International Journal of Engineering in Advanced Research Science and Technology

VOLUME-4
ISSUE-12

AN ENHANCED AND LOW POWER DOUBLE PRECISION FLOATING POINT MULTIPLIER DESIGN

[1] K.RAVI VARMA, [2] M.SANDYA

[1] M.Tech, Dept of ECE, Andhra Loyola Institute of Engineering &Technology Vijayawada, AP ,India

[2] Assistant Professor in ECE Andhra Loyola Institute of Engineering &Technology Vijayawada, AP,India

Abstract: ABSTRACT: The main objective of this project is to design an efficient IEEE-754 floating point multiplier .This project focuses on double precision normalized binary floating point multiplication in IEEE754 format. The proposed design is compliant with IEEE-754 format and handles over flow, under flow, rounding and various exception conditions. FLOATING-POINT arithmetic provides a wide dynamic range, freeing special purpose processor designers from the scaling and overflow/underflow concerns that arise with fixed-point arithmetic. Further this project can be enhanced by replacing the general multiplier architecture with modified booth multiplication. This modification yields reduction of partial products to half. Partial products reduction changes system overall performance in an efficient manner. Xilinx tool is used to perform this task with the help of VHDL language

KEYWORDS: Floating point multiplier, double precision, underflow, Modified booth, rounding, IEEE 754, VHDL, Xilinx.

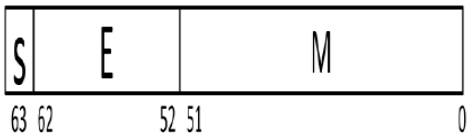
I. INTRODUCTION:

The real numbers represented in binary format are known as floating point numbers. Based on IEEE-754 standard, floating point formats are classified into binary and decimal interchange formats. Floating point multipliers are very important in DSP applications. This paper focuses on double precision normalized binary interchange format. Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 [3] standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Multiplying floating point numbers is a critical requirement for DSP applications involving large dynamic range. This paper focuses on double precision floating point binary interchange format. Figure 1 shows the IEEE 754 double precision floating point binary format representation; it consists of a one bit sign (S), an eleven bits exponent (E), and a fifty two bits fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand1. If the exponent is greater

than 0 and smaller than 2047, and there is 1 in the MSB of the significand then the number is said to be a normalized number, Significand is the mantissa with an extra MSB bit. It I shows the IEEE- 754 double precision binary format representation. Sign (S) is represented with one bit, exponent (E) and fraction (M or Mantissa) are represented with eleven and fifty two bits respectively. For a number is said to be a normalized number, it must consist of 'one' in the MSB of the significand and exponent is greater than zero and smaller than 1023. The real number is represented by equations (I) & (2).

$Z = (-I^{S}) * 2^{(E - Ria.l)} * (1. M)$ _____(1)

Value = (-ISign bit) * 2 (Exponent -1023) * (I.Mantissa) _____(2)



Multiplying two numbers in floating point format is done by 1- calculating the sign by XORing the sign of the two numbers, 2- adding the exponent of the two numbers then subtracting the bias from their result, and 3- multiplying the significand of the two numbers. In order to represent the multiplication result as a normalized number there should be 1 in the MSB of the result (leading one).

GENERAL MULTIPLICATION:

MULTIPLICATION is a complex arithmetic operation, which is reflected in its relatively high signal propagation delay, high power dissipation, and large area requirement. When choosing a multiplier for a digital system, the bit width of the multiplier is required to be at least as wide as the largest operand of the applications that are to be executed on that digital system. The bit width of the multiplier is, therefore, often much larger than the data represented inside the operands, which leads to unnecessarily high power dissipation and unnecessary long delay. This resource waste could partially be remedied by having several multipliers, each with a specific bit width, and use the particular multiplier with the smallest bit width that is large enough to accommodate the current multiplication. Such a scheme would assure that a multiplication would be computed on a multiplier that has been optimized in terms of power and delay for that specific bit width. However, using several multipliers with different bit widths would not be an efficient solution, mainly because of the huge area overhead.1 There have been several studies on operand bit widths of

integer applications and it has been shown that for the SPECint95 benchmarks more than 50% of the instructions are instructions where both operands are less than or equal to 16 bits [1] (henceforth called narrow-width operations). This property has been explored to save power, through operand guarding [1]–[3]. In operand guarding the most significant bits of the operands are not switched, thus power is saved in the arithmetic unit when multiple narrow-width operations are computed consecutively. Brooks et al. [1] showed that power dissipation can be reduced by gating of the upper part of narrow-width operands. For the SPECint95 and Media Bench benchmarks, the power reduction of an operand-guarded integer unit was 54% and 58%, respectively, which accounts for a total power reduction of 5–6% for an entire data path.

