

Design of Generic Floating Point Multiplier and Adder/Subtractor Units

Lamiaa S. A. Hamid
ECE Dept.

Misr International Uni.
Cairo, Egypt

Lamiaa.a.hamid@miuegypt.edu.eg

Khaled A. Shehata
ECE Dept.

Arab Academy for Science
and Technology

Cairo, Egypt
K_shehata@aast.edu

Hassan El-Ghitani
ECE Dept.

Misr International Uni.
Cairo, Egypt

Hassan.elghitani@miuegypt.edu.eg

Mohamed ElSaid
ECE Dept.

Ain Shamis University
Cairo, Egypt

mkelsaid@gmail.com

Abstract— A high speed generic Floating Point Unit (FPU) consisting of a multiplier and adder/subtractor units is proposed. A novel multiplication algorithm is proposed and used in the multiplier implementation. The new algorithm depends on dividing the multiplication operation into several smaller multiplications performed in parallel. The output from these multiplications is then manipulated in a manner to give the final result of the original multiplication operation. The adder/subtractor unit is implemented using the Leading One Detector (LOD) algorithm. In order to achieve high maximum speed, both units were deeply pipelined. The design is written using VHDL code and mapped to Virtex2, Virtex4 and Virtex5 FPGAs. Both units can operate at more than 400 MHz on Virtex4.

Keywords—floating point; FPGA; Pipelined Architecture

I. INTRODUCTION

Advances in process technology have led to dramatic increase in FPGAs densities and speeds. FPGAs are now becoming more suitable for supporting designs with dense computations and high operating frequencies. Consequently, FPGAs are becoming more suitable for supporting high speed floating point arithmetic units. Floating point units are widely used in digital applications such as digital signal processing and multimedia. In conventional floating point units, the most frequently used floating point operations are multiplication and addition/subtraction counting for more than 94% of all floating point instructions [1]. Hence the employment of highly performing multiplier and adder/subtractor modules is of high importance.

The design objective in this paper is to implement high speed generic multiplier and adder/subtractor single precision floating point units. In order to achieve a high maximum operating frequency for both the multiplier and the adder/subtractor units, each unit was optimized separately by optimizing its bottle neck block. The bottle neck of the floating point multiplier unit is the multiplier block. Many algorithms have been introduced aiming to speed up the multiplier block [2]. In this paper, to optimize the multiplier block we used a novel proposed multiplication algorithm where the operands to be multiplied are sliced into smaller blocks. This proposed multiplier is referred to as "Block Multiplier". The bottle neck of the adder/subtractor module is the normalization block which is responsible for adjusting the result to the IEEE 754 normalized format after the result from addition/subtraction has

been calculated. Many algorithms have been proposed to optimize a certain performance parameter in the normalization process [3], [4]. In this paper, we chose to implement the adder/subtractor using the standard algorithm, also known as the LOD algorithm. LOD algorithm is characterized by its simplicity and reduced area. To achieve high operating frequency, the design was deeply pipelined.

To attain a generic design, VHDL was used for design entry of the entire floating point unit. The adopted VHDL design style was intended for optimum synthesis performance. Since VHDL is a hardware description language independent on the hardware intended for implementation, our design is expected to show optimal performance over any FPGA platform it is implemented on. This has been illustrated by implementing the design on Virtex2p, Virtex4 and Virtex5 and comparing results with other implementations.

The rest of the paper is organized as follows. Section 2 explains the IEEE 754-2008 industry standard for floating point representation [5]. Sections 3 and 4 present the floating point block multiplier unit and adder/subtractor unit respectively. Section 5 includes the implementation and results while section 6 wraps up with the conclusion.

