Parallel-Array Implementations of A Non-Restoring Square Root Algorithm

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 parallel-array implementation of a new non-restoring square root algorithm (PASQRT). The carry-save adder (CSA) is used in the parallel array. The PASQRT has several features unlike other implementations. First, it does not use redundant representation for square root result. Second, each iteration generates an exact resulting value. Next, it does not require any conversion on the inputs of the CSA. And last, a precise remainder can be obtained immediately. Furthermore, we present an improved version — a root-select parallel-array implementation (RS-PASQRT) for fast result value generation. The RS-PASQRT is capable of achieving up to about 150% speedup ratio over the PASQRT. The simplicity of the implementations indicates that the proposed approach is an alternative to consider when designing a fully pipelined square root unit.

1. Introduction

The square root algorithms and implementations have been addressed mainly in three methods: *Newton-Raphson* method [3] [5] [12] [15], *SRT-Redundant* method [2] [7] [8] [11] [14], and *Non-Redundant* method [1] [4] [6] [9] [13].

The Newton-Raphson method has been adopted in many implementations. In order to calculate $Y = \sqrt{X}$, an approximate value is calculated by iterations. For example, we can use the Newton-Raphson method on $f(T) = 1/T^2 - X$. The zero of this function is at $T = 1/\sqrt{X}$. Applying Newton iteration to it will give an iterative method of computing $1/\sqrt{D}$ from D.

$$T_{i+1} = T_i - \frac{f(T_i)}{f'(T_i)} = T_i(3 - T_i^2 X)/2$$

where T_i is an approximate value of $1/\sqrt{X}$. After n-time iterations, an approximate square root can be obtained by equation $Y = \sqrt{X} \simeq T_n X$.

The algorithm needs a seed generator for generating T_0, 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, Wallace tree for example, to get a partial production and then use a carry propagate adder (CPA) 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 square root instructions. If it is not impractical, at least it is at high cost. And also, it will be a hard task to get an exact square root remainder.

The classical radix-2 SRT-Redundant method is based on the recursive relationship

$$\begin{aligned} X_{i+1} &= 2X_i - 2Y_i y_{i+1} - y_{i+1}^2 2^{-(i+1)} \\ Y_{i+1} &= Y_i + y_{i+1} 2^{-(i+1)} \end{aligned}$$

where X_i is ith partial remainder (X_0 is radicand), Y_i is ith partially developed square root with $Y_0 = 0$, y_i is ith square root bit, and $y_i \epsilon \{-1, 0, 1\}$. The y_{i+1} is obtained by applying the digit-selection function $y_{i+1} = Select(\tilde{X}_i)$, or for high-radix SRT-Redundant methods, $y_{i+1} = Select(\tilde{X}_i, \tilde{Y}_i)$, where $\tilde{X}_i$ and $\tilde{Y}_i$ are estimates obtained by truncating redundant representations of X_i and Y_i, respectively. In each iteration, there are four subcomputations: (1) One digit shift-left of X_i to produce $2X_i$, (2) Determination of y_{i+1}, (3) Formation of $F = -2Y_i y_{i+1} - y_{i+1}^2 2^{-(i+1)}$, and (4) Addition of F and $2X_i$ to produce X_{i+1}. The following procedure shows the derivation of X_{i+1}.

$$\begin{aligned} X_{i+1} &= (X_0 - Y_{i+1}^2)2^{i+1} \\ &= (X_0 - Y_i^2 - 2Y_i y_{i+1} 2^{-(i+1)} - y_{i+1}^2 2^{-2(i+1)})2^{i+1} \\ &= (X_0 - Y_i^2)2^{i+1} - 2Y_i y_{i+1} - y_{i+1}^2 2^{-(i+1)} \\ &= 2X_i - 2Y_i y_{i+1} - y_{i+1}^2 2^{-(i+1)} \end{aligned}$$

A CSA can be used to speedup the addition of F and $2X_i$. But, the F needs to be converted to the two's complement representation in order to be fed to the CSA. Moreover, for the determination of y_{i+1}, the selection function is rather complex, especially for high-radix SRT algorithms, although it depends only on the low-precision estimates of X_i and Y_i. At the final step, a CPA is needed to convert the square root from the redundant representation to the two's complement representation. Since the complicity of the circuitry, some of the implementations use an iterative version, that is, all the iterations share same hardware resources. Consequently, the implementations are not capable of accepting a new square root instruction on every clock cycle (refer to the throughput listed in Tab. 2).

