# Efficient Implementation of Pipelined Double Precision Floating Point Multiplier

**Riya Saini*, R.D.Daruwala****
M.Tech Student*
*(Department of Electrical Engineering, Veermata Jijabai Technological Institute, Matunga, Mumbai-19)
Professor & Dean**
** (Department of Electrical Engineering, Veermata Jijabai Technological Institute, Matunga, Mumbai-19)

## ABSTRACT

**Floating-point numbers are widely adopted in many applications due their dynamic representation capabilities. Floating-point representation is able to retain its resolution and accuracy compared to fixed-point representations. Unfortunately, floating point operators require excessive area (or time) for conventional implementations. The creation of floating point units under a collection of area, latency, and throughput constraints is an important consideration for system designers. 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 paper presents a fully parallel floating-point multiplier compliant with the IEEE 754 Standard for Floating-point Arithmetic. The design offers low latency and high throughput. 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. We implemented our algorithms using Xilinx ISE 12.1, with Xilinx Virtex-II Pro XC2VP100 FPGA as our target device.**

*Keywords* – **Floating Point Multiplier, FPGA, VHDL.**

## 1. INTRODUCTION

Field-programmable gate arrays (FPGAs) have long been attractive for accelerating fixed-point applications. Early on, FPGAs could deliver tens of narrow, low latency fixed-point operations. As FPGAs matured, the amount of parallelism to be exploited grew rapidly with FPGA size. This was a boon to many application designers as it enabled them to capture more of the application. It also meant that the performance of FPGAs was growing faster than that of CPUs [3].

The design of floating-point applications for FPGAs is much different. Due to the inherent complexity of floating point arithmetic mapping difficulties occurred. With the introduction of high level languages such as VHDL, rapid prototyping of floating point units has become possible. Elaborate simulation and synthesis tools at a higher design level aid the designer for a more controllable and maintainable product.

The IEEE standard for binary floating-point arithmetic [2] provides a detailed description of the floating-point representation and the specifications for the various floating-point operations. It also specifies the method to handle special cases and exceptions.

This paper describes the implementation of pipelining in design the floating-point multiplier using VHDL and its synthesis for a Xilinx Virtex-II FPGA using Xilinx's Integrated Software Environment (ISE) 12.1. Pipelining is one of the popular methods to realize high performance computing platform. Pipelining is a technique where multiple instruction executions are overlapped. In the top-down design approach, four arithmetic modules: addition, subtraction, multiplication, and division: are combined to form the floating-point ALU. The pipeline modules are independent of each other.

The organization of this paper is as follows: Section 2 describes background on floating point unit design. Section 3 describes algorithm for floating point multiplication. Section 4 describes our approach to multiply floating numbers. The simulation environment and results are briefed in Section 5 and concluding remarks are discussed Section 6.

## 2. BACKGROUND

### 2.1. Floating Point Representation

Standard floating point numbers are represented using an
exponent and a mantissa in the following format:

*(sign bit) mantissa × baseexponent +bias*

The *mantissa* is a binary, positive fixed-point value. Generally, the fixed point is located after the first bit,*m*0, so that *mantissa* = {*m*0.*m*1*m*2...*mn}*, where *mi* is the *ith* bit of the mantissa. The floating point number is "normalized" when *m*0 is one. The *exponent*, combined with a *bias*, sets the range of representable values. A common value for the bias is $-2k-1$, where $k$ is the bit-width of the exponent [4].The double precision floating point format has an 11 bit exponent and 52 bit mantissa plus a sign bit. This provides a wide dynamic range.

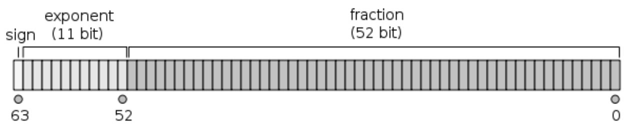

Figure 1. 64-bit Floating Point Format

The IEEE floating point standard makes floating point unit implementation portable and the precision of the results predictable. A variety of different circuit structures can be applied to the same number representations, offering flexibility.

