

# Fast Floating Point Square Root

Thomas F. Hain, David B. Mercer

**Abstract**—Hain and Freire have proposed different floating point square root algorithms that can be efficiently implemented in hardware. The algorithms are compared and evaluated on both performance and precision.

**Index Terms**—algorithms, square root, digital arithmetic, floating point arithmetic.

## I. INTRODUCTION

EXTRACTING square roots is a common enough operation that the function frequently finds the area on modern floating point processors dedicated to it. Unfortunately, the operation is also quite expensive both in time and space. Hain and Freire have proposed alternative algorithms that may prove to be superior to current methods [4].

Newton proposed an iterative method for approximating the root of a function. Given a function  $f$ , its first derivative  $f'$ , and an approximation  $x_i$ , a better approximation can be found by the following equation:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

One function that can be used to find the square root of  $n$  using Newton's method is

$$\begin{aligned} f(x) &= x^2 - n \\ f'(x) &= 2x \\ x_{i+1} &= x_i - \frac{x_i^2 - n}{2x_i} = \frac{1}{2} \left( x_i + \frac{n}{x_i} \right) \end{aligned}$$

However, this formula requires a division every iteration, which is itself quite an expensive operation. An alternative function is

$$\begin{aligned} f(x) &= x^{-2} - n^{-1} \\ f'(x) &= -2x^{-3} \\ x_{i+1} &= x_i - \frac{x_i^{-2} - n^{-1}}{-2x_i^{-3}} = \frac{x_i}{2} \left( 3 - n^{-1} x_i^2 \right) \end{aligned}$$

This function requires only an initial division to calculate  $n^{-1}$ . Iterations only require multiplications and a subtraction.

Square root algorithms based on Newton's iteration con-

verge on the result very quickly in  $O(\log p)$  where  $p$  is the number of bits of precision.

Other algorithms are also in use for calculating square roots, such as Goldschmidt's algorithm [5] used by the TI 8847 [6]. Goldschmidt's algorithm is an iterative algorithm beginning with  $x_0 = n$  and  $y_0 = n$  and then iterating:

$$\begin{aligned} x_{i+1} &= r_i^2 x_i \\ y_i &= r_i y_i \end{aligned}$$

with  $r_i$  chosen to drive  $x_i \rightarrow 1$  resulting in  $y_i \rightarrow \sqrt{n}$ . When implemented, Goldschmidt's algorithm is closely related to Newton's iteration.

In this paper, we compare two algorithms that calculate floating point square root in a way that is easily and efficiently implemented in hardware. Hain's algorithm has only been published as an internal report [3]. This algorithm is compared to a recent and comparable algorithm by Friere [1]. The algorithms are elucidated in Section II. A performance and precision analysis is presented in Section III, and is followed up with an experimental comparison in Section IV. The conclusions are finally presented in Section V.

## A. Representation of Floating Point Numbers

Floating point numbers are distinct from fixed point numbers, which have an implied decimal point (usually at the end of the number in the case of integers), in that they are described by both the digits of the number and the position of the decimal point. In decimal, we would use scientific notation of  $\pm n \times 10^e$ , where  $1 \leq n < 10$  and  $e \in \mathbb{Z}$  is the exponent (not the natural log base). In computers, however, the binary number system is more appropriate, and so the representation is usually of the form  $\pm n \times 2^e$  where  $1 \leq n < 2$ .

Throughout this paper, we will assume an internal floating point representation that uses an exponent base of two and provides convenient separate access to the sign, mantissa, and exponent. When implementation details require, the 32-bit IEEE 754 standard will be used, though the algorithms described here can be modified to work with other floating point representations.

## B. Conversion of Fixed Point Algorithms to Floating Point

The algorithms compared here are first presented as fixed point algorithms, and then it is shown how they can be converted to floating point algorithms. The process of converting the fixed point versions of Hain's and Freire's algorithms to floating point is similar, so the common concepts are presented here rather than having to be repeated for both algo-

Manuscript received March 4, 2005.

T. F. Hain is with the School of Computer & Information Sciences, University of South Alabama, Mobile, AL 36688 USA (phone: 251-460-6390; e-mail: [thain@usouthal.edu](mailto:thain@usouthal.edu)).