The Non-Redundant method is similar to the SRT method but it uses the two's complement representation for square root. The classical Non-Redundant method is based on the computations $R_{i+1} = X - Y_i^2$ and $Y_{i+1} = Y_i + y_{i+1}2^{-(i+1)}$ where R_i is ith partial remainder, Y_i is ith partial square root with $Y_1 = 0.1$, and y_i is ith square root bit with $y_1 = 1$. The resulting value is selected by checking the sign of the remainder. If $R_{i+1} \geq 0$, $y_{i+1} = 1$; otherwise $y_{i+1} = -1$. The computation of R_{i+1} can be simplified by eliminating the square operation by variable substitution:

$$\begin{aligned}
X_{i+1} &= (X - Y_i^2)2^i \\
&= (X - (Y_{i-1}^2 + 2Y_{i-1}y_i2^{-i} + y_i^2 2^{-2i}))2^i \\
&= (X - Y_{i-1}^2)2^i - 2Y_{i-1}y_i - y_i^2 2^{-i} \\
&= 2X_i - 2(Y_i - y_i2^{-i})y_i - y_i^2 2^{-i} \\
&= 2X_i - 2Y_iy_i + y_i^2 2^{-i}
\end{aligned}$$

The new iteration equations become

$$\begin{aligned}
X_{i+1} &= 2X_i - 2Y_iy_i + y_i^2 2^{-i} \\
Y_{i+1} &= Y_i + y_{i+1}2^{-(i+1)}
\end{aligned}$$

where X_i is ith partial remainder (X_1 is radicand). Different from the SRT methods, the resulting value selection is done *after* the X_{i+1}'s calculation, while the SRT methods do it *before* the X_{i+1}'s calculation.

It may also generate a wrong resulting value at the last bit position, and requires to convert such a $F = -2Y_iy_i + y_i^2 2^{-i}$ to get one operand that will be added to $2X_i$. Some Non-Redundant algorithms were said to belong to "*restoring*" or "*non-restoring*". For example, the one described above is said to be a non-restoring square root algorithm. But in fact, the word of restoring (non-restoring) means the restoring (non-restoring) on *square root*, but not *remainder*.

The proposed approach in this paper also uses the two's complement representation for the square root result. It is a non-restoring algorithm that does not restore the remainder. At each iteration the algorithm generates an exact resulting value, even in the last bit position. The algorithm does not require the F conversion and the calculation of $Y_i - 2^{-(i+1)}$ that appear in the SRT-Redundant and other Non-Redundant methods. An exact remainder can be obtained immediately without any correction if it is non-negative or with an addition operation if it is negative. In order to speedup the computation, a parallel array that uses the CSA is proposed. The determination of a square root value is based on the partial remainder that is the output of the CSA. An exact resulting value is determined by using the carry-lookahead on the sign bit of the partial remainder. The implementation is very simple and easy to understand. Furthermore, an improved version, a root-select parallel-array implementation (RS-PASQRT), is also addressed.

The paper is organized as follows. Section 2 describes the new non-restoring square root algorithm. Section 3 and 4 present two parallel-array implementations of the algorithm. The final section presents conclusions.

2. New Non-Restoring Square Root Algorithm

Assume that the radicand D is denoted by a 32-bit unsigned number. For every pair of bits of the radicand, the integer part of square root has one bit. Thus the integer part of square root Q for a 32-bit radicand D has 16 bits: $Q = Q_1Q_2...Q_{15}Q_{16}$. The remainder is defined $R = D - Q^2$. Because $D = (Q^2 + R) < (Q+1)^2$, we get $R < 2Q + 1$, i.e., $R \leq 2Q$ because the remainder R is an integer. This means that the remainder has at most one binary bit more than the square root.

