International Journal of advancement in electronics and computer engineering (IJAECE)
Volume 8, Issue 2, Feb. 2018, pp.101-105, ISSN 2278 -1412

Survey paper on Floating Point Multiplier Architectures on FPGA

Osho Patil[1], Paresh Rawat[2]
[1]*MTech Scholar, DC (ECE), TIEIT Bhopal (RGPV), anshu3040092@gmail.com, India;*
[2,] *HOD, ECE, TIEIT Bhopal (RGPV), parrawat@gmail.com, India;*

Abstract
This paper presents FPGAs used to be fixed-point. Floating-point operations are useful for computations involving large dynamic range, but they require significantly more resources than integer operations. Multipliers play an important role in today's digital signal processing and various other applications. With advances in technology, many researchers have tried and are trying to design multipliers which offer either of the following design targets – high speed, low power consumption, regularity of layout and hence less area or even combination of them in one Multiplier thus making them suitable for various high speed, low power and compact VLSI implementation. Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks.

Keywords — *Floating Point Arithmetic, Multipliers, Digital Arithmetic, FPGA*

I. INTRODUCTION

The complexity of the algorithms, floating point operations are very hard to implement on FPGA. The computations for floating point operations involve large dynamic range, but the resources required for this operations is high compared with the integer operations. This multiplier is mainly used to multiply two floating point numbers. Separate algorithm is essential for multiplication of these numbers. Here multiplication operation is simple than addition this is especially true if we are using a 32-bit format. One of the way to represent real numbers in binary is the floating point formats. There are two different formats for the IEEE 754 standard. Binary interchange format and Decimal interchange format. In the multiplication of floating point numbers involves a large dynamic range which is useful in DSP applications. This paper concentrates only on single precision normalized Binary interchange format. The below figure shows the IEEE 754 single precision binary format representation; consisting of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a computer realization of scientific notation. Over the years, a variety of floating-point representations have been used in computers. However, since the 1990s, the most commonly encountered representation is that defined by the IEEE 754 Standard. The advantage of floating-point representation over fixed-point and integer representation is that it can support a much wider range of values.

Field Programmable Gate Arrays (FPGAs) were first introduced almost two and a half decades ago. Since then they have seen a rapid growth and have become a popular implementation media for digital circuits. The advancement in process technology has greatly enhanced the logic capacity of FPGAs and has in turn made them a viable implementation alternative for larger and complex designs. Further, programmable nature of their logic and routing resources has a dramatic effect on the quality of final device's area, speed, and power consumption. The programmable logic and routing interconnect of FPGAs makes them flexible and general purpose but at the same time it makes them larger, slower and more power consuming than standard cell ASICs. However, the advancement in process technology has enabled and necessitated a number of developments in the basic FPGA architecture. These developments are aimed at further improvement in the overall efficiency of FPGAs so that the gap between FPGAs and ASICs might be reduced. Field programmable Gate Arrays (FPGAs) are pre-fabricated silicon devices that can be electrically programmed in the field to become almost any kind of digital circuit or system. For low to medium volume productions, FPGAs provide cheaper solution and faster time to market as compared to Application Specific Integrated Circuits (ASIC) which normally require a lot of resources in terms of time and money to obtain first device. FPGAs on the other

hand take less than a minute to configure and they cost anywhere around a few hundred dollars to a few thousand dollars.

II. LITERATURE SURVEY

S. Banescu, et.al [1] "Multipliers for floating-point double precision and beyond on FPGAs," The implementation of high-precision floating-point applications on reconfigurable hardware requires large multipliers. Full multipliers are the core of floating-point multipliers. Truncated multipliers, trading resources for a well-controlled accuracy degradation, are useful building blocks in situations where a full multiplier is not needed.

This work studies the automated generation of such multipliers using the embedded multipliers and adders present in the DSP blocks of current FPGAs. The optimization of such multipliers is expressed as a tiling problem, where a tile represents a hardware multiplier, and super-tiles represent combinations of several hardware multipliers and adders, making efficient use of the DSP internal resources. This tiling technique is shown to adapt to full or truncated multipliers. It addresses arbitrary precisions including single, double but also the quadruple precision introduced by the IEEE-754-2008 standard and currently unsupported by processor hardware. An open-source implementation is provided in the FloPoCo project.

