A New Non-Restoring Square Root Algorithm and Its VLSI Implementations

Yamin Li and Wanming Chu
Computer Architecture Laboratory
The University of Aizu
Aizu-Wakamatsu 965-80 Japan
yamin@u-aizu.ac.jp, w-chu@u-aizu.ac.jp

Abstract

In this paper, we present a new non-restoring square root algorithm that is very efficient to implement. The new algorithm presented here has the following features unlike other square root algorithms. First, the focus of the "non-restoring" is on the "partial remainder", not on "each bit of the square root", with each iteration. Second, it only requires one traditional adder/subtractor in each iteration, i.e., it does not require other hardware components, such as seed generators, multipliers, or even multiplexors. Third, it generates the correct resulting value even in the last bit position. Next, based on the resulting value of the last bit, a precise remainder can be obtained immediately without any correction or addition operation. And finally, it can be implemented at very fast clock rate because of the very simple operations at each iteration. We illustrate two VLSI implementations of the new algorithm. One is a fully pipelined high-performance implementation that can accept a new square-root instruction each clock cycle with each pipeline stage requiring a minimum number of gate counts. The other is a low-cost implementation that uses only a single adder/subtractor for iterative operation.

Keywords and phrases: Square root, non-restoring algorithm, Newton iteration, pipeline operation, VLSI design

1. Introduction

A number of square root (and division) algorithms have been developed and the *Newton* method has been adopted in many implementations [1] [6]. In order to calculate $Q = \sqrt{D}$, an approximate value is calculated by iterations. For example, we can use the Newton method on $f(T) = 1/T^2 - D$ to derive the iteration equation $T_{i+1} = T_i \times (3 - T_i^2 \times D)/2$, where T_i is an approximate value of $1/\sqrt{D}$. After n-time iterations, an approximate value of $Q = \sqrt{D}$ can be obtained by multiplying T_n by D. At the beginning, a seed (T_0) is generated by hardware circuitry, a ROM table for instance. At each iteration, multiplications and additions or subtractions are needed.

In order to speed up the multiplication, it is usual to use a fast parallel multiplier to get a partial production and then use an adder to get the production. Because the multiplier requires a rather large number of gate counts, it is impractical to place as many multipliers as required to realize fully pipelined operation for division *(div)* and square root *(sqrt)* instructions. In the design of most commercial RISC processors, a multiplier is used for all iterations of *div* or *sqrt* instructions. This means that the processors are not capable of accepting a new *div* or *sqrt* instruction for each clock cycle.

However, many applications require a fast pipelined square root operation. For the purpose of fast vector normalization, G. Knittel presented a design technique for pipelined operation that uses subtractors and multiplexors [2].

In this paper, we describe a new non-restoring square root algorithm that requires neither multipliers nor multiplexors. Compared with previous non-restoring algorithms, our algorithm is very efficient for VLSI implementation. It generates the correct resulting value at each iteration and does not require extra circuitry for adjusting the result bit. The operation at each iteration is simple: addition or subtraction based on the result bit generated in previous iteration. The remainder of the addition or subtraction is fed via registers to the next iteration directly even it is negative. At the last iteration, if the remainder is non-negative, it is a precise remainder. Otherwise, we can obtain a precise remainder by an addition operation.

This algorithm has been implemented in a multithreaded processor design which has been developed at University of Aizu using Toshiba TC180C/E/ TC183C/E Gate Array Library [7]. We also implemented and verified the algorithm on XILINX FPGA chip. The implementations are simple and more area-time efficient than many existing designs.

The paper is organized as follows. Section 2 describes previous non-restoring square root algorithms. Section 3 gives our new non-restoring square root algorithm. Section 4 and 5 introduce two VLSI implementations for the new algorithm. One is a fully pipelined implementation and the other is a low-cost implementation. The following section investigates the performance and cost compared with the Newton iteration algorithm. The final section presents conclusions.

2. Previous Non-Restoring Square Root Algorithm

Assume that an operand is denoted by a 32-bit unsigned number: $D = D_{31}D_{30}D_{29}...D_1D_0$. The value of the operand is $D_{31} \times 2^{31} + D_{30} \times 2^{30} + D_{29} \times 2^{29} + ...D_1 \times 2^1 + D_0 \times 2^0$. For every pair of bits of the operand, the integer part of square root has one bit (see Fig. 1). Thus the integer part of square root for a 32-bit operand has 16 bits: $Q = Q_{15}Q_{14}Q_{13}...Q_1Q_0$.