II. SINGLE PRECISION IEEE 754-2008 STANDARD

Floating point numbers are used to represent decimal numbers. The main advantage of floating point numbers is the wide dynamic range. IEEE 754-2008 standard specifies how binary floating point numbers are represented as well as how to carry out arithmetic operations on them. Single precision floating point binary numbers consist of 32 bits; 1 sign bit, 8 bits for exponent and 23 bits for mantissa. The exponent is represented in excess-127 code to facilitate exponent comparison needed when performing arithmetic operations. The 23 bit mantissa actually has a 24th implied bit. The floating point number is said to be normalized when it is adjusted such that the implied bit is '1'. Hence, it is dropped upon storage allowing for increased accuracy and retrieved when performing operations for correct calculations. For numbers that are smaller than the smallest normalized number, the implied bit is '0' and the number is referred to as a de-normalized or subnorm number. De-normalized numbers are generally rare and require complicated hardware for their handling [6]. This is the main reason of dealing with normalized numbers in our design.

Arithmetic operations may give a result with a number of bits more than what can actually be stored so the result has to be rounded to fit in the allowed number of bits. IEEE 754-2008 standard introduces five different rounding modes. Round to nearest even (REN) is the default rounding mode and is used in most software and hardware implementations and is the mode of interest in our design.

In IEEE floating point representation, zero is not directly represented due to assumption of leading '1'. Instead, zero is considered as a special value denoted by all zeros exponent and mantissa. IEEE 754-2008 standard also specifies how to deal with exceptions that could result from an arithmetic operation. Overflow and underflow are the most common exceptions. In this paper, they were only honored in the multiplier module where they are more probable.

III. FLOATING POINT MULTIPLIER DESIGN

A. Floating Point Multiplication

In floating point numbers, multiplication of two numbers is basically performed through adding their exponents and multiplying their mantissas. As mentioned before, for floating point multiplication, the bottle neck of the design is the 24*24 bit multiplier used to calculate the resulting 48 bit mantissa. To increase the maximum operating speed of the multiplier, the proposed design breaks up the 24*24 bit multiplication of operands A and B into nine 8*8 bit multiplications where each mantissa is sliced into three 8-bit slices such that $A=A_2A_1A_0$ $B=B_2B_1B_0$. Then, B_0 is multiplied in A_2 , A_1 and A_0 . Each of these three 8*8 bit multiplications gives a 16 bit result. The three 16 bit results are properly manipulated to give a 32 bit result R_0 of the 24*8 bits multiplication operation (i.e. $A*B_0$). In a similar manner B_1 and B_2 are multiplied in A_2 , A_1 and A_0 to give R_1 and R_2 . R_1 and R_2 are properly shifted to be added to R_0 thus giving the 48 bit result mantissa.

B. The Floating Point Multiplier Unit

The proposed floating point multiplier unit block diagram is illustrated in Fig. 1. Its general structure is similar to that introduced in [7] and [8]. The proposed design has the following main modules:

1) *Zero Detect*: It is used to set a zeroflag when any of the input operands is zero. This avoids unnecessary calculations throughout the multiplier module when a zero input is applied. The sign, exponent and mantissa of each input operand are separated to be manipulated differently throughout the multiplier module. To prepare the mantissas for the multiplication operation, the dropped implied bit of each mantissa is set to '1'.

2) *Add Exponents*: To determine the resultant exponent, the exponents are added and a 127 bias is subtracted. The bias subtraction compensates for the bias having been added in both exponents. The result is fit into a 10 bit exponent to allow checking for overflow or underflow in the post multiply normalization step. Since the 8 bit exponent of a single precision floating point number is an unsigned number, the 10th bit set to '1' indicates an underflow. An overflow is detected when the 10th bit is '0' and the 9th bit is '1'.

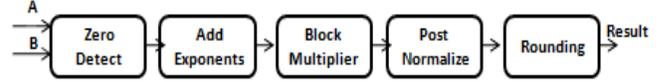


Fig. 1. Block Diagram of Floating Point Multiplier Module

3) *Block Multiplier*: Multiplies the two 24 bit mantissas of operands A and B through the proposed block multiplication method explained earlier. The result sign is determined through a simple XOR operation.

4) *Post Normalize*: The implied bit is located and dropped. The 48 bit mantissa from the multiplication operation is then truncated to a 26 bit mantissa which are the 23 bits assigned for the mantissa along with three extra bits to increase the accuracy of the rounding process. These three extra bits are the guard, round and sticky bits. The guard and round bits are just an extension added to the mantissa to allow for extra precision. Sticky bit is the logical "Or"ing of all the truncated bits. The exponent is adjusted and checked for overflow or underflow and the appropriate flag is set accordingly.

