IJCRT.ORG ISSN : 2320-2882
INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)
An International Open Access, Peer-reviewed, Refereed Journal

RTL REALIZATION OF SQUARE ROOT METHODS FOR ARITHMETIC LOGIC UNIT

Subhash Kumar Sharma
Assistant Professor
Department Of Electronics
MGPGCollege, Gorakhpur, U.P.

Umesh Kumar Gupta
Associate Professor
Department of Mathematics
MGPG College,Gorakhpur ,UP

Shri Prakash Dubey
Assistant Professor
Department of Physics
MGPGCollege, Gorakhpur, U.P.

ABSTRACT: This paper deals with three algorithms implementations for square root computation. Newton-Raphson's, Iterative and Binary Search Algorithm implementations are completely designed, verified and compared. The algorithms, entire systems realization and their functioning are described.

KEYWORDS: Newton-Raphson's Method, Iterative Method, Binary Search Algorithm.

1. INTRODUCTION

Digital systems for square root computation and division are still a challenge for IC designers. There are many algorithms and their implementations. A unit with 24-bit input is considered here. Two optimization criteria were used: chip area and speed. Therefore, the number of iterations for square root computation should be minimal. Attention has been paid to three types of square-rooters.

2. NEWTON-RAPHSON'S METHOD

Following Newton-Raphson's method (also known as Heron's), the square root value of number a is computed through the iterative formula $$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right) \quad (1)$$

Digital system of this formula implementation is shown on Fig. 1.

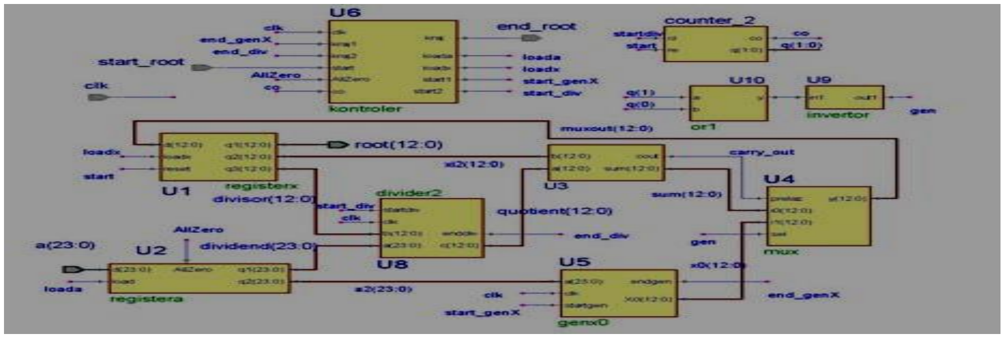

Fig.1. Block diagram of digital system for square-rooting based on Newton-Raphson's method.

It is built of few subsystems: 24-bit register *U2* used for storing the number that square-root should be computed for; 13-bit register *U1* in which temporary solution for square root is stored; subsystem *U8* for the division implementation; subsystem *U5* for initial solution-value computing; adder *U3* containing 13 full adders; *counter_2* that counts started divisions; sequencer *U6* that has the control over subsystems. The system ports are: signal *clk* for clock; *start root* for beginning of computation; *end_root* for the indication of the operation completion, bus *a(23:0)* for getting the input number *a*; bus *root(12:0)* for output data. How does the entire system function? At the beginning, number a, the dividend, is stored in register U2. Unit for initial value computation, *U5*, gets the value *a* from register *U2* and after time delay it produces initial solution *x0*. After completion of generating x0, this value is available on the output of register U1 as divisor. Subsystem for dividing divides values stored in registers *U2* and *U1*, and quotient is added to the value stored in register U1 and shifted right. After these operations, new solution x1 is stored in register *U1*. Only one additional dividing operation is needed to reach the final value of square root, because the initial value *x0* is computed on appropriate way, instead assumed as the arbitrary value. In second dividing operation, x1 is divisor. Quotient is added to x1, shifted right and stored in register *U1*. This final result can be read on bus *root (12:0).*

Further, the attention will be paid on subsystems implementing the parts of square-rooter.

