Australian Computer Science Communications, Vol.22, No.4, 2000, pp.9–16, IEEE Comp. Society Press

Cost/Performance Tradeoff of *n*-Select Square Root Implementations

Wanming Chu and Yamin Li

Computer Architecture Laboratory
The University of Aizu
Aizu-Wakamatsu 965-8580 Japan
w-chu@u-aizu.ac.jp, yamin@u-aizu.ac.jp

Abstract

Hardware square-root units require large numbers of gates even for iterative implementations. In this paper, we present four low-cost high-performance fully-pipelined n-select implementations (nS-Root) based on a non-restoring-remainder square root algorithm. The nS-Root uses a parallel array of carry-save adders (CSAs). For a square root bit calculation, a CSA is used once. This means that the calculations can be fully pipelined. It also uses the n-way root-select technique to speedup the square root calculation. The cost/performance evaluation shows that n=2 or n=2.5 is a suitable solution for designing a high-speed fully pipelined square root unit while keeping the low-cost.

1 Introduction

Square root is an important operation in scientific computations and multi-media applications. It calculates $Y = \sqrt{X}$, where X is the radicand and Y is the root. Although many existing designs of square root unit adopted iterative version, those designs still used a large number of gates. When we expand the designs to pipeline version, the cost will be an order of magnitude higher than that of iterative ones.

We present four pipelined square root implementations (nS-Root) based on a non-restoring-remainder square root algorithm. Our work shows that the high-performance fully pipelined square root unit can be implemented at low-cost.

The nS-Root uses a parallel array of the carry-save adders (CSAs) to calculate all the partial remainders in a pipeline manner. The n denotes the parallelism of the square root calculations. Increasing n will improve the performance but the area cost also increase. Our cost/performance evaluation shows that n=2 or 2.5 is a suitable choice.

We first review the previous square root algorithms. Then we introduce the non-restoring-remainder square root algorithm. We next describe the nS-Root implementations and their cost/performance evaluation with comparison to other implementations.

2 Brief Review of Previous Square Root Algorithms

The square root algorithms and implementations have been addressed mainly in three methods: *Newton-Raphson*, *SRT-Redundant*, and *Non-Redundant* methods.

2.1 Newton-Raphson method

The Newton-Raphson method has been adopted in many implementations [5] [7] [14] [17]. 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 will be at high cost to place as many multipliers as required in addition to the pipeline registers to realize fully pipelined operation for square root instructions.

In most real designs, a multiplier is shared by the operations of multiplication, divison, and square root. This means that the instruction level parallelism of those operations cannot be exploited in such designs. And also, it will be a little hard task to get an exact square root remainder.

2.2 SRT-Redundant method

The classical radix-2 SRT-Redundant method [2] [4] [9] [10] [13] [16] 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, this can be done using an on-the-fly converter. 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.

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. A systolic array implementation is described in [3].

2.3 Non-Redundant method

The Non-Redundant method [1] [6] [8] [15] 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:

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

The new iteration equations become

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

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 $F = -2Y_i y_i + y_i^2 2^{-i}$ 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*.

3 Non-Restoring-Remainder Square Root Algorithm

Assume that the radicand $D = D_1 D_2 ... D_{31} D_{32}$ 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: $Q = Q_1 Q_2 ... Q_{15} Q_{16}$. The remainder is defined $R = D - Q^2 = R_1 R_2 ... R_{16} R_{17}$. 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.