II. FLOATING POINT MULTIPLICATION ALGORITHM

As stated in the introduction, normalized floating point

numbers have the form of Z = (-1S) * 2 (E - *Bias*) * (1.M). To

Multiply two floating point numbers the following is done:

1. Obtaining the sign; i.e. Sa xor Sb
2. Adding the exponents; i.e. (E1 + E2 – *Bias*)
3. Multiplying the significand; i.e. (1.M1*1.M2)
4. Placing the decimal point in the significant result
5. Normalizing the result; i.e. obtaining 1 at the MSB of the

   Results significand
6. Rounding the result to fit in the available bits
7. Checking for underflow/overflow occurrence

Rounding and Exceptions

The IEEE standard specifies four rounding modes round to nearest, round to zero, round to positive infinity, and round to negative infinity. Table 1 show the rounding modes selected for various bit combinations of rmode. Based on the rounding changes to the mantissa corresponding changes has to be made in the exponent part also.

Modified Booth Algorithm

Booth multiplication algorithm consists of three major steps as shown in the structure of booth algorithm figure that includes generation of partial product called as recoding, reducing the partial product in

two rows, and addition that gives final product.

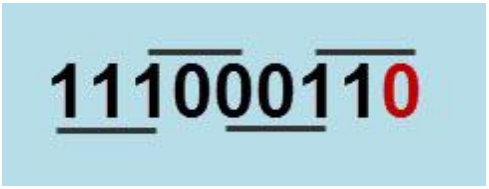
For a better understanding of modified booth algorithm & for multiplication, we must know about each block of booth algorithm for multiplication process.

Bit combination	Rounding Mode
00	round_nearest_even
01	round_to_zero
10	round_up
11	round_down

Modified Booth Algorithm Encoder

This modified booth multiplier is used to perform high-speed multiplications using modified booth algorithm. This modified booth multiplier's computation time and the logarithm of the word length of operands are proportional to each other. We can reduce half the number of partial product. Radix-4 booth algorithm used here increases the speed of multiplier and reduces the area of multiplier circuit. In this algorithm, every second column is taken and multiplied by 0 or +1 or +2 or -1 or -2 instead of multiplying with 0 or 1 after shifting and adding of every column of the booth multiplier. Thus, half of the partial product can be reduced using this booth algorithm. Based on the multiplier bits, the process of encoding the multiplicand is performed by radix-4 booth encoder.

The overlapping is used for comparing three bits at a time. This grouping is started from least significant bit (LSB), in which only two bits of the booth multiplier are used by the first block and a zero is assumed as third bit as shown in the figure.



The figure shows the functional operation of the radix-4 booth encoder that consists of eight different types of states. The outcomes or multiplication of multiplicand with 0, -1, and -2 are consecutively obtained during these eight states

The steps given below represent the radix-4 booth algorithm:

- Extend the sign bit 1 position if necessary to ensure that n is even.
- Append a 0 to the right of the least significant bit of the booth multiplier.

According to the value of each vector, each partial product will be 0, +y, -y, +2y or -2y.

Modified booth multiplier's (Z) digits can be defined with the following equation:

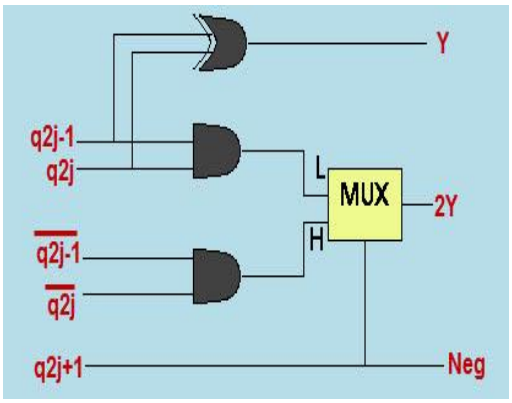
$Zj = q2j + q2j-1 -2q2j+1$ with $q-1 = 0$

The figure shows the modified booth algorithm encoder circuit. Now, the product of any digit of Z with multiplicand Y may be -2y, -y, 0, y, 2y.

Booth recoding table for radix-4

Multiplier Bits Block			Recoded 1-bit pair		2 bit booth	
i+1	i	i-1	i+1	i	Multiplier Value	Partial Product
0	0	0	0	0	0	Mx0
0	0	1	0	1	1	Mx1
0	1	0	1	-1	1	Mx1
0	1	0	1	0	2	Mx2
1	0	0	-1	0	-2	Mx-2
1	0	1	-1	1	-1	Mx-1
1	1	0	0	-1	-1	Mx-1
1	1	0	0	0	0	Mx0

