Simple Seed Architectures for Reciprocal and Square Root Reciprocal

Milos Ercegovac[a], Jean-Michel Muller[b] and Arnaud Tisserand[c,b]

[a] University of California, Los Angeles. Computer Science Department
4732 Boelter Hall, Los Angeles, CA 90095, USA
milos@cs.ucla.edu

[b] LIP, CNRS–ENS Lyon–INRIA–UCB Lyon
46 allée d'Italie. F-69364 Lyon, FRANCE
jean-michel.muller@ens-lyon.fr

[c] LIRMM, CNRS–Univ. Montpellier II
161 rue Ada. F-34392 Montpellier, FRANCE
arnaud.tisserand@lirmm.fr

***Abstract*— This paper presents a simple hardware architecture for computing the seed values for reciprocal and square root reciprocal. These seeds are used in the initialization of floating-point division and square root software iterations. The proposed solution is based on polynomial approximation with specific coefficients and a table lookup.**

INTRODUCTION

In general-purpose processors, floating-point division and square root are often performed using iterative methods such as the Newton-Raphson algorithm [1], [2], [3]. In order to reduce the number of iterations, dedicated hardware tables are frequently used to store medium accuracy initial values, or seeds, for the iterations. In this work, we focus on the computation of such seeds in hardware.

The proposed solution is based on a polynomial approximation with specific coefficients and a table lookup. The corresponding architecture is very simple and leads to small and fast circuits. There are several parameters in seed hardware operator: the approximated function ($1/x$ or $1/\sqrt{x}$), the argument width, internal accuracy requirements and optimization parameters. We have developed a VHDL code generator, called `seedgen`, to support all the possible parameters of our method. This program generates an optimized and synthesizable VHDL description of the seed operator based on our method. Compared to direct lookup table implementation on FPGAs (with content optimization using the synthesis tools), the proposed method leads to 2.5 reduction factor in size and 40% speed improvement.

I. BACKGROUND

A. Newton-Raphson Iteration

The Newton-Raphson algorithm evaluates a function by iteratively improving an initial approximation. This algorithm is based on a general method to obtain a single zero α of function f (i.e., $f(\alpha) = 0$ and $f'(\alpha) \neq 0$). If x_0 is close enough to α, the following iteration converges towards α:

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

where $f'(x)$ denotes the derivate of f with respect to x. The convergence of the method is quadratic, which means that the number of bits of accuracy roughly doubles after each iteration.

In order to speed up the overall computation, the first iterations that only provide a very small number of bits of accuracy should be avoided. For instance, if one uses an initial value x_0 with only one bit of accuracy, the computation requires at least 5 iterations ($1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32$) for 32-bit values. Using an initial seed with 8 bits of accuracy, it only requires 2 iterations ($8 \rightarrow 16 \rightarrow 32$).

In general, by using an initial approximation with a relative error not greater than 2^{-n}, the number of iterations to obtain the result with a relative error not greater than 2^{-m} is

$$K = \left\lceil log_2 \left(\frac{m}{n} \right) \right\rceil \tag{2}$$

Therefore, the choice of the initial approximation is critical for the speed of the algorithm. On the other hand, a seed with a large number of bits is costly to obtain. Consequently, finding a simple method of obtaining the seed value is of great practical interest in implementing the Newton-Raphson method. This speed-up method also applies in the Goldschmidt iteration for division and square root [2].

Direct lookup table implementations of such seeds are possible for small accuracy. For instance, the IBM 360/91 processor used generated 10-bit seed from a table (implemented as a ROM) addressed by 7 bits of the normalized divisor [4].

For larger accuracies, the bipartite method is more and more used [5]. For instance, in the AMD-K7 processor an optimized and simplified reciprocal and square root reciprocal estimate interpolation unit is implemented [6]. The reciprocal estimate

1-4244-0132-1/05/$20.00 ©2005 IEEE

provides at least 14.94 bits of accuracy while the square root reciprocal is accurate to at least 15.84 bits. This unit requires 69Kb of ROM and 3 cycles of latency.

