ALGORITHM for HIGH SPEED SHARED RADIX 4 DIVISION and RADIX 4 SQUARE-ROOT

Jan Fandrianto

Weitek Corporation
Sunnyvale, California 94086

ABSTRACT

An algorithm to implement radix four division and radix four square-root in a shared hardware for IEEE standard for binary floating point format will be described. The algorithm is best suited to be implemented in either off-the-shelf components or being a portion of a VLSI floating-point chip. Division and square-root bits are generated by a non-restoring method while keeping the partial remainder, partial radicand, quotient and root all in redundant forms. The core iteration involves a 8-bit carry look-ahead adder, a multiplexer to convert two's complement to sign magnitude, a 19-term next quotient/root prediction PLA, a divisor/root multiple selector, and a carry save adder. At the end, two iterations of carry look-ahead adder across the length of the mantissa are required to generate the quotient/root in a correctly rounded form. Despite its simplicity in the hardware requirement, the algorithm takes only about 30 cycles to compute double precision division or square-root. Finally, extending the algorithm to radix eight or higher division/square-root will be discussed.

INTRODUCTION

Division and square-root are among the required operations in IEEE floating point standard. Using Newton-Ralphson iteration method to compute these functions [1] does not satisfy the IEEE standard on the accuracy of the final result, even though the iteration has the speed advantage (converges quadratically) over many other algorithms, which often converge linearly. To obtain a correctly rounded number as specified by IEEE, hardware implementations of these two functions are generally done in a linear iterative algorithm, where one or more quotient/root bits are produced in each iteration. Naturally, the algorithm that can produce more and more bits per iteration, and/or can shorten the time to do one iteration, will give a better overall performance, but, usually at the expense of more complexity in the implementation. Higher radix non-restoring division (SRT) explained in a classic paper by Atkins [2] has been the basis of many hardware solutions. Radix 8 SRT division has been implemented in a VLSI chip - Weitek WTL2264 [3]. Restoring radix four division shared with radix two square-root has been built and reported by Taylor [4]. Non-restoring radix two square-root is described by Majerski [5]. The algorithm described in this paper will show how a radix four square-root fits nicely into a radix four SRT division hardware; resulting in a square-root performance that matches exactly with the divide performance.

ALGORITHM AND IMPLEMENTATION

This paper will not discuss how the result exponent is obtained since it is quite trivial to implement. It will emphasize on how the quotient and root bits are formed as well as the associated rounding technique at the end.

DIVISION

In this paper, division is implemented with radix 4 SRT : 2 quotient bits are produced per iteration; the partial remainder is kept in a redundant form : the sum and carry from carry save adders; the quotient bits are kept in a redundant form (Q and Q-1 form); and the next quotient bits are predicted using a symmetric prediction PLA.

Basic SRT division algorithm [2] is solving the recursive relationship :

$$p_{j+1} = r * p_j - q_{j+1} * d \qquad (1)$$

with the range restriction :

$$| p_{j+1} | <= \frac{n}{r-1} * d \qquad (2)$$

where

pj = partial remainder in j-th cycle
(po = dividend)
r = radix
qj = quotient digit selected in j-th cycle
d = divisor
n = number of divisor multiples
(not including zero)

The P-D (Partial Remainder - Divisor) Plot with r = 4 and n = 2 is shown in figure 1. To find the boundaries defining each redundant region, substitute the above equation with the minimum and maximum of p(j+1) :

