ISSN: 2277-9655
Impact Factor: 1.852

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

An Efficient Way of Decreasing the Latency of Flip Flops with Floating Point Multiplier

G Lalitha Rani
veenulalitha@gmail.com

Abstract

In this paper we proposed efficient implementation of floating point multiplier which is technology independent pipelined design. This also handles the case of overflow and underflow cases. This verified that they can decrease the flipflop latency over Xilinx flipflop core.

Keywords: MFLOPS, Multiplier, CAD Design Flow.

Introduction

In computing, floating point describes a method of representing an approximation of a real number in a way that can support a wide range of values. The numbers are, in general, represented approximately to a fixed number of significant digits (the mantissa) and scaled using an exponent. The base for the scaling is normally 2, 10 or 16. The typical number that can be represented exactly is of the form: The idea of floating-point representation over intrinsically integer fixed-point numbers, which consist purely of significant, is that expanding it with the exponent component achieves greater range. For instance, to represent large values, e.g. distances between galaxies, there is no need to keep all 39 decimal places down to femtometre-resolution (employed in particle physics). Assuming that the best resolution is in light years, only the 9 most significant decimal digits matter, whereas the remaining 30 digits carry pure noise, and thus can be safely dropped. This represents a savings of 100 bits of computer data storage. Instead of these 100 bits, much fewer are used to represent the scale (the exponent), e.g. 8 bits or 2 decimal digits. Given that one number can encode both astronomic and subatomic distances with the same nine digits of accuracy, but because a 9-digit number is 100 times less accurate than the 11 digits reserved for scale, this is considered a trade-off exchanging range for precision. The example of using scaling to extend the dynamic range reveals another contrast with fixed-point numbers: Floating-point values are not uniformly spaced. Small values, close to zero, can be represented with much higher resolution (e.g. one femtometre) than large ones because a greater scale (e.g. light years) must be selected for encoding significantly larger values. That is, floating-point numbers cannot represent point coordinates with atomic accuracy at galactic distances, only close to the origin. Floating point representation makes numerical computation much easier. You could write all your programs using integers or fixed-point representations, but this is tedious and error-prone.

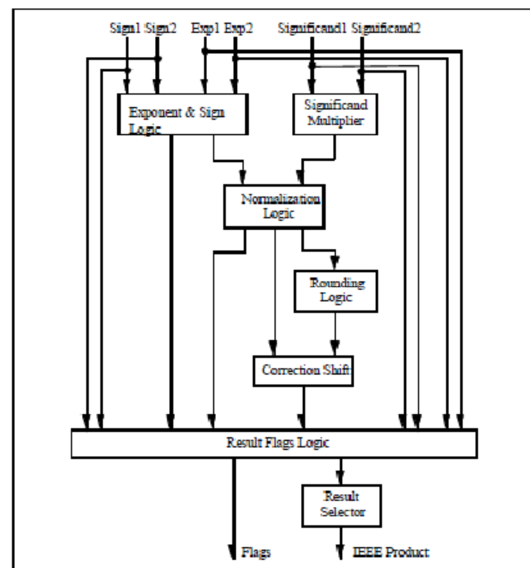


Fig 1.0 Simple floating point multiplication

For example, you could write a program with the understanding that all integers in the program are 100 times bigger than the number they represent. The integer 2345, for example, would represent the number 23.45. As long as you are consistent, everything works. This is actually the same as using fixed point notation. In fixed point binary notation the binary point is assumed to lie between two of the bits. This is the same as an understanding that the integer the bits represent should be divided by a particular power of two. But it is very hard to stay consistent. A programmer must remember where the decimal (or binary) point "really is" in each number. Sometimes one program needs to deal with several different ranges of numbers. Consider a program

that must deal with both the measurements that describe the dimensions on a silicon chip (say 0.000000010 to 0.000010000 meters) and also the speed of electrical signals, 100000000.0 to 300000000.0 meters/second. It is hard to find a place to fix the decimal point so that all these values can be represented. Notice that in writing those numbers (0.000000010, 0.000010000, 100000000.0, and 300000000.0) I was able to put the decimal point where it was needed in each number. We present a pipelined 32-bit Instruction Set Extension (ISE) for complex valued floating point operations. The ISE was implemented in the NIOS II processor, and the constraint on the number of inputs and outputs of the register bank was overcome by distributing the reads and writes of the instruction over several cycles. The hardware size was reduced by sharing hardware between instructions. The main contribution of this work is that the designed ISE performs division, multiplication, addition and subtraction on complex valued numbers. Comparing the use of the embedded multiplier and divider in a NIOS II processor to the designed ISE for an image processing problem, a speedup of 12.2 times was observed. 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. The common multiplication method is "add and shift" algorithm. In parallel multipliers number of partial products to be added is the main parameter that determines the performance of the multiplier. To reduce the number of partial products to be added, Modified Booth algorithm is one of the most popular algorithms. To achieve speed improvements Wallace Tree algorithm can be used to reduce the number of sequential adding stages. Further by combining both Modified Booth algorithm and Wallace Tree technique we can see advantage of both algorithms in one multiplier. However with increasing parallelism, the amount of shifts between the partial products and intermediate sums to be added will increase which may result in reduced speed, increase in silicon area due to irregularity of structure and also increased power consumption due to increase in interconnect resulting from complex routing. On the other hand "serial-parallel" multipliers compromise speed to achieve better performance for area and power consumption. The selection of a parallel or serial multiplier actually depends on the nature of application. In this lecture we introduce the multiplication algorithms and architecture and compare them in terms of speed, area, power and combination of these metrics.