D. B. Mercer, is with The SSI Group, 4721 Morrison Drive Mobile, AL 36609 (phone: 251-345-0000; e-mail: [dmercer@alum.mit.edu](mailto:dmercer@alum.mit.edu)).

rithms.

There are three parts to a floating point number: the sign, the exponent, and the mantissa. When computing the square root of a floating point number, the sign is the easiest bit to compute: if the sign of the input is positive, then the square root will be positive; if the sign of the input is negative, then there is no real square root. We will completely ignore the negative case here, as the test is trivial and uninteresting to our analysis.

The exponent is also quite easy to compute.

$$\sqrt{n \times 2^e} = n' \times 2^{\lfloor \frac{e}{2} \rfloor}$$

where both mantissas are greater than or equal to one and less than two. In binary arithmetic,  $\lfloor \frac{e}{2} \rfloor$  is easily computed as  $e \gg 1$ , but unfortunately, in IEEE 754 representation, the exponent is represented in a biased form, where the actual exponent,  $e$ , is equal to the number formed by bits 23–30 minus 127. This means we can't just do a right shift of the exponent, since that would halve the bias as well. Instead, the new exponent in IEEE 754 floating points would be calculated as:

$$e \leftarrow \lfloor \frac{e+1}{2} \rfloor + 63$$

Computing the mantissa of the root is, of course, the main problem being solved by Hain and Freire. However, it is not true in floating point notation that  $\sqrt{n \times 2^e}$  is equal to  $\sqrt{n} \times 2^{\lfloor \frac{e}{2} \rfloor}$ . Instead,

$$\sqrt{n \times 2^e} = \begin{cases} \sqrt{n} \times 2^{\lfloor \frac{e}{2} \rfloor}, & \text{if } e \text{ is even} \\ \sqrt{2n} \times 2^{\lfloor \frac{e}{2} \rfloor}, & \text{if } e \text{ is odd} \end{cases}$$

Since  $2 \leq 2n < 4$ , the fixed point representation of the mantissa of the input,  $x$ , must allow for two bits before the decimal point (when  $e$  is even, the most significant bit would be 0). In IEEE representation, where the mantissa has 24 bits, this means we either have to use 25 bits to represent the mantissa, or drop the least significant bit when the exponent is even so we can right shift a leading 0 into the most significant bit position. Dropping the least significant bit will result in unacceptable accuracy (accurate only up to 12 bits), so we must preserve that least significant bit.

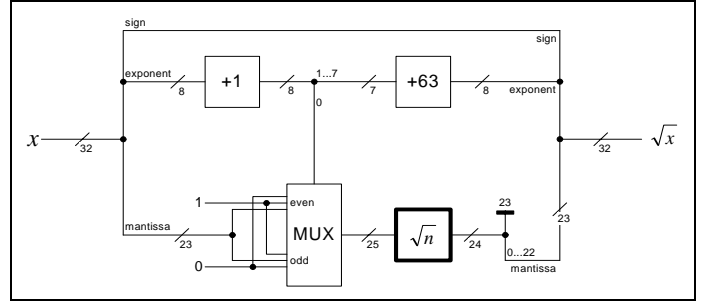
Thus, the fixed point number that will be passed to the two competing algorithms in this paper will be a 25-bit fixed-point number with the implied decimal point between the second and third bits. The expected output will be a 24-bit fixed point number with the implied decimal point between the first and second bits. Note that this is not the only way to generate fixed-point inputs and outputs. One could, for instance, multiply the mantissa by  $2^{23}$  to create an integer—and this is the approach proposed by Freire—but the approach chosen above is easier to implement in IEEE 754 floating point numbers.

Therefore, it can be assumed that fixed point algorithms are wrapped in the following pseudocode to create the floating point algorithms.

```

SQRT(x)
1  e ← bits 23–30 of x
2  n ← bits 0–22 of x, extended to 25 bits by prepending "01"
3  e ← e + 1
4  if e is odd
5    n ← n << 1
  endif
6  n ← f(n), where f is the implementation of the
   square root algorithm
7  e ← ⌊ e/2 ⌋ + 63
8  x ← "0" + 8 bits of e + bits 0–22 of n
9  return x

```



**Figure 1. Hardware to implement a generic square root algorithm on IEEE 754 floats.** An actual implementation of this would have to handle negative, denormalized, infinite, and NaN inputs, but is not shown here.