$$r * p_j = \pm \frac{2}{3} * d - q_{j+1} * d \qquad (3)$$

with q(j+1) = -2, -1, 0, +1, +2

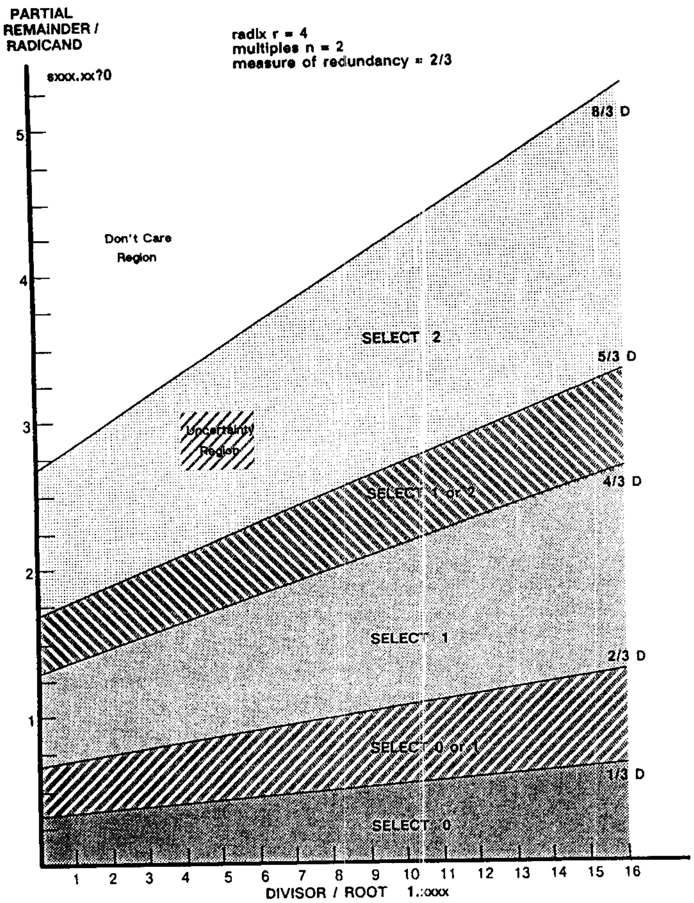

Fig 1 SRT DIVISION / SQUARE-ROOT GRAPH OF REDUNDANCY (P-D PLOT)

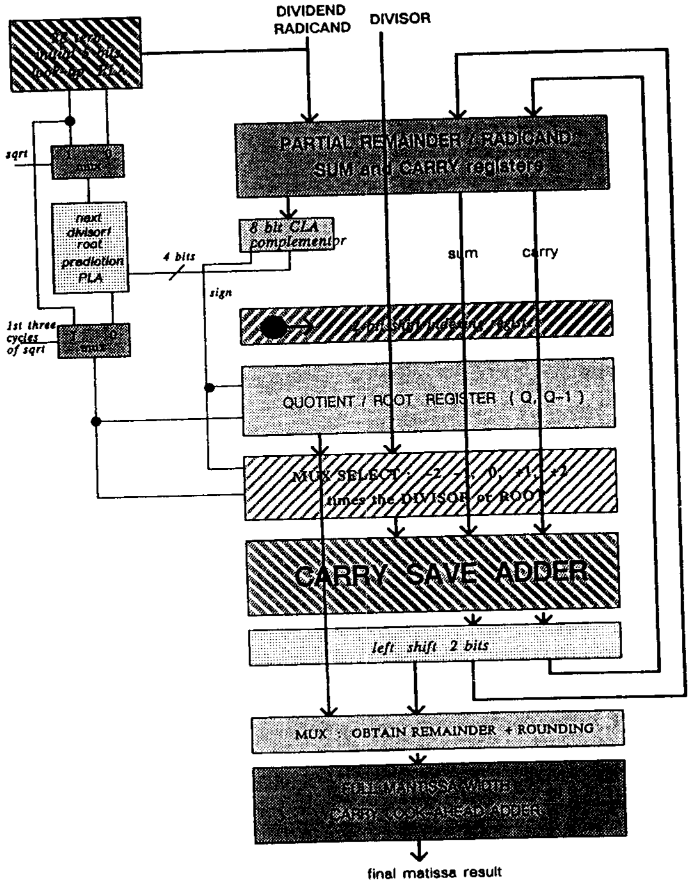
DIVIDEND RADICAND
DIVISOR
sqrt
1 0
mux
next divisor/ root prediction PLA
4 bits
sign
1st three cycles of sqrt
1 0
mux
PARTIAL REMAINDER / RADICAND SUM and CARRY registers
8 bit CLA complementor
sum
carry
QUOTIENT / ROOT REGISTER (Q, Q-1)
MUX SELECT : -2, -1, 0, +1, +2 times the DIVISOR or ROOT
CARRY SAVE ADDER
left shift 2 bits
MUX : OBTAIN REMAINDER + ROUNDING
FULL MANTISSA WIDTH CARRY LOOK-AHEAD ADDER
final matissa result

Fig 2 BLOCK DIAGRAM of SHARED RADIX-4 DIVISION and SQUARE-ROOT

This radix four SRT division is implemented with minimal redundancy (measure of redundancy = 2/3), i.e., with the choice of divisor multiples to be times one and times two of the divisor only. Although having a times three of the divisor as a divisor multiple choice (maximally redundant) will simplify the radix four next quotient prediction PLA, the generation of this multiple is non-trivial. This non-trivial generation, however, has been done to implement radix 8 SRT division in Weitek chip WTL2264.

