TITLE

Development of a Scalable FPGA-Based Floating Point Multiplier

TOPIC NUMBER

8. Field Programmable Gate Arrays

AUTHOR(S)

Name: Manuel A. Jiménez
Affiliation: Department of Electrical Engineering
Michigan State University
East Lansing, MI 48824

Name: Nayda G. Santiago
Affiliation: Department of Electrical Engineering
Michigan State University
East Lansing, MI 48824

Name[1]**:** Diane T. Rover
Affiliation: Department of Electrical Engineering
260 Engineering Building
Michigan State University
East Lansing, MI 48824
Telephone: (517) 355-5066
Fax: (517) 353-1980
E-mail: rover@egr.msu.edu

1. Contact person

Development of a Scalable FPGA-Based Floating Point Multiplier

Summary

This paper presents the implementation of a general purpose, scalable architecture used to synthesize floating point multipliers on FPGAs. Although several implementations of floating point units targeted to FPGAs have been previously reported, most of them are customized for specific applications. This new implementation is different in the sense that it accepts as a user parameter the operand size of the unit about to be synthesized and creates the requested unit. This feature makes our implementation a very convenient tool for rapid application prototyping. An evaluation of several multipliers of different typical sizes synthesized for testing purposes is also presented, highlighting the strengths and limitations of this approach in the creation of custom floating-point units. A comparison between our results and some previously reported implementations shows that our approach, in addition to the scalability feature, provides multipliers with significant improvements in area and speed.

Development of a Scalable FPGA-Based Floating Point Multiplier

Manuel A. Jiménez, Nayda G. Santiago, and Diane T. Rover

Electrical Engineering Department
Michigan State University
East Lansing, MI 48824

Abstract

This paper presents a scalable architecture for developing custom floating point multipliers targeted to FPGA synthesis. An evaluation of several multipliers of different typical sizes synthesized for testing purposes is also presented, highlighting the strengths and limitations of this approach in the creation of custom floating point units.

1 Introduction

The reconfigurability and programmability of Field Programmable Gate Arrays (FPGAs) make them attractive tools for implementing digital signal processing (DSP) applications. However, DSP applications are arithmetic intensive tasks, requiring in most cases floating point operations to be performed as part of the application itself [6]. This creates a necessity for some off-the-shelf floating point units which could be used in these applications without deviating the designer's attention from the main problem being solved. A number of custom implementations of fixed and floating point units have been reported for several FPGA architectures [5], [7], [2] most of them aimed at solving specific problems.

This paper presents the implementation of a general purpose, scalable architecture used to build floating point multipliers on FPGAs. What makes this implementation different and general purpose is its flexibility in accepting as a user parameter the operand size of the unit about to be synthesized. This feature makes our implementation a very convenient tool for rapid application prototyping.

The next section explains the format used to represent the FP numbers in the implementation. The multiplier structure, along with the approach for writing the scalable circuit description are explained in section three. Section four presents the results of synthesizing five different sizes of multipliers using the developed tool. This section also includes a comparison of units generated with our tool against similar units previously reported, allowing us to evaluate their efficiency.

2 Operand Format

In order to facilitate the integration of the multipliers into the diverse applications that might require them, we decided to follow as closely as possible the representation format of the IEEE 754 standard for single-precision numbers. The selection of this format allowed us not only to adhere to a well known representation format, but also to take advantage of some features of the standard. These features include the reduction of the representation error, the

ability of representing the inverse of any representable number without causing either overflow or underflow errors, and giving support to infinity values in operations [3].

The number of bits allocated for a number represented in this format is divided into three main fields, as illustrated in Figure 1. A brief description of these three fields follows.

i) The mantissa *f*, held in the rightmost field of the operand, is represented as an *m*-bit unsigned-magnitude number, plus an implicit hidden +1 bit. Mantissa values are normalized to reduce the number of leading zeros in the field, increasing the precision. Once normalized and augmented, the resulting value lies in the range [1,2).

ii) The sign bit S, held in the leftmost position of the *n*-bit word, is set to one to represent negative mantissas, or to zero otherwise.

iii) The exponent E, an *e*-bit wide field, uses an excess $2^{ebit-1}-1$ encoding (*bias*) to represent the actual exponent value. This allows, among other things, to represent all numbers in the format and their inverses without causing overflow/underflow conditions.

Some additional features supported by the 754 standard, and adopted in our representation include: a zero value is represented by E=0, *f*=0; an infinity value ($\pm\infty$) is represented as E=1..1, *f*=1..1 (all ones in both fields), and sign bit accordingly. Overflow and underflow indications are provided through two status bits. Denormalized numbers and NaN representations are not supported in this implementation.

