FCCM'97, IEEE Symposium on FPGAs for Custom Computing Machines, April 16 – 18, 1997, Napa, California, USA

Implementation of Single Precision Floating Point Square Root on FPGAs

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

Square root operation is hard to implement on FPGAs because of the complexity of the algorithms. In this paper, we present a non-restoring square root algorithm and two very simple single precision floating point square root implementations based on the algorithm on FPGAs. One is low-cost iterative implementation that uses a traditional adder/subtractor. The operation latency is 25 clock cycles and the issue rate is 24 clock cycles. The other is high-throughput pipelined implementation that uses multiple adder/subtractors. The operation latency is 15 clock cycles and the issue rate is one clock cycle. It means that the pipelined implementation is capable of accepting a square root instruction on every clock cycle.

1. Introduction

Addition, subtraction, multiplication, division, and square root are five basic operations in computer graphics and scientific calculation applications. The operations on integer numbers and floating point numbers have been implemented with standard VLSI circuits. Some of them have been implemented on FPGAs based custom computing machines. Shirazi *et al* presented FPGA implementations of 18-bit floating point adder/subtractor, multiplier, and divider [17]. Louca *et al* presented FPGA implementations of single precision floating point adder/subtractor and multiplier [10].

Because of the complexity of the square root algorithms, the square root operation is hard to implement on FPGAs for custom computing machines. In this paper, we present two very simple single precision floating point square root implementations on FPGAs based on a non-restoring square root algorithm. The first is a low-cost iterative implementation that uses only a single conventional adder/subtractor; the second is a fully pipelined implementation that is capable of accepting a square root instruction on every clock cycle.

We first review the various square root algorithms briefly. Then we introduce the non-restoring square root algorithm, based on which two single precision floating point square root implementations on FPGAs are presented here. Finally, we give the conclusion remarks.

2. Square Root Algorithms

The square root algorithms and implementations have been addressed mainly in three directions: *Newton-Raphson* method [4] [6] [12] [15], *SRT-Redundant* method [2] [3] [8] [11] [14], and *Non-Redundant* method [1] [5] [7] [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 $f(T) = 1/T^2 - X$ to derive the iteration equation $T_{i+1} = T_i \times (3 - T_i^2 \times X)/2$, where T_i is an approximate value of $1/\sqrt{X}$. After n-time iterations, an approximate square root can be obtained by equation $Y = \sqrt{X} \simeq T_n \times X$.

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. In the design of most commercial RISC processors, a multiplier is shared by all iterations of multiplication, division, and square root instructions. This becomes an obstacle of exploiting instruction level parallelism for multi-issued processors. 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 $X_{i+1} = 2X_i - 2Y_i y_{i+1} - y_{i+1}^2 2^{-(i+1)}$, $Y_{i+1} = Y_i + y_{i+1}2^{-(i+1)}$, 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 $F = -2Y_i y_{i+1} - y_{i+1}^2 2^{-(i+1)}$. (4) Addition of F and $2X_i$ to produce X_{i+1}. 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 complexity of the circuitry, some of the implementations use an iterative version, i.e., all the iterations share same hardware resources. Consequently, the implementations are not capable of accepting a square root instruction on every clock cycle. Notice that this method may generate a wrong resulting value at the last digit position, because it satisfies $X_i \leq 2(Y_i + 2^{-(i+1)})$, while a correct Y_i should satisfy $X_i \leq 2Y_i$.

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 $R_{i+1} = X - Y_i^2$, $Y_{i+1} = Y_i + y_{i+1}2^{-(i+1)}$, 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. 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 computations can be simplified by eliminating the square operation by variable substitution on $X_{i+1} = (X - Y_i^2)2^i$: $X_{i+1} = 2X_i - 2Y_i y_i + y_i^2 2^{-i}$, $Y_{i+1} = Y_i + y_{i+1}2^{-(i+1)}$, 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 need to convert such an $F = -2Y_i y_i + y_i^2 2^{-i}$ 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. The Non-Restoring Square Root Algorithm

The non-restoring square root algorithm also uses the two's complement representation for the square root result. It is a non-restoring algorithm that does not restore the remainder [9]. At each iteration the algorithm generates an exact resulting value, even in the last bit position. There is no need to do 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.