The block diagram of divide operation can be seen from figure 2. The recursive iteration given in (1) causes the selected divisor multiple to be subtracted from the current Partial Remainder (PR) to form the next PR. On each iteration, the subtraction is done in parallel bit by bit using Carry Save Adders (CSA), instead of performing a full mantissa width Carry Look-ahead Adder (CLA). Consequently, the iteration cycle time can be made much shorter, and also the cycle time will become relatively independent of the length of the mantissa precision. The CSA technique causes the PR to be in redundant sum and carry forms. This creates extra hardware complexities. First, an extra set of register is needed to hold the carry bits. Second, the PR still needs to be converted into a non-redundant form in every cycle to predict the next quotient digit, but, due to the size of the comparison constant in the overlapped redundant region of figure 1, a small 8 bit CLA on the upper bits is quite sufficient. Third, at the end of divide iterations, when the actual sign of the PR and the sticky bit need to be known, a full mantissa width CLA is needed to convert the PR into a non-redundant form. And last, a slightly more complex next quotient prediction PLA is needed. Obviously, the trade-off here is hardware complexity versus speed. For VLSI implementation, the extra complexity is minimal indeed; moreover, the speed advantage becomes more apparent for longer mantissa width (e.g. double precision, extended precision).

As the quotient digit is produced every cycle, conventionally the redundant digits are stored in positive and negative shift registers. The final quotient is then obtained by subtracting the content of the negative from the positive registers using a full mantissa width CLA. The algorithm described here keeps these redundant quotient digits in a slightly modified form, i.e. the "Q" and "Q minus 1 (Q-1)" form (figure 3). The bit strings are converted into Q and Q-1 forms sequentially as the positive/negative quotient digits are generated every cycle.

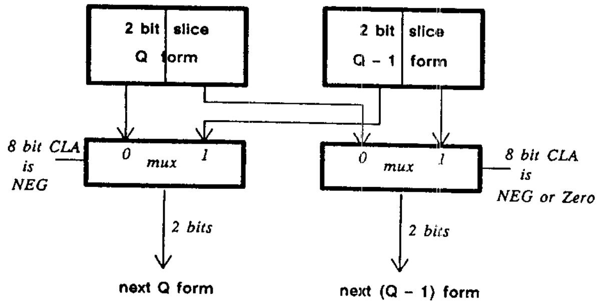

Fig 3. QUOTIENT / ROOT in Q and Q - 1 form

One advantage of this technique is that the final quotient may be readily available at the end of the iterations, thus eliminating the need to perform a full mantissa width CLA. The other is that it shares well with the root formation in the square-root. The disadvantage is almost none.

To summarize the total delay for each divide iteration cycle : At the beginning of each cycle, an 8 bit CLA is performed on the sum and carry forms of the current PR, and the result is then converted from two's complement into sign magnitude (figure 4). The magnitude part is then passed to the 19 term "next quotient prediction PLA (table 1)". The PLA outputs then select the divisor multiple. In parallel, these PLA outputs are transformed into Q and Q-1 forms and stored into the quotient registers. Using CSA, the selected divisor multiple is subtracted from the current PR to form the next PR. The result is stored in the PR registers, and the cycle repeats.

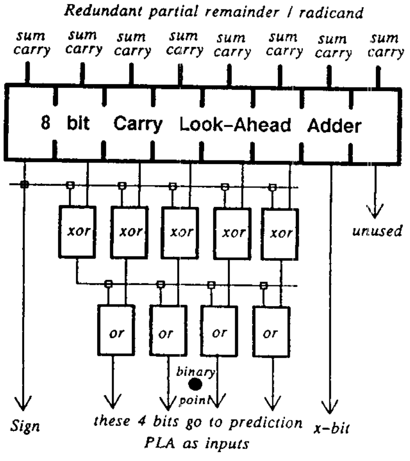

Fig 4. 8 BIT CLA and Converter to SIGN MAGNITUDE

SQUARE-ROOT

A close analog to division, the radicand is similar to the dividend, the partial radicand is to the partial remainder, the root is to the divisor, and also the root is similar in formation as the quotient. Square-root iteration requires the root bits to be known each time, because the root bits must be available to form the next partial radicand. Naturally, a one bit at a time restoring binary square-rooting is the simplest to implement, since the non-redundant root is readily formed and can be used directly in the subtraction of the current Partial Radicand (PR) to form the next PR. A redundant one bit at a time is slightly more complex, since the redundant root bits must be converted on the fly every time into a non-redundant form [5].

An extension of restoring one bit at a time square-root into a higher radix is possible by performing parallel CLAs on the full precision PR based on all possible root combinations. For instance, restoring radix four square-root would require 3 parallel CLAs to obtain comparisons on {01, 10, 11} possible next root digit combinations. This approach requires a lot of hardware and the iteration cycle time will be long due to the delay in performing full width CLA.

Higher radix non-restoring square-rooting process with redundant root selection has the speed advantage because redundancy permits less number of partial radicand bits to be compared. Therefore, a shorter CLA, independent of the length of the mantissa precision, is sufficient to predict the next root digit.

