INT. J. ELECTRONICS, 1992, VOL. 73, NO. 6, 1121-1132

Fast square root architecture based on prediction of result bits

MUKESH SHARMA† and MEGHANAD D. WAGH†

A new architecture for the fast evaluation of the square-root of normalized binary numbers is presented. It exploits the fact that depending upon the relative magnitudes of the partial result and the partial remainder at any stage of a non-restoring square-root algorithm, a large number of result bits can be predicted. This allows one to convert approximately two thirds of the additions in the square-root procedure to shifts at the cost of a marginal increase in the hardware.

1. Introduction

The square root is one of the important operations used in many diverse computational algorithms. Consequently, the design of fast square-root architectures has proved to be a topic of considerable research interest in recent years. Even though some other variations are available (Lim and Jullien 1989), most square-root algorithms evaluate the required result through a series of additions. Each addition provides a single bit (or digit) of the answer beginning with the most significant bit. To improve the performance of a square-root circuit requires either fast addition techniques or a modification of the basic square-root algorithm to require less additions.

The speed of the addition can be improved mainly by expressing the computation in non-redundant number systems based on a radix higher than two (Oklobdzija and Ercegovac 1982, Montuschi and Ciminiera 1989). Square-root architectures employing this principle and using radix 4 or 8 have been discussed by Fandrianto (1989) and Ercegovac and Lang (1990). The speed of computing each answer digit may be further improved by using the digit set $\{-1, 0\,1\}$ to express the answer (Metze 1965) or by using an intermediate non-binary redundant number system (Majerski 1985).

Other methods improve the square-root computation through modification of its basic algorithm to require less additions. These methods typically require minimal change of the normal square-root architecture and are therefore cost effective. Typical among these is the modification of the basic restoring square-root algorithm to a non-restoring version which requires an average of only n additions as against the original $3n/2$ to compute an n bit answer. New algorithms have been reported which allow one to skip some of the addition steps corresponding to the digit 0 in a non-binary square-root computation (Montuschi and Ciminiera 1991).

In this paper, we develop a new square-root algorithm that uses substantially less additions than the non-restoring algorithm with a very small hardware penalty. Our algorithm is based on the prediction of several answer bits in every cycle. This work is similar to the Wilson–Ledley algorithm for division (Wilson and Ledley 1961) which predicts a number of result bits in each cycle from the magnitude of the

Received 20 July 1992; acceted 22 July 1992.

†Department of Electrical Engineering and Computer Science, Lehigh University, Bethlehem, PA 18015, U.S.A.

0020-7217/92 $3.00

partial remainder. In the algorithm proposed here, the magnitude of the partial reminder relative to that of the partial result allows us to do the same thing in case of a square-root computation. The relative magnitude comparison is accomplished by identifying the leading significant bits of the two operands. The hardware complexity of the resultant architecture is marginally increased from the conventional architecture while the number of additions required to obtain the square root are decreased significantly.

Section 2 of this paper starts from the non-restoring square-root algorithm and derives the algorithm to predict the result bits. Section 3 is devoted to the realization of this algorithm to obtain the fast square-root architecture. We show that it is possible to convert many of the additions in the algorithm to mere shifts by tapping three specific bits of the partial remainder register. Since a shift operation is several times faster than an addition, the resultant architecture provides substantial speed improvement. Section 4 further discusses hardware related issues that directly control the complexity and speed of this architecture. Finally, Section 5 concludes the paper by summarizing the results obtained.

2. Algorithm to preduct the result bits

In this section we derive the algorithm to predict several answer bits in each addition cycle of a square-root computation. This prediction is based on the relative magnitudes of the partial remainder and the partial answer. In general, such a magnitude comparison could be time consuming; but we show that the knowledge of the positions of the most significant bits in the two quantities is sufficient to carry out the prediciton. By using a normalized input operand, one can easily figure out the position of the leading significant bit of the partial answer at every clock. The bounds on the partial remainder derived here shows that its leading significant bit can be predicted by examining only three bit positions. Thus the architecture implementing this algorithm only needs to examine three bits of the partial remainder to predict several result bits at the cost of a single addition in every cycle.

The commonly used non-restoring algorithm to compute the square root B of a $2n$ bit number A may be described as follows (Hwang 1979):

Define the initial values of the partial result, B_i, and the partial remainder, R_i, as

$$R_0 = 2^{-2(n-1)}A - 1 \quad \text{and} \quad B_0 = 0 \tag{1}$$

and for $1 \leqslant i \leqslant n$, compute them recursively as:

$$R_i = 4R_{i-1} - (8B_{i-1} + 5), \quad B_i = 2B_{i-1} + 1, \quad \text{if } R_{i-1} \geqslant 0 \tag{2}$$

$$R_i = 4R_{i-1} + (8B_{i+1} + 3), \quad B_i = 2B_{i-1}, \quad \text{if } R_{i-1} < 0 \tag{3}$$

The value of B_n then gives the desired square root B.