In hardware, this can be visualized by Figure 1. Note that optimizing this floating point wrapper is not relevant to this study; it is simply important that we use the same wrapper when comparing the two algorithms.

## II. ALGORITHMS

The algorithms presented in this paper are different implementations of fundamentally the same concept.

$$\sqrt{n} = (a \pm b)^2 = a^2 \pm 2ab + b^2$$

where  $a$  is an estimate of the square root, and  $b$  is a trial offset that is a power of two and is successively halved until the desired accuracy is reached. Therefore, each iteration provides us with an additional bit of accuracy.

This is in contrast to the standard Newtonian algorithms, which converge much faster but involve more complex (time-consuming) calculations.

### A. Hain's Algorithm

Hain's algorithm consists of determining the result bit by bit, by beginning with a trial offset,  $b$ , of the most-significant bit of the output. If  $x = n \times 2^p$ , then the initial trial offset is  $b_0 = 2^{\lfloor \frac{p}{2} \rfloor}$ . As each bit is considered, the trial offset is halved, so that  $b_i = 2^{\lfloor \frac{p}{2} \rfloor - i}$ .

At each iteration, the trial offset,  $b_i$ , is added to the root estimate,  $a_i$ , and if  $(a_i + b_i)^2 \leq x$ , then the bit represented by

$b_i$  must be one, so is added to our result.

$$a_i = \begin{cases} a_i & \text{if } x < (a_i + b_i)^2 \\ a_i + b_i & \text{otherwise} \end{cases}$$

Note that the comparison can be written a different way:

$$\begin{aligned} x &< (a_i + b_i)^2 \\ x &< a_i^2 + 2a_i b_i + b_i^2 \\ x - a_i^2 &< 2a_i b_i + b_i^2 \\ \frac{x - a_i^2}{b_i^2} &< \frac{2a_i b_i + b_i^2}{b_i^2} \end{aligned}$$

Hain capitalizes on the fact that that the two sides of the inequality are easier to initialize and update from iteration to iteration than they are to calculate at each iteration. If we call these variables  $s_i$  and  $t_i$  respectively, then  $s_{i+1}$  is calculated as:

$$\begin{aligned} s_{i+1} &= \frac{x - a_{i+1}^2}{b_{i+1}^2} \\ &= \begin{cases} \frac{x - a_i^2}{b_{i+1}^2}, & \text{if } s_i < t_i \\ \frac{x - (a_i^2 + 2a_i b_i + b_i^2)}{b_{i+1}^2}, & \text{otherwise} \end{cases} \\ &= \begin{cases} 4s_i, & \text{if } s_i < t_i \\ 4(s_i - t_i), & \text{otherwise} \end{cases} \end{aligned}$$

and  $t_{i+1}$  is calculated as:

$$\begin{aligned} t_{i+1} &= \frac{2a_{i+1} b_{i+1} + b_{i+1}^2}{b_{i+1}^2} \\ &= \begin{cases} \frac{x - a_i^2}{b_{i+1}^2}, & \text{if } s_i < t_i \\ \frac{x - (a_i^2 + 2a_i b_i + b_i^2)}{b_{i+1}^2}, & \text{otherwise} \end{cases} \\ &= \begin{cases} 2t_i - 1 = 2(t_i - 1) + 1, & \text{if } s_i < t_i \\ 2t_i - 3 = 2(t_i + 1) + 1, & \text{otherwise} \end{cases} \end{aligned}$$