2.1. Restoring Square Root Algorithm

First, we describe a restoring algorithm. Let us define $r_0 = D \times 2^{-32}$, partial square root $q_i = Q_1Q_2...Q_i$ with $q_0 = 0$. To determine the square root bit $Q_{i+1}, (i = 0, 1, 2, ..., 15)$, a tentative remainder

$$4r_i - (4q_i + 1)$$

is calculated where r_i is the partial remainder obtained at iteration i. If this tentative remainder is non-negative, then

$$\begin{aligned}
Q_{i+1} &= 1, \\
q_{i+1} &= 2q_i + 1, \\
r_{i+1} &= 4r_i - (4q_i + 1).
\end{aligned}$$

Otherwise,

$$\begin{aligned}
Q_{i+1} &= 0, \\
q_{i+1} &= 2q_i, \\
r_{i+1} &= 4r_i.
\end{aligned}$$

The meaning of *restoring* is that when the tentative remainder is negative, we restore the partial remainder by adding $(4q_i + 1)$ back to the tentative remainder or selecting the old partial remainder $4r_i$. The reason of why the algorithm works is explained as below. From the definitions of the r_i and q_i, we have

$$\begin{aligned}
r_i &= r_0 \times 2^{2i} - q_i^2, \\
q_i &= 2q_{i-1} + Q_i.
\end{aligned}$$

For example, $r_1 = 4r_0 - q_1^2$ and $R = r_{16} = r_0 \times 2^{32} - q_{16}^2 = D - Q^2$. The square calculation of q_i^2 can be eliminated by variable substitution:

$$\begin{aligned}
r_{i+1} &= r_0 \times 2^{2(i+1)} - q_{i+1}^2 \\
&= r_0 \times 2^{2(i+1)} - (2q_i + Q_{i+1})^2 \\
&= 4r_0 \times 2^{2i} - (4q_i^2 + 4q_iQ_{i+1} + Q_{i+1}^2) \\
&= 4r_i - (4q_iQ_{i+1} + Q_{i+1}^2)
\end{aligned}$$

We set $Q_{i+1} = 1$, then $r_{i+1} = 4r_i - (4q_i + 1)$. If the result is negative, setting Q_{i+1} made q_{i+1} too big, so we reset $Q_{i+1} = 0$ and restore the partial remainder by adding $(4q_i + 1)$ to the result or simply selecting $4r_i$.

2.2. Non-Restoring Square Root Algorithm

The non-restoring algorithm that does not restore partial remainder when it is negative is described as below.

```
$r_0 = D \times 2^{-32}, q_0 = 0,\ for\ i = 0\ to\ 15\ do$
    If $r_i \geq 0$          $r_{i+1} = 4r_i - (4q_i + 1)$
    else                     $r_{i+1} = 4r_i + (4q_i + 3)$
    If $r_{i+1} \geq 0$      $q_{i+1} = 2q_i + 1$
    else                     $q_{i+1} = 2q_i$
If $r_{16} < 0$              $r_{16} = r_{16} + (2q_{16} + 1)$
```

The final square root $Q = q_{16}$ and the final remainder $R = r_{16}$. The algorithm performs the same operation as the one of the restoring algorithm when the partial remainder r_i is non-negative. But for the negative partial remainder, we can restore it by adding $(4q_{i-1} + 1)$ to r_i. Notice that in this case, $q_i = 2q_{i-1}$. Thus,

$$\begin{aligned} r_{i+1} &= 4(4r_{i-1}) - (4q_i + 1) \\ &= 4(r_i + (4q_{i-1} + 1)) - (4q_i + 1) \\ &= 4(r_i + (2q_i + 1)) - (4q_i + 1) \\ &= 4r_i + (4q_i + 3). \end{aligned}$$

If r_{16} is non-negative, it becomes the final remainder. If it is negative, we restore it by adding $(4q_{15} + 1)$ or $(2q_{16} + 1)$ to r_{16}. The key point of this non-restoring algorithm is that when the partial remainder r_i is negative, the algorithm does not restore the previous partial remainder. Instead, it continues the calculation with $r_{i+1} = 4r_i + (4q_i + 3)$.