2.2. Floating Point Implementations in FPGAs

Several efforts to build floating point units using FPGAs have been made. These approaches have generally explored
bit-width variation as a means to control precision. A floating point library containing units with parameterized bitwidth was described in [6]. In this library, mantissa and exponent bit-width can be customized. The library includes a unit that can convert between fixed point and floating point numbers. The floating point units are arranged in fixed pipeline stages.

Several researchers [7, 9, 10] have implemented FPGA floating point adders and multipliers that meet IEEE754 floating point standards. Most commercial floating point libraries provide units that comply with the IEEE754 standard [7]. Luk [8] showed that in order to cover the same dynamic range, a fixed point design must be five times larger and 40% slower than a corresponding floating point design. In contrast to earlier approaches, our floating point unit generation tool automates floating point unit creation.

## 3. FLOATING POINT MULTIPLICATION ALGORITHM

The aim in developing a floating point multiplier routine was to pipeline the unit in order to produce a result every clock cycle. By pipelining the multiplier, the speed increased, however, the area increased as well. Different coding structures were tried in the VHDL code used to program the Xilinx chips in order to minimize size.

3.1 Algorithm

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 flow chart of the multiplier structure.

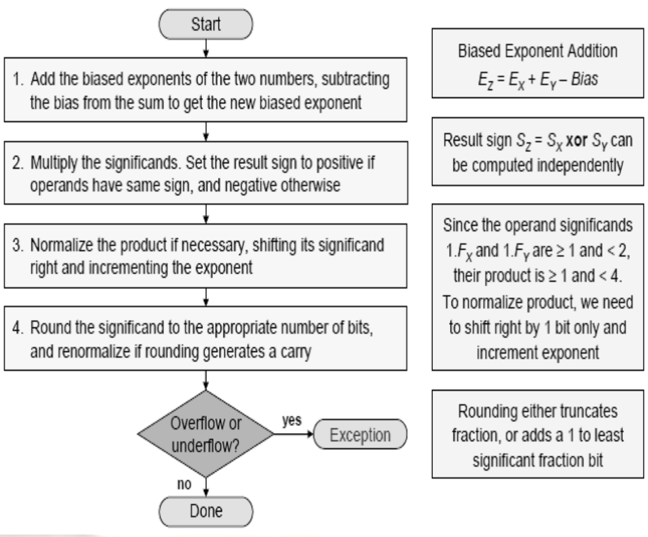
Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands. Set the result sign to positive if operands have same sign, and negative otherwise
3. Normalize the product if necessary, shifting its significand right and incrementing the exponent
4. Round the significand to the appropriate number of bits, and renormalize if rounding generates a carry
Overflow or underflow?
yes
Exception
no
Done
Biased Exponent Addition $E_Z = E_X + E_Y - Bias$
Result sign $S_Z = S_X$ xor $S_Y$ can be computed independently
Since the operand significands $1.F_X$ and $1.F_Y$ are ≥ 1 and < 2, their product is ≥ 1 and < 4. To normalize product, we need to shift right by 1 bit only and increment exponent
Rounding either truncates fraction, or adds a 1 to least significant fraction bit

Figure 2. Floating Point Algorithm

The two mantissas are to be multiplied, and the two exponents are to be added. In order to perform floating-point multiplication, a simple algorithm is realized:

1. Add the exponents and subtract bias.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

## 4. PIPELINED FLOATING POINT MULTIPLICATION MODULE

Multiplication entity has three 64-bit inputs and two 64-bit outputs. Selection input is used to enable or disable the entity. Multiplication module is divided into check zero, add exponent, multiply mantissa, check sign, and normalize and concatenate all modules, which are executed concurrently. Status signal indicates special result cases such as overflow, underflow and result zero.

In this paper, pipelined floating point multiplication is divided into three stages (Fig. 3).