The computation of the initial value *x0* of square root is done with the subsystem shown in Fig.2.

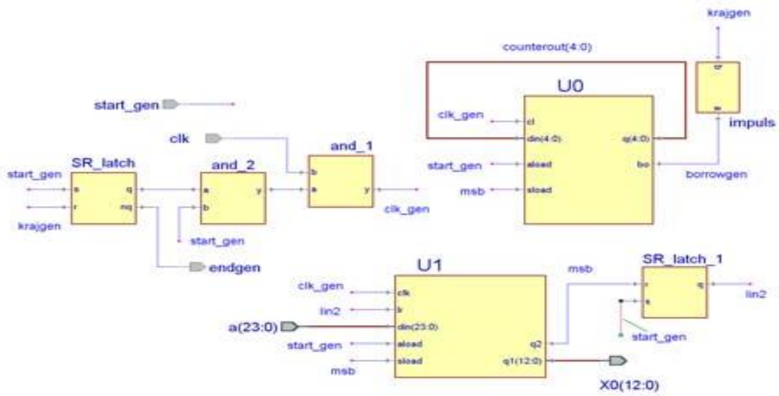

Fig.2. Block diagram of digital subsystem for initial square- root-solution generation.

Its inputs are: *clk* for clock, *start_gen* for operation start, *end_gen* for signaling if unit finished its operation, 24-bit input bus *a(23:0)* and output bus *x0(12:0).* It consists of one shift register U1, counter U0 and few simple logical gates. Input number *a* (for which the square root should be computed) is stored in 24 least significant bits of *25*-bit register *U1*. At the start of computation, the number of significant digits of the input number is unknown and therefore it should be computed first: counter *U0* counts the number of zeros before the most significant 1 in the input number *a*. After that, number *a* is stored again in register *U1*. The number of digits in initial solution is twice less than the number of input digits and it is stored in counter. Register *U1* starts shifting to the right and counter starts decrementing its value. When the counter reaches zero state, shifting is stopped and the value remained in *U1* is initial solution for square root.

The number of necessary clock iterations for initial value generation *Ngen* is: $N_{gen} = 25 - \left[\frac{K-1}{2} \right]$ where *K* is the number of significant bits of input *a*. *Ngen* is in the range from *25*, for computing square root of *1*, to *14*, for computing square root of $(FFFFFF)_{16}$. The division unit implements the manual or Longhand division algorithm. Its structure is shown in Fig.3.

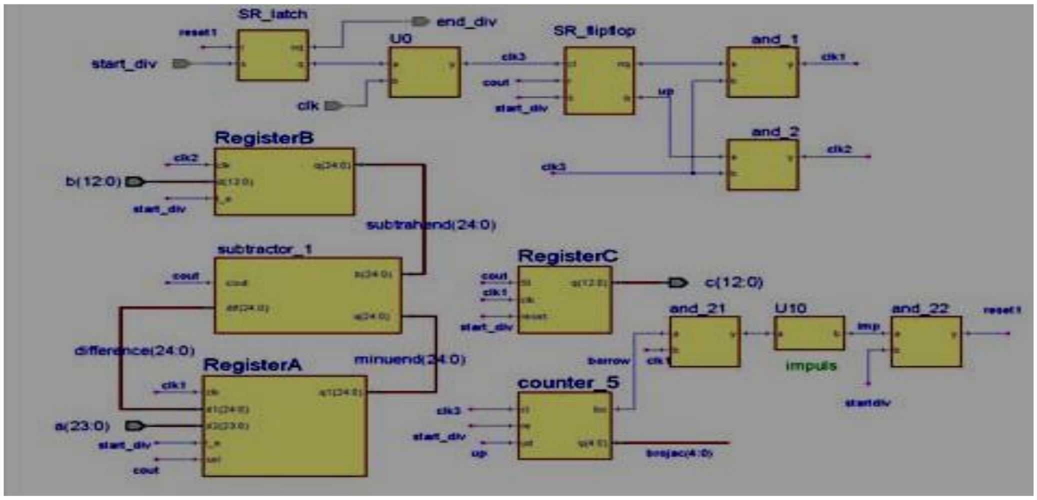

Fig.3. Block diagram of division implementation subsystem.