Putting these concepts together into an algorithm, we have Hain's algorithm, SQR\_T\_HAIN.

```

SQR_T_HAIN( $n$ ) [ $n$  is a  $p$ -bit integer]
1   $s \leftarrow 0$ 
2   $a \leftarrow 0$ 
3   $t \leftarrow 1$ 
4  for  $i=1$  to  $\frac{p}{2}$ 
5       $s \leftarrow 4s + \lfloor \frac{n}{2^{p-2i}} \rfloor$ ,  $n \leftarrow 4n \bmod 2^p$  [implemented as
        LSH2  $n$  into  $s$ ]
6      if  $s < t$ 
7           $t \leftarrow t - 1$ 
8           $a \leftarrow 2a$  [implemented as LSH 0 into  $a$ ]
        else
9           $s \leftarrow s - t$ 
10          $t \leftarrow t + 1$ 
11          $a \leftarrow 2a + 1$  [implemented as LSH 1 into  $a$ ]
        end if
12      $t \leftarrow 2t + 1$  [implemented as LSH 1 into  $t$ ]
13 next  $i$ 
14 return  $a$ 

```

### 1) Conversion To IEEE 754 Floats

Conversion of SQR\_T\_HAIN to handle floating point numbers is trivial. Although Hain's algorithm terminates after  $\frac{p}{2}$  iterations, since the integer square root of an integer has half as many bits as the input, the principles of the algorithm hold for nonintegers. That is, we can continue Hain's algorithm to as many bits of precision as desired so long as we know where to place the decimal point when we are done. In this implementation, however, this is easy, since the input is a number between 1 and 4, the result is between 1 and 2 (which is one bit less than the input, which is why the hardware implementation in Figure 2 only contains 24 bits in the output), so we know the decimal point will be between the first and second bits. Of course, after the first  $\frac{p}{2}$  iterations, the two bits we left-shift off  $n$  in step 5 will be zeroes.

Finally, since we know that the first bit of the result will be 1, we can short-circuit the first loop iteration by initializing our variables correctly in steps 1–3. The resulting algorithm for implementation in our IEEE 754 float square root processor is as follows. Note that subscripts indicate bits of the variable, and all registers are 24 bits long. (Although the input  $n$  is 25 bits long, it is immediately cut down to 23 bits, so, in fact, only a 23-bit register is required to hold  $n$ .)

```

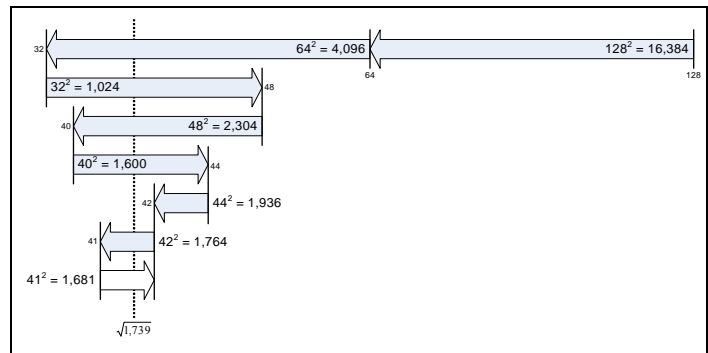
SQR_T_HAIN( $n$ ) [ $n$  is a 25-bit fixed point number  $1 \leq n < 4$  ]
1    $s \leftarrow (n \gg 23) - 1$ 
2    $n \leftarrow n \ll 2$ 
3    $a \leftarrow 1$ 
4    $t \leftarrow 5$ 
5   while  $a_{23} \neq 1$  [will loop 23 times]
6      $s \leftarrow (s \ll 2) + (n \gg 23)$ 
7      $n \leftarrow n \ll 2$ 
8     if  $s < t$ 
9        $t \leftarrow t - 1$ 
10       $a \leftarrow a \ll 1$ 
11     else
12        $s \leftarrow s - t$ 
13        $t \leftarrow t + 1$ 
14        $a \leftarrow (a \ll 1) + 1$ 
15     end if
16      $t \leftarrow (t \ll 1) + 1$ 
17     loop
18      $s \leftarrow s \ll 2$ 
19     if  $s \geq t$ 
20        $a \leftarrow a + 1$ 
21     end if
22     return  $a$ 

```

Steps 15–17 of the loop are to determine whether to round down or up, and are a partial repetition of the loop. No additional hardware is required to perform this check, though an additional adder is required for step 17. Mathematical properties of square roots are such that any square root with a one in this 25<sup>th</sup> position is irrational, so we don't have to worry about whether to round up or down (IEEE 754 specifies that such rounding would be toward the even number, but since the number is irrational, we are guaranteed to not have a result exactly halfway between two IEEE 754 numbers).