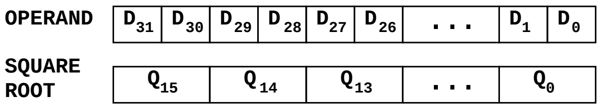

Figure 1. Format of operand and square root

At first, we reset $Q = 0$ and then iterate from $k = 15$ to 0. In each iteration, we set $Q_k = 1$, and subtract $Q \times Q$ from D. If the result is negative, then setting Q_k made Q too big, so we reset $Q_k = 0$. This algorithm modifies each bit of Q twice. This is called a *restoring square root algorithm*. An implementation example can be found in [5].

A *non-restoring square root algorithm* modifies each bit of Q once rather than twice. It begins with an initial guess of $Q = Q_{15}Q_{14}Q_{13}...Q_1Q_0 = 100...00$ (partial root) and then iterates from $k = 14$ to 0. In each iteration, D is subtracted by the squared partial root: $D - Q \times Q$. Based on the sign of the result, the algorithm adds or subtracts a 1 in Q_k position. An 8-bit example of the algorithm is shown in Fig. 2. Implementation examples can be found in [3] and [4].

We can see that the algorithm has following disadvantages. First, it requires an addition/subtraction (increase/decrease) operation to get a resulting bit value. Second, the algorithm may produce an error in the last bit position. Third, it requires an operation of $D - Q \times Q$ at each iteration with the result not being used for the next iteration. Although the multiplication of $Q \times Q$ can be replaced by substitute variable, the circuitry required is still complex.

```
D = 01,11,11,11
Q = 1000  D - Q x Q  = 00,11,11,11  is nonnegtive
  + 0100
Q = 1100  D - Q x Q  = 11,10,11,11  is negtive
  - 0010
Q = 1010  D - Q x Q  = 00,01,10,11  is nonnegtive
  + 0001
Q = 1011  D - Q x Q  = 00,00,01,10
```

Figure 2. An example of previous non-restoring square root algorithm

3. The New Non-Restoring Square Root Algorithm

The focus of the previous restoring and non-restoring algorithms is on *each bit of the square root* with each iteration. In this section, we describe a *new non-restoring square root algorithm*. The focus of the new algorithm is on the *partial remainder* with each iteration. The algorithm generates a correct resulting bit in each iteration including the last iteration. The operation during each iteration is very simple: addition or subtraction based on the sign of the result of previous iteration. The partial remainder generated in each iteration is used in the next iteration even it is negative (this is what non-restoring means in our new algorithm). At the final iteration, if the partial remainder is not negative, it becomes the final precise remainder. Otherwise, we can get the final precise remainder by an addition to the partial remainder.

The following is the new non-restoring square root algorithm written in the C language.

```
unsigned squart(D, r)      /*Non-Restoring sqrt*/
unsigned D;             /*D:32-bit unsigned integer
                                 to be square rooted */
int      *r;
{
  unsigned Q=0;         /*Q:16-bit unsigned integer
                                            (root)*/
  int       R=0;  /*R:17-bit integer (remainder)*/

  int       i;

  for (i=15;i>=0;i--)         /*for each root bit*/
  {
     if (R>=0)
     {                            /*new remainder:*/
        R=(R<<2)|((D>>(i+i))&3);
        R=R-((Q<<2)|1);                    /*-Q01*/
     }
     else
     {                            /*new remainder:*/
        R=(R<<2)|((D>>(i+i))&3);
```

```
      R=R+((Q<<2)|3);                        /*+Q11*/
    }
    if (R>=0) Q=(Q<<1)|1;                  /*new Q:*/
    else      Q=(Q<<1)|0;                  /*new Q:*/
  }

                            /*remainder adjusting*/
  if (R<0) R= R+((Q<<1)|1);
  *r=R;                          /*return remainder*/
  return(Q);                          /*return root*/
}
```

An 8-bit numerical example is given in Fig. 3. For good readable, some high order remainder bits are also shown, but they are not needed. It is sufficient for the remainder to have at most one bit more than the root.

```
D = 01,11,11,11     R=0     Q=0000  R: nonnegative
R = 01
  - 01            (-Q01)
R = 00,11                   Q=0001  R: nonnegative
  - 01,01         (-Q01)
R = 11,10,11                Q=0010  R: negative
  + 00,10,11      (+Q11)
R = 00,01,10,11             Q=0101  R: nonnegative
  - 00,01,01,01   (-Q01)
R = 00,00,01,10             Q=1011  R: nonnegative
```

Figure 3. An example of Non-Restoring Square Root

Before we explain the reason of why the algorithm works, we will describe a restoring square root algorithm that also uses the partial remainder for the next iteration.