3.1 Restoring Square Root Algorithm

First, we describe a restoring-remainder algorithm. Let us define $r_0 = D \times 2^{-32}$, partial square root $q_i = Q_1 Q_2 ... Q_i$ with $q_0 = 0$. To determine the square root bit $Q_{i+1}, (i = 0, 1, 2, ..., 15)$, a tentative remainder $r_{i+1}^* =$

$4r_i - (4q_i + 1)$ is calculated where r_i is the partial remainder obtained at iteration i.

$$\begin{array}{lll} \text{If } r^*_{i+1} \geq 0, & Q_{i+1} = 1, q_{i+1} = 2q_i + 1; \\ & r_{i+1} = r^*_{i+1} = 4r_i - (4q_i + 1); \\ \text{else} & Q_{i+1} = 0, q_{i+1} = 2q_i + 0; \\ & r_{i+1} = r^*_{i+1} + (4q_i + 1) = 4r_i; \end{array}$$

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 why the algorithm works is explained as below. From the definitions of the r_i and q_i, we have $r_i = r_0 \times 2^{2i} - q_i^2$, $q_i = 2q_{i-1} + Q_i$. 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:

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

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 $4r_i$.

3.2 Non-Restoring Square Root Algorithm

The non-restoring-remainder algorithm [11] that does not restore remainder when it is negative is described as below.

$$\begin{array}{lll} \multicolumn{3}{l}{r_0 = D \times 2^{-32}, \ q_0 = 0;} \\ \multicolumn{3}{l}{\text{for } i = 0 \text{ to } 15 \text{ do}} \\ & \text{If } r_i \geq 0, & r_{i+1} = 4r_i - (4q_i + 1); \\ & \text{else} & r_{i+1} = 4r_i + (4q_i + 3); \\ & \text{If } r_{i+1} \geq 0, & q_{i+1} = 2q_i + 1; \\ & \text{else} & q_{i+1} = 2q_i + 0; \\ \multicolumn{3}{l}{\text{If } r_{16} < 0, \ r_{16} = r_{16} + (2q_{16} + 1);} \end{array}$$

The final square root $Q = q_{16}$ and the final remainder $R = r_{16}$. This algorithm performs the same operation as the one of the restoring algorithm when the partial remainder r_i is non-negative. For the negative partial remainder, we should restore it by adding $(4q_{i-1}+1)$ to r_i or selecting the $4r_{i-1}$. Notice that in this case, $q_i = 2q_{i-1} + 0$. Thus, the next partial remainder

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

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 shifting r_i 2-bit left; while the $4q_i + 1$ (or $4q_i+3$) means shifting q_i 2-bit left and setting the least two significant bits to 01 (or 11).

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. [11] presented a low-cost integer implementation and [12] presented a single-precision floating point implementation on FPGA.

4 *n*S-Root Implementations

For some ASIC designs, a more efficient square root unit would be very useful. In this section, we present n-select (nS-Root) implementations of the non-restoring square root algorithm, where n is the number of root possibilities we can choose.

4.1 1S-Root Implementation

The 1S means that there is no other choice. We can consider the 1S-Root as the basis of the nS-Root implementations.

Except for the first-time iteration, the non-restoring-remainder algorithm can be presented as below. If $Q_i = 1$, $r_{i+1} = 4r_i - (4q_i + 1)$, else $r_{i+1} = 4r_i + (4q_i + 3)$. The first-time iteration always subtracts 1 from $4r_0$.

Because if $Q_i = 1$, $q_i = 2q_{i-1} + 1$, otherwise $q_i = 2q_{i-1} + 0$, the algorithm turns to: if $Q_i = 1$, $r_{i+1} = 4r_i - (8q_{i-1} + 5)$, else $r_{i+1} = 4r_i + (8q_{i-1} + 3)$. 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 $4r_i + (8\overline{q_{i-1}} + 3)$. We get a new presentation of the algorithm as below.

