International Journal of Computer Theory and Engineering, Vol.3, No.1, February, 2011
1793-8201

An Efficient Implementation of the Non Restoring Square Root Algorithm in Gate Level

Tole Sutikno, *Member, IACSIT*, *IEEE*

***Abstract*—This paper proposes an efficient strategy to implement modified non restoring algorithm based on FPGA in gate level abstraction of VHDL, which adopt fully pipelined architecture. A new basic building block is called controlled subtract-multiplex (CSM) is introduced. The main principle of the proposed method is similar with conventional non-restoring algorithm, but it only uses subtract operation and append 01, while add operation and append 11 is not used. The proposed strategy has conducted to implement FPGA successfully, and it is offer an efficient in hardware resource.**

***Index Terms*—FPGA, non-restoring, Gate Level, square root**

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 (DSP) algorithms, math coprocessor, data processing and control and even multimedia applications [1-6]. It is a classical problem in computational number theory and often encountered, which is a hard task to get an exact result [7-8].

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 have 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. Although the algorithms in [1-2] have a speed processing, they consumes too many hardware resource, while the algorithms in [9] although it cost less resource, but it has low speed. The similar architectures above have introduced by Xiaoliang [10], Thakkar [12] and Xiumin et al [13]. In the other study, Samawi et al [6] have introduced controlled add-sub (CAS) as basic building blocks. The effort is done to reduce hardware consumed, with moderate delay. The other architecture also has proposed is fully combinational architecture [4]. 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 [14].

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 is called controlled subtract-multiplex (CSM), and also it needs fewer pipeline stages compared with the proposed algorithm in [12]. Next, the performance of developed system will be compared to Samawi et al [6].

II. 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, 13]. It has several advantages, such as: every digit of the root found is

Manuscript received July 13, 2010.

Tole Sutikno is with the Department of Electrical Engineering, Universitas Ahmad Dahlan (UAD), Yogyakarta, Indonesia, e-mail: thsutikno@ieee.org, tole@ee.uad.ac.id.

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 (of course the process depends on 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 [15]. 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 . 00
     -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 . 00
-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

III. Proposed Square Root Algorithm

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.

```
  1  0  0  1 .
------------------
01 01 11 01 . 00   choose 1 squared, subtract, keep if positive
-1                 shift two bits at a time
---                subtract guess squared
00 01
-1 01  <---- sqrt so far with 01 appended : 2nd bit is a 0
------     no subtract because result would be negative
  01 11
 -10 01<---- sqrt so far with 01 appended: 3rd bit is a 0
--------   no subtract because result would be negative
  01 11 01
  -1 00 01
  --------
    11 00  <---- positive : 4th bit is a 1
```

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

Samavi, et al [6] has improved classical non-restoring digit-by-digit square root circuit by eliminate redundant blocks. Their circuit is referred to as the reduced area non-restoring circuit. However, it still based on constant digit of 01 or 11 and add-subtract as the main building block (still refer to Figure 1 b). This paper offers a simple alternative solution that it only uses subtracts operation and appends 01. As consequent, the subtract-multiplex is used as the main building block (refer to Figure 2). The principle of the proposed algorithm can be described as shown in Figure 3.

```
Step 0.  Start

Step 1.  Initialization 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 2.  Beginning at the binary point, divide the
         radicand into groups of two digits in both
         direction.

Step 3.  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 4.  Choose 1 squared, and then subtract.

         Fist developed root is "1" if the result of
         subtract is positive, and vice versa is "0"

Step 5.  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 6.  Go to step 5 until end group of two digits

Step 7.  End
```

Figure 3. The principle of proposed algorithm to solve square root

A simple hardware implementation of the proposed non-restoring digit-by-digit algorithm for unsigned 6-bit square root by an array structure is shown in Figure 4. The radicand is P (`P5,P4,P3,P2,P1,P0`), U (`U2,U1,U0`) as quotient and R (`R4,R3,R2,R1,R0`) as remainder. It can be shown that the implementation needs 3 stage pipelines. The

basic building blocks of the array are blocks called as *controlled subtract-multiplex* (CSM). Figure 5 present the details of a CSM. 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`.