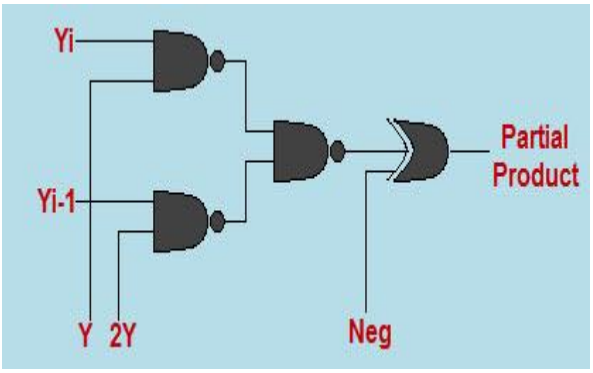
Booth Recoding Table for Radix-4



Booth's Encoder

But, by performing left shift operation at partial products generation stage, 2y may be generated. By taking 1′s complement to this 2y, negation is done, and then one is added in appropriate 4-2 compressor. One booth encoder shown in the figure generates three output signals by taking three consecutive bit inputs so as to represent all five possibilities -2X, -X, 0, X, 2X.

a) Partial Product Generator



Partial Product Generator

If we take the partial product as -2y, -y, 0, y, 2y then, we have to modify the general partial product generator. Now, every partial product point consists of two inputs (consecutive bits) from multiplicand and, based on the requirement, the output will be generated and its complements also generated in case if required. The figure shows the partial product generator circuit. The 2's complement is taken for negative values of y. There are different types of adders such as conventional adders, ripple-carry adders, carry-look-ahead adders, and carry select adders. The carry select adders (CSLA) and carry-look-ahead adders are considered as fastest adders and are frequently used. The multiplication of y is done by after performing shift operation on y – that is – y is shifted to the left by one bit.

III. RADIX-8 MODIFIED BOOTH ALGORITHM:

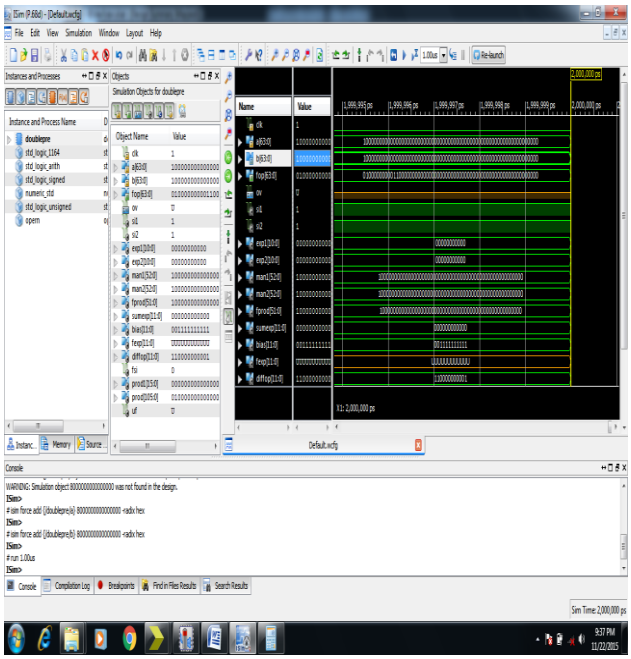
Radix-8 recoding applies the same algorithm as radix-4, but now we take quartets of bits instead of triplets. Each quartet is codified as asigned-digit using the table

Quartet value	*Signed-digit value*
0000	0
0001	+1
0010	+1
0011	+2
0100	+2
0101	+3
0110	+3
0111	+4
1000	-4
1001	-3
1010	-3
1011	-2
1100	-2
1101	-1
1110	-1
1111	0

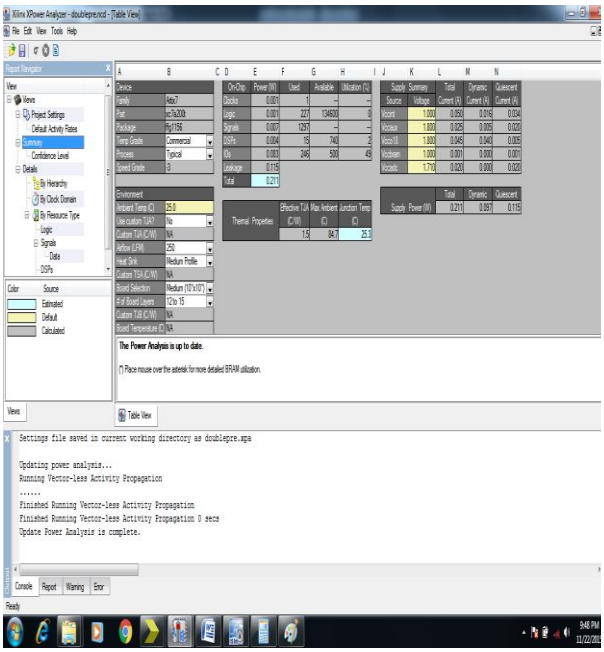
Here we have an odd multiple of the multiplicand, 3Y, which is not immediately available. To: generate it we need to perform this previous add:2Y+Y=3Y. But we are designing a multiplier for specific purpose and thereby the multiplicand belongs to a previously known set of numbers which are stored in a memory chip. We have tried to take advantage of this fact, to ease the bottleneck of the radix-8 architecture, that is, the generation of 3Y. In this manner we try to attain a better overall multiplication time, or at least comparable to the time we could obtain using radix-4 architecture (with the additional advantage of using a less number of transistors). To generate 3Y with 21-bit words we only have to add 2Y+Y, that is, to add the number with the same number shifted one position to the left.

