An FPGA based Implementation of Floating-point Multiplier

L. Rajesh, Prashant V. Joshi and Dr. S.S. Manvi

Abstract—In this paper we describe the parameterization, implementation and evaluation of floating-point multipliers for FPGAs. We have developed a method, based on the VHDL language, for producing technology-independent pipelined designs that allow compile-time parameterization of design precision and range, and optional inclusion of features such as overflow protection, gradual underflow and rounding modes of the IEEE floating-point format. The resulting designs, implemented in a Xilinx Spartan-3 device, achieve 288 MFLOPs with IEEE single precision floating-point numbers.

Keywords—FPGAs, Floating point arithmetic, multiplication, CAD design flow.

I. INTRODUCTION

Floating-point operations are useful for computations involving large dynamic range, but they require significantly more resources than integer operations. The rapid advance in Field-Programmable Gate Array (FPGA) technology makes such devices increasingly attractive for implementing floating-point arithmetic. FPGAs offer reduced development time and costs compared to application specific integrated circuits, and their flexibility enables field upgrade and adaptation of hardware to run-time conditions[1]

Our main aims of designing floating-point units (FPUs) for reconfigurable hardware implementation are: (a) to parameterise the precision and range of the floating-point format to allow optimizing the FPUs for specific applications, and (b) to support optional inclusion of features such as gradual underflow and rounding. An approach meeting these aims will achieve effective design tradeoff between performance and required resources.

II. FLOATING POINT FORMAT

The IEEE Standard for Binary Floating Point Arithmetic(ANSI/IEEE Std 754-2008)[2] is used. The single precision format is shown in Figure 1. Numbers in this format are composed of the following three fields:

- 1-bit sign, S: A value of ‘1’ indicates that the number is negative, and a ‘0’ indicates a positive number.
- Bias-127 exponent, e = E + bias: This gives us an exponent range from Emin = -126 to Emax = 127.
- Mantissa, M: A twenty three bit fraction, an bit is added to the fraction to form what is called the significand. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number; in this case the real number is represented by [2].

\[
\text{Value} = (-1)^{S} \times 2^{(E - Bias)} \times (1.M)
\]

Where \( M = m_{22}.2^{-1} + m_{21}.2^{-2} + m_{20}.2^{-3} + \ldots \ldots + m_{1}.2^{-22} + m_{0}.2^{-23} \)

Bias = 127.

III. FLOATING POINT MULTIPLICATION ALGORITHM

Multiplying two numbers in floating point format is done by 1- adding the exponent of the two numbers then subtracting the bias from their result, 2- multiplying the significand of the two numbers, and 3- calculating the sign by XORing the sign of the two numbers. In order to represent the multiplication result as a normalized number there should be 1 in the MSB of the result (leading one).

As stated in the floating point format, normalized floating point numbers have the form of as

\[
\text{Value} = (-1)S \times 2(E - Bias) \times (1.M).
\]

To multiply two floating point numbers the following is done:

1. Multiplying the significand\((1.M_1*1.M_2)\)
2. Placing the decimal point in the result
3. Adding the exponents; i.e.\((E_1 + E_2 - \text{Bias})\)
4. Obtaining the sign; i.e. s1 xor s2
5. Normalizing the result; i.e. obtaining 1 at the MSB of the results’ significand
6. Rounding the result to fit in the available bits
7. Checking for underflow/overflow occurrence

Consider a floating point representation similar to the IEEE 754 single precision floating point format, but with a reduced number of mantissa bits (only 4) while still retaining the hidden ‘1’ bit for normalized numbers:

\[
\begin{align*}
A &= 0.10001000 \quad 0100 = 40, \\
B &= 1.10000001 \quad 1110 = -7.5
\end{align*}
\]