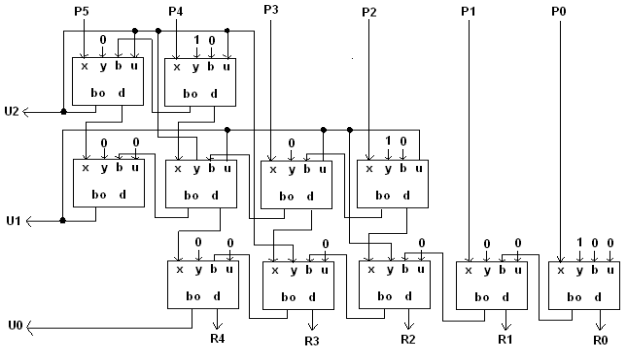


Figure 4. A simple hardware implementation of the non-restoring digit-by-digit algorithm for unsigned 6-bit square root

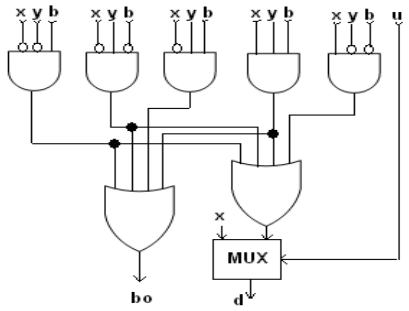


Figure 5. Internal structure of a CSM block

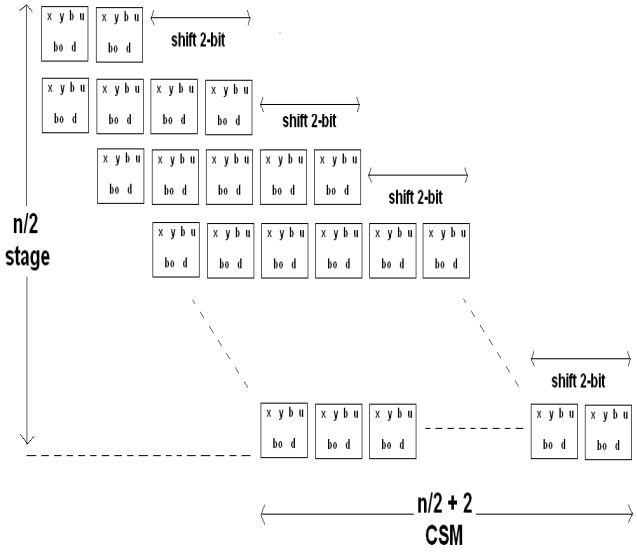


Figure 6. A simple hardware implementation of the non-restoring digit-by-digit algorithm for unsigned n-bit square root

The generalization of simple implementation of the non-restoring digit-by-digit algorithm for unsigned n-bit square root by an array structure is shown in Figure 6. Each row (stage) of the circuit in Figure 6 executes one-iteration of the non-restoring digit-by-digit square root algorithm, where it only uses subtracts operation and appends 01.

To optimize hardware resource utilization of the implementation above, specialized entities can be created as building block components. It will eliminate circuitry that is not needed. As example, the implementation in Figure 6 for unsigned 6-bit square root can be optimized become as shown in Figure 7. The specialized entities `A, B, C, D` and `E` are minimized CSM when input `ybu=100`, `yu=00`, `u=0`, `yu=10`, and `y=0` respectively, and the remainder is ignored. The generalization of optimized simple implementation of the non-restoring digit-by-digit algorithm for unsigned n-bit square root is shown in Figure 8.

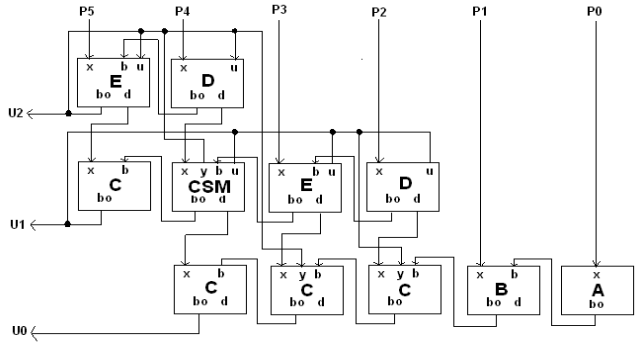


Figure 7. Optimized simple hardware implementation of the non-restoring digit-by-digit algorithm for unsigned 6-bit square root