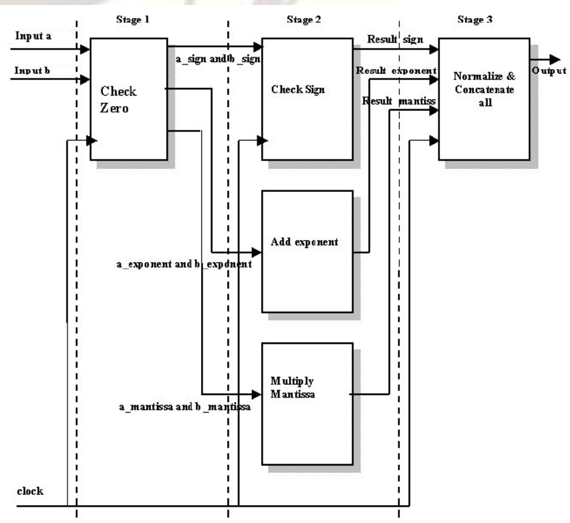

Figure 3. Pipelined Floating Point Multiplier

Stage 1 check whether the operand is 0 and report the result accordingly. Stage 2 determines the product sign add exponents and multiply fractions. Stage3 normalize and concatenate the product.

4.1 Check Zero module:

Initially, two operands are checked to determine whether they contain a zero. If one of the operands is zero, the zero_flag is set to 1. The output results zero, which is passed through all the stages and outputted. If neither of them are zero, then the inputs with IEEE754 format is unpacked and assigned to the check sign, add exponent and multiply mantissa modules. The mantissa is packed with the hidden bit '1'.

4.2 Add exponent module:

The module is activated if the zero flag is set. Else, zero is passed to the next stage and exp_flag is set to 0. Two extra bits are added to the exponent indicating overflow and underflow. The resulting sum has a double bias. So, the extra bias is subtracted from the exponent sum. After this, the exp_flag is set to 1.

4.3 Multiply mantissa module:

In this stage zero_flag is checked first. If the zero_flag is set to 0, then no calculation and normalization is performed. The mant_flag is set to 0. If both operands are not zero, the operation is done with multiplication operator. Mant_flag is set to 1 to indicate that this operation is executed. It then produces 46 bits where the lower order 32 bits of the product are truncated.

4.4 Check sign module:

This module determines the product sign of two operands. The product is positive when the two operands have the same sign; otherwise it is negative. The sign bits are compared using an XOR circuit. The sign_flag is set to 1.

4.5 Normalize and concatenate all modules:

This module checks the overflow and underflow after adding the exponent. Underflow occurs if the 9th bit is 1. Overflow occurs if the 8 bits is 1. If exp_flag, sign_flag and mant_flag are set, then normalization is carried out. Otherwise, 32 zero bits are assigned to the output.

During the normalization operation, the mantissa's MSB is 1. Hence, no normalization is needed. The hidden bit is dropped and the remaining bit is packed and assigned to the output port. Normalization module set the mantissa's MSB to 1. The current mantissa is shifted left until a 1 is encountered. For each shift the exponent is decreased by 1. Therefore, if the mantissa's MSB is 1, normalization is completed and first bit is the implied bit and dropped. The remaining bits are packed and assigned to the output port. The final normalized product with the correct biased exponent is concatenated with product sign.

## 5. RESULT AND ANALYSIS

Design is verified through simulation, which is done in a bottom-up fashion. The aim in designing the floating point units was to pipeline each unit a sufficient number of times in order to maximize speed and to minimize area.

The simulation output is obtained by using Xilinx simulation tool is as follows.