5) *Rounding*: The 26 bit resultant mantissa is rounded to a 23 bits using the REN technique. After rounding, the exponent is checked again for possible overflow. Finally, the sign, exponent and mantissa are appended together to give the single precision floating point multiplication result in the IEEE format along with the overflow and underflow flags.

IV Floating Point Adder/Subtractor Design

A. Floating Point Addition/Subtraction

In floating point addition (or subtraction), the two numbers must have equal exponents for their mantissas to be added (or subtracted) correctly. So in the adder/subtractor unit, the exponent of the smaller number is incremented such that both exponents are equal and the mantissa of the small number is then shifted right 'n' times where 'n' is the difference between the large and small exponents. After the addition/subtraction operation is performed, the resultant mantissa is normalized using the LOD method. The LOD detects the most significant '1' by counting the number of zeros (nz) before the most significant '1'. The mantissa is then shifted left 'nz' times.

Due to the excess shifting required before and after the addition/subtraction operation in the pre-normalization of the smaller number and the post normalization of the resultant mantissa respectively, addition/subtraction of floating numbers is considered more complicated than multiplication. To increase the maximum operating speed of the adder/subtractor unit all the shift operations in the pre-normalization and post-normalization steps were performed through barrel shifting. Barrel shifting has the advantage of shifting the data by any number of bits in one operation which makes barrel shifting suitable for the shifting operations required in the adder/subtractor unit that can be a shift by any number between 1 and 253 depending on the difference between the exponents of the input operands.

B. LOD Floating Point Adder/Subtractor Unit

The general block diagram of the floating point adder/subtractor module is illustrated in Fig. 2. Its general structure is similar to that introduced in [7] and [8]. The proposed design has the following main modules:

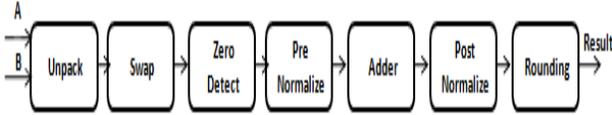


Fig. 2. Block Diagram of Floating Point Adder/Subtractor Module

1) *Unpack*: The sign, exponent and mantissa of both operands are separated. A flag, *aequalb flag*, is set if both inputs are equal. The *aequalb flag* will be used if the effective operation, determined in the adder/subtractor module, was subtraction to set a flag indicating a zero output. This prevents unnecessary addition/subtraction and pre-normalization operations from taking place.

2) *Swap*: Inputs are swapped if necessary such that operand A carries the larger floating point number and operand B is the smaller operand to be pre-normalized. A *swap flag* is set if the operands were swapped to be used in determining the effective operation in the adder/subtractor module.

3) *Zero Detect*: An appropriate flag is set if one or both input operands is a zero. This helps avoid unnecessary calculations and normalizations when a zero operand is detected. The resultant exponent and the difference between the two exponents are determined here.

4) *Prenormalize*: The smaller mantissa, of operand B, is prenormalized, that is it's shifted by the difference between the two input exponents. Three extra bits the guard bit, the round bit, and the sticky bit are added to both mantissas to increase the accuracy of the performed operation (addition or subtraction) and to be used in the rounding process. Sticky bit is the logical "Or"ing of any bits that are dropped during the pre-normalization of operand B.

5) *Adder/Subtractor*: The effective operation to be performed is calculated according to the signs of operands A & B, the input operation and the swap flag. The effective operation is performed and the zero flag is updated if the effective operation is subtraction and the *aequalb flag* is set.

6) *Post Normalize*: The resultant mantissa is normalized after the leading one is detected using the LOD method. The resultant exponent is adjusted accordingly.

7) *Rounding*: The resultant mantissa is rounded using the REN technique. The final output is given in IEEE format.

V Implementation

The implementation of both the multiplier and the adder/subtractor modules is performed in a top-down approach. Their sub-modules were written in VHDL using the Mentor Graphics FPGAdv 8.1 Tool. Each sub-module was first separately tested using ModelSim to validate correct functional behavior. The sub-modules were then connected to form the floating point multiplier and adder/subtractor units. Each unit was then tested for functionality. Deep pipelining was performed to reach maximum possible operating frequency. Xilinx ISE 9.2i was used for synthesis and implementation, place and route.