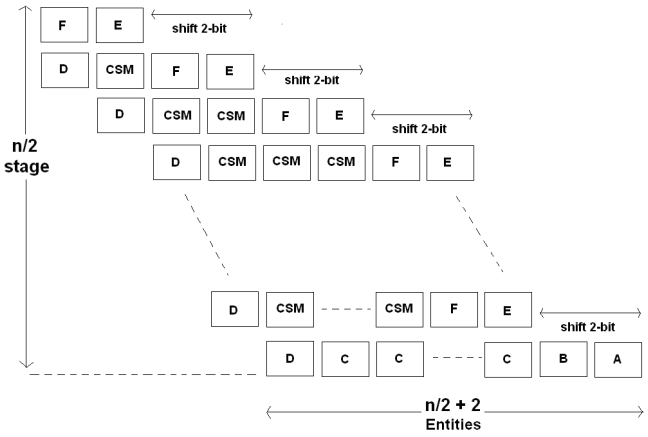


Figure 8. Optimized simple hardware implementation of the non-restoring digit-by-digit algorithm for unsigned n-bit square root

IV. THE IMPLEMENTATION OF THE NON RESTORING SQUARE ROOT ALGORITHM IN GATE LEVEL

The implementation of the proposed non restoring square root algorithm in gate level approach is conducted in VHDL language. The VHDL source codes for modules A, B, C, D, E, F and CSM is shown in Figure 9, 10, 11, 12, 13, 14 and 15 respectively.

```
-- modul A
library IEEE;
use IEEE.std_logic_1164.all;

entity A is
  port ( x  : in  std_logic;
         bo : out std_logic);
end S0b;

architecture circuits of A is
begin  -- circuits of S0b
  bo <= not x;
end;
```

Figure 9. Source code for module A

```
-- module B
library IEEE;
use IEEE.std_logic_1164.all;

entity B is
  port ( x  : in  std_logic;
```

```
        b  : in  std_logic;
        bo : out std_logic);
end S1b;

architecture circuits of B is
begin
  bo <= (not x) and b;
end;
```

Figure 10. Source code for module B

```
-- module C
library IEEE;
use IEEE.std_logic_1164.all;

entity C is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         b  : in  std_logic;
         bo : out std_logic);
end;

architecture circuits of C is
  signal t011, t111, t010, t001 : std_logic;
begin
  t011 <= (not x) and y and b;
  t111 <= x and y and b;
  t010 <= (not x) and y and (not b);
  t001 <= (not x) and (not y) and b;
  bo   <= t011 or t111 or t010 or t001;
end;
```

Figure 11. Source code for module C

```
-- module D
library IEEE;
use IEEE.std_logic_1164.all;

entity D is
  port ( x  : in  std_logic;
         b  : in  std_logic;
         bo : out std_logic);
end;

architecture circuits of D is
begin
  bo <= (not x) nand b;
end;
```

Figure 12. Source code for module D

```
-- module E
library IEEE;
use IEEE.std_logic_1164.all;

entity E is
  port ( x  : in  std_logic;
         u  : in  std_logic;
         d  : out std_logic;
         bo : out std_logic);
end;

architecture circuits of E is
begin
  bo   <= not x;
  d    <= not x when u='1' else x;
end;
```

Figure 13. Source code for module E

```
-- module F
library IEEE;
use IEEE.std_logic_1164.all;

entity F is
  port ( x  : in  std_logic;
         b  : in  std_logic;
         u  : in  std_logic;
         d  : out std_logic;
         bo : out std_logic);
end S1;

architecture circuits of F is
  signal t100, t001, td : std_logic;
begin
  t001 <= (not x) and b;
  t100 <= x and (not b);
  bo   <= t001;
  td   <= t100 or t001;
  d    <= td when u='1' else x;
end;
```

Figure 14. Source code for module F

```
-- module CSM
library IEEE;
use IEEE.std_logic_1164.all;

entity CSM is
  port ( x  : in  std_logic;
         y  : in  std_logic;
         b  : in  std_logic;
         u  : in  std_logic;
         d  : out std_logic;
         bo : out std_logic);
end Sm;

architecture circuits of CSM is
  signal t011, t111, t010, t001, t100, td : std_logic;
begin  -- circuits of CSM
  t011 <= (not x) and y and b;
  t111 <= x and y and b;
  t010 <= (not x) and y and (not b);
  t001 <= (not x) and (not y) and b;
  t100 <= x and (not y) and (not b);
  bo   <= t011 or t111 or t010 or t001;
  td   <= t100 or t001 or t010 or t111;
  d    <= td when u='1' else x;
end;
```