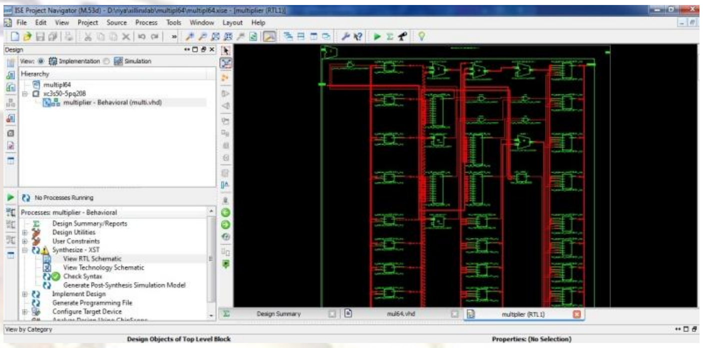

Figure 4. Synthesize result of multiplier

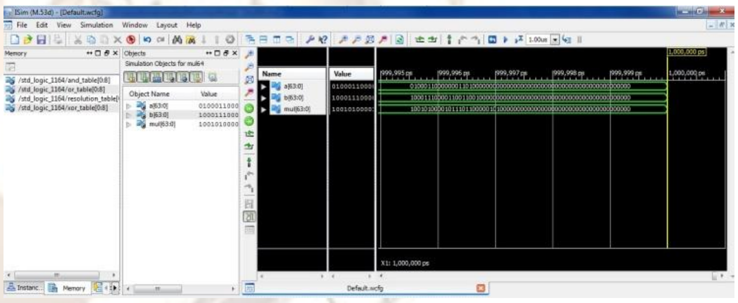

Figure 5. Simulation result of multiplier

## 6. CONCLUSION

Pipeline floating point multiplier design using VHDL is successfully designed, implemented, and tested. With the help of pipelined architecture i.e. concurrent processing there will be less combinational delay which means faster response and better throughput with less latency as compared with sequential processing but there will be a tradeoff between speed and chip area. Pipelined architecture provide a faster response in floating point multiplication but also consumes more area i.e. number of slices used on reconfigurable hardware are more as compared with standard architecture. Pipelining is used to decrease clock period. Using sequential processing there is larger latency but less number of slices are used on FPGAs as compared with pipelined architecture. Currently, we are conducting further research that considers the further reductions in the hardware complexity in terms of synthesis.

## REFERENCES

[1] M. de Lorimier and A. DeHon. Floating point sparsematrix-vector multiply for FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2005.

[2] The Institute of Electrical and Electronic Engineers, Inc. IEEE Standard for Binary Floating-point Arithmetic. ANSI/IEEE Std 754-1985.

[3] K. D. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterrey, CA, February 2004.

[4] I. Koren. *Computer Arithmetic Algorithms*. Brookside Court Publishers, Amherst, MA, 1998.

[5] IEEE Standards Board. IEEE standard for binary floating-point arithmetic. Technical Report ANSI/IEEE Std. 754-1985, The Institute of Electrical and Electronic Engineers, New York, 1985.

[6] P. Belanovi´*c* and M. Leeser. A Library of Parameterized Floating Point Modules and Their Use. In *Proceedings, International Conference on Field Programmable Logic and Applications*, Montpelier, France, Aug. 2002.

[7] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI Systems*, 2(3):365–367, Sept. 1994.

[8] A. A. Gaffar, W. Luk, P. Y. Cheung, N. Shirazi, and J. Hwang. Automating Customization of Floating-point Designs. In *Proceedings, International Conference on Field-Programmable Logic and Applications*, Montpelier, France, Aug. 2002.

[9] W. B. Ligon, S. McMillan, G. Monn, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *Proceedings, IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 206–215, Napa, CA, Apr. 1998.

[10] L. Louca, W. H. Johnson, and T. A. Cook. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In *Proceedings, IEEE Workshop on FPGAs for Custom Computing Machines*, pages 107–116, Napa, CA, Apr. 1996.

[11] E. M. Schwarz, M. Schmookler, and S. D. Trong. FPU implementations with denormalized numbers. IEEE Transactions on Computers, 54(7), July 2005.

[12] Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. June 2005 (Rev 4.3), [cited Aug 2005].