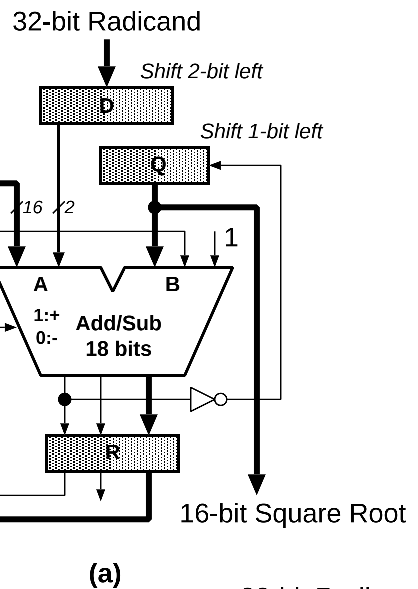
The $4r_i$ means to shift r_i 2-bit left; while the $4q_i + 1$ (or $4q_i + 3$) means to shift q_i 2-bit left and set the least 2-bit to 01 (or 11). Let us see an example in which $D = 01111111$ is an 8-bit radicand, hence Q will be a 4-bit square root. The calculation is illustrated below. We get $Q = 1011$ and $R = 00110$.

Set	$r_0 = 0.01111111$
	$q_0 = 0$
$r_0 \geq 0$	$r_1 = 001.111111 - 001 = 000.111111$
$r_1 \geq 0$	$q_1 = 1$
	$r_2 = 0011.1111 - 0101 = 1110.1111$
$r_2 < 0$	$q_2 = 10$
	$r_3 = 11011.11 + 01011 = 00110.11$
$r_3 \geq 0$	$q_3 = 101$
	$r_4 = 011011 - 010101 = 000110$
$r_4 \geq 0$	$q_4 = 1011$

Notice that q_i has i bits, so r_i has $i + 1$ bits. It is needed to calculate r_i with $i + 2$ bits width in order to check the sign of r_i. In each iteration, the algorithm requires only an addition or subtraction and generates a correct resulting value that does not need to be adjusted.

2.3. Low-Cost Implementation of the Algorithm

Two iterative low-cost versions of the circuit design for a 32-bit radicand are shown in Fig. 1. The 32-bit radicand is placed in register D. It will be shifted two bits left in each iteration. Register Q holds the square root result. It will be shifted one bit left in each iteration. Register R ($R2$ and $R0$ in Fig. 1(b)) contains the partial remainder. Registers Q and R are cleared at the beginning.



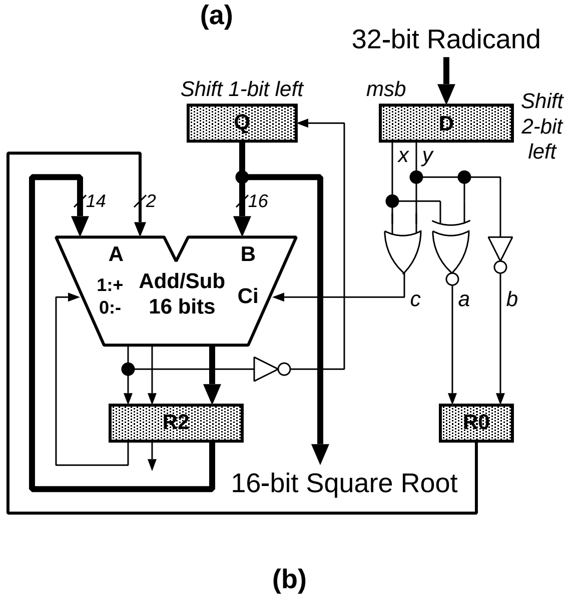


Figure 1. Low-cost integer square rooting

The operations of $4r_i$ and $4q_i$ are performed by suitable wiring. A single adder/subtractor is required. It subtracts if the control input is 0, otherwise it adds. In each clock cycle, one resulting value is generated by checking the sign of the partial remainder. The final remainder has 17 bits. In order to check its sign, an 18-bit adder/subtractor is needed (Fig. 1(a)).

