International Journal of Science Engineering and Advance Technology

FPGA Implementation of Floating Point Reciprocator Using Binomial Expansion Method

[1]B.Mahesh, [2]G.Sridevi
[1]M.Tech Scholar, Sri Sai Aditya Inst. Of Science & Technology, Surampalem
[2] Associate Professor, ECE
Sri Sai Aditya Inst. Of Science & Technology, Surampalem

ABSTRACT:

Floating-point support has become a mandatory feature of new micro processors due to the prevalence of business, technical, and recreational applications that use these operations. In these operations Floating-point division is generally regarded as a low frequency, high latency operation in typical floating-point applications. So due to this not much development had taken place in this field. But nowadays floating point divider has become indispensable and increasingly important in many scientific and signal processing applications.

In this paper we implement floating point reciprocator for both single precision and double precision floating point numbers using FPGA. Here the implementation is based on the binomial expansion method. By comparing with previous works the modules occupy less area with a higher performance and less latency. The designs trade off either 1 unit in last-place (ulp) or 2 ulp of accuracy (for double or single precision respectively), without rounding, to obtain a better implementation.

KEYWORDS:
Floating Point Reciprocator, Field Programmable Gate Array (FPGA), Single Precision Floating Point and Double Precision Floating Point.

I.INTRODUCTION

Floating point arithmetic is widely used in many scientific and signal processing applications. Modern applications comprise several floating point operations like addition, multiplication, division, and square root etc. Among the arithmetic operations division is generally the most difficult to implement in hardware. Division is a most common operation in many scientific and signal processing applications, so there is a need for efficient hardware implementations for division. Given the logic density of modern FPGAs, it is feasible to use FPGAs for floating-point applications. The IEEE standard for floating point (IEEE-754) defines the format of the numbers, and also specifies various rounding modes that determine the accuracy of the result. For many signal processing, and graphics applications, it is acceptable to trade off some accuracy (in the least significant bit positions) for faster and better implementations. A lot of work has been done on obtaining efficient implementations for this operation.

Generally, this operation can be done into two parts, first take the inverse of divisor and then multiply with dividend. Because of this, many hard-ware dividers focus on efficiently obtaining the reciprocal of floating-point number. Many algorithms were developed for divider which includes subtractive method, functional iterations which uses multipliers and algorithms for faster computation of

division like high radix algorithm. But most of these algorithms namely Newton raphson method, high radix algorithm and digit recurrence algorithm required huge look-up tables, along-with wider multipliers which affect the area and performance. But large area for division alone is not desirable. So we use binomial expansion method for computation. Our approach focuses on finding the reciprocal. It is based on the well known binomial-expansion, contains small look-up table, and uses partial block-multipliers, resulting in less area, less delay, and correct up to required level (accuracy trade off). We have restricted ourselves only to normalized numbers. All the exceptional cases are detected, and indicated as invalid input/output. Comparisons of our implementation with previous works mentioned in the literature show that we are able to obtain small look-up tables and overall very efficient hardware. We have used Xilinx ISE-10.1 synthesis tool, ModelSim SE 6.3f simulation tool.

II Technical Approach:

The format of a floating-point number is as follows:

For Single Precision

1-bit	8-bits	23-bits
Sign bit	exponent	mantissa

For Double Precision

1-bit	11-bits	52-bits
Sign bit	exponent	mantissa

In this paper, we do not discuss the exponent manipulation as it is a standard process. The benefits of our implementation are in the computation of the inverse of the mantissa. Let y be the inverse of the mantissa *a*. Then, y=1/(1.a), where in 1.a, 1 is hidden bit of mantissa. We have divided the mantissa in two parts, a_1 and a_2. a_1 is used to fetch some pre-calculated data from a look-up table. Now, since

$y=1/(a_1+a_2)$

$$y = (a_1+a_2)^{-1}$$
$$= a_1^{-1} - a_1^{-2}.a_2 + a_1^{-3}.a_2^2 - a_1^{-4}.a_2^3 + \ldots\ldots \quad (1)$$