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

Fig. 1 illustrates the square root calculations by using a parallel CSA array. The ith partial remainder r_i is pre-

		0	D_1	D_2										
	$+$	1	1	1										
Carry bits		D_1	D_2	0										
Sum bits		1	$\overline{D_1}$	$\overline{D_2}$	D_3	D_4								
Partial root		$+$	Q^1_0	0	1	1								
Carry bits			0	D_3	D_4	0								
Sum bits			A^2_1	$\overline{D_2}$	$\overline{D_3}$	$\overline{D_4}$	D_5	D_6						
Partial root			$+$	Q^2_0	Q^2_1	0	1	1						
Carry bits				B^3_1	0	D_5	D_6	0						
Sum bits				A^3_1	A^3_2	$\overline{D_4}$	$\overline{D_5}$	$\overline{D_6}$	D_7	D_8				
Partial root				$+$	Q^3_0	Q^3_1	Q^3_2	0	1	1				
Carry bits					B^4_1	B^4_2	0	D_7	D_8	0				
Sum bits					A^4_1	A^4_2	A^4_3	$\overline{D_6}$	$\overline{D_7}$	$\overline{D_8}$	D_9	D_{10}		
Partial root					$+$	Q^4_0	Q^4_1	Q^4_2	Q^4_3	0	1	1		
Carry bits						B^5_1	B^5_2	B^5_3	0	D_9	D_{10}	0		
Sum bits						A^5_1	A^5_2	A^5_3	A^5_4	$\overline{D_8}$	$\overline{D_9}$	$\overline{D_{10}}$	D_{11}	D_{12}
Partial root						$+$	Q^5_0	Q^5_1	Q^5_2	Q^5_3	Q^5_4	0	1	1
Carry bits							B^6_1	B^6_2	B^6_3	B^6_4	0	D_{11}	D_{12}	0
Sum bits							A^6_1	A^6_2	A^6_3	A^6_4	A^6_5	$\overline{D_{10}}$	$\overline{D_{11}}$	$\overline{D_{12}}$

... ...

Figure 1. 1S-Root implementation

sented by two groups of data, carry bits (B^i_j) and sum bits (A^i_j). The Q^i_j means $Q_j \oplus Q_i$ that implements q_i or $\overline{q_i}$: if $Q_i = 1$, $Q^i_j = \overline{Q_j}$, else $Q^i_j = Q_j$. Notice that $Q^i_0 = Q_i$. Because the 011 is always added to the lowest three bits of partial remainder, the B^i_j for $j = i-1,\ i,\ i+1,\ i+2$ and A^i_j for $j = i,\ i+1,\ i+2$ can be simplified as shown in the figure. The concept diagram of the figure is shown in Fig. 2. We will use such figure style in the following discussion.

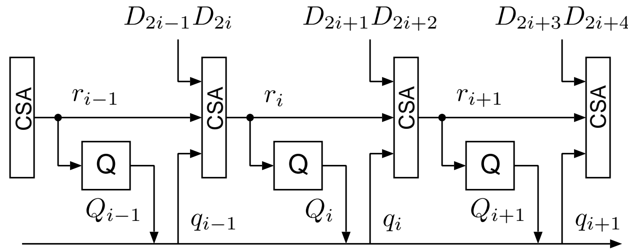

Figure 2. 1S-Root implementation

The input of the circuit is the radicand D and the output is the square root Q. The outputs of the CSA at stage i are named with A^i_j (sum bit) and B^i_{j-1} (carry bit) for $j = 1, 2, ..., i-1$.

The partially developed square root $q_i = Q_1Q_2...Q_i$ has i bits. Therefore the partial remainder r_i should be $i+1$ 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.

$$Q_i = \overline{A^i_1 \oplus B^i_1} \oplus (G^i_2 + P^i_2G^i_3 + ... + P^i_2P^i_3...P^i_{i-3}G^i_{i-2} \\ + 0 + P^i_2P^i_3...P^i_{i-2}A^i_{i-1}\overline{D_{2i-2}}(D_{2i-1} + D_{2i}))$$

where $G^i_j = A^i_jB^i_j$ and $P^i_j = A^i_j + B^i_j$. 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 [18] to speed the carry-lookahead circuit. 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).