The binary square root extraction is based on "completing the square" [6]

$$p_{j+1} = p_0 - Q_{j+1}^2$$

$$p_{j+1} = r * p_j - q_{j+1} 2^{-(j+1)} * (2Q_j + q_{j+1} 2^{-(j+1)}) \quad (4)$$

with the range restriction :

$$| p_{j+1} | <= \frac{n}{r-1} * Q \quad (5)$$

where p_j = partial radicand in j-th cycle
(p_o = radicand)
r = radix
q_j = root digit selected in j-th cycle
Q_j = partially developed root in j-th cycle
(Q = final root)
n = number of root multiples
(not including zero)

Thus the recursive relationship for square-root differs from the SRT division recursion only in the formation of the divisor/root multiples, and in the range restriction given in (5) that affects the size of the uncertainty region in the next root prediction PLA. Consequently, the potential of sharing the same logic and hardware with radix four division is enormous, and the performance speed-up is very attractive.

DIVISOR / ROOT $1\frac{x}{16}$

partial remainder radicand \ x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00.01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00.10	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
00.11	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
01.00	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
01.01	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
01.10	2	A	B	1	1	1	1	1	1	1	1	1	1	1	1	1
01.11	2	2	2	2	A	1	1	1	1	1	1	1	1	1	1	1
10.00	2	2	2	2	2	2	2	A	1	1	1	1	1	1	1	1
10.01	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1
10.10	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
10.11	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
> 11.00	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Note : There are 3 boxes of A : Normally = 2,
but if negative and xbit = 1 -> 1

There is one box of B : Normally = 1,
but if positive and xbit = 1 -> 2

Total number of terms : 19 terms

TABLE 1. NEXT DIVISOR / ROOT SELECTION PLA

TABLE 2

Look Up PLA for Prediction of the first 5 root bits

```
Inputs  = 7 bits : x[52] x[51] x[50] x[49] x[48] x[47] x[46]
Outputs = 5 bits : r[4] r[3] r[2] r[1] r[0]
Output Polarity  : Negative true
Terms   = 28

001000- 00010        00010-0 00001        000100- 00001
-000-00 01000        0-1111- 00011        -01010- 00001
00001-1 00111        10011-- 00001        000011- 00111
01111-- 11110        0010-1- 00101        01-11-- 00001
-100-0- 00001        01-00-- 00001        111-1-- 01001
1111--- 01001        -0000-- 01000        11-00-- 01010
1100--- 00001        10-00-- 00001        1-00--- 00010
-01--0- 00100        0001--- 00110        -01-0-- 00100
--010-- 00010        -100--- 00010        110---- 01010
10----- 01100
```

Previous Root : 0.Q or 0.(Q-1)

Select	ROOT MULTIPLES (magnitude)
0	0 . 0 0 0 0 0 0 0
+1	0 . 0 0 0 Q 0 0 1
-1	0 . 0 0 0 Q-1 1 1 1
+2	0 . 0 0 Q 0 1 0 0
-2	0 . 0 0 Q-1 1 1 0 0

Table 3. CHOICE of ROOT MULTIPLES

Refer to figure 2 for the operation diagram of radix four square-root. Prior to initiating the square-root iterations, the exponent of the radicand must first be examined. If the unbiased exponent is odd, the mantissa is left shifted by one bit. In order to start predicting the next 2 root bits, the root prediction PLA needs to know approximately where the root is. As shown in figure 1 on the x-axis, five bits of the root (including the hidden bit) must be known. There are many ways to obtain the initial 5 bits of the root. A look-up PLA is chosen to do this. Table 2 shows the look-up PLA table. The look-up PLA size is 28 terms with the most significant 6 bits of the mantissa plus the LSB of exponent (the odd and even exponent bit) as the inputs and the most significant 5 bits of the root as the outputs. Reducing the complexity of this initial look-up PLA is an interesting logic minimization problem.

Three initial square-root iterations will be performed using the 5 predicted root bits to obtain the partial radicand. Subsequently, the next 2 root bits will be produced by the root selection PLA on every iteration - a similar iterative process as done in division.

One tricky point in performing higher radix square-root is the generation of the root multiples (plus or minus times one of the root and plus or minus twice the root). Using equation (4), the root multiple can be calculated. Table 3 shows the root multiples to be selected. Since forming root multiples requires immediately the non-redundant form of the partially developed root (Qj), the advantage of storing the quotient/root digit string in "Q" and "Q-1" forms becomes clear. If the next root digit is positive, the "Q" form is used to generate the root multiple. The "Q-1" form is used if the next root digit is negative.

Again, similar to division, the delay path on each square-root iteration is : an 8 bit CLA is performed on the sum and carry of the current Partial Radicand (PR), and the 2's complement result is converted into a sign magnitude. The magnitude part together with 4 initial root bits (out of 5 bits excluding the hidden bit that are obtained from the look-up PLA) go to the next root prediction PLA (19 terms) to determine the next two root bit and the next root multiple. The sign determines whether subtraction or addition must be done to form the next PR. The cycle then repeats.