The value of a floating point number *F* in this format can be obtained as follows:

$$F = (-1)^{\dot{S}} \times 1.f \times 2^{E-bias}$$

3 Multiplier Structure

The multiplication of two floating point numbers involves three steps: adding the exponents, multiplying the mantissas, and operating on the signs of the values. The exponent addition is performed as an integer operation, requiring an adjustment in the result to subtract the redundant exponent bias. The mantissa multiplication is also performed as an integer operation, from which only the most significant *m* bits are taken. The sign of the result is computed by performing an exclusive-or operation on the signs of the input values. A postnormalization step might be required if the resulting product is equal to or larger than two. This step is commonly achieved by shifting its mantissa one position to the right, and adding one to the resulting exponent.

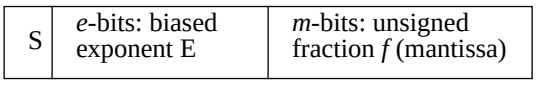

Figure 1: Format of a floating-point number.

In addition, the bounds of the results must be checked in order to detect possible overflow/underflow conditions which may arise during the process.

In order to perform the above sequence of operations within an acceptable time frame, the multiplier structure is organized as a three-stage pipeline. This arrangement allows the system to produce one result every clock cycle, after the first three values are entered into the unit.

Figure 2 shows a block diagram with the structure of the multiplier pipeline. Each stage in this pipeline performs one or more simultaneous operations, as described below.

Stage 1 performs the addition of the exponents, the multiplication of the mantissas, and the exclusive-or of the signs. Stage 2 takes these results as inputs, removing the redundant bias in the exponent, postnormalizing the resulting mantissa, and passing the sign. Stage 3 is devoted to rounding and representing the result according the overflow/underflow conditions which might be generated during the process.

3.1 Circuit Design

We designed the circuit for the multiplier in VHDL [4] as a structural description composed of several design hierarchies.

The whole design is accessed as a structural COMPONENT which accepts as GENERIC parameters the sizes of the operand fields. These parameters are passed along the whole hierarchical structure specifying the widths of the mantissa and exponent fields, and accordingly generating the appropriately sized units required. The following paragraphs provide an insight into the multiplier's hierarchy.

A floating-point multiplier is based on two types of integer units: an array multiplier and a look-ahead adder. Each

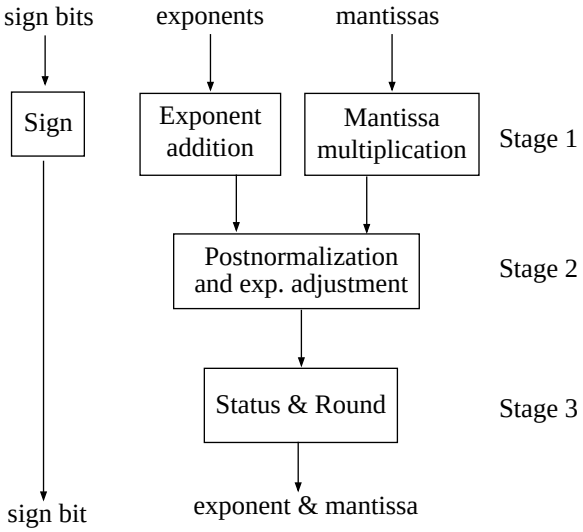

Figure 2: Block diagram of the multiplier pipeline.

of these units is designed as a scalable component which, upon instantiation, will accept a generic parameter specifying the width of the operands it will be synthesized for.

Three instances of the adder are used: The first adder in stage one of the pipeline performs the addition of the exponents. A second instance is used as part of the array multiplier to perform the addition of the last level of partial sums. The third adder is instantiated in stage two of the pipeline to perform the postnormalization and exponent adjustment step.

A single instance of the array multiplier is used in stage one of the pipeline to perform the multiplication of the mantissas. The most significant bit of the product is used as a postnormalization flag in stage two.

The overflow/underflow checking and corresponding corrections in the result are performed in stage three of the pipeline. Two status flags provide external notification of these conditions. The rounding step, also performed in stage three of the pipeline, is accomplished by simply truncating the result to the number of bits required by the operand size. Figure 3 shows a functional diagram of the FP multiplier pipeline, illustrating the interaction of the processes and components which comprise it.