4.2 1.5S Implementation

In the ith iteration, the computation of r_i depends on Q_{i-1}. There are two-case computations: If $Q_{i-1} = 1$, $r_i = 4r_{i-1} + (8\overline{q_{i-2}} + 3)$, else $r_i = 4r_{i-1} + (8q_{i-2} + 3)$. The Q_{i-1} is derived from r_{i-1}. Actually, the two-case computations of r_i can be started in parallel immediately after the r_{i-1} is known.

As shown in Fig. 3, two-case ($Q_{i-1} = 1$ and $Q_{i-1} = 0$) computations are performed simultaneously. The results

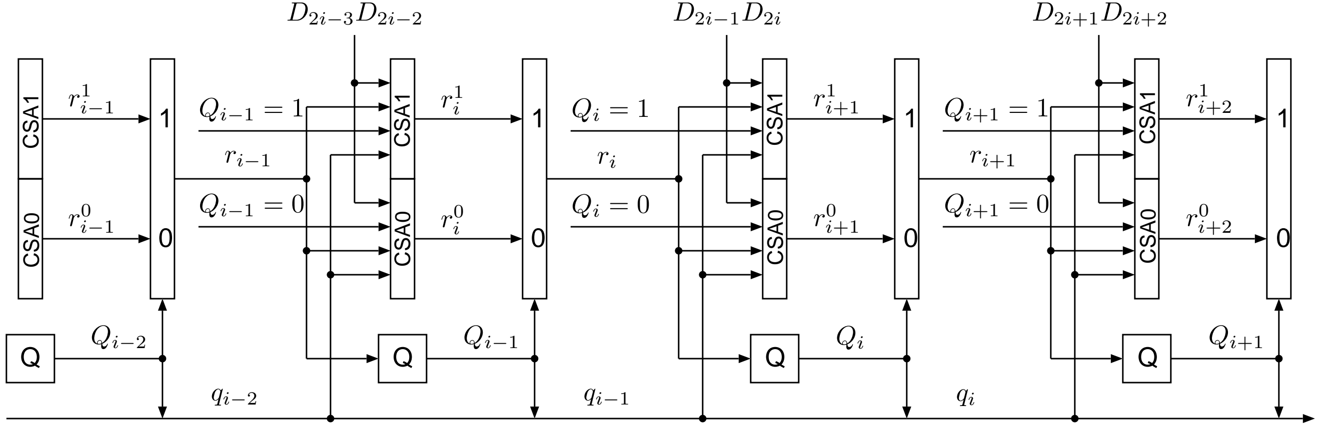

Figure 3. 1.5S-Root implementation

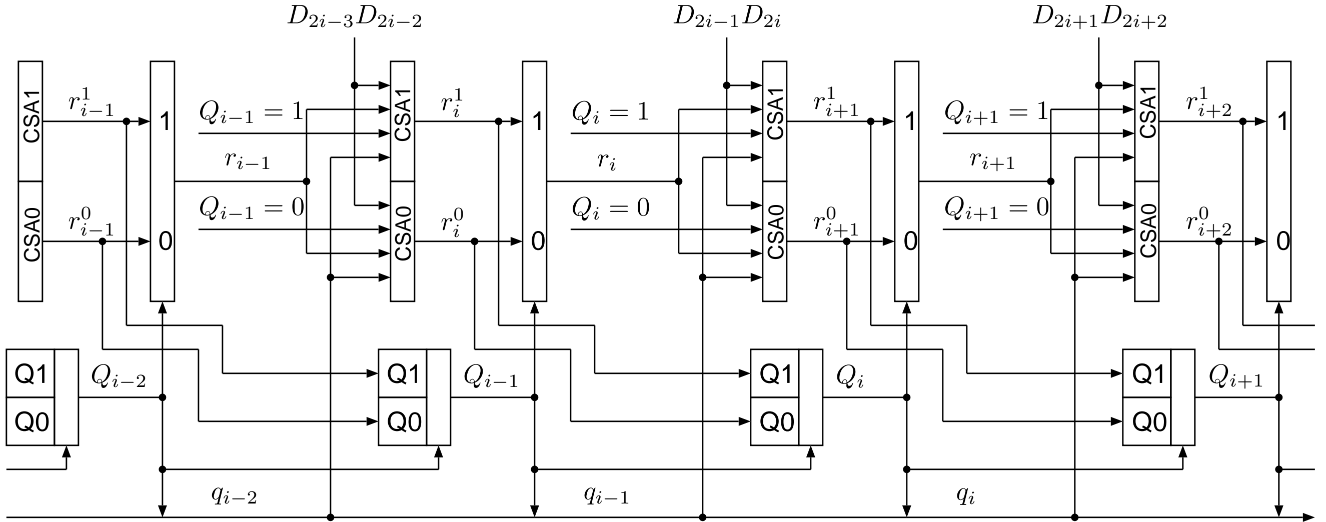

Figure 4. 2S-Root implementation

are labeled with r_i^1 and r_i^0 respectively. After Q_{i-1} is ready, we use a multiplexor to select a correct partial remainder (r_i in Fig. 3).

The time required by CSAs is hidden because the additions and the Q's generation are performed in parallel (Q's generation needs longer time than the addition). But here we used multiplexors which will introduce new delays.

For the CSAs, only the carry out generation needs 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.

In the implementation in Fig. 3, there still is only one root we can choose, but the number of CSAs is increased. We call it 1.5S-Root implementation.

4.3 2S-Root Implementation

The 2S means that there are two roots we can choose. In 1.5S-Root implementation, the Q's computation starts after the correct partial remainder is chosen. In the 2S-Root implementation, we start the two-case computations of Q_i in parallel before the correct partial remainder is chosen (Fig. 4).

As shown in Fig. 4, the correct Q_i can be selected by using multiplexors after the Q_{i-1} is known. And then, the Q_i is used to select the correct partial remainder r_{i+1}. The space (or chip area) required by 2S-Root is less than two times compared to 1S-Root because the sums of the CSA do not require new space as we mentioned above. Comparing to 1.5S-Root, the number of CSAs in 2S-Root is the