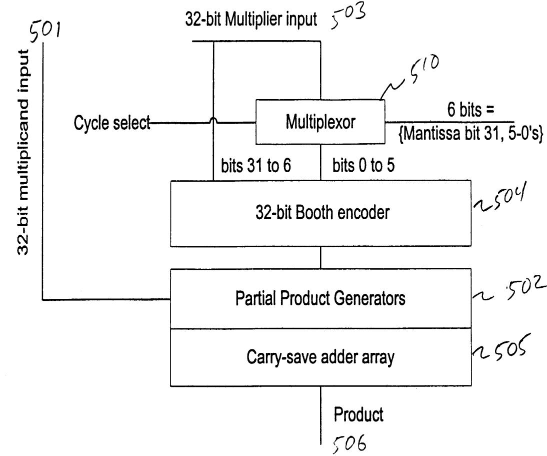


Fig1.1 Double precision floating point multiplier

Floating Point Arithmetic

Arithmetic operations on floating point numbers consist of addition, subtraction, multiplication and division the operations are done with algorithms similar to those used on sign magnitude integers (because of the similarity of representation) -- example, only add numbers of the same sign. If the numbers are of opposite sign, must do subtraction. Addition, subtraction, multiplication, division.

Floating Point Multiplication Algorithm

A pipelined multiplier based on the digit products can be designed using digit product generation logic and the digit adders.

25 * 35 = 875

Now for binary mutiplication

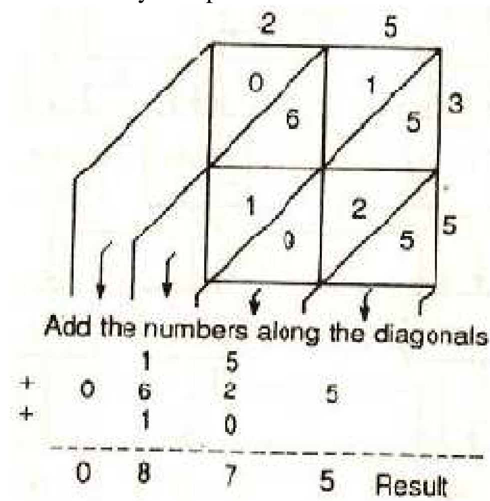
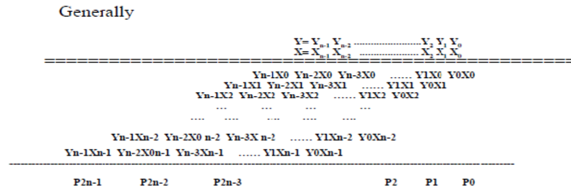


Fig 3.0 Multiplication Algorithm

The multiplication algorithm for an N bit multiplicand by N bit multiplier is shown below:



A) Serial Multiplier

Where area and power is of utmost importance and delay can be tolerated the serial multiplier is used. This circuit uses one adder to add the m * n partial products. The circuit is shown in the fig. below for m=n=4. Multiplicand and Multiplier inputs have to be arranged in a special manner synchronized with circuit behavior as shown on the figure. The inputs could be presented at different rates depending on the length of the multiplicand and the multiplier. Two clocks are used, one to clock the data and one for the reset. A first order approximation of the delay is O (m,n). With this circuit arrangement the delay is given as D =[(m+1)n + 1] tfa.