To multiply A and B

\[
\text{Equation}\]
1. Multiply significand: 1.0100 × 1.1110
   00000
   10100
   10100
   10100
   1001011000
2. Place the decimal point: 10.01011000
3. Add exponents: 10000100 + 10000001 = 10000101
   The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent;
   i.e. EA = EA−true + bias and EB = EB−true + bias And EA + EB = EA−true + EB−true + 2 bias
   So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.
   10000101
   -01111111
   10000110
4. Obtain the sign bit and put the result together: 10000110 10.01011000
5. Normalize the result so that there is a 1 just before the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1; moving one place to the right decrements the exponent by 1.
   1 10000110 10.01011000 (before normalizing)
   1 10000111 1.001011000 (normalized)
The result is (without the hidden bit):
   1 10000111 0010
6. The mantissa bits are more than 4 bits (mantissa available bits); rounding is needed. If we applied the truncation rounding mode then the stored value is: 1 10000111 0010.
Fig. 2 shows the multiplier structure; Exponents addition, Significand multiplication, and Result’s sign calculation are independent and are done in parallel. The significand multiplication is done on two 24 bit numbers and results in a 48 bit product, which we will call the intermediate product (IP). The IP is represented as (47 downto 0) and the decimal point is located between bits 46 and 45 in the IP. The following sections detail each block of the floating point multiplier.

IV. HARDWARE OF FLOATING POINT MULTIPLIER

A. Sign bit calculation

Multiplying two numbers results in a negative sign number if one of the multiplied numbers is of a negative value. By the aid of a truth table we find that this can be obtained by XORing the sign of two inputs.

B. Unsigned Adder/Subtractor (for exponent addition)

To reduce the delay caused by the effect of carry propagation in the ripple carry adder, we attempt to evaluate the carry-out for each stage (same as carry-in to next stage) concurrently with the computation of the sum bit [3, 4]. The two Boolean functions for the sum and carry are as follows [5,6]:

\[ \text{SUM} = A_i \oplus B_i \oplus C_i \]
\[ \text{Cout} = C_{i+1} = A_i \cdot B_i + (A_i \oplus B_i) \cdot C_i \]

Let \( G_i = A_i \cdot B_i \) be the carry generate function and \( P_i = (A_i \oplus B_i) \) be the carry propagate function, Then we can rewrite the carry function as follows:
\[ C_{i+1} = G_i + P_i \cdot C_i \]

Thus, for 4-bit adder, we can compute the carry for all the stages as shown below:
\[ C_1 = G_0 + P_0 \cdot C_0 \]
\[ C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0 \]
\[ C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0 \]
\[ C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0 \]

In general, we can write:
The sum function:
\[ \text{SUM}_i = A_i \cdot B_i \oplus C_i = P_i \oplus C_i \]
The carry function
\[ C_{i+1} = \sum_{k=0}^{i-1} \left( G_i \prod_{k=0}^{i-1} P_k + \prod_{k=0}^{i-1} P_k C_{in} \right) \]

Carry Look Ahead Adder can produce carries faster due to parallel generation of the carry bits by using additional circuitry.
The improvement in 4-bit adder is the key requirement to improve its efficiency, thus the authors propose the replacement of its 4-bit adder block with its carry look-ahead counter part. Recently, a modified carry look-ahead adder (abbreviated as MCLA) is proposed which is similar to CLA (carry look-ahead adder) in basic construction [7]. The drawback of MCLA [7] is that in spite of having significant speed improvement; it has significant increase in the area due to excessive number of NAND gates used for the faster carry propagation. This problem will significantly increase when the MCLA will be used for designing higher order CLA. This is the driving force that has led us to the proposal of new carry look-ahead adder modifying the structure of MCLA to make it more economical in terms of number of gates (area) without losing its speed efficiency. The MCLA [7] uses the modified full adder (MFA) as shown in Fig 3. In the proposed carry look-ahead adder shown in Fig. 4, we propose the replacement of 4th MFA in the MCLA by a full adder to reduce the area (number of gates) without sacrificing the speed improvement. It can be easily be verified that there will be reduction in number of gates to generate the final carry as shown in Fig.5. In order to have further saving in terms of number of gates, the proposed 4-bit CLA can be cascaded in series to design the higher width CLA as shown in Fig. 5.

![Fig3: MFA( modified full adder)](image1)

![Fig4: proposed 4-bit Carry Look-Ahead Adder](image2)

![Fig5: Cascading of the proposed CLA to Design Higher Width CLA](image3)

![Fig6: Proposed Carry Look-Ahead Nine’s Complementer](image4)

Fig6: Proposed Carry Look-Ahead Nine’s Complementer

Figure 6 shows the proposed Nine’s complementer using the proposed carry look-ahead adder. The proposed nine’s complementer sticks to the requirement of carry look-ahead approach having fast speed and reduced area(inherit property of proposed CLA).