M. K. Jaiswal, et.al [2] "Area-Efficient FPGAImplementation of Quadruple Precision Floating Point Multiplier," This paper presents FPGA based hardware architectures for floating point (FP) multipliers. The proposed multiplier architectures are aimed for single precision (SP), double precision (DP), double-extended precision (DEP) and quadruple precision (QP) implementation. This paper follows the standard computational flow for FP multiplication. The mantissa multiplications, the most complex unit of the FP multiplication, are built using efficient use of Karatsuba methodology integrated with the optimized used of in-built 25x18 DSP48E blocks available on the Xilinx Virtex-5 onward FPGA devices. It also combined with the other techniques (radix-4 booth encoding for small multipliers, partial products reduction using 4:2, 3:2, 2:2 counters; compression of multioperands adders) used at places, to improve the design. The proposed architectures out-performs the available state-of-the art, and used only 1-DSP48, 3 DSP-48, 6 DSP48 and 18 DSP48 for SP, DP, DEP, and QP multipliers respectively.

S. Srinath et.al [3], "Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks," The introduction of asymmetric embedded multiplier blocks in recent Xilinx FPGAs complicates the design of larger multiplier sizes. The two different input bit widths of the embedded multipliers lead to two different shifting factors for the partial product outputs. This makes even the most straightforward multiplier design less intuitive. In this paper, we present a methodology and set of equations to automatically generate the Verilog for a multiplier using asymmetric embedded multiplier cores. The presented technique also uses intelligent rearrangement of the multiplier block outputs into partial product terms to reduce the overall delay of the circuit. Multipliers created with our generator are faster and use fewer DSP blocks than those created using Xilinx Core Generator or by simply using the '*' operator in Verilog. It also uses fewer LUTs than those created using the '*' operator. Finally, the presented generator can create multipliers larger than possible with Core Generator, and is limited only by the number of available embedded multipliers.

F. de Dinechin, et.al [4] "Large multipliers with fewer DSP blocks," Recent computing-oriented FPGAs feature DSP blocks including small embedded multipliers. A large integer multiplier, for instance for a double-precision floating-point multiplier, consumes many of these DSP blocks. This article studies three non-standard implementation techniques of large multipliers: the Karatsuba-Of man algorithm, non-standard multiplier tiling, and specialized squarers. They allow for large multipliers working at the peak frequency of the DSP blocks while reducing the DSP block usage. Their overhead in term of logic resources, if any, is much lower than that of emulating embedded multipliers. Their latency overhead, if any, is very small. Complete algorithmic descriptions are provided, carefully mapped on recent Xilinx and Altera devices, and validated by synthesis results.

M. K. Jaiswal et.al [5] "VLSI Implementation of Double-Precision Floating-Point Multiplier Using Karatsuba Technique," The double-precision floating-point arithmetic, specifically multiplication, is a widely used arithmetic operation for many scientific and signal processing applications. In general, the double-precision floating-point multiplier requires a large 53×53 mantissa multiplication in order to get the final result. This mantissa multiplication exists as a limit on both area and performance bounds of this operation. This paper presents a novel way to reduce this large multiplication. The proposed approach in this paper allows to use less amount of multiplication hardware compared to the traditional method. The multiplication is done by using Karatsuba technique. This design is specifically targeting Field Programmable Gate Array (FPGA) platforms, and it has also been evaluated on ASIC flow. The proposed module gives excellent performance with efficient use of resources. The design is fully compatible with the IEEE standard precision. The proposed module has

shown a better performance in comparison with the best reported multipliers in the literature.

F. de Dinechin, et.al [6] "Automatic generation of polynomial-based hardware architectures for function evaluation" Many applications require the evaluation of some function through polynomial approximation. This article details an architecture generator for this class of problems that I'm-proves upon the literature in two aspects. Firstly, it benefits from recent advances related to constrained-coefficient polynomial approximation. Secondly, it refines the error analysis of polynomial evaluation to reduce the size of the multipliers used. As a result, architectures for evaluating arbitrary functions with precisions up to 64 bits, makingefficient use of the resources of recent FPGAs, can be obtained in seconds. An open-source implementation is pro-vided in the FloPoCo project.

G. Govindu, et. al [7] Analysis of high-performance floating-point arithmetic on FPGAs." FPGAs are increasingly being used in the high performance and scientific computing community to implement floating-point based hardware accelerators. In this paper we analyze the floating-point multiplier and adder/subtractor units by considering the number of pipeline stages of the units as a parameter and use throughput/area as the metric. We achieve throughput rates of more than 240 MHz (200 MHz) for single (double) precision operations by deeply pipelining the units. To illustrate the impact of the floating-point units on a kernel, we implement a matrix multiplication kernel based on our floating-point units and show that a state-of-the-art FPGA device is capable of achieving about 15GFLOPS (8GFLOPS) for the single (double) precision floating-point based matrix multiplication. We also show that FPGAs are capable of achieving up to 6x improvement (for single precision) in terms of the GFLOPS/W (performance per unit power) metric over that of general purpose processors. We then discuss the impact of floating-point units on the design of an energy efficient architecture for the matrix multiply kernel.