Figure 15. Source code for module CSM

The main program of the proposed implementation is as shown Figure 16.

```
-- main program
library IEEE;
use IEEE.std_logic_1164.all;

entity sqrt64 is
  port ( P : in  std_logic_vector(64 downto 0);
         U : out std_logic_vector(31 downto 0));
end sqrt64;

architecture circuits of sqrt64 is
  signal x3162, b3162,x3163, b3163, bxx  : std_logic;
  signal x3060, b3060, x3061, b3061, x3062, b3062, b3063 : std_logic;

  ……
  ……

  b0033 : std_logic;

begin  -- circuits of sqrt64
  --                                x      y      b      u      d      bo
  s3162: entity work.S0  port map(P(62),             b3163, x3162, b3162);
  s3163: entity work.S1  port map(P(63),       b3162, b3163, x3163, bxx );
  b3163 <= not bxx;

  ……
  ……

  s0032: entity work.Sb  port map(x0132, b3163, b0031,             b0032);
  s0033: entity work.Sn  port map(x0133,       b0032,              b0033);

  -- set output bits
  U(0)  <= b0033;  U(1)  <= b0134;
  U(2)  <= b0235;  U(3)  <= b0336;
  U(4)  <= b0437;  U(5)  <= b0538;
  U(6)  <= b0639;  U(7)  <= b0740;
  U(8)  <= b0841;  U(9)  <= b0942;
  U(10) <= b1043;  U(11) <= b1144;
  U(12) <= b1245;  U(13) <= b1346;
  U(14) <= b1447;  U(15) <= b1548;
  U(16) <= b1649;  U(17) <= b1750;
  U(18) <= b1851;  U(19) <= b1952;
  U(20) <= b2053;  U(21) <= b2154;
  U(22) <= b2255;  U(23) <= b2356;
  U(24) <= b2457;  U(25) <= b2558;
  U(26) <= b2659;  U(27) <= b2760;
  U(28) <= b2861;  U(29) <= b2962;
  U(30) <= b3063;  U(31) <= b3163;

end architecture circuits;  -- of sqrt64
```

Figure 16. Source code for module CSM

The implementation of the non restoring square root algorithm also has been tried in register transfer language (RTL) approach as shown in Figure 17. By using this approach, the VHDL source code can be composed in shorter and also it is easy to modify. Unfortunately, the RTL approach is not suitable to minimize the consumption of hardware resource. Therefore, the research focuses to implement the proposed non restoring square root algorithm in gate level approach to optimize the consumption of hardware resource.

```
-- source for 64-bit square root
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity m_sqrt is
generic ( n: positive:= 32 );
```

```
-- n: number of bits for output
--2n: number of bits for input

port    (
        input     : in std_logic_vector(2*n - 1 downto 0);
        -- input, 2*n bits
        output    : out std_logic_vector (n-1 downto 0)
        -- output, n bits
        );
end entity;

architecture RTL of m_sqrt is
--array to store partial remainder
type table1 is array(0 to n) of std_logic_vector(n+1  downto 0);
type table2 is array(0 to n) of std_logic_vector(n  downto 0);
begin

sqrt_calculation:
process(input) is
-- variables to store partial remainder
variable remain,r     : table1;
-- variables to store partial results of sqrt
variable qint         : table2;
variable q            : table2;
begin

    remain(0) := (others=>'0');     --r0 = -1 TUKAR YG NI!!!!
    qint  (0) := (others=>'0');     --q0 = 0
    r(0) :=  (others=>'0');

    for i in 1 to n loop
    if  (signed(remain(i-1)) >= 0) then
        r(i) := remain(i-1)(n-1 downto 0) & (input(2*(n-i+1)-1
                downto 2*(n-i+1)-2));
    else
        r(i) := r(i-1)(n-1 downto 0) & (input(2*(n-i+1)-1 downto
                2*(n-i+1)-2));
    end if;

    q(i):= qint(i-1)(n-2 downto 0) & "01";
    remain(i) := std_logic_vector ( unsigned(r(i)) -
                 unsigned(q(i)));
    qint(i)   := qint(i-1)(n-1 downto 0) & not(remain(i)(n+1));

    end loop;
    output <= qint(n)(n-1 downto 0);

end process;
end RTL;
```

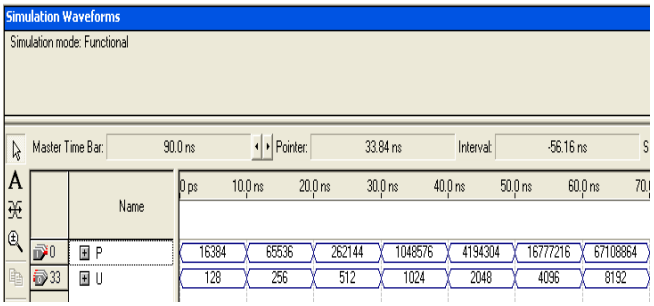
Figure 17. Source code for the proposed non restoring square root algorithm in RTL approach