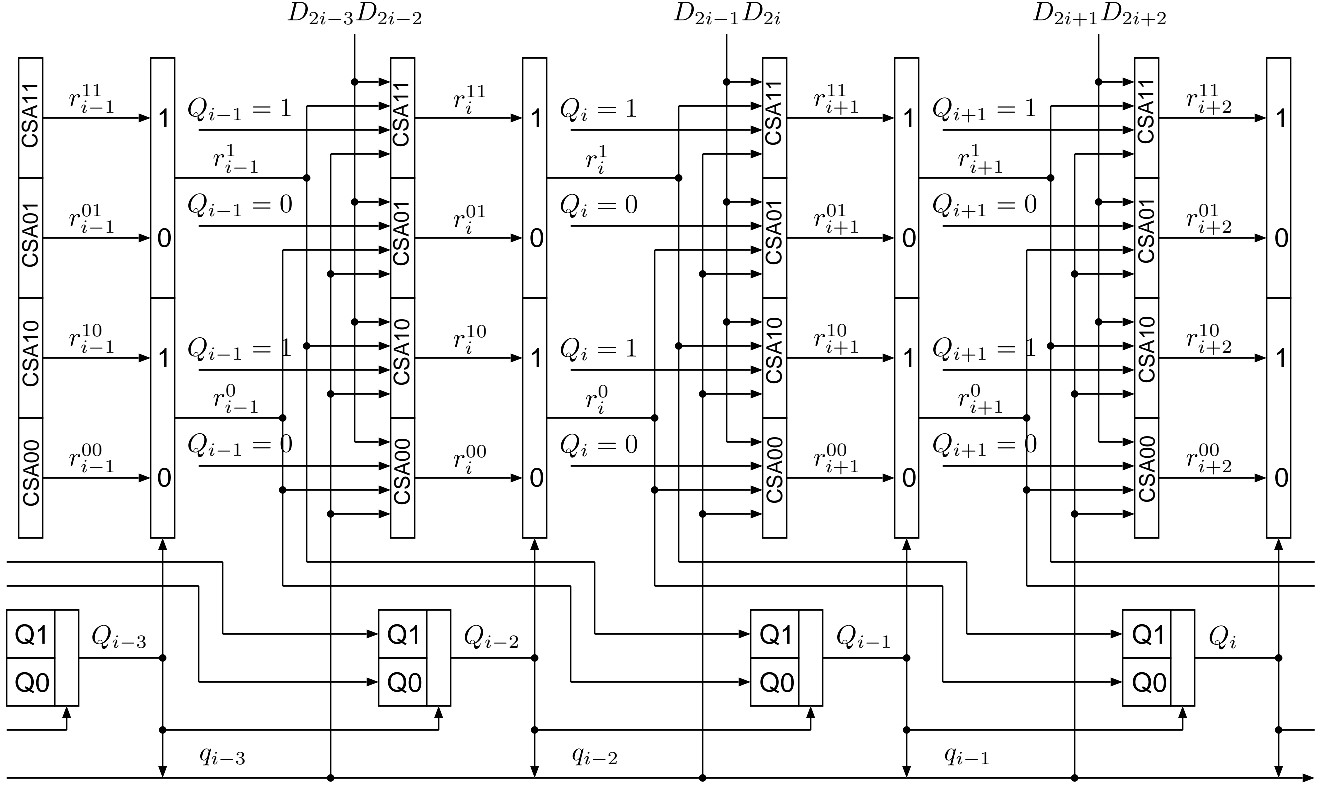

Figure 5. 2.5S-Root implementation

same, but the circuit for Q's computation is duplicated.

4.4 2.5S-Root Implementation

Similar to the expansion of 1S-Root to 1.5-Root, we can hide the addition time by calculating the all possible computations of partial remainder in parallel. This results in the 2.5S-Root (Fig. 5).

Before the correct remainder is selected, we can pass the two remainders to the next stage to calculate all the cases of the next remainder. See Fig. 5, for each Q, there are four CSAs and two multiplexors. After Q_{i-1} is ready, the upper multiplexor will output r_{i+1} with the assumption of $Q_i = 1$ (r^1_{i+1} in Fig. 5) and the lower multiplexor will output r_{i+1} with the assumption of $Q_i = 0$ (r^0_{i+1} in Fig. 5). The r^1_{i+1} and r^0_{i+1} are fed to the next stage where the four cases of the r_{i+2} are computed with the assumptions of $Q_{i+1} = 1$ and $Q_{i+1} = 0$ respectively. Similarly, we can design the circuits of 4S-Root and 4.5S-Root.

5 Cost/Performance Tradeoff

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 k units. In 1S-Root, an iteration takes $(k + 1)$ units, while in the 1.5S-Root, it is $(k+m)$, where m is the multiplexor's time units.

Typically, $m = 0.5$ [10] and $k = 2$ as described in the previous section. The speedup of 1.5S-Root is $(k + 1)/(k + m) = 3/2.5 = 120\%$. In 2S-Root, two iterations take $(k + 1 + 2m)$ units. The speedup of 2S-Root is $2(k + 1)/(k + 1 + 2m) = 6/4 = 150\%$. Similarly, the speedup of 2.5S-Root is $2(k + 1)/(k + 2m) = 6/3 = 200\%$.

Tab. 1 lists the comparison of the performance (operation latency and issue rate) and cost required by proposed approaches and others when performing double-precision floating point square root. The data for other implementations of double-precision are quoted from [9]. Tab. 1 also lists the iterative version of our algorithm's implementation. The s means the number of square root bits developed in each iteration. We also investigated the cost and speed for single precision floating point square root.