For the $D = D_{31}D_{30}D_{29}...D_1D_0$ and $Q = \sqrt{D} = Q_{15}Q_{14}Q_{13}...Q_1Q_0$, we start the calculation with the most significant pair $(D_{31}D_{30})$ of D. There are four possible values: 00, 01, 10, and 11. The integer part of the square root (Q_{15}) will be 0 (for 00) or 1 (for 01, 10, and 11). In general, we can determine the Q_{15} by subtracting 01 from $D_{31}D_{30}$. If the result is negative, then $Q_{15} = 0$, otherwise $Q_{15} = 1$. By subtracting $Q_{15} \times Q_{15}$ from $D_{31}D_{30}$, we can get the partial remainder $r_{15} = R^{15}_{31}R^{15}_{30} = D_{31}D_{30} - Q_{15} \times Q_{15}$. Here, the 15 in $R^{15}_{31}R^{15}_{30}$ means that the remainder is for the square root bit Q_{15}. The possible values of the remainder include 00, 01, and 10. We also use q_k to denote the partial root obtained at iteration k: $q_k = Q_{15}Q_{14}...Q_k$.

Now, consider the next pair $(D_{29}D_{28})$. The new partial remainder is $r_{15}D_{29}D_{28} = R^{15}_{31}R^{15}_{30}D_{29}D_{28}$. The present task is to find the Q_{14} so that the partial root $q_{14} = Q_{15}Q_{14}$ is the integer part of the square root for $D_{31}D_{30}D_{29}D_{28}$. We have Equation 1:

$$D_{31}D_{30}D_{29}D_{28} \geq (Q_{15}Q_{14}) \times (Q_{15}Q_{14}). \quad (1)$$

Since $(Q_{15}Q_{14}) \times (Q_{15}Q_{14}) = (2 \times Q_{15} + Q_{14}) \times (2 \times Q_{15}+Q_{14}) = 4\times Q_{15}\times Q_{15}+4\times Q_{15}\times Q_{14}+Q_{14}\times Q_{14}$ and $R^{15}_{31}R^{15}_{30}D_{29}D_{28} = 4\times R^{15}_{31}R^{15}_{30}+D_{29}D_{28} = 4\times(D_{31}D_{30}-Q_{15}\times Q_{15})+D_{29}D_{28} = D_{31}D_{30}D_{29}D_{28}-4\times Q_{15}\times Q_{15}$, we can get Equation 2 from Equation 1:

$$R^{15}_{31}R^{15}_{30}D_{29}D_{28} \geq 4 \times Q_{15} \times Q_{14} + Q_{14} \times Q_{14}. \quad (2)$$

If we assume $Q_{14} = 1$, the Equation 2 turns into

$$R^{15}_{31}R^{15}_{30}D_{29}D_{28} \geq 4 \times Q_{15} + 1. \quad (3)$$

The value of right side of Equation 3 is $Q_{15}01$. If the result of $R^{15}_{31}R^{15}_{30}D_{29}D_{28}-Q_{15}01$ is negative, then $Q_{14} = 0$, i.e., $q_{14} = Q_{15}0$, otherwise $Q_{14} = 1$, i.e., $q_{14} = Q_{15}1$. And we will have a new remainder: $r_{14} = R^{14}_{30}R^{14}_{29}R^{14}_{28} = R^{15}_{31}R^{15}_{30}D_{29}D_{28} - 4 \times Q_{15} \times Q_{14} + Q_{14} \times Q_{14}$, i.e., if $Q_{14} = 0$, the remainder is left unchanged; $R^{15}_{30}D_{29}D_{28}$ is just bypassed.

In a VLSI implementation, a multiplexor can perform the selection of the unchanged remainder or the changed remainder generated by subtractor (See Fig. 4).

Note that the R^{14}_{31} is always 0. The reason is as follows. If we assume T is the integer part of the square root of operand S, then we have the equation $S = (T \times T + R) < (T+1) \times (T+1)$ where R is the remainder. Thus, $R < (T+1) \times (T+1) - T \times T = 2 \times T + 1$, i.e., $R \leq 2 \times T$ because the remainder R is an integer. It means that the remainder has at most one binary bit more than the square root.