B. Division Iteration

For the computation of the quotient $q = a/d$, one can use the following 2-step method:

1) Evaluate $t = 1/d$ using the Newton-Raphson iteration based on the function $f(x) = \frac{1}{x} - d$. The corresponding iteration is:

$$x_{i+1} = x_i\left(2 - dx_i^2\right) \tag{3}$$

2) Compute the quotient using $q = t \times a$.

Each iteration requires two multiplications and one addition. The second step requires an additional full-length multiplication.

A table is often used to get the initial value x_0 based on a few most significant bits of d. As an example, on the Itanium processor, the `frcpa` instruction gives a seed for reciprocal $(1/d)$ with an accuracy of 8.886 bits (see [3] for details).

C. Square Root Iteration

Directly evaluating $\sqrt{c}$ using the Newton-Raphson algorithm with the function $f(x) = x^2 - c$ is not a good idea. The corresponding iteration is $x_{i+1} = \frac{1}{2}\left(x_i + \frac{c}{x_i}\right)$ which requires a division at each iteration. A better solution is based on the following 2-step method:

1) Newton-Raphson iteration based on the function $f(x) = \frac{1}{x^2} - c$ which has a zero equal to $\frac{1}{\sqrt{c}}$. The corresponding iteration is:

$$x_{i+1} = \frac{x_i}{2}\left(3 - cx_i^2\right) \tag{4}$$

2) Multiplication by c to get $\sqrt{c}$.

Each iteration requires three multiplications and one addition.

In this case also, a table is often used to get the initial value x_0 based on a few most significant bits of c. As an example, on the Itanium processor, the `frsqrta` instruction gives a seed for $1/\sqrt{c}$ with an accuracy of 8.831 bits (see [3] for details).

D. Minimax Polynomial Approximation

The initial approximations used in the following are based on the *minimax* polynomial approximation as a starting point. The degree-d minimax polynomial approximation to f on $[a, b]$ is the polynomial P^* that satisfies:

$$||f - P^*||_\infty = \min_{P \in \mathcal{P}_d} ||f - P||_\infty \tag{5}$$

where $\mathcal{P}_d$ is the set of polynomials with real coefficients and degree at most d and

$$||f - P||_\infty = \max_{a \le x \le b} |f(x) - P(x)|. \tag{6}$$

Minimax approximations can be computed using a well-known algorithm due to Remes [7] (available in the Maple numapprox package, for instance).

II. Reciprocal Seed

The degree-1 minimax polynomial of $f(x) = 1/x$ with $x \in [1, 2[$ (the range of the mantissa of a floating-point number) computed using Maple is

$$1.4571 - 0.5x \tag{7}$$

This polynomial provides an approximation with 4.5 bits of accuracy.

In this work, we use the minimax polynomial as a starting point because it is convenient and well implemented (e.g., Maple minimax function). But for degree-1 polynomial approximation, another method is possible to get the polynomial coefficients exactly, for details see [2, page 373]. Using this method one can get the polynomial $(3/4 + \sqrt{2}/2) - x/2$ which is theoretical polynomial that correspond to the minimax polynomial (7). Theoretically, this is the same polynomial, the slight difference in the constant coefficient is due to the rounding error during the Remez algorithm numerical execution. So, the two polynomials can be used as a starting point for our method since they provide similar accuracy and coefficients.

In the polynomial (7), the degree-1 coefficient is a power of 2. The constant coefficient is close to the value 1.5. So the following "close" polynomial can be used to approximate $1/x$:

$$p(x) = \frac{3}{2} - \frac{x}{2} \tag{8}$$

This modified polynomial p approximates $1/x$ on $[1, 2[$ with 3.5 bits of accuracy. We will use this very simple polynomial as a starting point to compute the seed value for the reciprocal function.