Assume that the radicand is denoted by a 32-bit unsigned number: $D = D_{31}D_{30}D_{29}...D_1D_0$. The value of the radicand 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 each pair of bits of the radicand, the integer part of square root has one bit. Thus the integer part of square root for a 32-bit radicand has 16 bits: $Q = Q_{15}Q_{14}Q_{13}...Q_1Q_0$. The remainder $R = D - Q \times Q$ has at most 17 bits: $R = R_{16}R_{15}R_{14}...R_1R_0$.

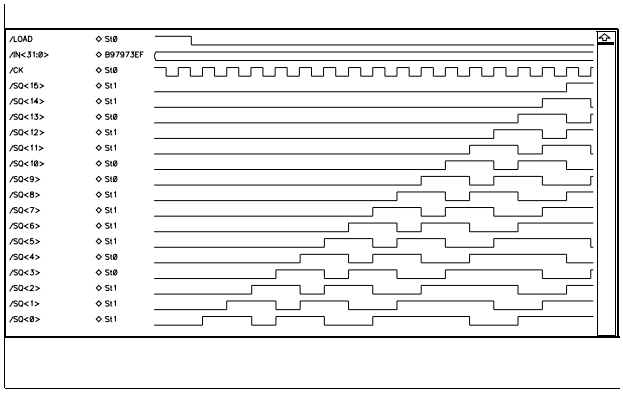
The reason of why R has at most 17 bits is explained as below. We have the equation $D = (Q \times Q + R) < (Q + 1) \times (Q + 1)$. Thus, $R < (Q + 1) \times (Q + 1) - Q \times Q = 2 \times Q + 1$, i.e., $R \leq 2 \times Q$ because the remainder R is an integer. It means that the remainder has at most one binary bit more than the square root.

Because we do not use redundant representation for square root, an exact bit can be obtained in each iteration. This makes the hardware implementation simple. Additionally, an exact remainder can be obtained although it is rarely requested by real applications. The non-restoring square root algorithm using non-redundant binary representation is given below.

1. *Set* $q_{16} = 0, r_{16} = 0$ *and then iterate from* $k = 15$ *to* 0.
2. *If* $r_{k+1} \geq 0$, $r_k = r_{k+1}D_{2k+1}D_{2k} - q_{k+1}01$,
   else $r_k = r_{k+1}D_{2k+1}D_{2k} + q_{k+1}11$,
3. *If* $r_k \geq 0$, $q_k = q_{k+1}1$ *(i.e.,* $Q_k = 1$*)*,
   else $q_k = q_{k+1}0$ *(i.e.,* $Q_k = 0$*)*,
4. *Repeat steps 2 and 3, until* $k = 0$.
   If $r_0 < 0$, $r_0 = r_0 + q_0 1$.

where $q_k = Q_{15}Q_{14}...Q_k$ has $(16 - k)$ bits, e.g., $q_0 = Q = Q_{15}Q_{14}Q_{13}...Q_1Q_0$, and r_k has $(17 - k)$ bits, e.g., $r_0 = R = R_{16}R_{15}R_{14}...R_1R_0$. Notice that $r_{k+1}D_{2k+1}D_{2k}$ means $r_{k+1} \times 4 + D_{2k+1} \times 2 + D_{2k}$. Similarly, $q_{k+1}1$ means $q_{k+1} \times 2 + 1$. The multiplications and additions are not needed. Instead, we use shifts and concatenations by suitable wiring. It can be found that the algorithm needn't do any conversion before the addition or subtraction.

Let us see an example in which $D = D_7D_6D_5D_4D_3D_2D_1D_0 = 01111111$ is an 8-bit radicand, hence Q will be a 4-bit square root. The example is illustrated below. We get $Q = 1011$ and $R = 00110$.



Set $q_4 = 0, r_4 = 0$ *and then iterate from* $k = 3$ *to* 0.
$\mathbf{k = 3}$:
$r_4 \geq 0,\ \ r_3 = r_4D_7D_6 - q_401 = 001 - 001 = 000,$
$r_3 \geq 0,\ \ q_3 = 1$ *(i.e.,* $Q_3 = 1$*),*
$\mathbf{k = 2}$:
$r_3 \geq 0,\ \ r_2 = r_3D_5D_4 - q_301 = 0011 - 0101 = 1110,$
$r_2 < 0,\ \ q_2 = q_30 = 10$ *(i.e.,* $Q_2 = 0$*),*
$\mathbf{k = 1}$:
$r_2 < 0,\ \ r_1 = r_2D_3D_2 + q_211 = 11011 + 01011 = 00110,$
$r_1 \geq 0,\ \ q_1 = q_21 = 101$ *(i.e.,* $Q_1 = 1$*),*
$\mathbf{k = 0}$:
$r_1 \geq 0,\ \ r_0 = r_1D_1D_0 - q_101 = 011011 - 010101 = 000110,$
$r_0 \geq 0,\ \ q_0 = q_11 = 1011$ *(i.e.,* $Q_0 = 1$*),*