For clarity, we will demonstrate the calculation of the next square root bit, Q_{13}. Consider the new pair $(D_{27}D_{26})$, we get the new remainder: $R^{14}_{30}R^{14}_{29}R^{14}_{28}D_{27}D_{26}$. Similarly, we have

$$D_{31}D_{30}D_{29}D_{28}D_{27}D_{26} \geq (Q_{15}Q_{14}Q_{13}) \times (Q_{15}Q_{14}Q_{13}). \quad (4)$$

Since $(Q_{15}Q_{14}Q_{13}) \times (Q_{15}Q_{14}Q_{13}) = (2 \times Q_{15}Q_{14} + Q_{13}) \times (2 \times Q_{15}Q_{14} + Q_{13}) = 4 \times Q_{15}Q_{14} \times Q_{15}Q_{14} +4 \times Q_{15}Q_{14} \times Q_{13} + Q_{13} \times Q_{13}$ and $R^{14}_{30}R^{14}_{29}R^{14}_{28}D_{27}D_{26} = D_{31}D_{30}D_{29}D_{28}D_{27}D_{26} - 4 \times Q_{15}Q_{14} \times Q_{15}Q_{14}$, we can get Equation 5:

$$R^{14}_{30}R^{14}_{29}R^{14}_{28}D_{27}D_{26} \geq 4 \times Q_{15}Q_{14} \times Q_{13} + Q_{13} \times Q_{13}. \quad (5)$$

Again, if we assume $Q_{13} = 1$, the Equation 5 turns into

$$R^{14}_{30}R^{14}_{29}R^{14}_{28}D_{27}D_{26} \geq 4 \times Q_{15}Q_{14} + 1 = Q_{15}Q_{14}01. \quad (6)$$

Also, $Q_{13} = 0$ if $R^{14}_{30}R^{14}_{29}R^{14}_{28}D_{27}D_{26} - Q_{15}Q_{14}01$ is negative and $Q_{13} = 1$ otherwise. The remainder $r_{13} = R^{13}_{29}R^{13}_{28}R^{13}_{27}R^{13}_{26}$ will be unchanged when $Q_{13} = 0$, or $r_{13} = R^{13}_{29}R^{13}_{28}R^{13}_{27}R^{13}_{26} = R^{14}_{30}R^{14}_{29}R^{14}_{28}D_{27}D_{26} - Q_{15}Q_{14}01$ when $Q_{13} = 1$. Also note that R^{13}_{30} is always 0. We got $q_{13} = Q_{15}Q_{14}Q_{13}$.

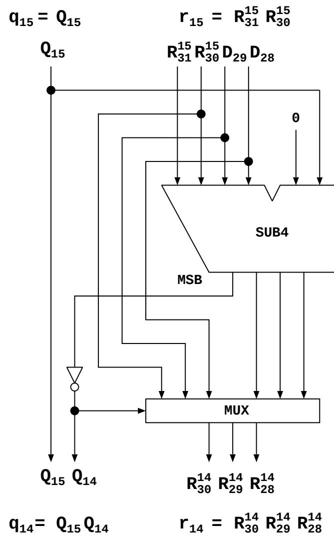

Figure 4. A multiplexor for restoring algorithm

We can repeat this calculation until the $q_0 = Q_{15}Q_{14}...Q_0$ is calculated. We can now explain why the non-restoring algorithm works. Let q_k be the partial square root and r_kab be the partial remainder at step k where ab is the next pair of operand and $a = D_{2\times k-1}$ and $b = D_{2\times k-2}$. The algorithm computes

$$r_{k-1} = r_kab - q_k01 \tag{7}$$

if the sign of the previous result is not negative. The new partial remainder is $r_{k-1}cd$ where cd is the next pair of operands and $c = D_{2\times(k-1)-1}$ and $d = D_{2\times(k-1)-2}$, i.e., the new remainder

$$\begin{aligned} r_{k-1}cd &= 4 \times r_{k-1} + cd \\ &= 4 \times (r_kab - q_k01) + cd \\ &= r_kabcd - q_k0100. \end{aligned} \tag{8}$$