VI. RESULTS

The multiplier and adder/subtractor functional simulations showed that both modules accept an input and give an output with every clock cycle. The first output appears after a latency of 23 and 22 clock cycles for the multiplier and adder/subtractor units respectively. Fig.3 shows a snapshot from the functional simulation of the multiplier unit. In this example the operands are -22 (H41B00000) and 4 (H40800000). The result is available after 23 clock cycle and equal to -88 (HC2B00000). Fig. 4 shows a snapshot from the functional simulation of the adder/subtractor unit. In this example the operands are 12(H41400000) and 1(H3F800000) are added. The result is available after 22 clock cycles and equal to 13(H41500000).

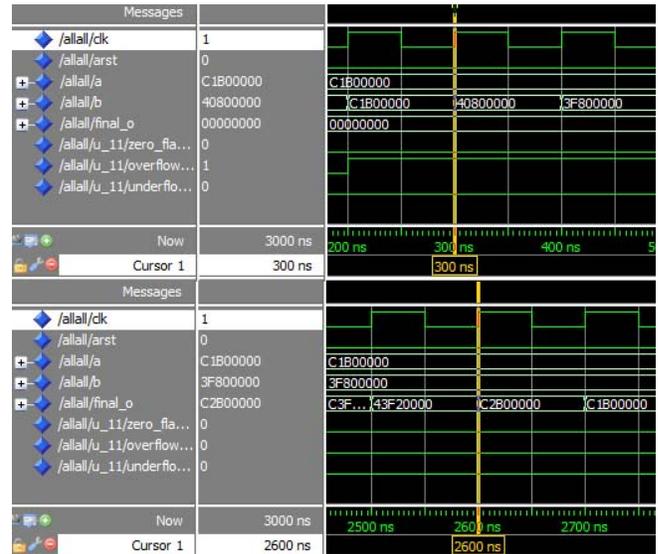


Fig. 3 Functional Simulation of Multiplier

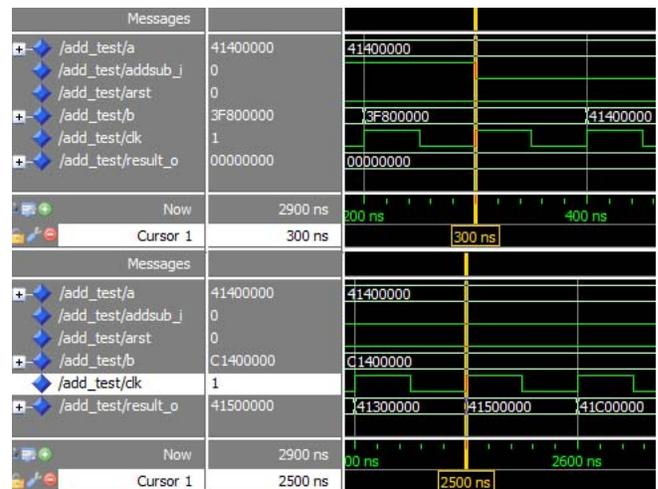


Fig. 4 Functional Simulation of Adder

Both the multiplier and the adder/subtractor modules were implemented on three generations of Xilinx FPGAs namely Virtex2p (speed grade -7), Virtex4 (speed grade -12) and Virtex5 (speed grade -3). Maximum operating frequency and area were obtained by the Xilinx tool after the place and route process. The results obtained are summed up in Table I.

To further investigate the improvement in performance of the proposed block multiplication, the multiplier module was also implemented using a simple 24* 24 multiplier. The comparison in synthesis results between the floating multiplier using the simple multiplier module and using the block multiplier module is listed in Table II. The comparison shows superiority in the maximum operating frequency for the design using the proposed block multiplier over the design using the simple multiplier for the tested FPGAs.