### B. Freire's Algorithm

Freire's algorithm for finding square roots is based on the idea that the list of all possible square roots for a particular precision is finite and ordered, and therefore a binary search can be performed on the range to find the matching square root value. The list is ordered because square roots are monotonically increasing (for every  $x > y$ ,  $\sqrt{x} > \sqrt{y}$ ), and it is finite, because there are a finite number of numbers that can be expressed with the  $p$  bits used by a computer to store numbers. The algorithm is described by the following pseudocode, and illustrated in Figure 2.



**Figure 2. Finding the square root of 1,739 using Freire's binomial search algorithm.** In the figure,  $p = 16$ , so there are  $2^{16} = 65,536$  possible numbers in the number system (0–65,535). The maximum square root is therefore  $\sqrt{2^{16} - 1} < 256$ , so the initial test value is  $256 \div 2 = 128$ . The increment begins as half the initial test value, and is halved with each iteration: 64, 32, 16, 8, etc. Since  $128^2 > 1,739$  the increment is subtracted from the test value, and then tested again. At each iteration, if the square of the test value is greater than 1,739, then the increment is subtracted from the test value; if the square is less, then the increment is added to the test value. After seven iterations, we have determined that the answer lies somewhere between 41 and 42, and we have run out of bits for our precision. To finally determine which one to choose, we could perform one more iteration to see how  $41.5^2$  compares to 1,739, but, in fact, there is some faster mathematics that can be performed on some of the residual variables to determine which one to choose. In this example, 42 is chosen.

```

SQR_T_FREIRE( $n$ ) [ $n$  is a  $p$ -bit integer]
1    $b^2 \leftarrow 2^{p-2}$ 
2    $a \leftarrow 2^{\frac{p}{2}-1}$ 
3    $i \leftarrow \frac{p}{2} - 1$ 
4    $b \leftarrow a$ 
5    $a^2 \leftarrow b^2$ 
6   do
7      $b^2 \leftarrow b^2 \div 4$  [implemented as  $b^2 \gg 2$  ]
8      $b \leftarrow b \div 2$  [implemented as  $b \gg 1$  ]
9     if  $n \geq a^2$ 
10       $a^2 \leftarrow a^2 + a \times 2^i + b^2$ 
11      [ $a \times 2^i$  implemented as  $a \ll i$  ]
12       $a \leftarrow a + b$ 
13     else
14       $a^2 \leftarrow a^2 - a \times 2^i + b^2$ 
15       $a \leftarrow a - b$ 
16     end if
17      $i \leftarrow i - 1$ 
18     loop while  $i > 0$ 
19     if  $n \leq a^2 - a$ 
20        $a \leftarrow a - 1$ 
21     else if  $n > a^2 + a$ 
22        $a \leftarrow a + 1$ 
23     end if
24     return  $a$ 

```

In the algorithm,  $a$  represents the result, which is iteratively improved by adding or subtracting the delta  $b$ , which is halved each iteration. Instead of squaring  $a$  each iteration to compare to  $n$  (which involves a costly multiplication),  $a^2$  is kept in its

own variable and updated according to the formula

$$(a^2 \pm b^2) = a^2 \pm 2ab + b^2.$$

The  $b^2$  is known at the beginning ( $2^{p-2}$ ), and as  $b$  is halved each iteration (line 7),  $b^2$  is quartered (line 6), both of which can be accomplished with bitwise right-shifts. Furthermore, the apparent multiplication of  $2ab$  can be eliminated by realizing that  $b = 2^{i-1}$ , so  $2ab = a \times 2^i$ , which can again be implemented by a bitwise shift of  $i$  bits to the left (lines 9 and 11). Therefore, each iteration of the loop requires three bitwise shift operations, four additions or subtractions, and two comparisons (one of which is the comparison of  $i$  to 0), but no multiplications.