**Carry Look Ahead BCD Subtractor**

By looking the key components designed in carry look-ahead fashion, the carry look-ahead BCD subtractor can be designed by integrating them. Figure 7 shows the implementation of the carry look-ahead BCD subtractor.

![Fig7: Proposed Carry Look-Ahead BCD Subtractor](image5)

**C. Multiplier for Unsigned Data**

Multiplication involves the generation of partial products, one for each digit in the multiplier, as in Fig 8. These partial products are then summed to produce the final product. The multiplication of two n-bit binary integers results in a product of up to 2n bits in length [8].

![Fig8: A Schematic of multiplier](image6)

**D. Normalizer**

The result of the significand multiplication (intermediate product) must be normalized to have a leading ‘1’ just to the left of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47

1- If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed.
2. If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1.

The shift operation is done using combinational shift logic made by multiplexers. Fig 9 shows a simplified logic of a Normalizer that has an 8 bit intermediate product input and a 6bit intermediate exponent input.

V. UNDERFLOW/OVERFLOW DETECTION

Overflow/underflow means that the result’s exponent is too large/small to be represented in the exponent field. The exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one.

An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition may be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it’s an underflow that can never be compensated; if the intermediate exponent = 0 then it’s an underflow that may be compensated during normalization by adding 1 to it. When an overflow occurs an overflow flag signal goes high and the result turns to ±Infinity (sign determined according to the sign of the floating point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs). Denormalized numbers are signaled to Zero with the appropriate sign calculated from the inputs and an underflow flag is raised. Assume that E1 and E2 are the exponents of the two numbers A and B respectively; the result’s exponent is calculated by (6)

\[ E_{result} = E1 + E2 - 127 \]

E1 and E2 can have the values from 1 to 254; resulting in Eresult having values from -125 (2-127) to 381 (508-127); but for normalized numbers, Eresult can only have the values from 1 to 254. Table I summarizes the Eresult different values and the effect of normalization on it.

VI. PIPELINING THE MULTIPLIER

In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier.

The pipelining stages are imbedded at the following locations:

1. In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).
2. After the significand multiplier, and after the exponent adder.
3. At the floating point multiplier outputs (sign, exponent and mantissa bits).

Fig 10 shows the pipelining stages as dotted lines.

Three pipelining stages mean that there is latency in the output by three clocks. The synthesis tool “retiming” option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

VII. COMPARISONS AND VHDL SIMULATION

First the VHDL simulation of two multipliers is considered. The VHDL code for both multipliers, using a fast carry look ahead adder and a ripple carry adder, are generated. The multiplier uses 24-bit values. The worst case was applied to the two multipliers, where the gate delay is assumed to be 5ns.
Under the worst case, the multiplier with a ripple adder uses time=979.056ns, while the multiplier with the carry look-ahead uses time=659.292ns. Hence we can overcome the ripple carry adder by carry-look ahead adder.

The whole multiplier was tested against the Xilinx floating point multiplier core generated by Xilinx coregen. Xilinx coregen was customized to have two flags to indicate overflow and underflow, and to have a maximum latency of three cycles. Xilinx core implements the "round to nearest" rounding mode. A testbench is used to generate the stimulus and applies it to the implemented floating point multiplier and to the Xilinx core then compares the result. The floating point multiplier was also checked using Design checker. The design was synthesized using precision synthesis tool [9] targeted to XC3S1500-5FG456 Spartan-3 device.

VIII. CONCLUSIONS AND FUTURE WORK

This paper presents an implementation of a floating point multiplier that supports the IEEE 754-2008 binary interchange format; the multiplier doesn’t implement rounding and just presents the significand multiplication result as is (48 bits); this gives better precision if the whole 48 bits are utilized in another unit. The design has three pipelining stages and after implementation on a Xilinx Spartan3 FPGA it achieves 288 MFLOPs.

REFERENCES