Both fixed units, adder and multiplier, integrate other subcomponents whose number is variable depending on the size of the operands specified. The hierarchy of entities in the FP multiplier is summarized below:

```
FP_Mult(m,e)                -- Top level component
    Look_aheadN(n)          -- Scalable integer adder
        Lk_ah1              -- Single-bit look ahead adder
        Lk_ah2              -- Two-bit look ahead adder
        Lk_ah3              -- Three-bit look ahead adder
        Lk-ah4              -- Four-bit look ahead adder
    Array_MultN(m+1)        -- Scalable array multiplier
        mpyh_cell           -- Half multiplier cell
        mpyf_cell           -- Full multiplier cell
        Look_aheadN(m)      -- Scalable integer adder
            (adder sub-components as above)
    -- end-of-hierarchy --
```

A scalable integer adder is formed with a combination of multiple carry look-ahead modules connected in ripple carry fashion to form a single *n*-bit adder. As a whole, the integer adder is very flexible in the sense that it accepts as a generic parameter the width of the operands (*nbit*), and generates an appropriately sized unit using $\lfloor (nbit)/4 \rfloor$ 4-bit look-ahead adders plus a look-ahead unit of size mod(*nbit*,4). The top level entity determines how many four-bit adders are required for a given operand width, as well as the size of the look-ahead unit for the remaining bits (if any) to complete the unit.

A similar approach is followed for the scalability of the array multiplier. The top level entity of this component accepts a parameter specifying the operand size (*nbit*). This value is used to generate a two-dimensional array of multiplier cells of order $((nbit-1)\times(nbit-1))$. Each multiplier cell is essentially a (3,2) counter with an AND gate

which accumulates the intermediate sums and carry bits. The last row in the array is completed with a look-ahead adder of size (*nbit*) whose output corresponds to the upper-half of the product.

The whole structure for the floating point unit can be integrated into a given application by instantiating it as a component, whose interface specification is the simple description given below (the default values e=6 and m=5 are overriden by the component instantiation):

```
ENTITY fp_mult IS
   GENERIC (ebit : INTEGER := 6; mbit : INTEGER :=5);
   PORT (fp1,fp2 : IN BIT_VECTOR(ebit+mbit DOWNTO 0);
         clk  : IN BIT;
         stat : OUT BIT_VECTOR (1 DOWNTO 0);
              -- msb = overflow, lsb = underflow
         fpr : OUT BIT_VECTOR(ebit+mbit DOWNTO 0));
END fp_mult;
```

4 Results and Analysis

To evaluate the efficiency of our approach, five different floating point multipliers were synthesized and tested. The target device used in the synthesis is a Xilinx 4010 FPGA, which provides 400 available CLBs [8]. The testing platform used in our evaluation is the SPLASH 2 Custom Computing Machine [1] hosted on a Sun Sparcstation. This platform allows us to perform exhaustive testing of the correct performance of units, providing a suitable interface for

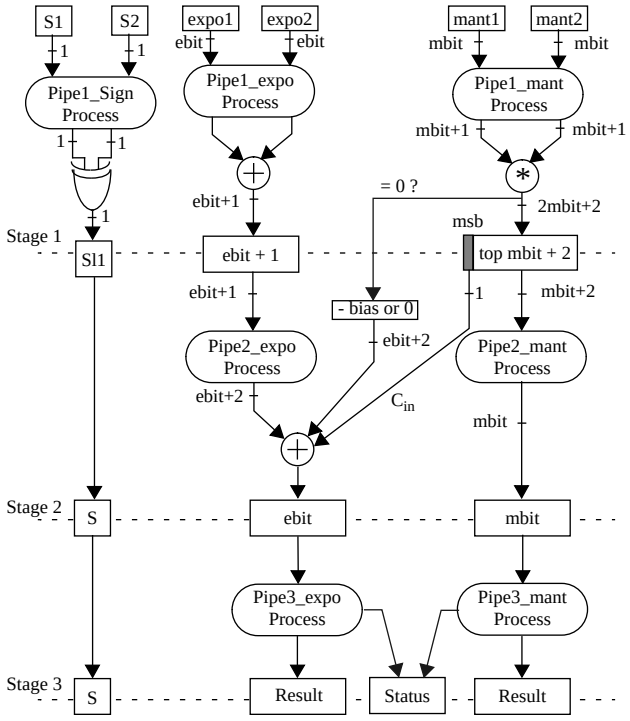

Figure 3: Functional diagram of the FP multiplier.

data exchange in most of the cases. A SPLASH board is composed of 16 Xilinx 4010 FPGAs (acting processing elements, PEs) plus an additional 4010 device devoted to control tasks. Data exchange among the processing elements on SPLASH is achieved through a 36-bit bus which connects the 16 PEs as a linear array synchronized by a global clock. For our tests, one of the PEs was targeted to contain the multiplier under test and the remaining PEs were configured as data passing elements to retrieve the results from the array.

Table 1 summarizes the results of the syntheses, indicating the unit sizes, area requirements, and maximum speed in each case. The second column shows the percent of FG function generators required by each of the synthesized units. This value is an indicator of the area resources required to implement the actual unit. The column indicating the occupied CLBs includes those containing the function generators, plus the feedthrough CLBs used for routing the design. The partitioning, placement, and routing (PPR) tools used were set to medium effort for the tested units. This implies that imposing tighter routing constraints the CLB occupation might be reduced further.