Two examples of double precision square-rooting process using this algorithm is included in the appendix, showing how the partial radicand registers change on every iteration cycle.

UNCERTAINTY REGION of the NEXT QUOTIENT / ROOT PREDICTION PLA

The P-D plot in figure 1 shows on the X-axis that the divisor/root is in the range of +1 to +2. This is indeed dictated by the IEEE standards : the hidden bit is to the left of the binary point, and the mantissa is represented as a positive magnitude. Let us examine the uncertainty range on the X-axis for each divide and square-root.

For division, the divisor is already known to its full precision, its position on the X-axis is fixed. Through trial and error on the uncertainty region, only 4 MSB of the divisor (excluding the hidden bit) are chosen to be examined. This truncated divisor-bits has an uncertainty range of the size at most +1 Unit of the Least significant Place (ULP). In this case, the uncertainty range for the divisor is 1/16, which satisfies the divisor interval requirement. In the case for square-root, initially, only the radicand is known; and the root position in the range of +1 to +2 is not yet known. Using several upper bits of the radicand, it is possible to obtain the upper 5 root bits (including the hidden bit). A look-up PLA implementation is quite practical, since the size is not too large (28 terms). Since subsequent root digits to be produced are still redundant, this truncated root-bits has an uncertainty range extending to not only +1 ULP (+1/16) but also -1 ULP (-1/16). The uncertainty range for the root is twice the divisor, therefore the final size of the uncertainty range on the X-axis for the shared prediction PLA is 1/8; even though, the stepping size on the X-axis is still 1/16 (4 MSB examined).

The Y-axis of the P-D plot from figure 1 represents the partial remainder/radicand (PR). The plot only shows the positive half of the Y-axis. The negative half is symmetrical to the positive half around zero. Although at the start, the dividend/radicand is positive, subsequent non-restoring subtractions may produce negative PRs. Unfortunately, the two's complement representation for the PR is asymmetrical around zero. On the other hand, the overlapped region in figure 1 allows the comparison constant to be truncated to 8 MSB of the PR. The truncated PR, while represented in two's complement, will have +2 ULP uncertainty range, because the PR is truncated from its original sum and carry representation. The uncertainty range of the magnitude representation of the PR thus extends to +2 and -2 ULP. When the selection of the next quotient/root digit is done from two's complement PR, the result is an asymmetrical PLA [4]. The size of this asymmetrical PLA is quite large, because not only the PLA has to cover the positive and negative regions of figure 1, but also the negative half does not map nicely into the positive half. Therefore, if the negative two's complement truncated PR is first mapped into a sign magnitude representation, a smaller size symmetrical PLA can be utilized. The logic for this conversion is shown in figure 4. This conversion technique not only maps the negative half of the PR into its symmetrical counterpart, but also causes the uncertainty range on the magnitude of the PR to reduce to +1.25 ULP and -0.25 ULP.

To a certain extend, if the size on the X-axis of the uncertainty region is increased, the size on the Y-axis may be reduced [2]. The truncated divisor/root bits that go to the PLA does not change on every iteration, but the truncated PR is always changing. Since the formation of the truncated PR involves a CLA delay (this delay depends on the width of the truncated PR), it is desirable to examine fewer number of PR bits, even at the expense of examining more divisor/root bits. The size chosen for the uncertainty region in this paper is indeed an optimal one.

ROUNDING

In order to comply with IEEE specified rounding, both divide and square-root iterations must be continued until the quotient bits at Guard and Round positions are produced. The un-rounded quotient bits must be selected from Q or Q-1 registers. The selection is determined by the sign of the last PR, thus a full mantissa width CLA must be performed on the last PR to compute its sign.

The sticky bit is zero if and only if the final partial remainder/radicand is exactly zero; thus a big OR gate following the CLA can be used to perform a zero detect logic. A one bit normalization on the mantissa may be needed for divide operation if the bit value at the hidden bit position of the quotient is zero. Square-root operation never needs normalization.

Finally, depending on the rounding mode, the LSB, the Round bit and the Sticky bit, the quotient/root may need to be incremented at the LSB position. If so, again, performing a full mantissa width CLA will add one on the least significant bit position of the mantissa. Increment the exponent if the mantissa overflows due to rounding.

RADIX 8 OR HIGHER
SHARED DIVISION AND SQUARE-ROOT

Staging several hardware with each performing radix 4 division/square-root in parallel can achieve a higher radix implementation. Radix 16 SRT division with stages has been reported by Taylor [7]. The discussion here pursues implementations of higher radix not by means of staging, rather by means of one stage hardware.