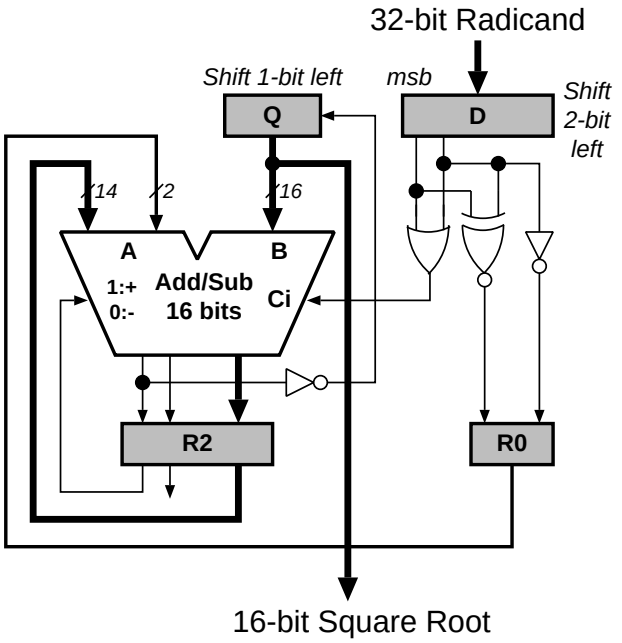


Figure 1. A very-low cost circuitry for integer square root.

An iterative low-cost version of the circuit design for a 32-bit radicand is 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 (combination of $R2$ and $R0$) contains the partial remainder. Registers Q and R are cleared at the beginning. A 16-bit conventional adder/subtractor is required. It subtracts if the