The evaluation cost of this polynomial is very small in practice. It is illustrated in Figure 1. Using 2's complement the value $-x/2$ is obtained by complementing all bits of x, shifted one position to the right, and adding a 1 in the LSB position. As $x \in [1, 2[$, the first bit of x is 1 (at position 0). If the binary representation of x is $(1.x_1x_2x_3 \ldots x_n)_2$, then the binary expansion of $-x/2$ is $(1.0\overline{x_1}\overline{x_2}\overline{x_3} \ldots \overline{x_n})_2$ plus 1 LSB. So the computation of $3/2 - x/2$ only costs one carry propagation corresponding to the additional 1 LSB. This kind of a very simple polynomial approximation was proposed in [8].

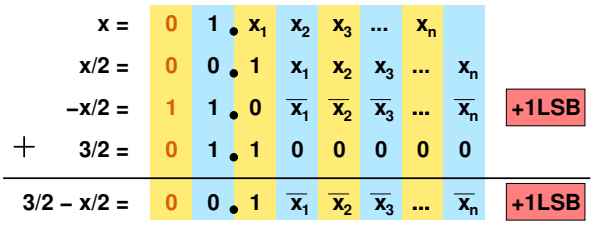

Fig. 1. Efficient evaluation of the polynomial $p(x) = \frac{3}{2} - \frac{x}{2}$.

In order to improve the result of this approximation (p only provides 3.5 bits of accuracy), we add a *correcting term* $t(x)$

to the result of this modified polynomial. Our method is based on this very simple polynomial approximation (8) and a table to store the correcting terms. So the value of the seed is:

$$s(x) = p(x) + t(x) = \frac{3}{2} - \frac{x}{2} + t(x) \tag{9}$$

where the correcting term $t(x)$ is the difference between the actual function $1/x$ and the result of $p(x)$ rounded to the output width:

$$t(x) = \frac{1}{x} - \left(\frac{3}{2} - \frac{x}{2}\right) \tag{10}$$

The corresponding architecture is presented in Figure 3.

The table used to the correcting terms has n-bit addresses (the bits of the operand x) and w-bit words. The final result (seed) $s(x)$ has $n + g$ bits where g is the number of guard bits ($g > 1$ for the reciprocal function). The alignment of the n bits of p and the w bits of t is illustrated in Figure 2 using an example.

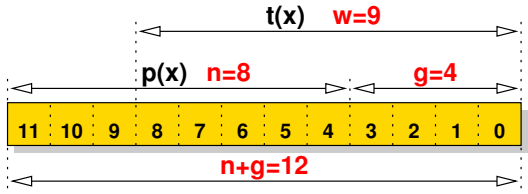

Fig. 2. Alignment of p and t in the final result. For $n = 8$ and $g = 4$ the generator gives $w = 9$.

The main hardware cost of our solution is one $(2^n \times w)$-bit table and one $(n + g)$-bit addition with input carry. Optimizations at circuit level are presented in Section IV.

In addition to the VHDL description of the architecture, `seedgen` generates a plot of the approximation results. The corresponding plot is presented Figure 4 in the case $n = 3$ bits and $g = 1$ bit for the reciprocal function. The generated plot is based on a `gnuplot` script [9].

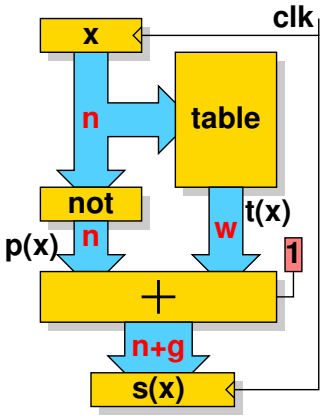

Fig. 3. Architecture of the seed generator.