The algorithm explained in this paper can be extended to radix 8 or higher. SRT division algorithm can be applied to any radix (2^n); and the recursive relationship for square-root (4) can still map into the SRT recursion. The major obstacles to higher radix than radix four division and square-root are the non-trivial generation of divisor/root multiples (e.g. 3d, 5d, 7d, etc) and the complexity of the next quotient/root prediction PLA. All these requirements translate directly to more hardware.

For division, the divisor multiples generation may be formed at the beginning of the divide iterations - a one time delay. But for square-root, the non trivial generation of the root multiples must be formed on the fly on every iteration. Utilizing the properties of signed digit representation [8][9] on the quotient/root, and using modified carry save adders, it is possible to generate the root multiples on the fly. And naturally, divisor multiples can also be formed using the same hardware.

Complex prediction PLA can be simplified by transforming the divisor into a suitable range [10]. Transforming the divisor into a different range does not pose much side effect on the quotient, given that the dividend is also transformed by the same factor. In a shared division/square-root method, the root must also be modified through the same transformation scheme as the divisor. This requires the final result for the root to be modified back by the inverse factor.

CONCLUSION

In this paper, a unique algorithm to share a radix four division and a radix four square-root has been demonstrated. In addition to the obvious performance advantage with higher radix, the hardware solution for this algorithm is relatively simple. Figure 2 shows the block diagram of shared division and square-root hardware. Extending this shared division and square-root algorithm to higher radix than radix four is possible with more complex hardware requirements.

ACKNOWLEDGEMENTS

The author is grateful to Selfia Halim and George S. Taylor for their incisive comments.