Table 1. Change 18-bit adder to 16-bit adder

	x	*y*	*c*	*a*	*b*	*c active*
+11	*0*	*0*	*0*	*1*	*1*	*high*
	0	*1*	*1*	*0*	*0*	*high*
	1	*0*	*1*	*0*	*1*	*high*
	1	*1*	*1*	*1*	*0*	*high*
−01	*0*	*0*	*0*	*1*	*1*	*low*
	0	*1*	*1*	*0*	*0*	*low*
	1	*0*	*1*	*0*	*1*	*low*
	1	*1*	*1*	*1*	*0*	*low*

We can simplify the least 2-bit operations of -01 and $+11$ just by using three gates based on Tab. 1 (the *x*, *y*, *a*, *b*, and *c* in the table are marked in Fig. 1(b)). It is right because, in a conventional adder/subtractor, the carry is high-active for addition and low-active for subtraction. With this simplification, the calculations can be done by a 16-bit adder/subtractor. Notice that for both the implementations, the circuit for the final remainder adjustment was not shown in the figure.

A total of 16 cycles is needed to get the final square root. A similar circuit can be used in a low-cost floating point square root design [10]. Such low-cost integer and floating point implementations may be better if there are few square root operation in an application software.

3. Parallel-Array Implementation (PASQRT)

For some ASIC designs, a more efficient square root unit would be very useful. In this section, we present a parallel-array implementation of the non-restoring square root algorithm.

Except for the first-time iteration, the non-restoring algorithm can be presented as below.

$$\begin{array}{lrcl} \text{If } Q_i = 1 & r_{i+1} & = & 4r_i - (4q_i + 1) \\ \text{else} & r_{i+1} & = & 4r_i + (4q_i + 3) \end{array}$$

The first-time iteration always subtracts 1 from $4r_0$. If $Q_i = 1$, then $q_i = 2q_{i-1} + 1$. If $Q_i = 0$, then $q_i = 2q_{i-1}$. Now, the algorithm turns to

$$\begin{array}{lrcl} \text{If } Q_i = 1 & r_{i+1} & = & 4r_i - (8q_{i-1} + 5) \\ \text{else} & r_{i+1} & = & 4r_i + (8q_{i-1} + 3) \end{array}$$

Because, for any binary numbers u and v, $u - v = u + (-v) = u + \overline{v} + 1$, we can replace $4r_i - (8q_{i-1} + 5)$ with $4r_i + (\overline{8q_{i-1}} + 3)$. We get a new presentation of the algorithm as below.

```
1. $r_0 = D \times 2^{-32}$, $q_0 = 0$
2. Always          $r_1 = 4r_0 + (-1)$
   If $r_1 \geq 0$        $q_1 = Q_1 = 1$
   else            $q_1 = Q_1 = 0$
3. $for\ i = 1\ to\ 15\ do$
   If $Q_i = 1$      $r_{i+1} = 4r_i + (\overline{8q_{i-1}} + 3)$
   else            $r_{i+1} = 4r_i + (8q_{i-1} + 3)$
   If $r_{i+1} \geq 0$    $q_{i+1} = 2q_i + 1$
   else            $q_{i+1} = 2q_i$
4. If $r_{16} < 0$      $r_{16} = r_{16} + (2q_{16} + 1)$
```

The Fig. 2 illustrates the square root calculations by parallel CSA (carry-save adder) array. We define the radicand $D = D_1 D_2 ... D_{30} D_{31}$. The ith partial remainder r_i is presented by two groups of data, carry bits (B_j^i) and sum bits (A_j^i). The Q_j^i means $Q_j \oplus Q_i$ that implements $8q_i$ or $\overline{8q_i}$: if $Q_i = 1$, $Q_j^i = \overline{Q_j}$, else $Q_j^i = Q_j$. Because the 011 is always added to the lowest three bits of partial remainder, the B_j^i for $j = i-1,\ i,\ i+1,\ i+2$ and A_j^i for $j = i,\ i+1,\ i+2$ can be simplified as shown in the figure.

