Hardware Implementation of Double precision Floating-Point Reciprocator on FPGA

Naga Valli. Vegesna  
Dept of ECE  
Shri Vishnu Engineering College for women  
E-mail: valli.vegesna@gmail.com

V. Srinivasa Rao  
Dept of ECE  
Shri Vishnu Engineering College for women  
E-mail: vemu1974@gmail.com

Abstract—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 most important in many modern applications. Most of the previous implementations required larger area and latencies. In this paper we have presented an efficient FPGA implementation of a reciprocator for double-precision floating point numbers. The method is based on the use of small look-up tables and partial block multipliers. The modules occupy less area and less latency.

Keywords: Double precision, Floating-point arithmetic, reciprocator, partial block-multipliers, FPGA.

I. INTRODUCTION

Floating point arithmetic like addition, multiplication, division and square root etc are mostly used in modern applications. Mainly in scientific and signal processing applications. The greater dynamic range and lack of need to scale the numbers makes development of algorithms much easier. Implementing of arithmetic operations for floating point numbers in hardware is very difficult. Among the operations division is most difficult to implement in hardware. So to make floating point division simple we are going for binomial expansion method.

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 for faster and better implementations.

There are many algorithms to implement division. Many algorithms were developed for division which includes subtractive method, functional iterations, Digit recurrence method, seed architecture which uses multipliers and algorithms for faster computation of division like high radix algorithm. But these algorithms uses huge look up tables along with wider multipliers which affects the area and performance.

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

II. APPROACH

The format of a floating-point number is as follows in Fig:1 for Single Precision and Double Precision:

Fig:1 format for single and double precision

We are using Xilinx ISE synthesis tool, ModelSim 6.4c simulation tool, and FPGA as our platform.

Let \( y \) be the inverse of the mantissa \( a \). Then,

\[
X = \frac{1}{a}, \text{ where } 1.a \text{ is hidden bit of mantissa.}
\]

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

\[
X = 1/(a_1+a_2) = (a_1+a_2)^{-1}
\]

\[
a_1^{-1}a_2^{-2}a_1+a_1^{-1}a_2^2a_1-33 \ldots \ldots (i)
\]

The content of each term of equation (1) will be as follows:
First we convert the required decimal number in to IEEE 754 floating point number using IEEE 754 decimal to
precision and double precision separation method which contains small look up tables, the finite number of bits
used to represent the numbers. i.e. u(v + w) may not be exactly equal to (uv + uw). This is 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

An algorithm for single precision and double precision floating point reciprocal is implemented using Binomial
Expansion method which contains small look up tables, and partial block multipliers, resulting in less area, less
delay. We have shown the implementations for single precision and double precision separately as different
issues arise in each case.

First we convert the required decimal number in to IEEE 754 floating point number using IEEE 754 decimal to
floating point converter and mantissa is inverted.

If we go for higher terms, contribution to main results decreases. Thus, depending upon our precision number of terms can be
taken from equation (1) for calculating inverse, based on value of m.

For our implementation, based on experiments over a large number of random test cases, we have chosen the number of
terms as described below. 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.

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

\[ Y = a_1^{-1}(a_1^{-1}a_2a_3 + a_1^{-1}a_2^{-1}a_3^{-1}) \]

The above equation can be little more simplified but it affects 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.

A. Single-precision Floating-point

The architecture of single-precision floating-point reciprocal is shown in Fig. 1. It includes a Block-Memory
(BRAM) which contains pre-calculated values of \( a_1^{-1}(24 \text{ bits}) \), \( a_2(17 \text{ bits}) \), and \( a_3(17 \text{ bits}) \) 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 most significant bit
(LSB’s) for addition/subtraction purpose. 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. 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 reciprocal is shown in Fig. 2. It also includes a single
BRAM which contains pre-calculated values of only \( a_1^{-1}(54 \text{ bits}) \) 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 “1111111” 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 MUL18x18 block) that have been used. Second, third and
seventh stage has 51-bit partial multiplier, which is shown in Fig. 3. 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 it’s specific input nature (17-bits of first input is 0x10000 in hex), it contains only three-MULT18x18 block Fig. 4. 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 (shown in Fig. 5) 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 can increase further using pipelining as discussed in the case of single precision, for better performance.

IV. RESULTS

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. For the error performance 5-

Fig. 3. Architecture for double-precision floating-point reciprocator.

Fig. 4. Partial 51-bit multiplier for stages 2, 3 and 7

Fig. 5. Partial 51-bit multiplier for stage 6

Fig. 6. 34-bit block multiplier for stage 4

Millions randomly generated test cases were used to check the errors. The error performance is shown in Table-II for both versions of floating-point numbers. The error was obtained by comparing results from the proposed module with the results produced by a C compiler on a workstation. In all cases, it was found that the maximum error in the case of single precision was 2 ulp (unit last place), while in the case of double precision numbers, it was 1 ulp. The error we got is without rounding.
V. CONCLUSION
We have implemented an efficient reciprocal unit on FPGA for both single and double precision floating-point numbers. 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 kept to a minimum, and are the least reported in the literature for double precision implementation. Initial latency for our module is also less (4 for single and 8 for double-precision), that too with promising frequency, which we can improve by pipelining them very easily. The error performance is also within acceptable range (1-ulp for double-precision).

The implementation can thus form a useful core for use in hardware dividers, especially for applications like signal processing that could be more tolerant of inaccuracies in the least significant bits.

SINGLE PRECISION: DOUBLE PRECISION

Resource Comparison:

<table>
<thead>
<tr>
<th>Method</th>
<th>Single-precision</th>
<th>Double-precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNR</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>[2]</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>[5][10]</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>[6][11]</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>[7]</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>[9]</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>[12]</td>
<td>-</td>
<td>impractical</td>
</tr>
<tr>
<td>Proposed</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

VI. REFERENCES