Note that in the non-restoring algorithm above, B_i may be computed from B_{i-1} through a left shift with either a 1 or a 0 sliding into the least significant bit position. This new bit of the result B is obtained at the cost of one addition required to obtain R_i from R_{i-1}. We now show that by comparing the magnitude of B_{i-1} and R_{i-1}, one can save many of these addtions. To do this, define s and t to be the positions of the most significant bits in B_{i-1} and R_{i-1}, respectively. (In two's complement number system, the position of the most significant bit is the position of the leading

1 for a positive number and position of the leading 0 for a negative number. For example, for six-bit representations, if $B_{i-1}=\langle 001011\rangle$ and $R_{i-1}=\langle 111010\rangle$, then $s=3$ and $t=2$.) Note that B_{i-1}, representing the partial result, is always positive. We now consider the following two cases based upon the sign of R_{i-1}. In each case, we are able to predict $s-t+2$ bits of B at the cost of a single addition.

Case 1: $R_{i-1}\geqslant 0$ *and* $t\leqslant s$

Since $R_{i-1}\geqslant 0$, application of (2) gives

$$R_i=4R_{i-1}-(8B_{i-1}+5) \quad \text{and} \quad B_i=2B_{i-1}+1$$

If this $R_i<0$, one would apply (3) to obtain

$$R_{i+1}=4R_i+(8B_i+3) \quad \text{and} \quad B_{i+1}=2B_i$$

Substituting for R_i and B_i in these equations gives

$$R_{i+1}=16R_{i-1}-16B_{i-1}-9 \quad \text{and} \quad B_{i+1}=4B_{i-1}+2.$$

If this new R_{i+1} is still negative, (3) would be applied again to obtain R_{i+2} and B_{i+2} as

$$R_{i+2}=64R_{i-1}-32B_{i-1}-17 \quad \text{and} \quad B_{i+1}=8B_{i-1}+4$$

Proceeding in this manner, we find that if $R_i, R_{i+1}, R_{i+2}, \ldots, R_{i+k-1}$ are all negative, then R_{i+k} and B_{i+k} would be evaluated as

$$\left.\begin{aligned} R_{i+k}&=2^{2(k+1)}R_{i-1}-2^{k+3}B_{i-1}-(2^{k+2}+1)\\ B_{i+k}&=2^{k+1}B_{i-1}+2^k \end{aligned}\right\} \tag{4}$$

We now show that it is possible to predict the value of k for which R_i to R_{i+k-1} are all negative by examining the magnitudes of B_{i-1} and R_{i-1} characterized only by s and t. The definitions of s and t in the case of positive B_{i-1} and R_{i-1} clearly imply that

$$R_{i-1}<2^{t+1} \quad \text{and} \quad B_{i-1}\geqslant 2^s$$

Using these limits, one obtains

$$R_{i+k-1}=2^{2k}R_{i-1}-2^{k+2}B_{i-1}-(2^{k+1}+1)<2^{k+t+1}(2^k-2^{s-t+1})$$

Clearly, for all k values satisfying $1\leqslant k\leqslant(s-t+1)$, one obtains $R_{i+k-1}<0$, implying that (4) is true for all these k values. Therefore one can directly compute $R_{i+s-t+1}$ and $B_{i+s-t+1}$ from R_{i-1} and B_{i-1} as

$$R_{i+s-t+1}=2^{2(s-t+2)}R_{i-1}-[2^{s-t+4}B_{i-1}+2^{s-t+3}+1] \tag{5}$$

$$B_{i+s-t+1}=2^{s-t+2}B_{i-1}+2^{s-t+1} \tag{6}$$

It should be noted that (5) represents a single addition because the terms in the square brackets form a single operand. Equation (6) represents the prediction of $(s-t+2)$ consecutive bits of B consisting of a 1 followed by a string of $(s-t+1)$

zeros. Thus in this case, our algorithm allows one to advance the square-root calculation through $s-t+2$ cycles of the nonrestoring alogrithm within the cost of just one addition.