$$\begin{array}{cccccccccccccc}
 & 0 & D_1 & D_2 \\
+ & 1 & 1 & 1 \\ \cline{1-4}
 & D_1 & D_2 & 0 \\
 & 1 & \overline{D_1} & \overline{D_2} & D_3 & D_4 \\
 & + & Q_0^1 & 0 & 1 & 1 \\ \cline{2-6}
 & & 0 & \overline{D_3} & \overline{D_4} & 0 \\
 & & A_1^2 & \overline{D_2} & \overline{D_3} & \overline{D_4} & D_5 & D_6 \\
 & & + & Q_0^2 & Q_1^2 & 0 & 1 & 1 \\ \cline{3-8}
 & & & B_1^3 & 0 & \overline{D_5} & \overline{D_6} & 0 \\
 & & & A_1^3 & A_2^3 & \overline{D_4} & \overline{D_5} & \overline{D_6} & D_7 & D_8 \\
 & & & + & Q_0^3 & Q_1^3 & Q_2^3 & 0 & 1 & 1 \\ \cline{4-10}
 & & & & B_1^4 & B_2^4 & 0 & \overline{D_7} & \overline{D_8} & 0 \\
 & & & & A_1^4 & A_2^4 & A_3^4 & \overline{D_6} & \overline{D_7} & \overline{D_8} & D_9 & D_{10} \\
 & & & & + & Q_0^4 & Q_1^4 & Q_2^4 & Q_3^4 & 0 & 1 & 1 \\ \cline{5-12}
 & & & & & B_1^5 & B_2^5 & B_3^5 & 0 & \overline{D_9} & \overline{D_{10}} & 0 \\
 & & & & & A_1^5 & A_2^5 & A_3^5 & A_4^5 & \overline{D_8} & \overline{D_9} & \overline{D_{10}} & D_{11} & D_{12} \\
 & & & & & + & Q_0^5 & Q_1^5 & Q_2^5 & Q_3^5 & Q_4^5 & 0 & 1 & 1 \\ \cline{6-14}
 & & & & & & B_1^6 & B_2^6 & B_3^6 & B_4^6 & 0 & \overline{D_{11}} & \overline{D_{12}} & 0 \\
 & & & & & & A_1^6 & A_2^6 & A_3^6 & A_4^6 & A_5^6 & \overline{D_{10}} & \overline{D_{11}} & \overline{D_{12}} \\
 & & & & & & \ldots
\end{array}$$

Figure 2. Square rooting by CSA parallel array

The $q_i = Q_1 Q_2 ... Q_i$ has i bits. Therefore the r_i has $i+1$ bits. In order to check the sign of r_i, it is needed to calculate r_i with only $i+2$ bits. Here we can use the *carry-lookahead* circuit to determine Q_i.

$$\begin{array}{rcl}
Q_i & = & \overline{A_1^i \oplus B_1^i \oplus C_2^i} \\
C_2^i & = & G_2^i + P_2^i G_3^i + ... + P_2^i P_3^i ... P_{i-3}^i G_{i-2}^i + 0 + \\
 & & + P_2^i P_3^i ... P_{i-2} A_{i-1}^i \overline{D_{2i-2}} (D_{2i-1} + D_{2i}) \\
G_j^i & = & A_j^i B_j^i \\
P_j^i & = & A_j^i + B_j^i
\end{array}$$

The hardware implementation is shown as in Fig. 3. We call it *parallel array for square rooting* (PASQRT). The input of the circuit is the radicand D and the output is the square root Q. Each small block denotes a CSA, its structure is shown in the up-right corner in the figure. The outputs of the CSA are named with A_j^i (sum bit) and B_{j-1}^i (carry bit). The block Q is used for generating square root bit, for example, $Q_1 = D_1 + D_2$. The $+3$ does not require to use CSA, it is simplified in the figure based on Fig 2.