The content of each term of equation (1) is as follows

$a_1^{-1} = 0.\overbrace{xxxxxxxx}^{\text{Full significant bits}}$

$a_1^{-2}.a_2 = 0.\overbrace{00....00}^{\text{m-zero bits}}\overbrace{xx...xx}^{\text{significant bits}}$

$a_1^{-3}.a_2^2 = 0.\overbrace{00....00}^{\text{2m-zero bits}}\overbrace{xx...xx}^{\text{significant bits}}$

$a_1^{-4}.a_2^3 = 0.\overbrace{00....00}^{\text{3m-zero bits}}\overbrace{xx...xx}^{\text{significant bits}}$

and so on....

where m is the number of bits of a1.

We can see that as we move towards higher terms their contributions to main result are decreasing. Thus, depending upon our precision choice we can take suitable number of terms from equation (1) for calculating inverse, based on value of m. In case of single-precision we have taken the first three terms, while for the case of double- precision 7 terms have been taken. The value of m we have chosen is 8 for both cases. These values were selected based on available FPGA resources, as will be shown soon. We have simplified the desired terms in such a way so that we can use less hardware with low latency and good accuracy.

For single-precision we have taken all the three terms as available, like

$$y = a_1^{-1} - a_1^{-2}.a_2 + a_1^{-3}.a_2^2 \quad \ldots\ldots(2)$$

For double precision, simplified form will be as,

$$y = a_1^{-1} - a_1^{-1}[(a_1^{-1}a_2 - a_1^{-2}a_2^2)(1 + a_1^{-2}a_2^2 + a_1^{-4}a_2^4)] \quad \ldots\ldots(3)$$

Though we can simplify above equations a little more, it will affect the area, latency and accuracy. The accuracy is affected due to the fact that floating-point operations are not completely associative, i.e. u(v + w) may not be exactly equal to (uv +uw). This is due to the finite number of bits used to represent the numbers.

III. Implementation

We have shown the implementations for single precision and double precision separately as different issues arise in each case. Some of the design decisions are based on the fact that multipliers of size 18×18 are readily available as hard IP cores on many common FPGA families. We have based our computations on the Xilinx Virtex II platform. However, the basic ideas of saving some of the block multiplications hold even if a different sized multiplier core is used, although the exact numbers would change.

A. Single-precision Floating-point:

The architecture of single-precision floating-point divisor is shown in Fig. 1.It includes a Block-Memory (BRAM) which contains pre-calculated values of $a^{-}{}_{1}{}^{1}$(24 − bits), $a^{-}{}_{1}{}^{2}$(17 − bits), and $a^{-}{}_{1}{}^{3}$(17 − bits) in a single data-word(58-bits), with 8-bit (content of a_1) as address bits. The contents of the BRAM have been calculated using a separate program written in C, with float data type for the numbers. The content of $a^{-}{}_{1}{}^{1}$ has been extended to 30-bits (by appending 6-bits"111111" at least significant bit (LSB's)) for addition/subtraction purpose.