Dividend is 24-bit number, divisor is 13-bit number and quotient is 13-bit number. At the beginning of dividing procedure, *Register B* (initially storing divisor) is shifted to the left until its value becomes greater then value stored in *Register A* (dividend). The 5-bit counter counts the number of shifts. Then, counter starts counting down, and the procedure of quotient-digits computing follow. *Register A* is shifted left for one bit. Further, *Register A* changes its value, but *Register B* is keeping its value. The dividing procedure is performed through the successive subtractions (*Register A – Register B*). Each time the result is negative, the value *Register A* is shifted left for one position and zero is appended to *Register C* value on least significant position. Else, result of subtraction, shifted left for one position, is stored in *Register A and* digit 1 is appended to Register *C*. Since the number of quotient digits is equal to the number of shifts, the division operation is finished when counter reaches zero. The number of clock cycles needed for dividing is two times greater than number of significant digits of the quotient. The total number of clock periods (for initial value generation, two divide iterations, entering data in registers and control signals production) is in range from 40, for computing square root of 1, to 72, for computing square root of $(FFFFFF)_{16}$. System is described in VHDL and synthesized by Xilinx.

3. ITERATIVE ALGORITHM:

Next, an old, traditional method for square root computation will be considered. It is known as Iterative method or Longhand square root computation method. Obtained results confirm the idea that traditional methods are usually the best ones. The algorithm is very fast and its hardware implementation has small chip area. The algorithm computes square root on the same way like people do manually: It starts with grouping the digits in pairs, starting from the decimal point. The first digit of result is the greatest digit (e.g. *bm*) whose square is less than the first group value. The positive remainder after square of *bm* has been subtracted from the first group value, should be concatenated with next pair of digits and treated integral (e.g. value *cm*). Present result is formed of found digits (at the beginning it is only *bm*) Next digit (e.g. *bm-1*) in the resulting number, is the greatest number that meets the condition

$$(20 * present\_result + bm\text{-}1) * bm\text{-}1 \leq cm \quad (2)$$

The result of subtraction

$$cm - (20 * present\_result + bm\text{-}1) * bm\text{-}1 \quad (3)$$

should be concatenated with next pair of digits (like previously) and treated integral (e.g. value *cm-1*) hereafter. Present result is appended by new digit *bm-1*. The procedure is repeated until all input groups of digits are considered.

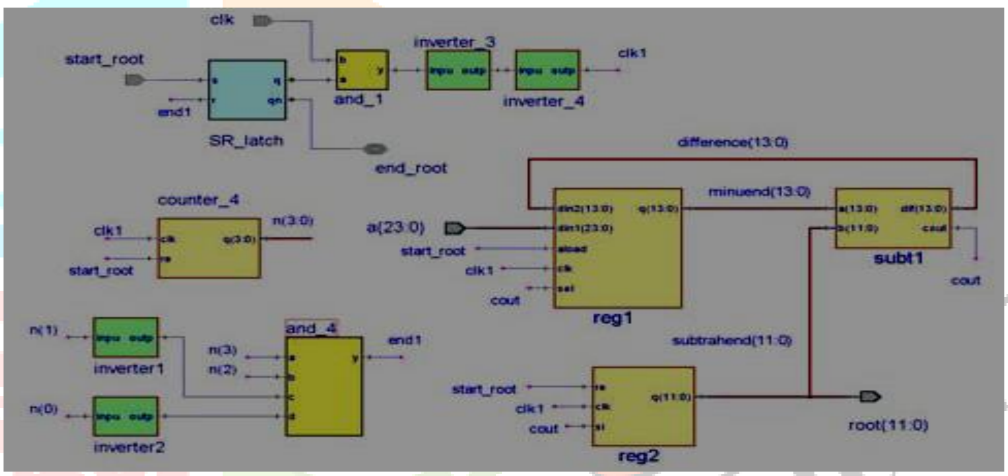

Fig.4. Block diagram of square rooter based on iterative algorithm.