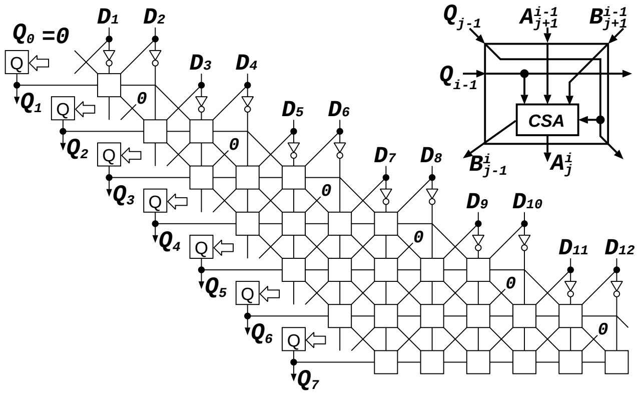

$Q_0 = 0$
D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 D_{10} D_{11} D_{12}
Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7
Q
0
Q_{j-1} A_{j+1}^{i-1} B_{j+1}^{i-1}
Q_{i-1}
CSA
B_{j-1}^i A_j^i

Figure 3. PASQRT

The circuit for generating a bit of resulting value is simpler than a carry-lookahead adder (CLA) because the CLA needs to generate all of the carry bits for fast addition, but here, it requires to generate only a single carry bit. We can use a special technology [16] to speed the generation of the C_2^i. It was developed by Rowen, Johnson, and Ries and used in MIPS R3010 floating point coprocessor for divider's quotient logic, fraction zero-detector, and others. By using this technology, the Q_i can be obtained with four-level gates, i.e., two times compared to CSA (the CSA is implemented with two-level gates).

4. Root-Select Parallel-Array Implementation (RS-PASQRT)

In the ith iteration, the computation of r_i depends on Q_{i-1}. There are two-case computations. If $Q_{i-1} = 1$, then $r_i = 4r_{i-1} + (\overline{8q_{i-2}} + 3)$; otherwise, $r_i = 4r_{i-1} + (8q_{i-2} + 3)$. The Q_{i-1} is derived from r_{i-1}. This is illustrated in Fig. 4(a).

Actually, the two-case computations of r_i can be started in parallel immediately after the r_{i-1} is known. Consequently, the determination of Q_i in the two cases can be started early. The correct case can be selected by using multiplexors after the Q_{i-1} is known. This is illustrated in Fig. 4(b). We call it *root-select parallel-array for square rooting* (RS-PASQRT). The space (or chip area) required by RS-PASQRT will be about twice compared to PASQRT. Notice that for the CSA, only the carry out generation needs

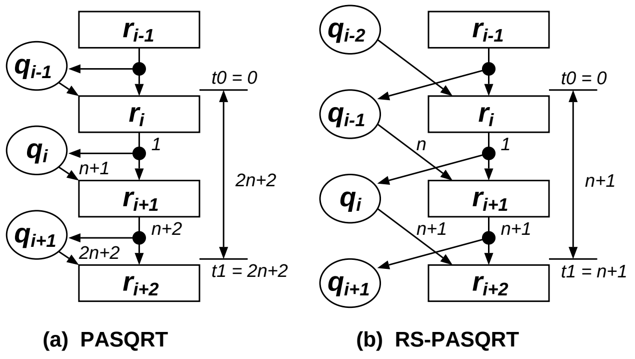


Figure 4. RS-PASQRT

to be duplicated, the sum generation ($s = a \oplus b \oplus c$) does not need to be duplicated because $a \oplus \overline{b} \oplus c = \overline{s}$. This will save CSA more than 50% space.

The question is what the performance improvement is. Let us define the time required by a carry-save adder as a unit time (t_{FA}). Assume that the determination of a square root bit from the partial remainder takes n units. Referring to Fig. 4, in the PASQRT, one iteration takes $n + 1$ units, while in the RS-PASQRT, two iterations take $n + 1$ units, therefore the speedup is two. The speedup estimation does not consider the time required by the multiplexors. If we use m to denote the multiplexor's time units, then

$$Speedup = \frac{2n+2}{n+1+2m}.$$

Typically, $m = 0.5$ [8] and $n = 2$ as described in the previous section. The speedup is 150%. Tab. 2 lists the comparison of the times required by proposed approaches and others when performing double-precision floating point square root. The data for others are quoted from [7].