III. PROBLEM STATEMENT

This paper represents previous paper problem statement are the mantissa multiplications, the most complex unit of the FP multiplication.

IV. METHOD ARCHITECTURE

The floating point multiplier implementation is relatively simple compared to other floating point arithmetic operations. The crucial part of the floating point multiplication lies in mantissa multiplication. This is the bottleneck in the performance of FPU multiplication. The mantissa of quadruple precision floating point numbers is 113-bit (including 1 hidden bit) in length and in general, this needs implementation of a large 113 × 113 multiplier in hardware, which is very expensive in terms of area as well as performance.

This massive part of the multiplication arithmetic operation. The basic underlying concept used here is karatsuba multiplication technique. We have divided the mantissa operands in a beautiful manner which leads us to achieve the better result by incorporation of the Karatsuba Method, on the FPGA based platform. Before proceeding to the detail implementation we first explain the basics of the Karatsuba Method. Karatsuba Multiplication is a fast multiplication algorithm. It reduces the multiplication of two n-digit numbers from simple $n2$ to at most $3n\log 2\ 3 \approx 3n1.585$ single digit multiplication. The basic steps for this algorithm depend on the divide and conquer paradigm and proceeds in the following ways.

Let $W\&X$ are two n-digit numbers. By decomposing these number in two parts, for some base B, we can rewrite them as below,

$$W = W_1.B^m + W_0$$

$$X = X_1.B^m + W_0$$

Where $W0$ & $X0$ are of m-digit. Now, we can write the Product of $W\&X$ as follows,

$$WX = (W_1.B^m + W_0)(X_1.B^m + X_0)$$

$$W_1.X_1.B^{2m} + (W_1.X_0 + W_0.X_1)B^m + W_0.X_0$$

$$= \sim B^{2m} + \quad .B^m + ¥$$

Where,

$$\sim = W_1.X_1, \quad = W_0X_1$$

$$¥ = W_0.X_0$$

These tell us to use four multiplications to get the complete result. But, Karatsuba method leads to the use of only three multiplications to get the complete result. This can be achieved as follows. We can modify the β as below,

$$= (W_1.X_0 + W_0.X_1) + (W_1.X_1 + W_0.X_0) - (W_1.X_1 + W_0.X_0)$$
$$= (W_1 + W_0)(X_1 + X_0) - W_1.X_1 - W_0.X_0$$
$$= (W_1 + W_0)(X_1 + X_0) - \sim - ¥$$

We can also have an another variant of β as follows,

$$= \sim + ¥ - (W_1 - W_0)(X_1 - X_0)$$

Which requires only one multiplication instead of two, with some extra overhead of addition and subtraction. Thus to get complete product of $W\&X$ we need three Multiplication instead of four. Similarly by extending this technique, and splitting the Operands into three parts, we reduce number of multipliers from 9 to 6. The details are described as follows:

We can divide the operands $W\&X$ as follows

$$W = W_2.B^{2m} + W_1.B^m + W_0$$

$$X = X_2.B^{2m} + X_1.B^m + X_0$$

Where W_1, X_1, W_0 *and* X_0 are of m-digit. Now, the product of $W\&X$ will be as follows.

$$W.X = (W_2.B^{2m} + W_1.B^m + W_0)$$
$$\times (X_2.B^{2m} + X_1.B^m + X_0)$$
$$W_2.X_2.B^{4m} + (W_2.X_1 + W_1.X_2)B^{3m} +$$
$$(W_2.X_0 + W_0.X_2)B^{2m} + W_1.X_1.B^{2m}$$
$$+ (W_1.X_0 + W_0.X_1)B^m + W_0.X_0$$

$$= \sim_2.B^{4m} + \sim_1.B^{2m} + \sim_0 . \ _2.B^{3m} + \ _1B^{2m} + \ _0.B^m$$

Where,

$$\sim_2 = W_2.X_2, \sim_1 = W_1.X_1, \sim_0 = W_0.X_0$$

And

$$_2 = (W_2.X_1 + W_1.X_2) + (W_2.X_2 + W_1.X_1)$$

$$_0 = W_1.X_0 + W_0.X_1$$

Up to this level we need 9 multiplier to accomplish the task. The number of multiplications can be reduced by modifying the $_{2,}\ _2 and\ _0$ as below.