It can be found that the proposed simple implementations have about same level performance compared to other complex implementations, which can be readily appreciated. More importantly, the proposed implementations are very easy to be fully pipelined with the issue rate of one

Table 1. Cost/performance comparison

Precision	Algorithms	Latency (t_{FA})	Issue rate (cycles)	Cost	
				CSAs	Others
Double	Lang's radix-256	126	21	819	2048×54-bit ROM table, 13-bit adder, 6-to-2 adder, selector, converter
	Fandrianto's radix-8	190	16	115	128×5-bit table, two 256×3-bit tables, two 9-bit carry-lookahead adders
	IBM-RS6000 Newton-Raphson	204	12	2809	128×8-bit table, a 9-, two 53-bit CPAs
	WEITEK W4164/4363	136	8	4096	ROM table
	Matula's radix-2^{16}	216	7	1197	ROM table
Double	1S-Root	159	1	1378	Q generators
	1.5S-Root	133	1	2067	Q generators, multiplexors
	2S-Root	106	1	2067	Q generators, multiplexors
	2.5S-Root	80	1	4134	Q generators, multiplexors
	Iterative version (s=7)	168	8	392	Q generators
Single	1S-Root	72	1	276	Q generators
	1.5S-Root	60	1	414	Q generators, multiplexors
	2S-Root	48	1	414	Q generators, multiplexors
	2.5S-Root	36	1	828	Q generators, multiplexors
	Iterative version (s=6)	72	4	144	Q generators

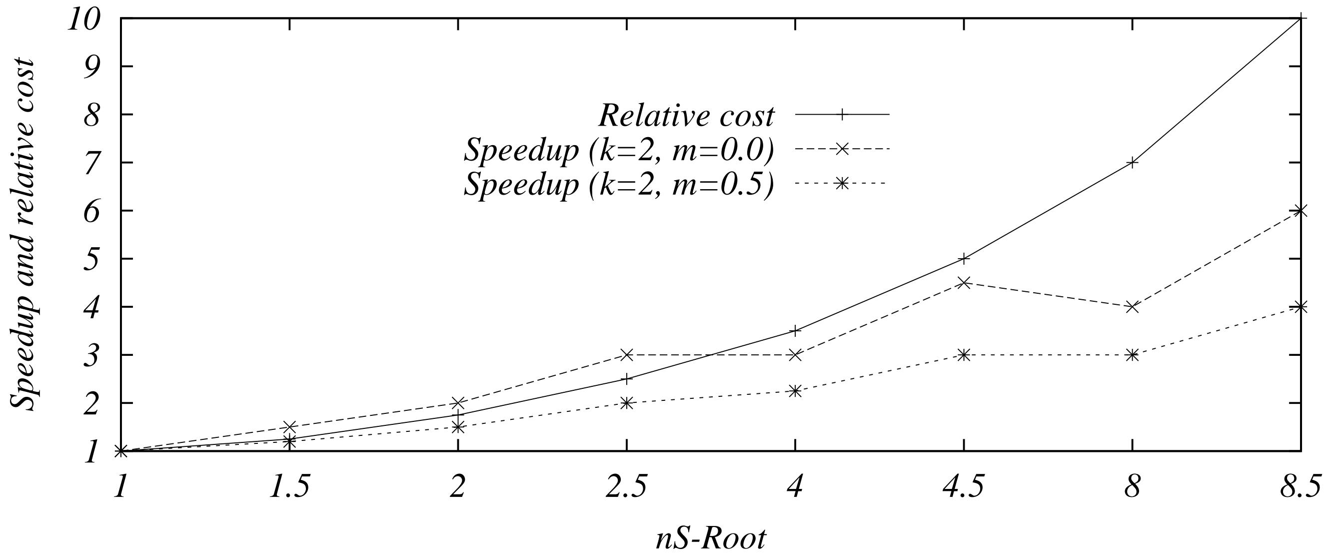

Figure 6. Speedup and relative cost of *n*S-Root

clock cycle, while others use iteration method that results in the issue rate 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.