Computation is significantly simpler if it is performed in radix 2. Like previous computation in radix 10, binary digits of number which square root should be found are grouped in pairs. In every iteration, two digits are appended to the right side of the present result (digit 0 for multiplying by two and digit 1 for possible digit of result). In radix 2 only digits 1 and 0 can be assumed for the next digit of result. Subtraction is performed, and if minuend is grater then subtrahend, digit 1 is appended to present result. Result of subtraction, concatenated with next pair of digits, form minuend for next subtraction. Else, present result is concatenated with zero. Minuend concatenated with next pair of digits forms minuend for next iteration. Schematic of digital system implementing iterative algorithm is given in Fig.4. System ports are: *clk* for clock, *start_root* for computation beginning, input 24-bit bus *a(23:0)* for input data, output 11-bit bus *root(11:0)* for output data, output signal *end_root* for indication that computation is finished. It consists of 36-bit register *Reg1*, 12-bit register *Reg2*, subtractor *Subt1* that consists of 14 full adders and 14 inverters, 4-bit counter *Counter_4* and few simple logic gates. At the beginning, 24 least significant bits of *Reg1* get the input value *a*. Other 12 most significant bits are set to zeros. *Reg2* also gets zero value. Minuend is made of 14 most 70 significant bits of *Reg1*. Present result is stored in *Reg2*. At the beginning of computation, present result is zero. Subtrahend is made of present result and two binary digits, zero or one, appended to the right side of present result. If minuend is greater than subtrahend, present result is appended by binary digit one. Difference is stored on minuend's place, in 14 higher bits of *Reg1*. After that, *Reg1* is shifted left for two positions. If minuend is smaller than subtrahend, subtraction results are not stored. *Reg1* is just shifted left for two positions. The algorithm is very fast. It uses only 12 iterations for the square-root completion. Every iteration is completed exactly in one clock period.

4. BINARY SEARCH ALGORITHM:

This algorithm can be found solving many different tasks in programming. Here, this algorithm is modified to speed up root computation. The computing procedure is simple: The resulting square root value e.g. $a_m\ a_{m-1}\ a_{m-2}\ \ldots\ a_1\ a_0$, has twice less number of bits comparing to input number, 12 bits in our case. The algorithm finds the value of square root bit-by-bit. First, the most significant bit a_m, is assumed to be 1 and other bits are zeros. Number 100...0 is squared and subtracted from the input number. If the remainder is positive, the assumed bit is correct. If not, the bit is 0. In the next iteration, the next bit a_{m-1} is assumed to be 1. Number a_m100...0 is squared and subtracted from the input number and the same considerations stand. The algorithm proceeds until LSB is considered. In order to speed up the algorithm, square computation is modified: If the temporary square-root solution is number $A = a_m a_{m-1} a_{m-2} \ldots a_{k0} \ldots 0$ (bits am, $am\text{-}_1$, $am\text{-}_2, \ldots, a_k$ are computed in previous iterations, others are still unknown and replaced by zeros), guess is $A' = a_m a_{m-1} a_{m-2} \ldots a_k 10 \ldots 0$, *i.e.*, assumed bit on position k-1 is 1. The square of the guess is computed using the square of temporary solution.

$$A' = (a_m a_{m-1} \ldots a_k 10 \ldots 0)^2 = (a_m\ a_{m-1} \ldots\ldots\ldots a_k \times 2^k + 2^{k-1})^2 \qquad (4)$$

$= (A + 2^{k-1})^2 = A^2 + 2^k A + 2^{2k-2}$ where A is the temporary solution. So, if square of temporary solution is known, square computation of guess is reduced to few shifting and addition operations. The algorithm is very fast - it uses only 12 clock iterations for the square-root completion, like previously described iterative algorithm.