If r_{k-1} is not negative, the algorithm is the same as the restoring algorithm:

$$q_{k-1} = 2 \times q_k + 1 = q_k1, \tag{9}$$

$$r_{k-2} = r_{k-1}cd - q_{k-1}01. \tag{10}$$

But if r_{k-1} is negative, both the algorithms (restoring and non-restoring) compute

$$q_{k-1} = 2 \times q_k + 0 = q_k0, \tag{11}$$

while the restoring algorithm computes

$$r_{k-2} = r_kabcd - q_{k-1}01, \tag{12}$$

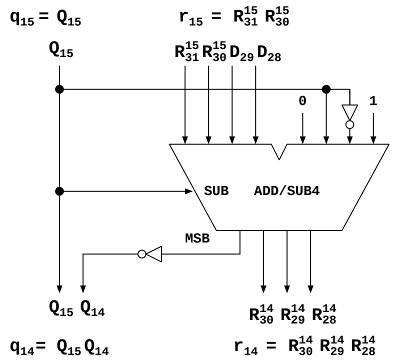

Figure 5. No multiplexor needed for non-restoring algorithm

where r_kab is the previous remainder passed via multiplexor. Referring to Equation 8, we can re-write Equation 12 as following.

$$r_{k-2} = r_{k-1}cd + q_k0100 - q_{k-1}01. \tag{13}$$

Referring to Equation 11, we can re-write Equation 13 as following.

$$r_{k-2} = r_{k-1}cd + q_{k-1}100 - q_{k-1}01. \tag{14}$$

Because $q_{k-1}100 - q_{k-1}01 = (8 \times q_{k-1} + 100) - (4 \times q_{k-1} + 01) = 4 \times q_{k-1} + 11 = q_{k-1}11$, we get

$$r_{k-2} = r_{k-1}cd + q_{k-1}11. \tag{15}$$

Equation 15 is just what the non-restoring algorithm does. In the VLSI implementation, only a single traditional adder/subtractor is required to generate a partial remainder and a bit value of partial root. The least significant bit (LSB) of the partial root is used for controlling the operation of adder/subtractors. If the LSB of partial root is 1, the adder/subtractor subtracts $Q01$ from R; if LSB of partial root is 0, the adder/subtractor adds $Q11$ to R. Therefore, the second least significant bit is just the inverse of the LSB of partial root (See Fig. 5).

Non-restoring in the previous algorithm means that the bit of the partial root is set one time, and then, based on the sign of $D - Q \times Q$, the partial root is adjusted by adding or subtracting a 1 in the corresponding position. However, non-restoring in our algorithm means that the remainder in each iteration is used for next iteration even if it is negative. Thus, the remainder is not restored by using the previous remainder as is done in the restoring algorithm via a multiplexor. In this sense, our non-restoring square root algorithm is very similar to the non-restoring division algorithm [1].