The basic hardware requirements for the 1S-Root are $\sum_{i=1}^{53}(i-1) = 26 \times 53$ CSAs and corresponding square root result bit generators, while in other implementation,

IBM R6000 for example, a multiplier will require about 53×53 CSAs. Furthermore, the number of CSAs 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).

Fig 6 illustrates the speedup and relative cost of the nS-Root implementations. The speedup labeled with $m = 0.0$ is the upper bound where we assume the zero delay time of multiplexor. We can find that the 2S- or 2.5S-Root would be a good solution from the cost/performance point of view.

6 Conclusion Remarks

Four nS-Root (n=1, 1.5, 2, 2.5) low-cost fully pipelined square root implementations based on a non-restoring-remainder square root algorithm were presented in this paper. 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 the issue rate of one clock cycle. We also investigated the cost for all the implementations. It is helpful for making a tradeoff between the cost and performance by selecting a suitable n when we design a pipelined square root unit.

References

[1] J. Bannur and A. Varma. The vlsi implementation of a square root algorithm. In *Proc. IEEE Symposium on Computer Arithmetic*, pages 159–165. IEEE Computer Society Press, 1985.

[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*, pages 55–64, February 1990.

[3] M. Ercegovac and T. Lang. Module to perform multiplication, divison, and square root in systolic arrays for matrix computations. *Journal of Parallel and Distributed Computing*, 11:212–221, 1991.

[4] J. Fandrianto. Algorithm for high speed shared radix 8 divison and radix 8 square root. In *Proc. 9th IEEE Symposium on Computer Arithmetic*, pages 68–75, 1989.

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

[6] K. Johnson. Efficient square root implementation on the 68000. *ACM Transaction on Mathematical Software*, 13(2):138–151, February 1987.

[7] H. Kabuo, T. Taniguchi, A. Miyoshi, H. Yamashita, M. Urano, H. Edamatsu, and S. Kuninobu. Accurate rounding scheme for the newton-raphson method using redundant binary representation. *IEEE Trans. on Computers*, 43(1):43–51, 1994.

[8] G. Knittel. A vlsi-design for fast vector normalization. *Comput.& Graphics*, 19(2):261–271, 1995.

[9] T. Lang and P. Montuschi. Higher radix square root with prescaling. *IEEE Transaction on Computers*, 41(8):996–1009, 1992.

[10] T. Lang and P. Montuschi. Very-high radix combined division and square root with prescaling and selection by rounding. In *Proc. 12th IEEE Symposium on Computer Arithmetic*, pages 124–131. IEEE Computer Society Press, 1995.

[11] Y. Li and W. Chu. A new non-restoring square root algorithm and its vlsi implementations. In *Proc. of 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 538–544, Austin, Texas, USA, October 1996. IEEE Computer Society Press.

[12] Y. Li and W. Chu. Implementation of single precision floating point square root on fpgas. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 226–232, Napa, California, USA, April 1997. IEEE Computer Society Press.

[13] S. Majerski. Square-rooting algorithms for high-speed digital circuits. *IEEE Transaction on Computers*, 34(8):724–733, 1985.

[14] P. Markstein. Computation of elementary functions on the ibm risc rs6000 processor. *IBM Jour. of Res. and Dev.*, pages 111–119, January 1990.

[15] J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In *Proc. 2nd International Conference on Theorem Provers in Circuit Design*, pages 52–71, 1994.

[16] J. Prabhu and G. Zyner. 167 mhz radix-8 divide and square root using overlapped radix-2 stages. In *Proc. 12th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE Computer Society Press, 1995.

[17] C. Ramamoorthy, J. Goodman, and K. Kim. Some properties of iterative square-rooting methods using high-speed multiplication. *IEEE Transaction on Computers*, C-21(8):837–847, 1972.

[18] C. Rowenand, M. Johnson, and P. Ries. The mips r3010 floating-point coprocessor. *IEEE MICRO*, pages 3–62, June 1988.