Table 1: Synthesis results of FP units.

Size of FP unit	FG Funct. Gen.	Max. Clk Speed	Used Flip Flops	Occup. CLBs
12-bit (e=6, m=5)	13%	10.8 MHz	5.5%	19%
16-bit (e=7, m=8)	25%	7.3 MHz	7.0%	32%
20-bit (e=8, m=11)	41%	5.3 MHz	8.5%	51%
24-bit (e=9, m=14)	62%	4.4 MHz	10%	74%
18-bit (e=7, m=10)	35%	6.2 MHz	7.7%	43%

The tested devices were designed in increments of one bit in the exponents and three bits in the mantissas, to produce units of 12, 16, 20, and 24 bits respectively. The field sizes for the first unit are 6-bit exponent and 5-bit mantissa. An additional 18-bit multiplier was also synthesized and tested, to compare our results with those reported by Shirazi et al. in [7] for a similar implementation. The compared units operate on 18-bit floating point numbers represented with a 10-bit mantissa (plus a hidden 1 bit) and a 7-bit exponent in a 3-stage pipeline. As can be appreciated in Table 2, our scalable implementation shows substantial improvements in both area and speed over the unit used as a reference.

The evaluation performed on the tested units also included a search of the limiting design factors, with the purpose of identifying targets for further improvements. This search was aimed essentially to identify the limitations in area, speed, and operand size.

Table 2: Comparison of results for 18-bit unit.

Parameter	Custom unit in [7]	Scalable unit	Improve-ment
FG Funct. Generators	44%	35%	20.5%
Flip Flops	14%	7.7%	45.0%
Stages	3	3	-
Speed	4.9MHz	6.2MHz	26.5%

The utilization of area resources is very close to our expectations. The structural approach taken for the circuit descriptions played a decisive role in these results, especially in the array multiplier, which is responsible for most of the used area. As an example, the 18-bit FP unit requires an 11-bit array multiplier, which uses 28% of the resources (80% of the FP unit area). This is equivalent to 113 CLBs, rendering an area complexity $O(nbit^2)$. This complexity result confirms our expectations that we can map a cell of the array multiplier to a CLB of the FPGA.

The critical factor limiting the maximum operating frequency of the designs was found to be the clock distribution network. As an example, in the 12-bit FP unit, the delay of the clock net is 92.4 ns, roughly the inverse of the maximum frequency of the unit. The next largest delay below that of the clock net is 32.6 ns in one of the data paths.

The scalable description does not explicitly limit the size of the largest FP multiplier that can be requested. In fact, for simulation purposes only FP multipliers for operands as large as 64 bits wide could be successfully specified. However, for synthesis purposes, the size of the largest unit is limited by the number of CLBs available in the target FPGA. Specifically, units with mantissa widths as large as 16 bits were found to take very long PPR CPU time, and not all paths were successfully routed. FP units with larger mantissa sizes will not fit in a single X4010 FPGA.

5 Conclusion

A scalable architecture for implementing floating point multipliers targeted to FPGA synthesis has been successfully developed and evaluated. Its strengths and limitations have been discussed.

Test results demonstrate its suitability for creating modestly sized FP units, which in most cases provide the accuracy and dynamic range characteristic needed in a number of DSP applications.

In addition to their scalability, the performance of the units created with this approach has been demonstrated to be higher than similar units previously reported in both area and speed.

Finally, the approach presented in this paper can also be used in the development other typical arithmetic structures. Currently, a floating-point adder using the same strategy is under development.

Acknowledgements

The authors would like to thank Sea Choi at MSU's Computer Science Department and Hari Vattikota at Xilinx Corp. for their valuable help in the development of this project.

References

[1] J. M. Arnold and M. A. McGarry. *Splash 2 Programmer's Manual*, Version 1.2. IDA Supercomputing Research Center, Jul. 1994.

[2] C. J. Chou, S. Mohanakrishnan, and J. Evans. FPGA Implementation of Digital Filters. In *Proceedings of the Fourth International Conference on Signal Processing Appications and Technology*, pages 80-88, Sep. 1993.

[3] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, Englewood Cliffs, 1993.

[4] Z. Navabi. *VHDL Analysis and Modeling of Digital Systems*. Mc Graw-Hill, Inc., 1993.

[5] J. H. Novak and E. Brunvand. Using FPGAs to Prototype a Self-Timed Floating Point Co-Processor. In *Proceedings of the IEEE 1994 Custom Integrated Circuits Conference*, pages 85-88, Jan. 1994.

[6] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*, Second Edition. Macmillan, 1992.

[7] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155-162, Apr. 1995.

[8] Xilinx, Inc. *The Programmable Logic Data Book*. 1994.