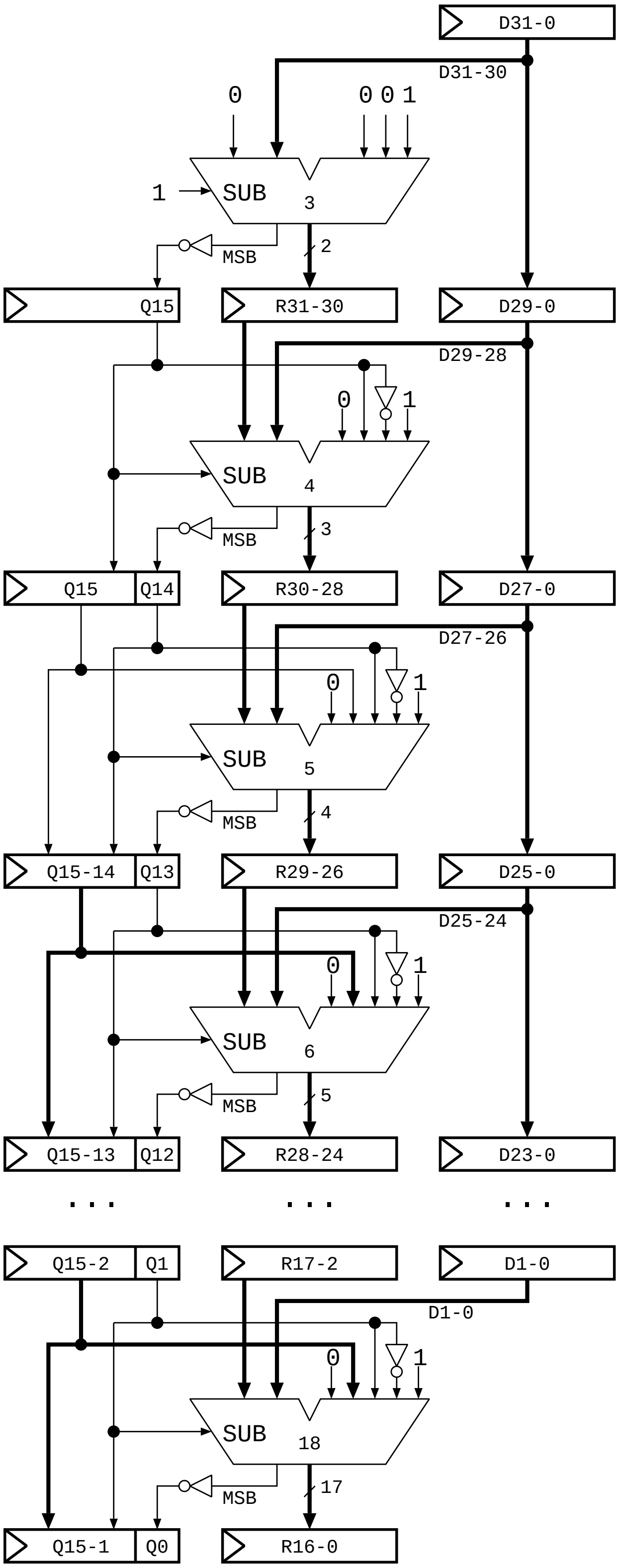

Figure 6. Non-restoring square root pipeline

Table 1. Operation of first stage

Input		Output		
D_{31}	D_{30}	Q_{15}	R_{31}	R_{30}
0	0	0	1	1
0	1	1	0	0
1	0	1	0	1
1	1	1	1	0

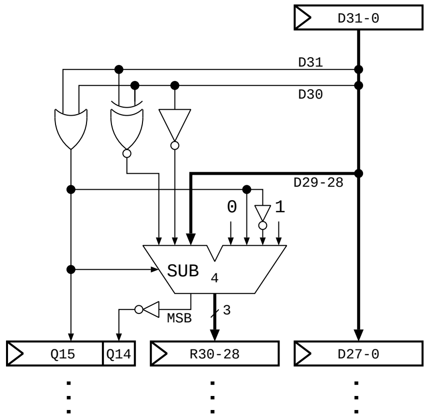

Figure 7. Simplification of the first two steps

4. VLSI Design of Fully Pipelined Version

The pipelined circuitry for a 32 bit unsigned number is shown in Fig. 6. There are 16 adder/subtractors. The numbers of bits for these adder/subtractors are 3, 4, 5, 6, ..., 18. The gate count required is reduced because no multiplexor is needed.

Furthermore, based on Table 1, we can merge the first two steps into one step (see Fig. 7) with the following expressions.

$$Q_{15} = D_{31} \vee D_{30},$$
$$R_{31} = \overline{D_{31} \oplus D_{30}},$$
$$R_{30} = \overline{D_{30}}.$$

Such fully pipelined circuitry can initiate a new *sqrt* instruction with each clock cycle.

5. VLSI Design of Iterative Version

An iterative low-cost version of the VLSI design for 32-bit operands is shown in Fig. 8. The 32-bit operand is placed in register D and it will be shifted two bits left in each iteration. Register Q holds the square root result and