APPENDIX

```
ODD EXPONENT
What is the radicand (two 32bit integers) ?
          40023456 789abcde

predicted root bits :0110000
sel2=1 sel1=0 negate=0
sum :0010010001101000101011001111000100110101011110011011110000
cry :0000000000000000000000000000000000000000000000000000000000000000
root:1011111111111111111111111111111111111111111111111111111111111111
sel2=1 sel1=0 negate=1
sum :0110111001011101010011000011101100101010000110010000111100
cry :0010001101000101011001111000100110101011110011011110000100
root:0111000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=0 negate=0
sum :1111010001100000101011101100101000000111010100111011100000
cry :0001001000101010001000000100100101010000010010000000100000
root:1111111111111111111111111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :0110011011010101110001011111001010100011100100010011111100
cry :1011001101010101011101100101101010111010110111011100000100
root:1100111110111111111111111111111111111111111111111111111111111111
sel2=1 sel1=0 negate=1
sum :0110100011111101001100010101111110011010110011000000001100
cry :0011111010101111101111111101010111011110111011111111100100
root:0110000011000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=0
sum :1101101001001010001110100010100100010000100011111111100000
cry :0100011101101001100010101010110011010110011000000000100000
root:1100111110111011111111111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=1
sum :0100101001100001001111011110100011100100010000000011111100
cry :0111101101011101110101010110111010110111011111111100000100
root:0011000001000111000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :0000010111101111101000100001100101001100111111111111100000
cry :1101001000101000101010110100010100100010000000000000100000
root:0011000001000101110000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=0
sum :1001111000001011001001010111000110111011111111111100000000
cry :1000001101101101000100000000100000000000000000000100000000
root:1100111110111010011011111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=1
sum :0100101101110001011010100001100100010000000000001111111100
cry :0111100101011001001010111100110111011111111111110000000100
root:0011000001000101100111000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :0000100110110111011101110101001100111111111111111111100000
cry :1100101010001001010100000100100010000000000000000000100000
root:0011000001000101100101110000000000000000000000000000000000000000
sel2=0 sel1=0 negate=0
sum :1100110111101110110000000110111011111111111111111100000000
cry :0100010000101010101110100000000000000000000000000100000000
root:1111111111111111111111111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :1101100011101110000101100100010000000000000000001111111100
cry :0110111101110111110100110111011111111111111111110000000100
root:1100111110111010011010011110111111111111111111111111111111111111
sel2=1 sel1=0 negate=0
sum :1110000010001110101100110111000000000000000000000000011100
cry :0111111111110010100110110011111111111111111111111111100100
root:1001111101110100110100111010111111111111111111111111111111111111
sel2=0 sel1=0 negate=1
sum :0000000000100011111011111100000000000000000000000000000000011100
cry :1111111110110100100110011111111111111111111111111111111111100100
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=0 negate=1
sum :1111111001011101110110011111111111111111111111111111111111100000
cry :0000000100000100010011000000000000000000000000000000000000100000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=0 negate=1
sum :1111110101100110010101111111111111111111111111111111111100000000
cry :0000000000100010010000000000000000000000000000000000000100000000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=0 negate=1
sum :1111010100010000010111111111111111111111111111111000000000000
cry :0000000100010010000000000000000000000000000000000100000000000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :1101000000001001011111111111111111111111111111100000000000000000
cry :0000100010000000000000000000000000000000000000100000000000000000
root:0011000001000101100101100010111111111111110000000000000000000000
sel2=0 sel1=1 negate=0
sum :1010001100110011101001110100000000000000011100000000000000000000
cry :1000000000001000101100010111111111111100010000000000000000000000
root:1100111110111010011010011101000000000000011011111111111111111111
sel2=0 sel1=1 negate=1
sum :1011001000000101111111111011111111110101011111111111111111100
cry :0001100111010101000010101000000000000011000000000000000000000100
root:0011000001000101100101100010111111111110011100000000000000000000
sel2=0 sel1=0 negate=1
sum :0110111001010101100011000100000001100010001111111111100000
cry :1000001000101100111101010111111101110011100000000000000100000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :1011000111100101111001001111110001000110111111111100000000
cry :0001000000100100001000100000000110001000000000000000100000000
root:0011000001000101100101100010111111111110010111111000000000000
sel2=0 sel1=1 negate=1
sum :0100011000010001010000110100001010100010100000010000000000000
cry :1000001100101101001100010111101010110010111110100000000000000
root:0011000001000101100101100010111111111110010111011100000000000
sel2=0 sel1=1 negate=1
sum :1101010111100111100100000101111110111000100101110000000000000
cry :0001000000101000100110110101010110010110110100000000000000000
root:0011000001000101100101100010111111111110010111010111000000000
sel2=1 sel1=0 negate=0
sum :1101011000101010011101001001011101000000011010011100000000000
cry :1000001100101100100100101111110111110110101010000000000000000
root:1001111101110100110100111010000000000001101000101001011111111
sel2=0 sel1=1 negate=1
sum :0010100111001000110101110010101011010110000100111011111100
cry :1011100101100110100101011010101000010011010010000000000000100
root:0011000001000101100101100010111111111110010111010110111100
sel2=1 sel1=0 negate=0
sum :1000001110101111010100101011110011101100000110110100010000
cry :1100101000100100101110010101011010110010110010010111100000
root:1001111101110100110100111010000000000001101000101001000101 1

root:0011000001000101100101100010111111111110010111010110111011
Radicand    = 40023456 789abcde
Final root  = 3ff822cb 17ff2eb7


EVEN EXPONENT
What is the radicand (two 32bit integers) ?
          3ffabcde 98765431

predicted root bits :0101010
sel2=0 sel1=1 negate=0
sum :0001101010111100110111101001100001110110010101000011000100
cry :0000000000000000000000000000000000000000000000000000000000000000
root:1110111111111111111111111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :1101010100001100100001011001111000100110101011110011101100
cry :0101010111100110111101001100001110110010101000011000100100
root:1101101111111111111111111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :0110110001010110001110101000100110101111110001010011011100
cry :1010111011101111010111011111101101101010111110111101100100
root:1101011011111111111111111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=1
sum :0101011101111001101011100010111110010101000111000100011100
cry :0111001110111101111101111110110111111111111101111111100100
root:0010100111000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :0011010000010001011001110000100110101011110011101111100000
cry :1001111111001101001100010110110010101000011000100000100000
root:0010100101110000000000000000000000000000000000000000000000000000
sel2=0 sel1=0 negate=1
sum :0000101010110001010110011001010000001110101100111100000000
cry :1110101010001001000010000100010101000010000100000100000000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :1000000011100001010001110100010100110010100011100000000000
cry :0101010000000100001000000000100000000010000100000100000000000
root:0010100101011111000000000000000000000000000000000000000000000000
```

```
sel2=0 sel1=0 negate=1
sum :1111011011011000000111011001010010001000001100000000000000000000
cry :0000001001001010000000000000000000001000010000001000000000000000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :1101001001001000011101100101000000110000100000000000000000000000
cry :0001001001000000000000000000000010000000000100000000000000000000
root:0010100101011101111110000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=0
sum :1010010101010110000110010101000011000000000000000000000000000000
cry :1001001001000011100000000000000000000010000000000000000000000000
root:1101011010100010000110111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :1000011011011110000010101011110011101111111111111111111111111100
cry :1011001000010000110010101000011000100000000000000000000000000100
root:1101011010100010000101101111111111111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :1000100110110011010110110001010011000000000000000000000000011100
cry :1011010010010000010101011111011101111111111111111111111111100100
root:1101011010100010000101011011111111111111111111111111111111111111
sel2=1 sel1=0 negate=0
sum :1010111000000100011011010111000100000000000000000000000000011100
cry :1010010110010010101011011011111111111111111111111111111111100100
root:1010110101000100001010101011111111111111111111111111111111111111
sel2=0 sel1=0 negate=0
sum :1001101101001011101010011100010000000000000000000000000000011100
cry :0110100000100001011011011111111111111111111111111111111111100100
root:1111111111111111111111111111111111111111111111111111111111111111
sel2=0 sel1=0 negate=0
sum :0011001001010100111011110001000000000000000000000000000000011100
cry :1101101101011111011011111111111111111111111111111111111111100100
root:1111111111111111111111111111111111111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :0101101111010001111110001000000000000000000000000000000000011100
cry :1101101011111111011111111111111111111111111111111111111111100100
root:1101011010100010000101010011111110111111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :0101111000110010010110100000000010000000000000000000000000011100
cry :1101011110011011111010111111110111111111111111111111111111100100
root:1101011010100010000101010011111101101111111111111111111111111111
sel2=0 sel1=1 negate=0
sum :0111110000101110100100110000111001000000000000000000000000011100
cry :1011010110010010110110011110101101111111111111111111111111100100
root:1101011010100010000101010011111101011011111111111111111111111111
sel2=0 sel1=1 negate=0
sum :0111110001111001011111110110100110010000000000000000000000011100
cry :1010010100010100100010010111101011011111111111111111111111100100
root:1101011010100010000101010011111101010110111111111111111111111111
sel2=0 sel1=1 negate=1
sum :0011111100111111100011001011000000110010000000000000000000011100
cry :1010000110000000111010111011110101101111111111111111111111100100
root:0010100101011101111010101100000010101001110000000000000000000000
sel2=0 sel1=1 negate=0
sum :1101111110001010001101101011100111101000111111111111111100000
cry :0100100011101111010101101000010100101110000000000000000100000
root:1101011010100010000101010011111101010110011011111111111111111
sel2=0 sel1=0 negate=1
sum :0000011100011101110101000000111001000010010000000000011111100
cry :1111010101010000101101011110101101110011011111111111100000100
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=0 negate=1
sum :1100100100110101100001111001010011000100111111111111111100000
cry :0010100010000100101000000101001000010010000000000000000100000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :1000011011000100100111110001101101011011111111111110000000000
cry :0100000000100100000000000100000000000000000000000000100000000
root:0010100101011101111010101100000010101001100111111100000000000
sel2=0 sel1=1 negate=1
sum :1011111011110101110101010110111111001001100000010000000000
cry :0000001000100100010101000000000000100110011111100000000000
root:0010100101011101111010101100000010101001100111110111000000
sel2=1 sel1=0 negate=1
sum :0101011000110001101011101011110010110011100000011100000000
cry :0101001110101110101000100000011001001100111111000000000000
root:0101001010111011110101011000000101010011001111101011000000
sel2=0 sel1=0 negate=1
sum :0101110010010011011001001110111010110001000111011100000000
cry :1001010111011101001101000010001010011101110001000000000000
root:0000000000000000000000000000000000000000000000000000000000000000
sel2=0 sel1=1 negate=1
sum :0010010100111001010000110011000010110011011001110000000000
cry :1010010010001001001000010001010010001000001000000000000000
root:0010100101011101111010101100000010101001100111110100111111

root:0010100101011101111010101100000010101001100111110100111101
Radicand   = 3ffabcde 98765431
Final root = 3ff4aef5 6054cfa8
```

