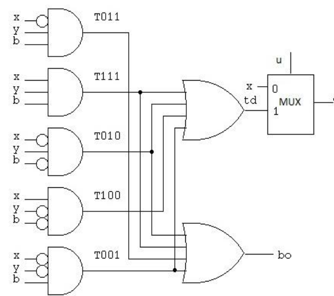


Fig 5. Internal structure of a SM block

V. SYNTHESIS REPORT

Table 3. The comparison of 32-bit squareroot circuits

No	Design Analysis	32-bit square root	
		Unsigned	Signed
1	Design statistics	BELLS:301 IOBUFFER:48	BELLS:419 IOBUFFER:49
2	Delay	181.674 ns	187.355 ns
3	Logic Level	122	157
4	Memory Used	255948kb	256204kb
5	CPU time	10.19 secs	11.19 ecs

REFERENCES

[1]L. Yamin and C. Wanming, "Implementation of Single Precision on Floating Point Square Root on FPGAs," in FCCM'97, IEEE Symposium for Custom Computing Machines, Napa, California, USA, 1997, pp. 226-232.

[2] L. Yamin and C. Wanming, "Parallel-array implementations of a non-restoring square root algorithm," in Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings. 1997 IEEE International Conference on, 1997, pp. 690-695.

[3] K. Piromsopa, et al., "An FPGA Implementation of a fixed-point square root operation," presented at the Int. Symp. on Communications and Information Technology (ISCIT 2001), ChiangMai, Thailand, 2001.

[4] D. R. Llamocca-Obregon, "A Core Design to Obtain Square Root Based on a Non-Restoring Algorithm," presented at the IBERCHIPS Workshop, Salvador Bahia, Brazil, 2005.

[5] XiaojunWang, "Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms," 2003

[6] S. Samavi, et al., "Modular array structure for non-restoring square root circuit," Journal of Systems Architecture, vol. 54, pp. 957-966,2008.

[7] H. Dong-Guk, et al., "Improved Computation of Square Roots in Specific Finite Fields," Computers, IEEE Transactions on, vol. 58,pp. 188-196, 2009.

[8] S. Lachowicz and H. J. Pfleiderer, "Fast Evaluation of the Square Root and Other Nonlinear Functions in FPGA," in Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on, 2008, pp. 474-477.

[9] W. Chu; and Y. Li;, "Cost/Performance Tradeoff of n-Select Square Root Implementations," in 5th Australasian Computer Architecture Conference (ACAC 2000), Canberra, ACT 2000, pp. 9-16.

[10] J. Xiaoliang, "Implementation of Square Root Arithmetic Based on FPGA," Modern Electronics Technique, vol. 30, 2007.

[11] P. Montuschi and M. Mezzalama, "Survey of square rooting a lgorithms," in Computers and Digital Techniques, IEE Proceedings Italy, 1990, pp. 31 - 40.

[12] Nicholaas C.DeTroye "Digital Root Extraction Circuit" in patent no. 4,748,581, May 31, 1988.

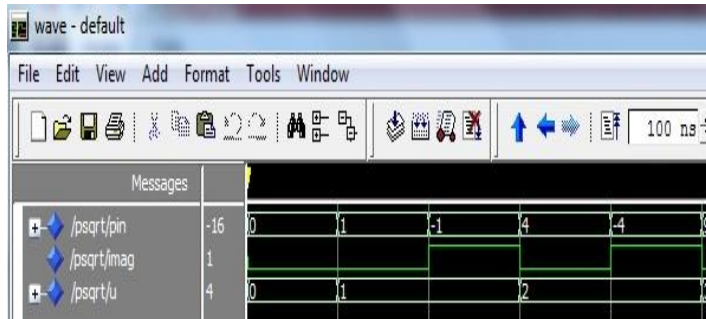
[13] Sau-Gee Chen, Hsinchu, Chieh-Chih Li, "Apparatus for finding the Square root of a number" in patent no. 5,430,669, Jul. 4, 1995.