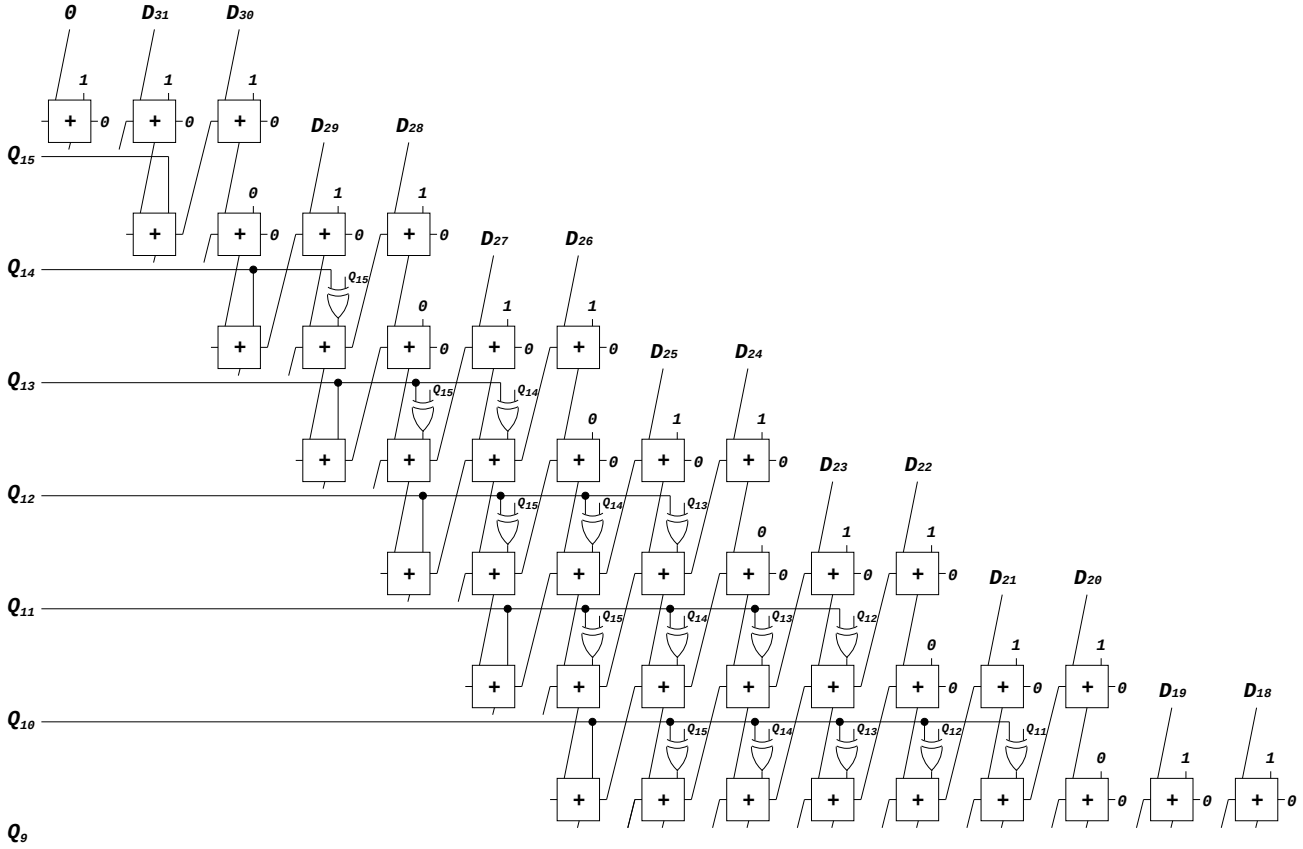


Figure 3. PASQRT – parallel array for square root.

can be replaced with $\overline{q_{k+2}}011$. Similarly, if $Q_{k+1} = 0$, the $+q_{k+1}11$ can be replaced with $+q_{k+2}011$. Now, the algorithm turns to

1. *Always* $r_{15} = 0D_{31}D_{30} + 111$,
2. *If* $r_{15} \geq 0$, $q_{15} = 1$ *(i.e.,* $Q_{15} = 1$*),*
   else $q_{15} = 0$, *(i.e.,* $Q_{15} = 0$*),*
3. *iterate from* $k = 14$ *to* 0,
4. *If* $Q_{k+1} = 1$, $r_k = r_{k+1}D_{2k+1}D_{2k} + \overline{q_{k+2}}011$,
   else $r_k = r_{k+1}D_{2k+1}D_{2k} + q_{k+2}011$,
5. *If* $r_k \geq 0$, $q_k = q_{k+1}1$ *(i.e.,* $Q_k = 1$*),*
   else $q_k = q_{k+1}0$ *(i.e.,* $Q_k = 0$*),*
6. *Repeat steps 4 and 5, until* $k = 0$.
   If $r_0 < 0$, $r_0 = r_0 + q_0 1$.

The first two steps are only for dealing with the first iteration, i.e., for calculating the r_{15} and q_{15}. The carry-save adders can be used to form a parallel-array implementation as shown as in Fig. 3. We call it as *parallel array for square rooting* (PASQRT). The XOR gates in the figure are used to implement q_{k+2} or $\overline{q_{k+2}}$ based on Q_{k+1}. The $+011$ needn't use CSAs, it can be simplified.

For determination of Q_k, it is needed to know the sign of the partial remainder r_k. Because the remainder has at most one binary bit more than the square root, we just need to check the R_{17-k} bit of the r_k. The carry-lookahead circuit can be used here with some simplification. The following is an example of determination of Q_{11} (Fig. 4).

$$
\begin{aligned}
Q_{11} &= \overline{A_6 \oplus B_6} \oplus C_5 \\
C_5 &= G_5 + P_5 \cdot G_4 + P_5 \cdot P_4 \cdot P_3 \cdot C_2 \\
G_i &= A_i \cdot B_i \\
P_i &= A_i + B_i \\
C_2 &= \overline{D_{24}} \cdot (D_{23} + D_{22})
\end{aligned}
$$

where A_i and B_i are the outputs of CSA, A_i is sum and B_i is carry. C_5 is the carry-out when adding the low 6 bits of A and B. In the C_5 equation, there is no $P_5 \cdot P_4 \cdot G_3$ term because $B_3 = 0$. C_2 can be formed directly from the radicand. Generally, for the determination of Q_k, the functions are as the following.