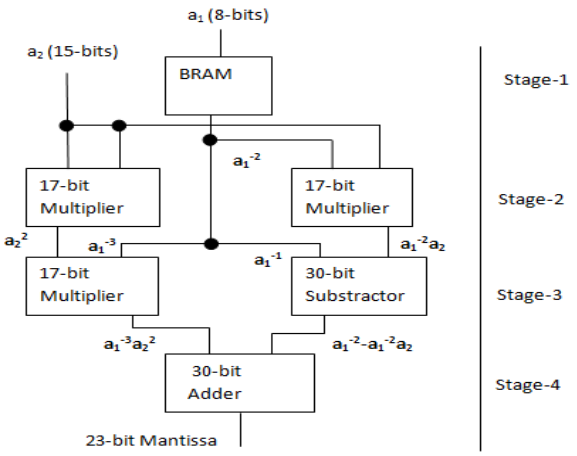
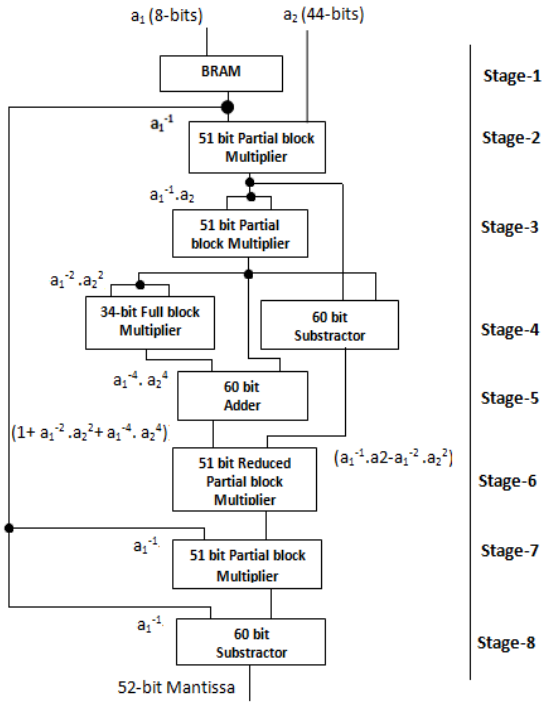


Fig. 1. Architecture for single-precision floating-point reciprocator

Here we can also do above operation with only value of $a^{-}{}_{1}{}^{1}$, but it will increase the total operation latency and size of multipliers. In both cases we will use only a single BRAM on FPGA, so we prefer the first approach. The architecture has latency of four, though we can include the BRAM access in the first stage with a slight loss in maximum operating frequency. By using pipelined multiplier we can approximately double the overall frequency. We have shown the result with the latency four.

Fig.2.Architecture for double-precision floating-point reciprocator



Our aim here is to only show the use of less necessary hardware. We can do pipelining in the given architecture very easily.

B. Double-precision Floating-point:

The architecture of double-precision floating-point divisor is shown in Fig. 2. It also includes a single BRAM which contains pre-calculated values of only $a^{-}{}_{1}{}^{1}$(54 − *bits*) with 8-bit (content of a_1) as address bits. The content of BRAM has been calculated using a C-program,

with double as data-type of floating-point numbers. The content of a^{-1}_{1} has been extended to 60-bits (by appending 6-Bits ”111111” at LSB’s) for addition/ subtraction purpose. Here we have a huge saving on block-memory compared to other methods discussed later.

There are three type of multiplier (based on Xilinx MULT18x18 block) that have been used. Second, third and seventh stage has 51-bit partial multiplier. It uses only six-MULT18x18 block instead of nine, to produce more than 52-bit (MSB) of correct result, which is all that we need. Stage six is also a 51-bit partial multiplier, but due to its specific input nature (17-bits of first input is 0x10000 in hex), it contains only three-MULT18x18 block. The fourth stage multiplier is a 34-bit full multiplier, but instead of using IP-core for it we have designed it using four MULT18x18 block which is taking less (about 2/3) glue logic and is faster than the IP-core available from Xilinx. Overall latency of module is eight, which we can increase further using pipelining as discussed in the case of single precision, for better performance.

IV. Results

The Hardware utilization and performance of both the single-precision and double-precision is shown in Table-I. Since our implementation neglects some of the lower order bits in the computation, it is important to estimate the impact of this on the overall accuracy of results.