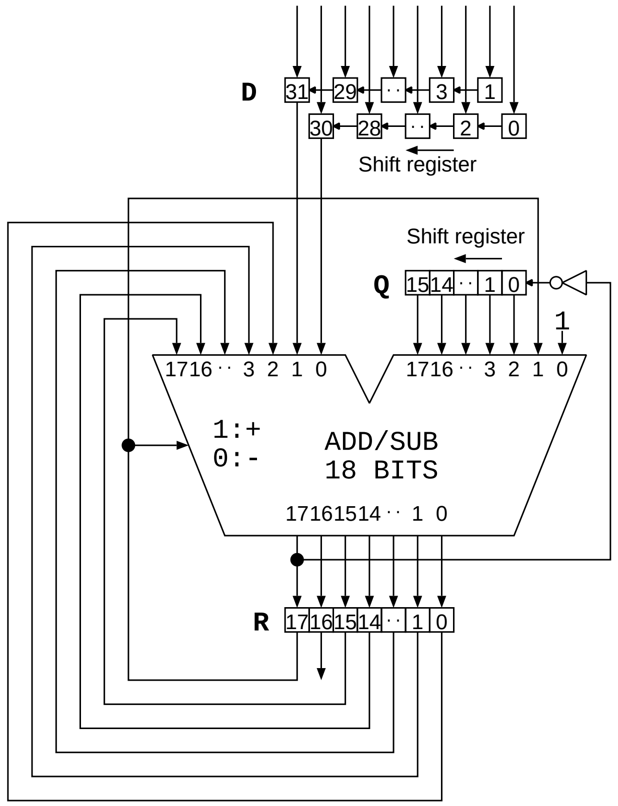

Figure 8. Iteration of non-restoring square root

it is initialized to zero at the beginning. It will be shifted one bit left in each iteration. Register R contains the partial remainder. It is cleared at the beginning in order to start the first iteration properly.

A single adder/subtractor is required. It subtracts if the control input is 0, otherwise it adds. One resulting bit is generated in each clock cycle, and a total of 16 cycles is needed to get the final square root. Such low-cost circuitry can initiate a new *sqrt* instruction every 16 clock cycles. Compared to [4], the simplicity of the circuitry for our new algorithm can be readily appreciated.

6. Performance and Cost Evaluation

In this section, we discuss the performance and cost of our algorithm compared with Newton iterative implementation. As mentioned in the introduction section, it is impractical to implement the fully pipelined version for Newton iteration (if it is not impractical, at least it is high cost), we only discuss on iterative implementations for both the algorithms.

Consider that a correct result of $Q = \sqrt{D}$ can be generated by Newton iteration two times for $D = D_{31}D_{31}...D_1D_0$ and $Q = Q_{15}Q_{14}...Q_1Q_0$ by Equation 16:

$$T_{i+1} = T_i \times (3 - T_i^2 \times D)/2, \qquad (16)$$

where T_i is an approximate value of $1/\sqrt{D}$. At first, a seed (T_0) is generated by a ROM table. After two-time iterations, the square root Q can be obtained by $T_2 \times D$ (see Fig. 9).

Calculation time is measured by the number of clock cycles. We assume the time of a clock cycle $t = t_{AS} + t_{MX}$ where the t_{AS} is the delay time of adder/subtractor and the t_{MX} is the delay time of the multiplexor. The cycle time of the circuitry for the new non-restoring algorithm is less than t because no multiplexor is used, i.e., $t = t_{AS}$. Multiplication in Newton iteration takes two cycles: the partial product is generated in the first cycle and the product is generated in the second cycle by the adder. We assume that partial product generation can be finished within the cycle time t.

Fig. 9 shows a possible implementation at the above assumed cycle time t. Table 2 shows the sequence of Newton iteration. This requires 17 cycles to get a square root, one cycle more than our algorithm.

The contrast between cycles and gate counts of the two algorithms is listed in Table 3. We find that the new algorithm requires fewer cycles when the bit width of operand is 16 and 32, and the Newton method requires fewer cycles when the bit width of the operand is 64. The obvious advantage of the new non-restoring algorithm is that only a traditional adder/subtractor with few shift registers can perform square root operations efficiently.

It is clear that multiple functional units can be easily fabricated in a single chip processor with current VLSI technology. However, if some functional units, such as the multiplier and adder/subtractor for instance, are shared by different types of instructions, there will be resource conflicts, and the processor will not be capable of exploiting the instruction level parallelism very well. For example, if a multiplier in a processor is shared by the *mul, div, sqrt* instructions, it is impossible to issue these instructions in parallel even if there is no data dependency. Therefore, it is important for multiple-issued processors to have dedicated functional units that can perform special operations independently of each other.

7. Conclusion

A new non-restoring square root algorithm was presented in this paper. The algorithm is highly suitable for VLSI designs and can exploit the advances in VLSI technology. Two VLSI implementations have illustrated the benefit of the algorithm: high speed was achieved at minimum cost because neither a multiplier nor a multiplexor is required.