V. Results And Analysis

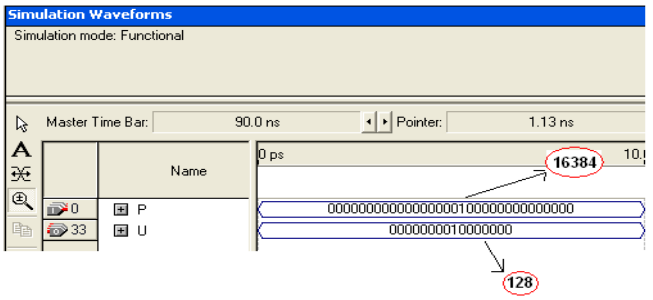
In the previous sections, optimized simple hardware implementation method of the non-restoring digit-by-digit algorithm for square root was explained. In this section, simulation results of 32-bit and 64-bit square root based on Altera APEX 20KE FPGA using the above method are presented, as shown in Figure 18.

In this simulation, `P` is radicand and `U` is quotient. The results showed that the implementation has succeeded and worked properly.

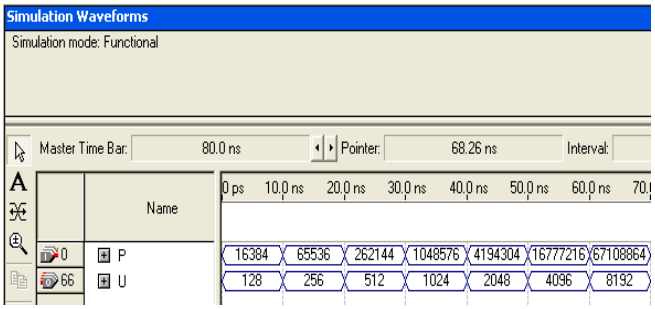
Based on compilation report, to implement 32-bit and 64-bit square root using optimized simple hardware implementation method of the non-restoring digit-by-digit algorithm using Altera FPGA APEX 20KE are needed 256 and 1023 logic element (LE) respectively. The comparison of results obtained from different implementation method is shown in Table 1. This comparison of LE or logic cell (LC) usage is listed based on references [6] and [16]. It has shown a fantastic value for reducing of hardware resource consumed. This is due adoption fully pipelined architecture and also simplification of CSM as shown in Figure 8.



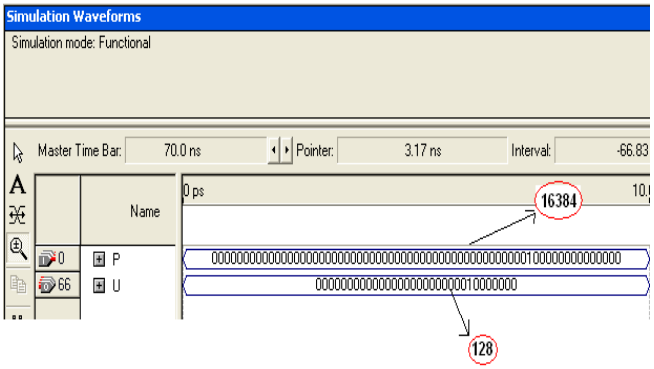
(a)



(b)



(c)



(d)

Figure 18. Simulation result of n-bit square root using optimized simple hardware implementation method of the non-restoring digit-by-digit algorithm: (a) 32-bit in decimal display, (b) 32-bit in binary display, (c) 64-bit in decimal display, (d) 64-bit in binary display

TABLE 1. The Comparison of Logic Element Usage

No	Method	LEs Usage	
		32-bit square root	64-bit square root
1	Classical-NR	1008	4092
2	Reduced-Area-NR	632	2464
3	Modular-NR	624	2468
4	Simple-X-Module	648	2488
5	Proposed	**256**	**1023**

Based on [16], for Altera APEX 20KE & Xilinx Virtex-E, 1 LC = 1 LE, and 1 CLB = 4 LE

TABLE 2. THE LIST OF LC/LE USAGE USING VARIOUS ALTERA FPGA FAMILIES

No	FPGA Families	LEs Usage	
		32-bit square root	64-bit square root
1	FLEX10K	256	1024
2	ACEX1K	256	1024
3	Cyclone	256	1025
4	Cyclone II	256	1025
5	Cyclone III	256	1025
6	Stratix	256	1025
7	Stratix GX	256	1025
8	APEX20KE	256	1023
9	Arria GX	185	736
10	Stratix II	185	736
11	Stratix III	185	736