Case 2: $R_{i-1}<0$ *and* $t\leqslant s$

In this case one may apply (3) to obtain

$$R_i=4R_{i=1}+(8B_{i-1}+3) \quad \text{and} \quad B_i=2B_{i-1}$$

If R_i, $R_{i+1},\ldots,$ R_{i+k-1} are all non-negative, then R_{i+1}, $R_{i+2},\ldots,$ R_{i+k} would be obtained by the application of (2) to give

$$R_{i+k}=2^{2(k+1)}R_{i-1}+2^{k+3}B_{i-1}+(2^{k+2}-1)$$

$$B_{i+k}=2^{k+1}B_{i-1}+(2^k-1) \tag{7}$$

To find the largest value of k for which (7) is true, characterize the magnitudes of negative R_{i-1} and positive B_{i-1} as

$$R_{i-1}>-2^{t+1}, \quad \text{and} \quad B_{i-1}>2^s$$

Hence,

$$R_{i+k-1}=2^{2k}R_{i-1}+2^{k+2}B_{i-1}+(2^{k+1}-1)>2^{k+t+1}(2^{s-t+1}-2^k)$$

Thus for all k satisfying $1\leqslant k\leqslant(s-t+1)$, $R_{i+k-1}>0$ and (7) is true. Thus once again, one can compute $R_{i+s-t+1}$ and $B_{i+s-t+1}$ directly from R_{i-1} and B_{i-1} as

$$R_{i+s-t+1}=2^{2(s-t+2)}R_{i-1}+[2^{s-t+4}B_{i-1}+2^{s-t+3}-1] \tag{8}$$

$$B_{i+s-t+1}=2^{s-t+2}B_{i-1}+(2^{s-t+1}-1) \tag{9}$$

Thus even in this case, one can predict $(s-t+2)$ consecutive bits of B as a string made up of a 0 followed by $(s-t+1)$ 1's at the cost of one addition required to obtain $R_{i+s-t+1}$ from R_{i-1}.

The results of this section lead to the square-root algorithm shown in Fig. 1. Note that when the conditions of neither of the two cases discussed above apply, we merely resort to the usual steps (2) and (3) of the non-restoring square-root algorithm. It can be seen that in each pass of the algorithm, one performs exactly one addition to obtain a new R value. (The bracketed terms form a single operand of this adition.) The modification of B can be interpreted as left shifts of B with 0's or 1's entering from right.

Note that the two cases that allow us to predict $s-t+2$ bits of the square root in one addition require that $t\leqslant s$. If the most significant bit of R_{i-1} is at a higher position than that of B_{i-1}, then one would not be able to take advantage of this prediction technique.

However, as is shown by the following theorem, the magnitude of R_{i-1} always lies between 0 and $4B_{i-1}+3$. Thus there is a high probability that $t\leqslant s$.

Theorem 1

In the non-restoring algorithm described by (1), (2) and (3); for any $i\geqslant 1$,

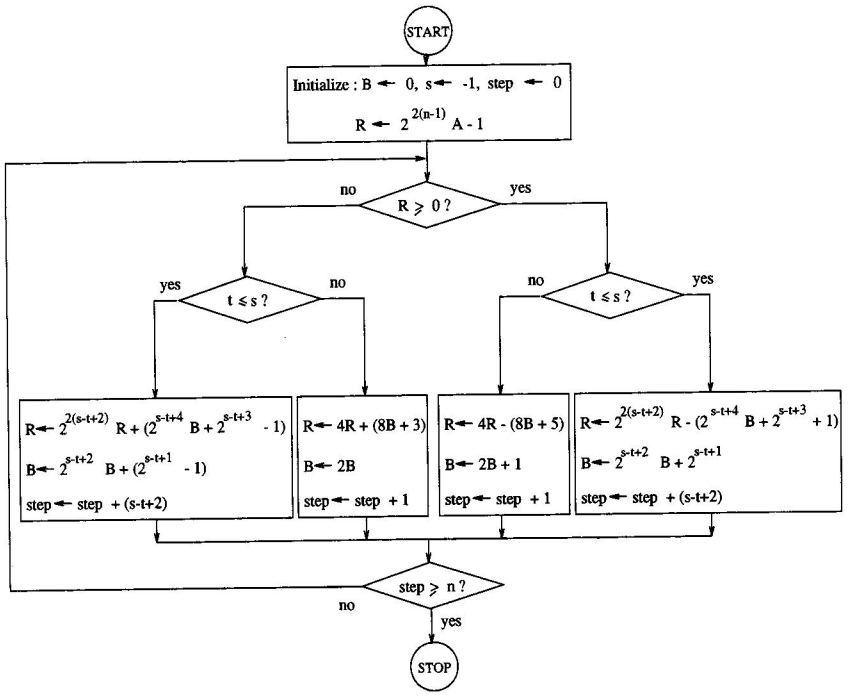

Figure 1. Square root algorithm based on prediction of answer bits.