TABLE-I

Parameters	Single Precision	Double Precision
Mul 18X 18	3	55
Slices	95	760
BRAM	1	1
Latency	Less	More

Single Precision

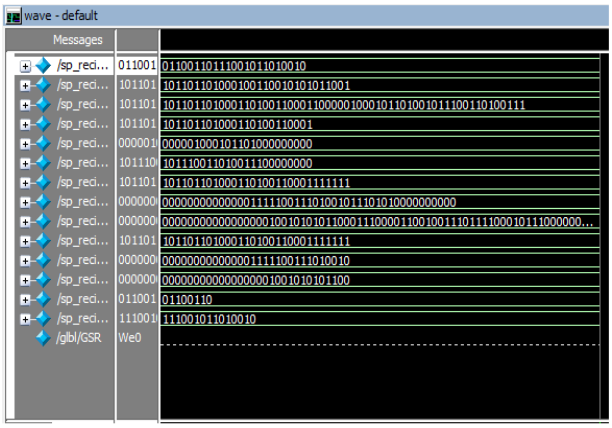


Fig.3 Simulation Results of Single-Precision floating point numbers

Double Precision

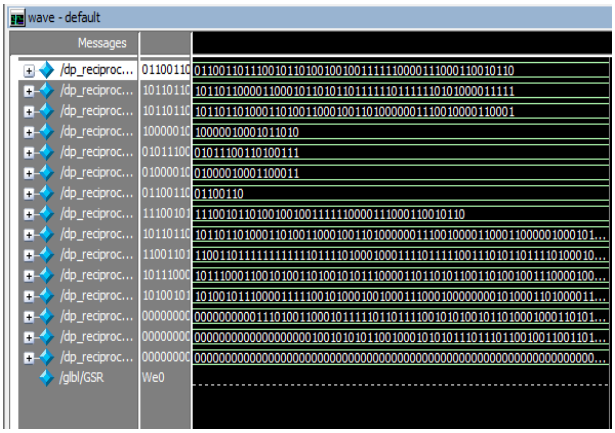


Fig.4 Simulation Results of Double-Precision floating point numbers

V. Comparison

The basis of our implementation is a well known technique of using look-up tables and multipliers. The main benefits are in the optimizations of the resource usage. In this section, we compare our implementation against many previous approaches mentioned in the literature.

Our comparisons are based around the Xilinx hardware resources. Even on this platform, many different multiplier implementations are available with differing speed-area-latency tradeoffs. By using different instances, we can obtain

suitable tradeoffs. Similarly, on a different platform with different basic resources, the main ideas developed in this paper will still hold. Only the details of the hardware usage will differ. For many of the comparisons, direct FPGA implementations of the methods are not available. In such cases, we have estimated the resource usage based on the components in the design. The number of block RAM cores required to implement a given look-up table and number of MULT18x18 block required to implement a given multiplication has been estimated from the Xilinx core generator software.

One of the most popular methods used for computing reciprocals is the Newton Raphson iterative procedure [2]. The Newton-Raphson iteration for reciprocal of A is given by, $x_{i+1} = x_i(2 - x_i.A)$. For each iteration it requires two multiplications and one subtraction. The value of $x0$ is usually taken from a look-up table. Thus for two iterations (results are based on [2]), in the case of single precision it requires one look-up table in 8-bit address space, two 8×16 multiplication and two 16×32 multiplication (equivalently 1 BRAM and 6 MULT18x18). For double-precision it requires one look-up table in 15-bit address space, two 15×30 multiplication and two 30×60 multiplication (equivalently 28 BRAM and 20 MULT18x18).

In Tradeoffs of designing floating point division and set square root on virtex FPGA's [5], the authors have reported the floating –point division and square-root using SRT 1 division method on FPGA. In terms of performance, the pipelined approach is closest to our proposed implementation, but requires significantly 212 number of slices and 14 clock cycles. The maximum frequency is approximately 66MHz.

Digit Recurrence Divider [4] presents another SRT based implementation that has similar area to ours but considerably less throughput and speed with 520 number of slices for single precision and 1400 slices for double precision.

The method Advanced Component in the variable precision Floating Point [3] presented a library for floating-point operations. In the case of single precision it requires 7 BRAMS and 8 MULTI 18x18 at a maximum frequency of operation 129MHz and 361 slices. For double precision it requires 62 BRAMS at a maximum frequency of 108MHz and 1145 slices.