$$_2 = (W_2.X_1 + W_1.X_2) + (W_2.X_2 + W_1.X_1)$$

$$-(W_2.X_2 - W_1.X_1)$$
$$= (W_2 + W_1)(X_2 + X_1) - \sim_2 - \sim_1$$

Similarly,

$$_1 = (W_2 + W_0)(X_2 + X_0) - \sim_2 - \sim_0$$
$$_0 = (W_1 + W_0)(X_1 + X_0) - \sim_1 - \sim_0$$

We can also have an another variant of βs as follows,

$$_2 = \sim_2 + \sim_1 - (W_2 - W_1)(X_2 - X_1)$$
$$_1 = \sim_2 + \sim_0 - (W_2 - W_0)(X_2 - X_0)$$
$$_0 = \sim_1 + \sim_0 - (W_1 - W_0)(X_2 - X_0)$$

In the present context, for the quadruple precision multiplication, we require 113×113 multiplier. For, this we have divided the operands in two unequal parts, 51-bit and 62-bit.Here the splitting of the operands is based on the availability of multiplier IP core on the Xilinx FPGA platform. Thus, as discussed above for the karatsuba method, we need one 51-bit multiplier, one 62-bit multiplier and one 63-bit multiplier.

Further, the keys lies in the effective implementation of these sub-multipliers. Based on the availability of 17 × 17 multiplier, we have implemented the 51-bit multiplier by three partitioning extension of Karatsuba method, and 62- bit & 63-bit multiplier have been implemented by two partitioning method. The 62-bit & 63-bit multiplier, further have divided in two parts, requires three 34-bit multipliers, for each.

The Implementation of each 34-bit multiplier needs three 17×17 multiplier by two partition method. Both of the 62- bit and 63-bit multiplier will require 9, 17 × 17 multiplier, individually. And, the 51-bit multiplier will need 6, 17×17multiplier, by three partitioning method. Thus, a total of 24, 17 × 17 multiplier blocks are required to do 113 × 113 multiplication, which has a large saving in terms of area. Whereas, by general approach 113 × 113 multiplication do need 49, 17×17 multiplier blocks. The detail implementation of these multiplication has been explained in the next section.

V. CONCLUSIONS

This paper presents Floating-point operations are useful for computations involving large dynamic range, but they require significantly more resources than integer operations. Multipliers play an important role in today's digital signal processing and various other applications.

REFERENCES

[1] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran,"Multipliers for floating-point double precision and beyond on FPGAs," SIGARCH Comput. Archit. News, vol. 38, pp.73–79, Jan 2011.

[2] M. K. Jaiswal and R. C. C. Cheung, "Area-Efficient FPGA Implementation of Quadrupl Precision Floating Point Multiplier, "in The IEEE 26th

International Parallel and Distributed Processing Symposium Workshops & PhD Forum(IPDPSW 2012). Shanghai, China: IEEE Computer Society, May 2012, pp. 369–375.

[3] X. Wang and M. Leeser, "Vfloat: A variable precision fixed and floating-point library for reconfigurable hardware," ACM Trans. Reconfigurable Technol. Syst., vol. 3, no. 3, pp. 16:1–16:34, Sep. 2010.

[4] S. Srinath and K. Compton, "Automatic generation of high-performance multipliers for fpgas with asymmetric multiplier blocks," in Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, ser. FPGA '10, 2010, pp. 51–58.

[5] M. K. Jaiswal and R. C. C. Cheung, "Area-efficient architectures for double precision multiplier on FPGA, with run-time-reconfigurable dual single precision support, "Microelectronics Journal, vol. 44, no. 5, pp. 421–430, May 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026269213000591

[6] F. de Dinechin, "Large multipliers with fewer DSP blocks," in International Conference on Field Programmable Logic and Applications, 2009, pp. 250–255.

[7] M. K. Jaiswal and R. C. C. Cheung, "VLSI Implementation of Double-Precision Floating-Point Multiplier Using Karatsuba Technique," Circuits, Systems, and Signal Processing, vol. 32, pp. 15–27, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00034-012-9457-3

[8] A. Karatsuba and Y. Of man, "Multiplication of Many-Digital Numbers by Automatic Computers," in Proceedings of the USSR Academy of Sciences, vol. 145, 1962, pp. 293–294.

[9] "IEEE standard for floating-point arithmetic," IEEE Std 754-2008, pp. 1–70, Aug 2008.

[10] Xilinx, "Logic ORE IP Floating-Point Operatorv5.0." [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf

[11] "Logic ORE IP Floating-Point Operator v7.1." [Online].Available: http://www.xilinx.com/support/documentation/ipdocumentation/ru/floating-point.html