Freire actually does propose a slight improvement to the algorithm shown, based on the observation that  $i$  is really only used to shift a left each iteration to effect the  $2ab$  multiplication. Since  $i$  begins at  $\frac{p}{2}-1$  and decreases down to 0, we can instead begin  $a$  shifted left  $\frac{p}{2}-1$  (i.e.,  $a \leftarrow 2^{\frac{p}{2}-1} \cdot 2^{\frac{p}{2}-1} = 2^{p-2}$ ) and shift it right one bit each iteration. Now  $b$ , which was being added to or subtracted from  $a$  each iteration also needs to be shifted left by the same amount, and instead of right-shifting it one bit each iteration, it is right-shifted two bits. Now with  $b$  being right-shifted two bits every iteration, it is only one bit-shift away from the old  $b^2$ , so we no longer have to keep track of that value as well. The improved algorithm becomes:

```

SQRT_FREIRE( $n$ ) [ $n$  is a  $p$ -bit integer]
1    $a \leftarrow 2^{p-2}$ 
2    $b \leftarrow 2^{-3}$ 
3    $a^2 \leftarrow 2^{p-2}$ 
   do
4     if  $n \geq a^2$ 
9        $a^2 \leftarrow a^2 + a + (b \gg 1)$ 
10       $a \leftarrow (a + b) \gg 1$ 
     else
11       $a^2 \leftarrow a^2 - a + (b \gg 1)$ 
12       $a \leftarrow (a - b) \gg 1$ 
     end if
13      $b \leftarrow b \gg 2$ 
14   loop while  $b \neq 0$ 
15   if  $n \leq a^2 - a$ 
16      $a \leftarrow a - 1$ 
17   else if  $n > a^2 + a$ 
18      $a \leftarrow a + 1$ 
     end if
19   return  $a$ 

```

This is the algorithm Freire converted to handle floating point numbers.

### 1) Conversion to floating Point

Freire's algorithm as shown takes a  $p$ -bit number and re-

turns its  $\frac{p}{2}$ -bit square root. This works for integers because the integer square root of an integer has half as many bits as the input. However, in floating point, we want the output to have the same number of bits as the input. This will require us to have twice as many iterations through the loop and double-length registers to hold intermediate values. To obtain the 24-bit output, we must start with a 48-bit input, so we would begin by shifting the 25-bit input left 23 bits into a 48-bit register.

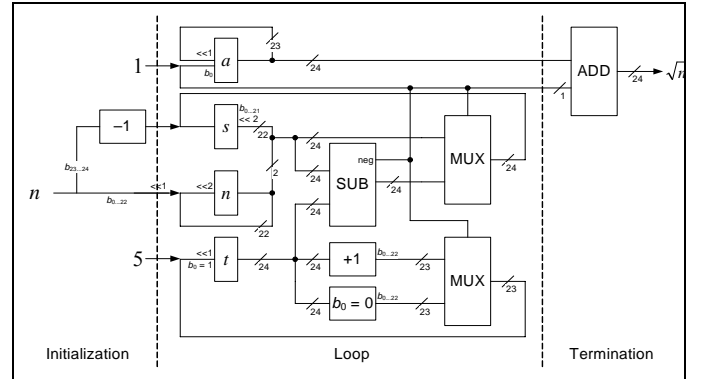
As with Hain's algorithm, since we know that the first bit is going to be one, we know the result of the first comparison, which we can short-circuit by initializing the variables as if the first loop iteration has already been completed. The initializations in steps 1–3 can be replaced by respectively  $a \leftarrow 3 \ll 44$ ,  $b \leftarrow 2^{43}$ , and  $a^2 \leftarrow 9 \ll 44$ . The rest of the algorithm remains exactly the same.

## III. PERFORMANCE AND PRECISION ANALYSIS

As these algorithms are intended to be implemented in hardware, an analysis of their performance must take into account their hardware implementation. Precision must also be exact in order for the algorithms to be considered accurate enough for most processors today. Fortunately, as we shall see, both algorithms produce the full 24-bit accuracy in order to produce the closest possible approximation in 32-bit IEEE-754 floats.

### A. Hain's Algorithm

From a performance perspective, we can implement Hain's algorithm in hardware similar to Figure 4. It is clear from the hardware implementation that Hain's loop requires only the time required to perform a subtraction and a multiplex, and lock the results into the registers. Even the subtraction can be short-circuited, however, once the result is determined to be negative, since the actual result will not be used in that case.