$$-(4B_{i-1}+1)\leqslant R_{i-1}<4B_{i-1}+3 \tag{10}$$

Proof (*by mathematical induction*)

For $i=1$, (1) shows that (10) is true. If (10) is true for any $i=k$, its truth for $i=k+1$ is established thus:

For non-negative R_{k-1}, application of (2) gives

$$R_k=4R_{k-1}-(4B_k+1) \tag{11}$$

But since (10) is assumed true for $i=k$,

$$0\leqslant R_{k-1}<4B_{k-1}+3=2B_k+1 \tag{12}$$

Combining (11) and (12) one obtains

$$-(4B_k+1)\leqslant R_k<4B_k+3$$

which shows that (10) is also true for $i=k+1$.

Similarly for negative R_{k-1}, from (3) one obtains

$$R_k=4R_{k-1}+(4B_k+3) \tag{13}$$

Also since (10) is assumed true for $i=k$,

$$-(2B_k+1)=-(4B_{k-1}+1)\leqslant R_{k-1}<0 \tag{14}$$

Bits inspected			Operations for each step		
n_1	m_2	m_3	R	B	FF
0	0	0	$R \leftarrow 4R$	$B \leftarrow 2B + \overline{FF}$	1
0	0	1	$R \leftarrow 4R - (8B + 1 + 4\overline{FF})$	$B \leftarrow 2B + \overline{FF}$	0
0	1	0	$R \leftarrow 4R - (8B + 5)$	$B \leftarrow 2B + \overline{FF}$	0
0	1	1	$R \leftarrow 4R - (8B + 5)$	$B \leftarrow 2B + \overline{FF}$	0
1	0	0	$R \leftarrow 4R + (8B + 3)$	$B \leftarrow 2B + FF$	0
1	0	1	$R \leftarrow 4R + (8B + 3)$	$B \leftarrow 2B + FF$	0
1	1	0	$R \leftarrow 4R + (8B + 3 + 4FF)$	$B \leftarrow 2B + FF$	0
1	1	1	$R \leftarrow 4R$	$B \leftarrow 2B + FF$	1

Table 1. Synchronous modifications to R, B, and the FF based upon m_1, m_2, and m_3.

Combining (13) and (14) one can see that (10) is true for $i = k+1$. □

3. Implementation of the algorithm

The square-root algorithm presented in §2 is attractive because it can predict several bits of the result in a single cycle using one addition. An efficient implementation of this idea is, however, not trivial. In order to determine the type of modifications to be done to R and B, one should know the sign of R and the value of $s-t$. The sign of R can be determined merely by examining its most significant bit, a signal denoted by m_1 in the following discussion.

The determination of $s-t$ is a difficult task. We therefore break up each cycle of modifying R and B into several steps. In each step we append a single bit to B. Thus, for example, a cycle with $R \geqslant 0$ on which one can predict $s-t+2$ bits of B is carried out through $s-t+2$ steps; appending a 1 to B in the first step and 0's in the rest. Together, these steps imply $B \leftarrow 2^{s+t+2}B + 2^{s-t+1}$ as specified by the algorithm of Fig. 1. In order to distinguish the first step of a cycle from the rest of the steps, we use a flip-flop (FF) which holds a 0 during the first step of each cycle and a 1 during the rest of steps of that cycle.

Consider a cycle which begins when the leading 1 of B is at position s_0. Note from (10) that the leading significant bit of R cannot be at a position higher than s_0+2. Thus all the bits of R at positions s_0+3, s_0+4, ... are identical to signal m_1 (representing the sign of R) and provide no additional information. We create signals m_2 and m_3 by tapping bits of R at positions s_0+2 and s_0+1, respectively, at the beginning of a cycle and then moving the taps gradually to higher significant bits during each step of the cycle. It can be shown that signals m_1, m_2 and m_3 are sufficient to decide the operations at each step.

Table 1 tabulates the modifications done to R and B at each step based upon $\langle m_1 m_2 m_3 \rangle$ at that step. We now show for $R \geqslant 0$ that the combined effect of the actions in Table 1 is the same as that of the algorithm in Fig. 1. The case for $R < 0$ can be similarly analysed.

Case 1: $R \geqslant 0$ and $t_0 \leqslant s_0$ where t_0 and s_0 are the significant bit positions of R_0 and B_0, the values of R and B at the beginning of the cycle.

Note that when $s_0 - t_0 \geqslant 0$, m_2 and m_3 are identical to the sign bit of R_0 and $\langle m_1 m_2 m_3 \rangle$ equals $\langle 000 \rangle$. From Table 1, this would prompt the modifications