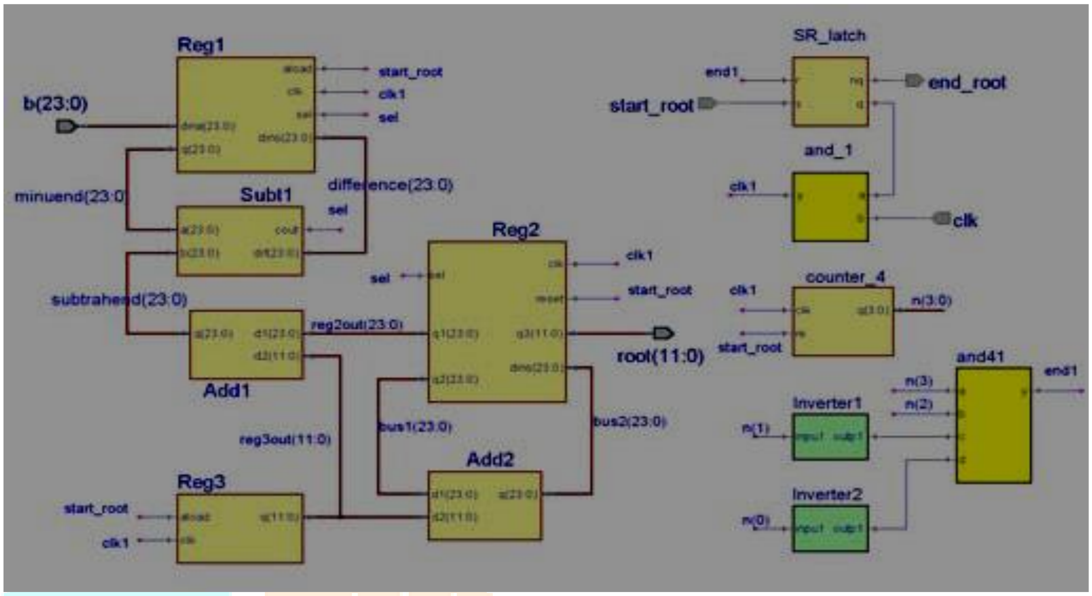

Fig.6. Block diagram of square rooter based on binary search algorithm.

Schematic of digital system implementing binary search algorithm is shown in Fig.6. The system ports are: *clk* port for clock, signal *start_root* for computation beginning, input 24-bit bus *b(23:0)* for input data, output 12-bit bus *root(11:0)* for output data, output signal *end_root* for indication that computation has been finished. System is composed of: registers *Reg1*, *Reg2* and *Reg3*; subtractor *Subt1*, adders *Add1* and *Add2* composed only of logical OR gates; 4-bit synchronous counter *Counter_4* and few simple logical gates in control logic. These parts function as follows. *Reg1* is 24-bit register that stores the difference of input value *b* and square of temporary solution (*A2*). At the beginning of computation, temporary solution is *A=00..0* (all binary numbers are unknown), so *Reg1* is loaded with number *B*. *Reg2* is also 24-bit register and stores value 2^kA, i.e., temporary solution shifted for k places (*k* is the number of unknown binary digits in temporary solution). At the start of computation *Reg2* is reset to all zeros. *Reg3* is 12-bit register that holds 2^{k-1}. At the beginning *k* is 12 and *Reg3* stores hexadecimal value *800*. It means that 12 bits are unknown. After every clock rising edge, number of unknown digits is decremented and the value in *Reg3* is shifted right. Subtractor *Subt1*, composed of 24 full adders and the same number of inverters, gives the control signal *sel* on *carry_out* output. If minuend is greater than subtrahend then *sel* is 1 and difference should be loaded into *Reg1*. Else, *sel* is 0 and *Reg1* retains its old value. Negative data should not be stored. Adders *Add1* and

Add2 are composed of logical OR gates. They get values of *Reg2* and *Reg3* outputs on their inputs.