**Figure 3. Hardware to implement Hain's square root algorithm.** All registers are 24 bits long. Bit shifts can be hardwired. Loop terminates after 23 iterations, when  $a_{23} = 1$ .

The loop repeats  $p-1$ , or  $\Theta(p)$ , times. The loop initialization and termination steps require only a subtraction (initialization) and addition (termination). Even these operations can be implemented with half-adders, since they are only adding and subtracting one.

We can therefore calculate the running time of Hain's algorithm by the following formula:

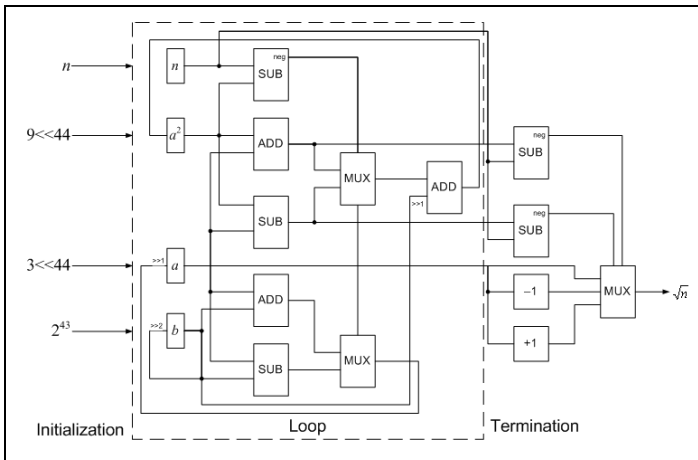
$$T_{Hain}(p) = t_{sub} + (p-1)(t_{sub} + t_{mux}) + t_{add} \\ = (p+1)t_{add} + (p-1)t_{mux} \quad [\text{assuming } t_{sub} = t_{add}]$$

This indicates that Hain's algorithm is very fast. For large enough  $p$ , traditional Newtonian methods will outperform, since they take  $\Theta(\log p)$ , though their constants will be significantly higher.

The mathematical basis of Hain's algorithm indicates that it should produce  $l + 1$  bits of precision, where  $l$  is the number of loop iterations. Since the loop iterates  $p-1$  times, we end up with a full  $p$  bits of precision, as required by IEEE 754. Hain's final addition is due to the fact that the  $(p+1)^{th}$  bit of the result may be a one, in which case the result must be rounded up. There should be no issues with IEEE rounding (which require that a result halfway between two  $p$ -bit floats be rounded toward the even number), since all  $p$ -bit numbers with square roots of at least  $p$  significant bits are irrational. This precision is tested in the next section.

### B. Freire's Algorithm

Freire's algorithm can be implemented in hardware similar to Figure 4. In contrast to Hain's algorithm, Freire's main loop has three layers of calculations. Furthermore his calculations are performed in twice the precision of Hain's, so in addition to the additional hardware and wires required to calculate and carry the extra bits, the additions and subtractions will take about 22% longer (best-case assumption, assuming carry-lookahead adders; if ripple-carry adders are used, then the time will double; and if carry-skip or carry-select adders are used, additions will take about 41% longer).



**Figure 4. Hardware to implement Freire's square root algorithm.** Diagram taken directly from Freire [1]. In his diagram, register EAX is  $n$ , EBX is  $a$ , ECX is  $b$ , and EDX is  $a^2$ . All registers are 48 bits long. Loop terminates after 22 iterations, when  $b = 0$ . Omitted from Freire's diagram, however, are the calculations that must be performed after the loop terminates to handle rounding. The termination calculations are about as complicated as a single loop iteration and require additional hardware.

Freire's run-time can be calculated as follows:

$$T_{Freire}(p) = (p-2)(2t'_{add} + t_{mux}) + (t'_{add} + 2t_{mux}) \quad [\text{assuming } t'_{sub} = t'_{add}] \\ = (2p-3)t'_{add} + p \cdot t_{mux}$$

If we assume that  $t'_{add} \approx 1.22t_{add}$ , and that  $t_{mux} \ll t_{add}$ , then Freire's algorithm takes about 2.44 times as long as Hain's.

Freire's algorithm is every bit as precise as Hain's. Its post-loop processing essentially simulates another iteration of the loop to determine whether one needs to be added to or subtracted from the result. This will be tested in the next section.