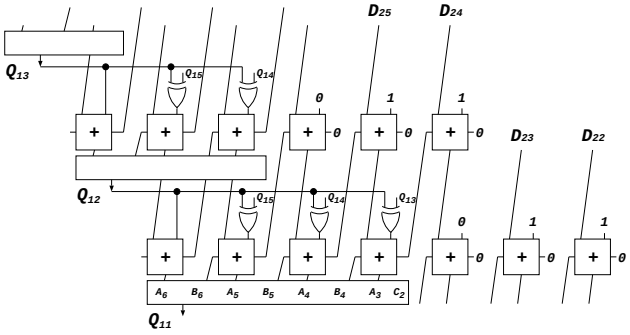


Figure 4. Determination of square root bit.

$$
\begin{aligned}
Q_k &= \overline{A_{17-k} \oplus B_{17-k}} \oplus C_{16-k} \\
C_{16-k} &= G_{16-k} + P_{16-k} \cdot G_{15-k} + ...+ \\
&\quad +P_{16-k} \cdot P_{15-k} \cdot ... \cdot P_3 \cdot C_2 \\
G_i &= A_i \cdot B_i \ (G_3 = 0) \\
P_i &= A_i + B_i \ (P_3 = A_3) \\
C_2 &= \overline{D_{2k+2}} \cdot (D_{2k+1} + D_{2k})
\end{aligned}
$$

The circuit 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. The C_{16-k} needs a large fan-in gate. This would not be suitable for common precharge/discharge circuit implementation. To solve this problem, we can adopt an unconventional fast, large fan-in circuit. It was developed by Rowen, Johnson, and Ries [16] and used in MIPS R3010 floating point coprocessor for divider's quotient logic, fraction zero-detector, and others.

Unfortunately, we could not find a method to implement such unconventional circuit on FPGAs, especially on Xilinx xc4000 family. Instead, we investigated to use the "dedicated high-speed carry-propagation circuit" the xc4000 family provided.

5. Implementations of Single Precision Floating Point Square Root on FPGAs

The value of a normalized single precision floating point number D is $(-1)^s \times (2^{e-127}) \times (1.f)$, where s is sign, e is biased exponent, and f is fraction. In our implementation, we consider only the non-negative radicand and square root. The $(1.f)$ will be shifted one- or zero-bit left so that the new exponent e' makes $e' - 127$ even. The resulting biased exponent will be $(e - 127)/2 + 127$. We can use an adder to do it because

$$(e - 127)/2 + 127 = e/2 + 63 + e\%2,$$

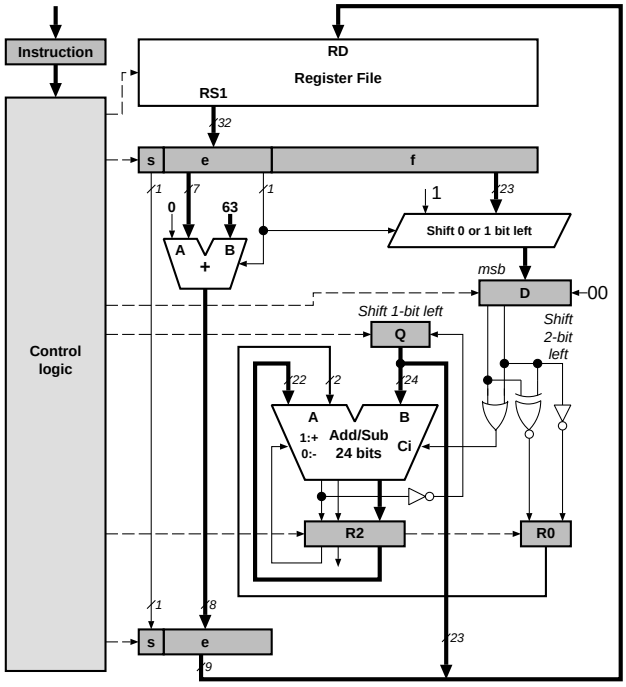


Figure 5. A low-cost implementation of floating-point square root.

where $\%$ denotes a modular operation, i.e., $e\%2$ is the least significant bit of e. The shifted fraction will be $1x.xx...xx$ or $01.xx...xx$. In the both cases, the resulting value of Q will be $1.yy..yy$. This means that the resulting fraction is already normalized. If the unit deals with subnormal (denormalized) numbers, a normalization stage will be needed at the final stage. Notice that zeros are needed to be attached to the fraction in order to obtain enough bits of the square root.

Fig. 5 shows a low-cost implementation of the single-precision floating-point square root unit. After loading data into register D, twenty-four clock cycles are needed for generating a 24-bit result.

Fig. 6 shows a pipelined implementation of the single-precision floating-point square root unit. We used the "high-speed carry-propagation circuit" (cy4) in the design. Because the high-order square root bits can be generated very fast, it is not necessary to use a whole pipeline stage for one bit. We divide the calculations of Q_k, $(k = 23, 22, ..., 1, 0)$, into 15 pipeline stages as shown in Tab. 1. Pipeline registers are placed between the stages. The implementation is fully pipelined and capable of accepting a new square root instruction on every cycle.