In A Combined Interval and Floating Point Reciprocal Unit [6], the gates are 50,762 and latency time of 11 cycles and Time period of 5.62ns for double pre cision.

By the Reciprocation, Square root, Inverse Square Root using small multipliers [2], six 18x18 multipliers are used in single precision and twenty 18x18 multipliers are used in double precision.

With A Parameterizable Handle IC divider Generator for FPGA's with Embedded H/w Multiplier [7] there is a reduction in Latency to 4 with a Time Period of 33.03ns. The no of slices are 226 for single precision.

In terms of performance, the floating point library from Sandia Labs (Open Source High Performance Floating Point Modules) [10] is the best. The Sandia implementation reported here obtain high frequency of operation equal to 262MHz for single precision and 223MHz for double precision. The latency is nearly 38 cycles for single precision and for the double precision it is 67 cycles. The Number of slices used is 1174 for single precision and 4173 for double precision.

VI. Conclusion

Here the floating point Reciprocator for both IEEE single and double precision floating point numbers are implemented. The method uses the idea of neglecting higher order terms in the partial block multiplication to reduce the number of multipliers. At the same time, the look-up table requirements are kep to a minimum, and are the least reported in the literature for double precision implementation and shown double precision is more accurate than single precision. The simulation is done using ModelSim 6.3f and synthesized in Xilinx 10.1 and it is observed that the latency and area are more for double precision when compared to single precision and the speed can be improved by adding pipelining concept.

.

VII. References

[1] E. Antelo, T. Lang, P. Montuschi and A. Nannarelli, "Low latency digit-recurrence reciprocal and square-root reciprocal algorithm and architec-ture", 17th IEEE Symposium on Computer Arithmetic, P.P 147- 154, June-2005.

[2] H. Bessalah, M. Anane, M. Issad, N. Anane and K. Messaoudi, "Digit recurrence divider: Optimization and verification", International Conference on Design & Technology of Integrated Systems in Nanoscale Era, P.P 70-75, Sept-2007.

[3] R. Goldberg, G. Even and P. M. Seidel, "An FPGA implementation of pipelined multiplicative division with IEEE Rounding", *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, Pages 185-196, Apr-2007.

[4] J. Hopf, "A parameterizable HandeIC divider generator for FPGAs with embedded hardware multipliers", IEEE International Conference on Field-Programmable Technology, P.P 355-358, Dec-2004.

[5] D. Milos Ercegovac, Tomas Lang, Jean-Michel Muller, Arnaud Tisserand, "Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers", IEEE Transactions on Computers, Issue 7, VOL. 49, July-2000.

[6] K. Scott Hemmert, D. Keith Underwood, "Open Source High Performance Floating-Point Modules", 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06), pages 349-350, April-2006.

[7] K. Shamna and S.R Ramesh, "Design and Implementation of an Optimized Double Precision Floating Point Divider on FPGA", International Journal of Advanced Science and Technology, Vol. 18, May, 2010

[8] Xiaojun Wang, B. E. Nelson, "Tradeoffs of designing floating-point division and square root on Virtex FPGAs", 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003), P.P 195- 203, Apr-2003

[9] Xiaojun Wang, Sherman Braganza and Miriam Leeser, "Advanced Components in the Variable Precision Floating-Point Library", 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM-06), P.P 249-258, April-2006.

[10] Xilinx Floating-point unit v2.0. [Online]. Available: www.xilinx.com

http://babbage.cs.qc.edu/IEEE-754/Decimal.html

B.Mahesh is pursuing M.Tech with the specialization of VLSI in Sri Sai Aditya Institute of Science & Technology, Surampalem. He received the B.Tech degree in Electronics & Communication Engineering from Aditya Engineering College, Surampalem in 2010.

G.Sridevi working as Associate Professor in Department of Electronics and Communication Engineering, registered PhD in the area of Image Processing at JNTUK, Kakinada. Having Teaching Experience of about 12 years.