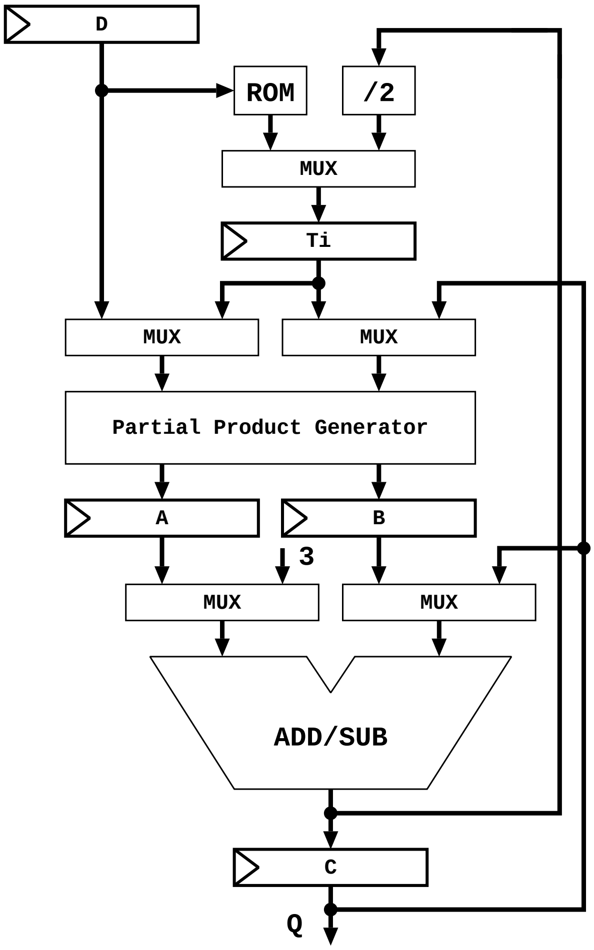

Figure 9. An implementation of Newton iteration

Table 2. Two-time Newton iterations for square root calculation

Cycle	Calculation	Operation
1	T_0	$T_0 = ROM[D]$
2	T_0^2	$A, B = Partial\ Prod.$
3	T_0^2	$C = A + B$ (Prod.)
4	$T_0^2 D$	$A, B = Partial\ Prod.$
5	$T_0^2 D$	$C = A + B$ (Prod.)
6	$3 - T_0^2 D$	$C = 3 - T_0^2 D$
7	$T_0(3 - T_0^2 D)$	$A, B = Partial\ Prod.$
8	$(T_0(3 - T_0^2 D))/2$	$T_1 = (A + B)/2$ (Prod./2)
9	T_1^2	$A, B = Partial\ Prod.$
10	T_1^2	$C = A + B$ (Prod.)
11	$T_1^2 D$	$A, B = Partial\ Prod.$
12	$T_1^2 D$	$C = A + B$ (Prod.)
13	$3 - T_1^2 D$	$C = 3 - T_1^2 D$
14	$T_1(3 - T_1^2 D)$	$A, B = Partial\ Prod.$
15	$(T_1(3 - T_1^2 D))/2$	$T_2 = (A + B)/2$ (Prod./2)
16	$T_2 D$	$A, B = Partial\ Prod.$
17	$T_2 D$	$C = A + B$ (Prod.)

Table 3. Required time and space contrast of two implementations

	No. of Cycles			No. of Gates		
Data Bits	16	32	64	16	32	64
Newton	10	17	24	2,300	5,800	18,000
Non-res.	8	16	32	550	1,100	2,200

References

[1] J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[2] G. Knittel, “ A VLSI-Design for Fast Vector Normalization” *Comput. & Graphics*, Vol. 19, No. 2, 1995. pp261 - 271.

[3] J. Bannur and A. Varma, “The VLSI Implementation of A Square Root Algorithm”, *Proc. IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, Washington D.C., 1985. pp159 - 165.

[4] J. O’Leary, M. Leeser, J. Hickey, M. Aagaard, “Non-Restoring Integer Square Root: A Case Study in Design by Principled Optimization”, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, 1994. pp52 - 71.

[5] K. C. Johnson, “Efficient Square Root Implementation on the 68000”, *ACM Transaction on Mathematical Software*, Vol. 13, No. 2, 1987. pp138 - 151.

[6] H. Kabuo, T. Taniguchi, A. Miyoshi, H. Yamashita, M. Urano, H. Edamatsu, S. Kuninobu, “Accurate Rounding Scheme for the Newton-Raphson Method Using Redundant Binary Representation”, *IEEE Transaction on Computers*, Vol. 43, No. 1, 1994. pp43 - 51.

[7] Y. Li and W. Chu, “Design and Performance Analysis of A Multiple-threaded Multiple-pipelined Architecture,” *Proc. of the Second International Conference on High Performance Computing*, New Delhi, India, December 1995. pp483 - 490.