Radix-8 Booth recoding applies the same algorithm as that of Radix-4, but now we take quartets of bits instead of triplets. Each quartet is codified as a signed digit using Table II. Radix-8 algorithm reduces the number of partial products to n/3, where n is the number of multiplier bits. Thus it allows a time gain in the partial products summation

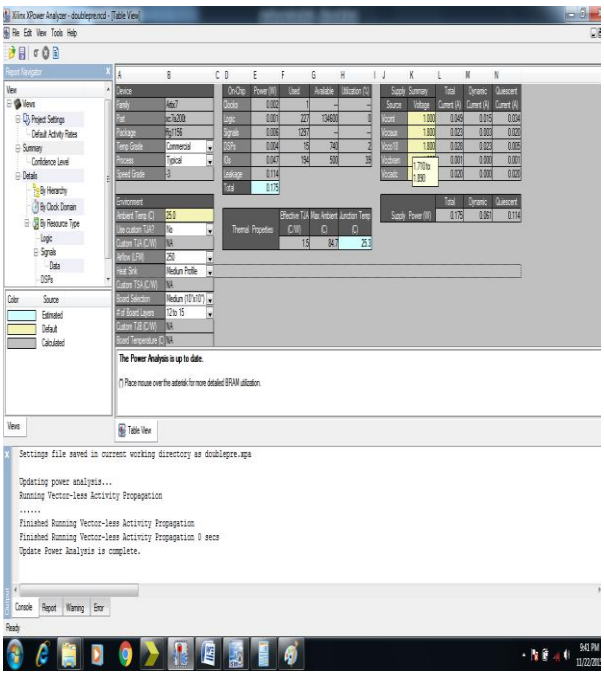
IV. RESULT:



Power analysis of existing code:



Power analysis of enhanced code:



Timing Summary of existing code:

Speed Grade: -3

Minimum period: 0.766ns (Maximum Frequency: 1306.336MHz)

Minimum input arrival time before clock: 0.695ns

Maximum output required time after clock: 1.935ns

Maximum combinational path delay: 10.564ns

Timing Summary of enhanced code:

Speed Grade: -3

Minimum period: 0.766ns (Maximum Frequency: 1306.336MHz)

Minimum input arrival time before clock: 0.695ns

Maximum output required time after clock: 1.935ns

Maximum combinational path delay: 10.560ns

V. CONCLUSION:

Finally, this project is implemented in an efficient manner with floating point and radix8 modified booth encoding algorithm for multipurpose applications. This concept is having the capability of handling the overflow, underflow cases, and this multiplier support truncation rounding mode was implemented. It has been performed the design and implementation of a 32 bit radix-8 booth

multiplier. It has been proved that it can be useful to apply a radix-8 architecture in high speed multipliers because of the gain in time obtained due to reduction of partial products to n/3.

VI. REFERENCES:

[1] M.Al-Ashrafy, A.Salem and W.Anis,"An Efficient Implementation of Floating Point Multiplier " Electronics Communications and Photonics Conference(SIECPC) 2011 Saudi International, pp.1- 5,2011.

[2] F.de Dinechin and B.Pasca. Large multipliers with fewer DSP blocks.In Field Pro- grammable Logic and Applications. IEEE, Aug. 2009. [3] IEEE 754-2008, IEEE Standard for Floating-Point ic, 2008.

[4] Patterson, D. & Hennessy, J. (2005), computer Organization and Design : The Hardware/software Interface , Morgan Kaufmann.

[5] B. Lee and N. Burgess, "Parameterisable Floatingpoint Operations on FPGA," Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, 2002

[6] A. Jaenicke and W.Luk, "Parameterized Floating- Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp. 897-900.

[7]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.

[8] John G. Proakis and Dimitris G. Manolakis (1996), "Digital Signal Processing: Principles,.Algorithms and Applications", Third Edition.

[9] N. Shirazi, A. Walters, and P.Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines,"Proceedings of the IEEESymposium on FPGAs for Custom Computing Machines (FCCM"95), pp.155–162, 1995.

[10] 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.