The accuracy (corresponding to the maximum error) provided by our architecture is $n+g+1$ bits for the reciprocal. The

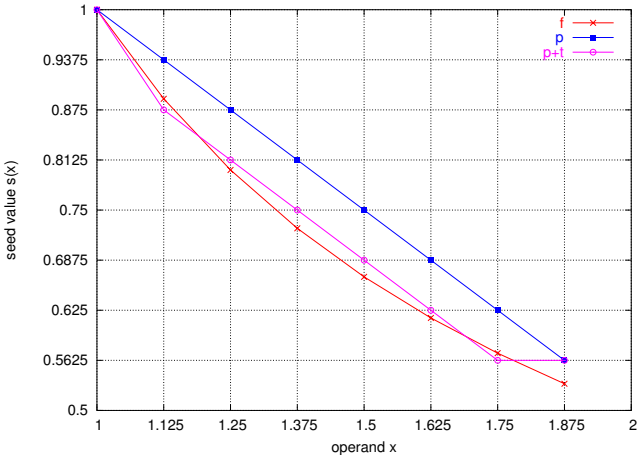

Fig. 4. Approximation for $f = 1/x$, $n = 3$ and $g = 1$.

choice of the correcting terms $t(x)$ can be done to ensure the seed value $s(x)$ is within a $1/2$ulp distance to the theoretical value of $1/x$. The plot presented in Figure 5 illustrates the approximation error corresponding to the case $n = 3$ bits and $g = 1$ bit. This plot is also generated using `seedgen`.

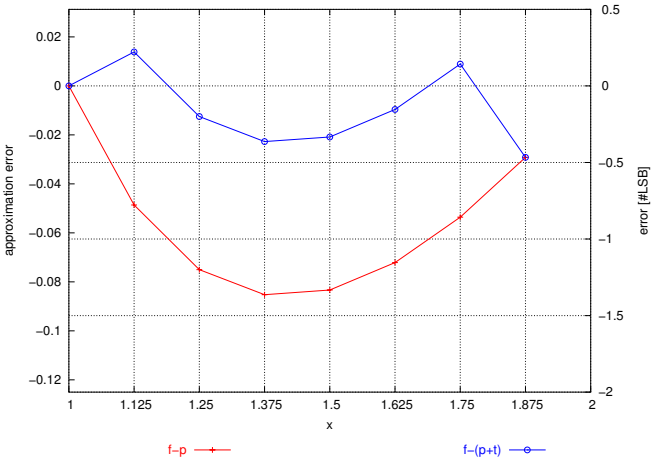

Fig. 5. Approximation error for $f = 1/x$, $n = 3$ and $g = 1$.

The polynomial p overestimates the actual value of f (see Figure 4 for an illustration). Then the sign of the difference between p and f is always greater than or equal to zero. At the implementation level, the correction can be implemented using two solutions:

- store positive offsets in the table t and perform a subtraction $p - t$;
- store negative offsets in the table t and perform an addition $p + t$.

The best solution depends on the basic cells available at low level.

Figures 3 and 2 show that the alignment of p and t operands allow to improve the adder architecture used to perform $p+t$. Indeed, in some cases one can use incrementer cells instead of

adder cells for the $w-g$ LSBs. In practice, this optimization is provided at the synthesis level by most of standard tools.

III. Square Root Reciprocal Seed

A similar solution is proposed for the square root reciprocal. The degree-1 minimax polynomial of $f(x) = 1/\sqrt{x}$ with $x \in [1,2[$, and 5.7 bits of accuracy is

$$1.2739 - 0.292x \tag{11}$$

This polynomial is close to:

$$p(x) = \frac{5}{4} - \frac{x}{4} \tag{12}$$

which gives an approximation to $1/\sqrt{x}$ on $[1,2[$ with 4 bits of accuracy. The evaluation cost of this polynomial is also very small in practice as illustrated on Figure 6.

Using 2's complement the value $-x/4$ is obtained by complementing all bits of x shifted 2 bits to the right and add 1 LSB. As $x \in [1,2[$, the binary expansion of $-x/4$ is $(1.10\overline{x_1}\overline{x_2}\overline{x_3}\ldots\overline{x_n})_2$ plus 1 LSB. The computation of $5/4 - x/4$ only costs one carry propagation corresponding the additional 1 LSB. So the same kind of architecture presented in Figure 3 can be used in case of the square root reciprocal.