REFERENCES

[1] Hart et.al., "Computer Approximation", New York, John Wiley and Sons, Inc., 1978 edition

[2] D. E. Atkins, "Higher-Radix Division Using Estimates of the Divisor and Partial Remainder." IEEE Transactions on Computers, 17, No 10, October 1968, pp. 925-934

[3] Weitek Corp., "WTL2264/WTL2265 Floating Point Multiplier/Divider and ALU", data sheet, July 1986

[4] G. S. Taylor, "Compatible Hardware for Division and Square Root", Proceedings of the 5th Symposium on Computer Arithmetic, May 1981, pp. 127-134

[5] S. Majerski, "Square-rooting Algorithms for High Speed Digital Circuits." IEEE Transactions on Computers, c34, No 8, August 1985, pp. 724-733

[6] K. Hwang, "Computer Arithmetic: Principles, Architecture and Design", New York, John Wiley and Sons, Inc., 1978

[7] G. S. Taylor, "Radix 16 SRT Dividers with Overlapped Quotient Selection Stages", Proceedings of the 7th Symposium on Computer Arithmetic, June 1985, pp. 64-71

[8] A. Avizienis, "Signed Digit Number Representations for Fast Parallel Arithmetic", IRE Transactions on Electronic Computers, EC-10, September 1961, pp. 389-400

[9] D. E. Atkins, "Design of the Arithmetic Units of Illiac III : Use of Redundancy and Higher Radix Methods", IEEE Transactions on Computers, C-19, No 8, August 1970, pp. 720-732

[10] M. D. Ercegovac, "A Higher Radix Division with Simple Selection of Quotient Digits", Proceedings of the 6th Symposium for Computer Arithmetic, June 1983, pp. 94-98