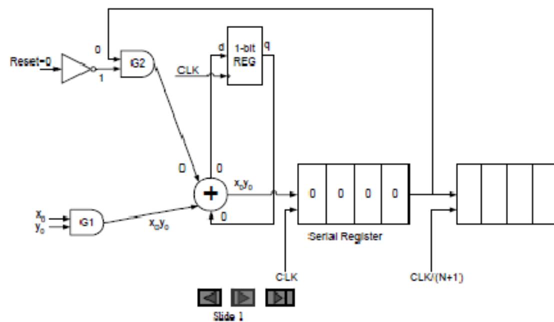


Fig 3.1 Serial Multiplier

B) Signed multiplication

Direct two's complement array multiplication can perform "direct" multiplication of two's complement numbers without requiring the complementing stages, significantly speeds up the multiplication process. This appendix will discuss two direct two's complement multiplication algorithms and their implementation. These two direct two's complement multiplication algorithms are Tri-section modified Pezaris two's complement multiplication, Baugh-Wooley two's complement multiplication These two algorithms are generally used in systems where the operands are less than 16-bit. They are relatively simpler than Booth multiplier whose structure is based on recoding the 2's complement operand in order to reduce the number of partial products to be added. Listed below are four arithmetic equations that describe the input/output relationships of the four types of generalized full adders.

Type 0: $C2^1 + S2^0 = X2^0 + Y2^0 + Z2^0$

Type 1: $C2^1 + (-S)2^0 = X2^0 + Y2^0 + (-Z)2^0$

Type 2: $(-C)2^1 + S2^0 = (-X)2^0 + (-Y)2^0 + Z2^0$

Type 3: $(-C)2^1 + (-S)2^0 = (-X)2^0 + (-Y)2^0 + (-Z)2^0$

Operands Multiplication and Rounding

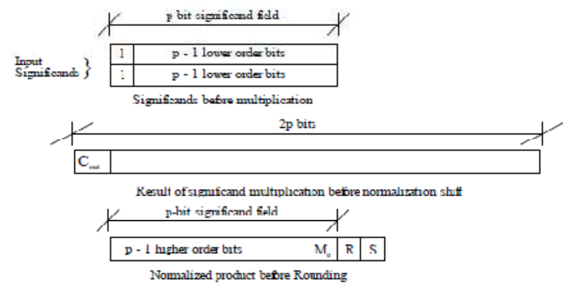


Fig 3.1 Significant multiplication ,and Rounding

The exponents of the two registers are subtracted. The difference is positive, indicating that the exponent in register A (on the left is larger. Control selects the exponent from register A (by asserting 0 at the multiplexer on the left) to pass to the next section of the adder to be used as the preliminary result for the exponent. The difference between the two exponents is 2, indicating the significant in register B must be shifted right two places. Before entering the ALU or the shift register, the 23-bit significant are expanded to 32 bits by inserting the leading implicit 1 and filling in leading 0's. (To provide for round off, trailing 0's may also be appended to the original 23 bits.) Control selects the (expanded) contents of register B to be placed in the shift register and the contents of register A (expanded) to be sent directly to the ALU. The contents of register B are shifted right two places and the two terms are added. In this example, the 23 bits of the significant are mapped into bits 24 -- 2 during the process of expanding to 32 bits. Bits 0 and 1 are set to 0 initially and used for calculating round off. The implicit leading 1 is set in bit

ISSN: 2277-9655
Impact Factor: 1.852

25 and bits 26 -- 31 hold leading 0's. The input to the ALU (after shifting) is shown in the diagram below. (Note! Since the last two bits of the significant in register B are both 0, shifting right just moves these two 0's into the additional trailing bits.)

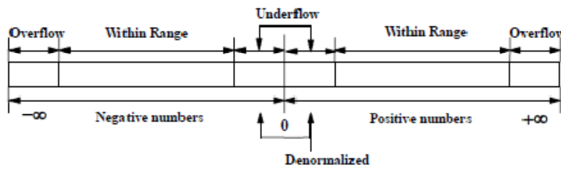


Fig 3.2 Range of Floating point numbers

Conclusion

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 significant multiplication result as is (48 bits); this gives better precision if the whole 48 bits are utilized in another unit; i.e. a floating point adder to form a MAC unit achieves 301 MFLOPs. This verified that they can decrease the flipflop latency over Xilinx flipflop core.

References

[1] Computer Arithmetic Systems, Algorithms, Architecture and Implementations. A. Omondi. Prentice Hall, 1994.
[2] Computer Architecture A Quantitative Approach, chapter Appendix A. D. Goldberg. Morgan Kaufmann, 1990.
[3] Reduced latency IEEE floating-point standard adder architectures. Beaumont-Smith, A.; Burgess, N.; Lefrere, S.; Lim, C.C.; Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on , 14-16 April 1999
[4] Rounding in Floating-Point Addition using a Compound Adder. J.D. Bruguera and T. Lang. Technical Report. University of Santiago de Compostela. (2000)
[5] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.
[6] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," IEEE Transactions on VLSI, vol. 2, no. 3, pp. 365– 367, 1994.
[7] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95), pp.155 162, 1995.
[8] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," Proceedings of 83 the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96), pp. 107–116, 1996.
[9] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp. 897-900.