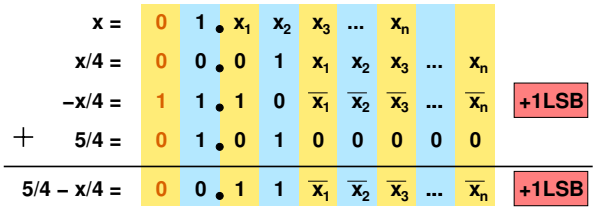

Fig. 6. Efficient evaluation of the polynomial $p(x) = \frac{5}{4} - \frac{x}{4}$.

As the x is shifted two positions to the right the number of guard bits is $g \geq 2$ in the square root reciprocal case. Now the generator has to compute the correct constant bits of $-x/4$ and the correcting terms[1] $t(x)$ with

$$t(x) = \frac{1}{\sqrt{x}} - \left(\frac{5}{4} - \frac{x}{4}\right) \tag{13}$$

The accuracy (corresponding to the maximum error) provided by our architecture is also $n+g+1$ bits (1/2 LSB) for the square root reciprocal.

IV. VHDL Generator

We have developed a program that generates optimized and synthesizable VHDL descriptions of seed architectures using our method. This program, called `seedgen`, is a C program in text mode and distributed under the GPL license. It is available on the Web [10]. All the `seedgen` parameters are given on the command line. Table I, given in the last page of the paper, reports the main parameters.

[1]The correcting terms t here are specific to the square root reciprocal functions.

V. Accuracy Results

Our solution provides seeds (for reciprocal and square root reciprocal) with an accuracy of at most 1/2 LSB with words on $n+g$ bits. This corresponds to a maximum error of at most $2^{-(n+g+1)}$. In order to verify this accuracy, we present in Table II, given in the last page of the paper, the minimum accuracy (corresponding to the maximum error) and the average accuracy (corresponding to the average error) reported by the generator (using an exhaustive check).

A. Comparison to Direct Lookup Table

In order to compare our method with standard solutions, we implemented seeds based on direct lookup table on FPGAs. The generated direct lookup table architecture is a ROM description with n-bit addresses and $n+g+1$-bit words. The stored values provide 1/2 LSB accuracy for all inputs as in operators generated by `seedgen`. When this ROM description is implemented on FPGAs, the synthesis tool performs a lot of logical optimization based on the content of the ROM. This leads to significantly smaller and faster architecture than using unoptimized table with $2^n \times (n+g+1)$ bits.

Table III presents the comparison results for the reciprocal on the Spartan3 FPGA and ISE tools. In this table, 3 solutions are compared. The first one (direct) is the ROM description without logical optimization (only area is reported). The second one (optimized) is the ROM description with the logical optimizations performed by the synthesis tool. A reduction factor less than 2 is obtained compared to the direct ROM description. The last solution is the results from `seedgen`. Our results show a reduction factor up to 2.5 and around 40% speed improvement compared to the optimized solution.

		direct		optimized		`seedgen`	
n	g	ROM size	area [#CLB]	area [#CLB]	period [ns]	area [#CLB]	period [ns]
7	2	1280	80	57	11.7	33	9.0
8	2	2816	176	109	12.7	48	9.2
9	2	6144	384	227	14.7	89	10.6
10	2	13312	832	448	16.4	178	11.9

TABLE III
Comparison results between direct lookup table implementation (without and without logical optimization) and our method on Spartan3 FPGA.

Conclusion

A method for the initial approximation to reciprocal and square root reciprocal functions in hardware was proposed. These initial approximations or seeds are used in the initialization of floating-point division and square root software iterations in order to speedup the computation.