A comparison of the latency, issue rate, and cost required by the two implementations is listed in Tab. 2. It can be found that the iterative implementation requires few area

Table 1. Clock cycles required for calculating Q_{23} .. Q_0

Result bits	Q_{23} .. Q_{18}	Q_{17}..Q_{15}	Q_{14}..Q_{13}	Q_{12} .. Q_{11}	Q_{10} .. Q_0
Adder bits	1 .. 6	7 .. 9	10 .. 11	12 .. 13	14 .. 24
Clock cycles	1	1	1	1	11

Table 2. Cost/performance comparison of two FPGA implementations

	Performance		Cost (Xilinx xc4000 CLB)	
	Latency (cycles)	Issue (cycles)	CLB function generators	CLB flip-flops
Iterative	25	24	82	138
Pipelined	15	1	408	675

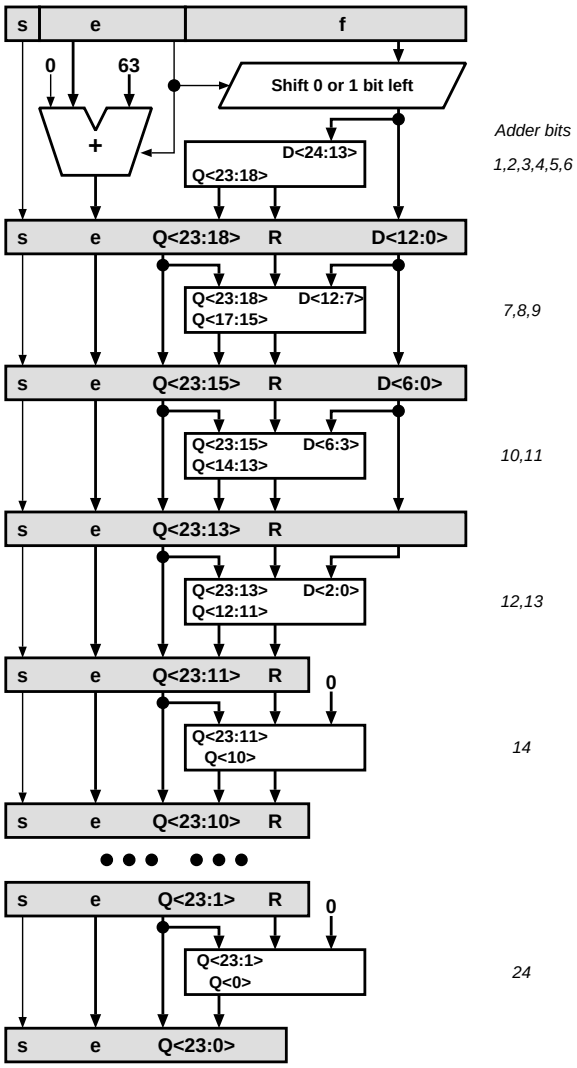


Figure 6. A pipelined implementation of floating-point square root.

(the CLB function generators is almost equal to the area required by the two adders), and the pipelined implementation achieves high performance at a reasonable cost (about five times compared to that of the iterative implementation).

6. Conclusion Remarks

Two single precision square root implementations on FPGAs, based on a non-restoring algorithm, were presented in this paper. The iterative implementation requires few area, and the pipelined implementation achieves high performance at a reasonable cost. Both the implementations are very simple, which can be readily appreciated.

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 floating point unit on FPGAs.

References

[1] J. Bannur and A. Varma, "The VLSI Implementation of A Square Root Algorithm", *Proc. of 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] M. Ercegovac and T. Lang, "Radix-4 Square Root Without Initial PLA", *IEEE Transaction on Computers*, Vol. 39, No. 8, 1990. pp1016-1024.

[4] J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, Second Edition, Morgan

Kaufmann Publishers, Inc., 1996. Appendix A: Computer Arithmetic by D. Goldberg.

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

[8] T. Lang and P. Montuschi, "Very-high Radix Combined Division and Square Root with Prescaling and Selection by Rounding", *Proc. of 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] L. Louca, T. A. Cook, W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs", *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM96)*, IEEE Computer Society Press, 1996. pp107-116.

[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. of 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. of 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.

[17] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines", *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM95)*, IEEE Computer Society Press, 1995. pp155-162.