It can be found that the proposed simple implementations have about same level performance compared to other complex implementations, which can be readily appreciated. Additionally, the implementations are very easy to be fully pipelined with the throughput of one clock cycle, while other implementations use iteration method that results in the throughput of 7 to 21 clock cycles.

As for the area cost, the high-radix SRT and Newton-Raphson implementations require some multipliers and lookup tables that take a rather large number of gate counts. The proposed implementations need neither multipliers nor tables. Referring to Fig. 3, the basic hardware requirements for the PASQRT are $\sum_{i=1}^{53}(i-1) = 26 \times 53$ adders and corresponding square root result bit generators, while in other implementations a multiplier will require about 53×53 adders. Furthermore, the number of adders required by the proposed implementations can be reduced because the low 53-bit input is zero (to generate 53-bit square root result requires 106-bit radicand).

Table 2. Execution time comparison

	Double-precision							Single-precision	
	Lang's radix-256	Fandrianto's radix-8	IBM-RS6000 Newton-Rap.	WEITEK W4164/4363	Matula's radix-2^{16}	PASQRT Non-redun.	RS-PASQRT Non-redun.	PASQRT Non-redun.	RS-PASQRT Non-redun.
Latency(t_{FA})	126	190	204	136	216	159	106	72	48
Throughput(cycles)	21	16	12	8	7	1	1	1	1

5. Conclusion Remarks

Two parallel-array implementations based on a new non-restoring square root algorithm were presented in this paper. The implementations use two's complement representation for square root bit. An exact resulting value can be generated in each iteration. A rough estimation indicates that the proposed simple approach can achieve an equivalent speed to other implementations. Better than others, the proposed approach can be easily pipelined with a throughput of one clock cycle. The modern multi-issued processors require multiple dedicated, fully-pipelined functional units to exploit instruction level parallelism, hence the simplicity of the functional units becomes an important issue. The proposed implementations are shown to be suitable for designing a fully pipelined dedicated square root unit.

References

[1] J. Bannur and A. Varma, "The VLSI Implementation of A Square Root Algorithm", *Proc. IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, 1985. pp159-165.

[2] M. Birman, A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes, "Developing the WTL3170/3171 Sparc Floating-Point Coprocessors", *IEEE MICRO* February, 1990. pp55-64.

[3] J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., 1996. Appendix A: Computer Arithmetic by D. Goldberg.

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

[5] 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.

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

[7] T. Lang and P. Montuschi, "High Radix Square Root with Prescaling", *IEEE Transaction on Computers*, Vol. 41, No. 8, 1992. pp996-1009.

[8] T. Lang and P. Montuschi, "Very-high Radix Combined Division and Square Root with Prescaling and Selection by Rounding", *Proc. 12th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, 1995. pp124-131.

[9] Y. Li and W. Chu, "A New Non-Restoring Square Root Algorithm and Its VLSI Implementations," *Proc. of 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Austin, Texas, USA, October 1996. pp538-544.

[10] Y. Li and W. Chu, "Implementation of Single Precision Floating Point Square Root on FPGAs," *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, California, USA, April 1997.

[11] S. Majerski, "Square-Rooting Algorithms for High-Speed Digital Circuits", *IEEE Transaction on Computers*, Vol. 34, No. 8, 1985. pp724-733.

[12] P. Markstein, "Computation of Elementary Functions on the IBM RISC RS6000 Processor" *IBM Jour. of Res. and Dev.*, January, 1990. pp111-119.

[13] 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.

[14] J. Prabhu and G. Zyner, "167 MHz Radix-8 Divide and Square Root Using Overlapped Radix-2 Stages", *Proc. 12th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, 1995. pp155-162.

[15] C. Ramamoorthy, J. Goodman, and K. Kim, "Some properties of iterative Square-Rooting Methods Using High-Speed Multiplication", *IEEE Transaction on Computers*, Vol. C-21, No. 8, 1972. pp837-847.

[16] C. Rowen, M. Johnson, and P. Ries, "The MIPS R3010 Floating-Point Coprocessor", *IEEE MICRO*, June, 1988. pp53-62.