Table III and IV compare the post place and route results of the proposed multiplier and the adder/subtractor units respectively with other published results. The performed comparison is fair since the proposed designs and those presented in [4] and [6] do not support denormalized numbers. The proposed multiplier has a higher speed than the one presented in [6] when implemented on Virtex2. When implementing the multiplier on Virtex4 FPGA, the one presented in [4] has a higher speed due to the use of DSP48 blocks to construct the multiplier module. The use of these blocks optimized the design presented in [4] for Virtex4. The proposed adder module has higher operating frequency when implemented on both Virtex2 and Virtex4.

TABLE I. BLOCK MULTIPLIER & ADDER PARAMETERS OVER DIFFERENT XILINX FPGAS

	Multiplier		Adder/Subtractor	
	Speed (MHz)	Area (Slices)	Speed (MHz)	Area (Slices)
Virtex2p Xc2vp7ff896 -7	288	1005	317	1323
Virtex4 Xc4vfx100 -12	437	903	401	1324
Virtex5 Xc5vlx110 -3	442	579	407	664

TABLE II. SYNTHESIS COMPARISON BETWEEN SIMPLE & BLOCK MULTIPLIERS

	Proposed Multiplier		Simple Multiplier	
	Speed (MHz)	Area Slices	Speed (MHz)	Area (Slices)
Virtex2p Xc2vp7ff896 -7	284	1049	181	412
Virtex4 Xc4vfx100 -12	430	864	106	343
Virtex5 Xc5vlx110 -3	415	457	217	578

TABLE III. COMPARISON WITH OTHER MULTIPLIER MODULES

	Multiplier Designs		
	Proposed	Karlstrom [4]	USC [6]
Virtex2p -7	288	NA	250
Virtex4 -12	437	450*	NA

*Optimized for Virtex4 by using DSP4 to construct multiplier.

TABLE IV. COMPARISON WITH OTHER ADDER MODULES

	Adder Designs		
	Proposed	Karlstrom [4]	USC [6]
Virtex2p -7	317	288	250
Virtex4 -12	401	361	NA

VII CONCLUSIONS

Floating point multiplier and adder/subtractor units are presented. Both were written entirely in VHDL to optimize implementation on any FPGA. A novel architecture is proposed for the multiplier block. A deeply pipelined LOD architecture is used to implement the adder/subtractor unit. Both designs were proved to operate at high operating frequencies when implemented to several Xilinx FPGAs. The proposed multiplier operates at 430 MHz while the adder/subtractor operates up to 401 MHz when implemented on Virtex4 FPGA.

REFERENCES

- [1] A. Amarica, M. Vladutiu, L. Prodan, M. Udrescu and O. Boncalo, "Design of addition and multiplication units for high performance interval arithmetic processor," in Proceedings of the International Conference on Computer Design, 2007, pp. 1-4.
- [2] S.V.Siddamal, R.M. Banakar and B.C. Jinaga, "Design of high speed floating point multiplier," in the 4th IEEE International Symposium on Electronic Design, Test and Application, 2008, pp.285-289.
- [3] J. Liang, R. Tessier and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in Field-Programmable Custom Computing Machines, 2003, pp. 185-194.
- [4] P. Karlstrom, A. Ehliar and D. Liu, "High performance, low latency FPGA based floating point adder and multiplier units in a Virtex 4," in Computers and Digital Techniques, volume 2, issue 4, 2008, pp. 305-313.
- [5] IEEE standards board, IEEE standard for floating-point arithmetic, 2008.
- [6] G. Govindu, L. Zhuo, S. Choi and V. Prasanna, "Analysis of high performance floating point arithmetic on FPGAs," in Proceeding of the 18th International Parallel and Distributed Processing Symposium, April 2004, pp. 149-156
- [7] M. Reaz, S. Islam and M. Suliman, "Pipeline floating point ALU design using VHDL," in Proceedings of Semiconductor Electronics, 2002, pp. 204-208.
- [8] G. Marcus, P. Hinojosa, A. Avila and J. Nolasco-Flores, "A fully synthesizable single precision floating point adder/subtractor and multiplier in VHDL for general and educational use," in Proceedings of the 5th IEEE Interantional Caracas Conference on Devices, Circuits and Systems, November 2004, pp. 319-323