One of them provides *subtrahend (23:0)* on its output bus:

Subtrahend (23:0):=Reg2 + $(Reg3)^2 = 2^kA+2^{2k-2}$ (5)

The other provides the data input for Reg2:

Reg2:=Reg2>>1+ $(Reg3)^2 = 2^{k-1}A+2^{2k-2} = 2^{k-1}(A+2^{k-1}) = 2^{k-1}(a_m\, a_{m-1}\, a_{m-2} \ldots a_k\, 10\ldots0) = 2^{k-1}A'$ (6)

The *>>1* denotes that *Reg2* value is shifted one bit to the right. There is no carry transition, so OR logical gates are used instead of full adders, providing significant saving in chip area. Number *Reg2>>1* = $2^kA>>1$ has 2^{k-1} zeros at least significant positions, while $(Reg3)^2=2^{2k-2}$. In every iteration, subtraction is performed. Subtrahend, 2^kA+2^{2k-2}, is subtracted from minuend, *Reg1* value ($B-A^2$). If minuend is greater than subtrahend, guess *A'* is correct and digit 1 is correctly assumed on bit position *k-1*. New temporary solution gets value from previous guess *A'*. New values are stored in *Reg1* and *Reg2*:

Reg1 :=$(B-A^2) - 2^kA+2^{2k-2} = B-(A+2^{k-1})^2 = B-A'^2$ (7)

Reg2:= $2^{k-1}A'$ (8)

If minuend is less than subtrahend, guess *A'* is incorrect and digit 1 is not correctly assumed on bit position *k-1*, i.e., 0 is the solution for bit position *k-1*. Temporary solution *A* retains its value, so *Reg1* value is the same as previous one.

Value in *Reg2* is shifted right for one bit position: Reg2:= (Reg2>>1) = $(2^kA>>1) = 2^{k-1}A$ (9)

After 12 ($m+1=12$) clock iterations, *Reg2* holds correct value of square root which can be read on bus *root (11:0)* $2^kA = 2^0A = A = a_m\, a_{m-1}\, a_{m-2} \ldots a_1\, a_0$ (10)

The implementation is verified in VHDL, and synthesized in Xilinx synthesis tool.

5. RESULTS:

Comparing the implementations independently of the physical realization, we can see that the implementation based on Newton-Raphson's method requires greater number of clock periods (in range from 40 to 70 depending on input data) than the other methods (exactly 12 clock cycles) for completing the square root solution. All systems are verified in VHDL and implemented in same standard cell technology. So, the properties of the resulting solutions can be regularly compared. Namely, the first VHDL simulation was performed in ModelSim tool. Logical synthesis is done in Xilinx.

6. CONCLUSION:

Three square root implementations were designed and their properties were compared. System implementation based on iterative algorithm provides the solution with smallest chip area and power consumption, and maximal clock frequency. All proposed solutions are very flexible and can be modified if square root of a number with more or less than 24 bits is required, if it is required to find bits after decimal point, etc. These demands can be accomplished by minor changes in VHDL code. It is worth to notice that Newton-Raphson's algorithm based square root computation circuit has the advantage of having built-in a division subsystem. So, it can be applied in all circumstances where both are needed, integer division and square root computation. In that case, significant saving in chip area can be accomplished.

REFERENCES:

1. M.Cornea-Hansen, B.Norin, “IA-64 Floating Point Operations and IEEE Standards for Binary Floating Point Arithmetic”, *IEEE Trans. Automat. Control*, vol. AC-22, pp. 210-222, April 1977.
2. M.Cornea-Hansen, R.Golliver “Correctness Proofs Outline for Newton-Raphson Based Floating Point Divide and Square Root”, New York: Wiley, 1975.
3. M.Cornea-Hasegan, “Proving IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms,” *Intel Technology Journal*,Q2, 1998 at http://developer.intel.com/technology/itj/q21998.htm
4. J.P.Grossman, “Roll Your Own Divide/Square Root Unit”, June 24, 1999
5. Majerski, S. “Square-Rooting Algorithms for High- Speed Digital Circuits”, IEEE Transactions on Computers, Vol. C-34, No. 8, August 1985, pp. 724-733
6. P.Freire, “Square Root Algorithms” http://www.pedro freire. com/sqrt
7. K.C.Chang, “Digital Systems Design with VHDL and Synthesis: An Integrated Approach”, IEEE Computer Society, Los Alamitos, California 1999.