The proposed algorithm is also tried to implement using various FPGA families. Table 2 shows the list of LC/LE usage. The number of employed LE indicates the size of the implemented circuit “hardware resource”. They showed that proposed method is most efficient of hardware resource. This is reasonable, because it only uses subtract operation and append 01, without add operation and append 11, and also adopt fully pipelined architecture. The results have proven that the proposed strategy is easy to implement, less resource consume and suitable to be expanded for larger number to solve complicated square root problem in FPGA implementation.

VI. CONCLUSION

This contribution presented a modification of conventional non-restoring digit-by-digit calculation method for implementation in FPGA hardware which adopts fully pipelined architecture. The main principle of the proposed method is two-bit shifter and subtractor-multiplexor operations, only uses subtract operation and append 01, without add operation and append 11. The proposed strategy has conducted to implement FPGA based unsigned 32 bit and 64-bit binary square root successfully. The results have shown that the proposed method is most efficient of hardware resource compare to other researches. The strategy also can be expanded to larger number easily, to solve complicated square root problem in FPGA implementation.

REFERENCES

[1] L. Yamin and C. Wanming, "Implementation of Single Precision Floating Point Square Root on FPGAs," in IEEE Symposium on FPGA 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 Workhsop, Salvador Bahia, Brazil, 2005.
[5] XiaojunWang, "Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms," Doctor of Philosophy, Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts, 2007.
[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 algorithms," in Computers and Digital Techniques, IEE Proceedings E, Italy, 1990, pp. 31 - 40.
[12] A. J. Thakkar and A. Ejnioui, "Design and implementation of double precision floating point division and square root on FPGAs," in Aerospace Conference, 2006 IEEE, 2006, p. 7 pp.
[13] W. Xiumin, et al., "A New Algorithm for Designing Square Root Calculators Based on FPGA with Pipeline Technology," in Hybrid Intelligent Systems, 2009. HIS '09. Ninth International Conference on, 2009, pp. 99-102.
[14] G. Renxi, et al., "Hardware Implementation of a High Speed Floating Point Multiplier Based on FPGA," in 4th International Conference on Computer Science & Education, Nanning, Guangxi, P.R.China, 2009.
[15] S. Dattalo. (2000, March 17, 2010). Square Root Theory. Available: http://www.dattalo.com/technical/theory/sqrt.html
[16] March 30, 2010). Comparing Altera APEX 20KE & Xilinx Virtex-E Logic Densities. Available: http://www.altera.com/products/devices/apex/features/apx-compdensity.html

Tole Sutikno received his B.Eng. degree in electrical engineering from Diponegoro University (UNDIP), Semarang, Indonesia, his M.Eng. degree in Power Electronics from Gadjah Mada University UGM), Yogyakarta, Indonesia, in 1999 and 2004, respectively. He is currently PhD Student in Energy Conversion, Universiti Teknologi Malaysia (UTM), Johor, Malaysia. He is currently lecturer at Electrical Engineering Department, Ahmad Dahlan University (UAD), Yogyakarta, Indonesia. He is also as Editor-in-Chief of Telecommunication, Computing, Control and Electronics (TELKOMNIKA) Scientific Journal and advisor of Robotic Development Community (RDC) in the university. His research interests include the field of power electronics motor drive systems and field programmable gate array (FPGA) applications.

Tole Sutikno (M’07 IEEE) received his B.Eng. degree in electrical engineering from Diponegoro University (UNDIP), Semarang, Indonesia, his M.Eng. degree in Power Electronics from Gadjah Mada University UGM), Yogyakarta, Indonesia, in 1999 and 2004, respectively. Since 2001 he has been a lecturer in Electrical Engineering Department, Universitas Ahmad Dahlan (UAD), Yogyakarta, Indonesia. He is also as Editor-in-Chief of TELKOMNIKA Journal, ISSN 1693-6930 (Indonesian Journal of Electrical Engineering). Currently, he is pursuing PhD degree in Energy Conversion Department, Universiti Teknologi Malaysia (UTM), Johor, Malaysia. His research interests include the field of power electronics, motor drive systems and field programmable gate array (FPGA) applications. Email: thsutikno@ieee.org or tole@ee.uad.ac.id