The proposed solution is based on polynomial approximation with specific coefficients and a table lookup. The obtained architectures lead to small and fast circuits. The method has been implemented in C program distributed under GPL license. This program, called `seedgen`, automatically

parameter	meaning	value	
		possible	default
`-f`	approximated function (f)	$\begin{cases} \texttt{r} & \text{for } \frac{1}{x} \\ \texttt{s} & \text{for } \frac{1}{\sqrt{x}} \end{cases}$	–
`-n`	argument size (n)	$2 \leq n \leq 16$	–
`-g`	number of guard bits (g)	$\begin{cases} 1 \leq g \leq 4 & \text{for } \frac{1}{x} \\ 2 \leq g \leq 4 & \text{for } \frac{1}{\sqrt{x}} \end{cases}$	$\begin{cases} g = 1 & \text{for } \frac{1}{x} \\ g = 2 & \text{for } \frac{1}{\sqrt{x}} \end{cases}$
`-o`	optimization	`n` for negative offset	no optimization
`-r`	register	$\begin{cases} \texttt{o} & \text{output only} \\ \texttt{i} & \text{input only} \\ \texttt{b} & \text{both input and output} \end{cases}$	no register

TABLE I

MAIN PARAMETERS SUPPORTED BY `seedgen`.

$f \downarrow$	$g \downarrow$	$n \rightarrow$	4	5	6	7	8	9	10	11	12
$\frac{1}{x}$	1	min.	6.04	7.02	8.01	9.00	10.00	11.00	12.00	13.00	14.00
		avg.	6.90	7.91	9.10	10.07	11.02	11.99	12.97	14.00	14.99
	2	min.	7.06	8.07	9.01	10.01	11.00	12.00	13.00	14.00	15.00
		avg.	8.16	9.24	10.05	11.10	12.02	12.98	14.00	15.01	16.01
	3	min.	8.07	9.03	10.01	11.00	12.00	13.00	14.00	15.00	16.00
		avg.	9.16	10.11	11.14	12.04	12.93	13.99	15.02	15.99	17.01
$\frac{1}{\sqrt{x}}$	2	min.	7.11	8.03	9.05	10.03	11.00	12.00	13.00	14.00	15.00
		avg.	8.02	8.87	10.15	10.98	12.00	12.98	13.98	15.02	16.02
	3	min.	8.03	9.05	10.05	11.01	12.00	13.00	14.00	15.00	16.00
		avg.	8.76	10.25	11.05	11.97	12.97	13.97	15.01	16.02	17.01
	4	min.	9.05	10.08	11.03	12.00	13.00	14.00	15.00	16.00	17.00
		avg.	10.27	11.16	11.95	12.87	13.97	15.00	15.99	17.00	17.99

TABLE II

ERROR MEASUREMENTS (MINIMAL AND AVERAGE ACCURACY) IN NUMBER OF CORRECT BITS.

generates optimized and synthesizable VHDL operators for various parameters and hardware constraints.

Compared to direct lookup table implementation on FPGAs (with content optimization using the synthesis tools), the proposed method leads to 2.5 reduction factor in size and 40% speed improvement.

REFERENCES

[1] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[2] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[3] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[4] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. The IBM system/360 model 91: Floating-point execution unit. *IBM Journal*, 11(1):34–53, January 1967.

[5] D. Das Sarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In *12th IEEE Symposium on Computer Arithmetic*, pages 17–28. IEEE Computer Society, July 1995.

[6] S. F. Oberman. Floating point division and square root algorithms and implementation in the AMD-k7 microprocessor. In *14th IEEE Symposium on Computer Arithmetic*, pages 106–115. IEEE Computer Society, April 1999.

[7] E. Remes. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, 198:2063–2065, 1934.

[8] M. Ito, N. Takagi, and S. Yajima. Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification. *IEEE Transactions on Computers*, 46(4):495–498, April 1997.

[9] Gnuplot. `http://www.gnuplot.info/`.

[10] A. Tisserand. seedgen: A circuit generator for reciprocal and inverse square root seeds. `http://www.lirmm.fr/~tisseran/devel/seedgen`, GPL license, 2005.