[14] Yoshitsugu Kitora. "Square Root Extractor" in patent no.5, 331,586,Jul. 19, 1994

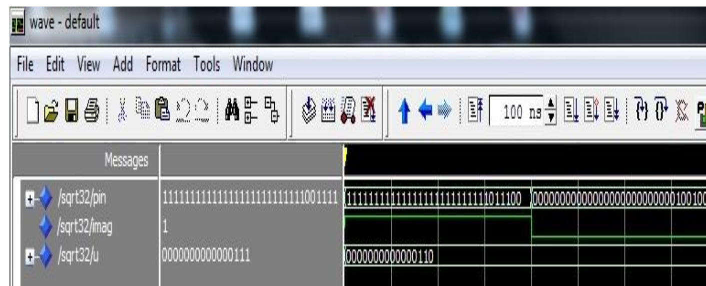
[15] Keshav K. Parhi, " A Systematic approach for design of digit serial signal processing architectures", IEEE transactions on circuits and systems , vol. 38, no.4, april 1991.

B.E.Saglam, G.Cosgul, Comments on " A Systematic approach for design of digit serial signal processing architectures", IEEE transactions on circuits and systems-II , vol. 47, no.4, april 2000.

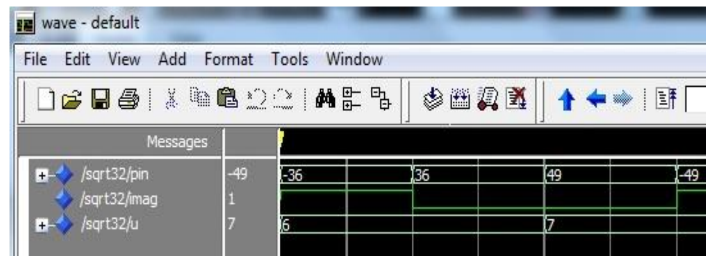
(a)



(b)

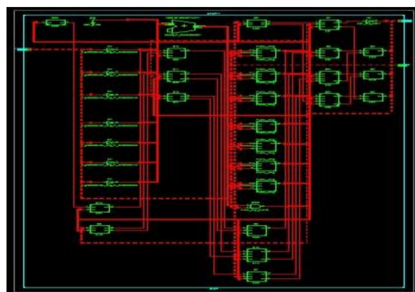


(c)

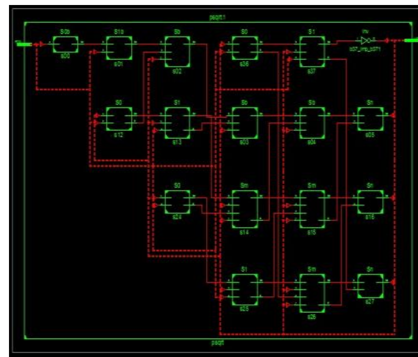


(d)

Fig 6. Simulation result of square root using simple hardware implementation method of the non-restoring digit-by-digit algorithm:
(a) 8-bit in binary display , (b) 8-bit in decimal display
(c) 32-bit in binary display (d) 32-bit in decimal display

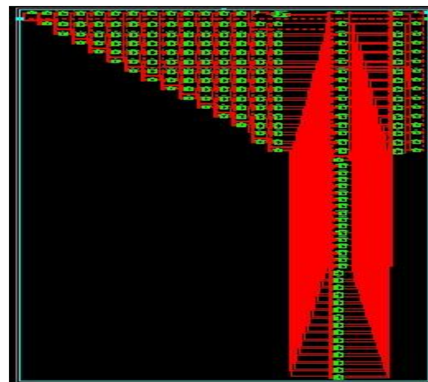


(a)

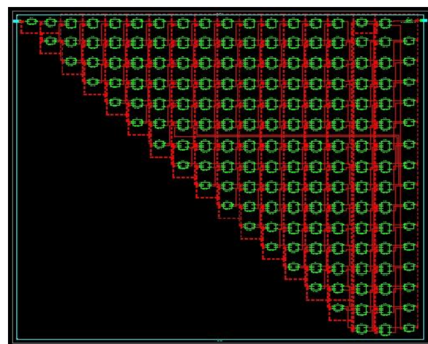


(b)

Fig 7. RTL Schematic (a) 8-bit signed square root ,
(b) 8-bit unsigned square root



(a)



(b)

Fig 8. RTL Schematic (a) 32-bit signed square root,
(b) 32-bit unsigned square root.