A final issue worth considering is the space consumed by Freire's algorithm compared to Hain's. It is clear that Freire's would occupy at least twice as much space as Hain's due to the size doubling of the registers. Furthermore, Freire's approach requires more arithmetic and logical components, so it is not unreasonable to presume that Freire's algorithm would take 2.5 or 3 times the amount of space on a chip as Hain's.

## IV. EXPERIMENTAL COMPARISON

A program<sup>1</sup> was written to test the precision of these algorithms, to determine whether the assertions made in the previous section are valid. Although performance was also measured and the results presented here, performance tests of the algorithms written in software are only of use as evidence for or against rather than proof of the performance conclusions made in the previous section.

### A. Methodology

The two algorithms were implemented using the same floating-point-to-fixed-point conversion so that the only difference would be in the way the two algorithms compute the square root. The two functions were then used to compute the square root for every 32-bit IEEE-754 floating point number between one and four (inclusive of one, exclusive of four). These limits were chosen because they include every possible 25-bit fixed point mantissa sent to the square root functions for all normalized floats from  $2^{-126}$  to  $2^{128}$ . Expanding the limits would change only the exponent without changing the mantissa.

Precision was checked by squaring the result and comparing the square to the input value. If the comparison showed that the square of the calculated root was smaller than the input, then the result had the smallest possible quantum ( $2^{-23}$ ) added to it to see if the resulting square would be closer to the input. If it were not, then the function had returned the closest possible 32-bit IEEE-754 number to the square root. The opposite was done if the square of the calculated root was larger than the input (i.e., a quantum was subtracted from the result and the square compared to the input).

As a sanity check, Hain's square root algorithm was also implemented in microcode on a 2 slice/16-bit Texas Instruments 74AS-EVM-16 processor using 32-bit even-odd register pairs. Freire's algorithm was not implemented, but rather Hain's square root operation was compared to a microcoded

<sup>1</sup> Obtainable from the authors.

floating point divide operation on the same architecture.

### B. Results

Both functions calculated all 16,777,216 square roots accurately in every bit.

The programmed implementation of Hain's algorithm proved to be about 86% faster than Freire's. However, as mentioned earlier, the relative timings would depend on the specific hardware implementations, where some of the microoperations operations can be executed in parallel.

The microcoded comparisons showed that Hain's floating point square root implementation took only 89% more clock cycles than the floating point divide operation. That is, it ran slightly faster than two floating point divides. Again, this should be regarded as indicative only, since no architectural optimizations were performed. It is conceivable, however, that any optimizations on one operation could be applied to the other, so that the relative timings may, in fact, be a good indicator.

## V. CONCLUSIONS

While they are equivalent in precision, Hain's algorithm is superior to Freire's both in hardware cost and time. Although Hain's algorithm converges on its result in  $O(p)$  time versus other methods that converge in  $O(\log p)$  time, it is likely that Hain's algorithm is superior for small  $p$  (say 32, or 64), and further comparisons could be performed to determine the break-even point where conventional methods begin to outperform Hain.

## REFERENCES

- [1] Freire, P. <http://www.pedrofreire.com/sqrt>. 2002.
- [2] Goldberg, D. "Computer Arithmetic". Xerox Palo Alto Research Center, 2003. Published as App. H in *Computer Architecture: A Quantitative Approach*, Third Edition by Hennessy, J.L. and Patterson, D.A., Morgan Kaufmann, 2002. Available at <http://books.elsevier.com/companions/1558605967/appendices/1558605967-appendix-h.pdf>.
- [3] Hain, T. "Floating Point Arithmetic Processor". Internal Report, CIS, University of South Alabama, Mobile, AL 36688, 1989.
- [4] Jovanović, B., Damnjanović, M.: "Digital Systems for Square Root Computation", *Zbornik Radova XLVI Konferencije Etran, Herceg Novi, Montenegro*, June 2003, Vol 1, pp. 68-71.
- [5] R. Goldschmidt, Application of division by convergence, Master's thesis, MIT, June, 1964.
- [6] Sun Microsystems, Inc, "Numerical Computation Guide", <http://docs.sun.com/source/806-3568/ncgTOC.html>.