DESIGN OF FLOATING POINT MULTIPLIER BASED ON BOOTH ALGORITHM USING VHDL
Ms. Anuja A. Bhat & Prof. Rutuja Warbhe
Department of EXTC, B. D. College of Engineering, Sewagram Wardha, RTMNU, INDIA
Department of Electrical Engineering, D.Y.P.I.T, Pimpri, Pune, SPPU, INDIA

DOi: 10.5281/zenodo.580860

Keywords: Booth Algorithm, Floating Point Multiplier, Xilinx, Floating Point Subtractor, VHDL.

Abstract
In this paper, high Speed, low power and less delay 32-bit IEEE 754 Floating Point Subtractor and Multiplier is presented using Booth Multiplier. Multiplication is an important fundamental function in many Digital Signal Processing (DSP) applications such as Fast Fourier Transform (FFT). Since multiplication dominates the execution time of most DSP algorithms, so there is need of high speed multiplier. The main objective of this research is to reduce delay, power and to increase the speed. The coding is done in VHDL, synthesis and simulation has been done using Xilinx ISE simulator. The modules designed are 24-bit Booth Multiplier for mantissa multiplication in Floating Point Multiplier, 32-bit Floating Point Subtractor and 32-bit Floating Point Multiplier. The Computational delay obtained by Floating Point Subtractor, booth multiplier and floating point multiplier is 16.180 nsec, 33.159 nsec and 18.623 nsec respectively.

Introduction
The fundamental and the core of all the digital signal processors (DSPs) are its multipliers, and the speed of the DSPs is mainly determined by the speed of its multiplier. Multipliers are key components of many high performance systems such as microprocessors, FIR filters, digital signal processors, etc. Performance of a system is generally determined by the performance of the multiplier because the multiplier is generally the slowest element in the system. Since multiplication dominates the execution time of most DSP application so there is need of high speed multiplier. Complex number operations are the backbone of many digital signal processing algorithms, which mostly depend on extensive number of multiplication. Complex multiplication is of immense importance in Digital Signal Processing (DSP) and Image Processing (IP). To implement the hardware module of Discrete Fourier Transformation (DFT), Discrete Cosine Transformation (DCT), Discrete Sine Transformation (DST) and modern broadband communications; large numbers of complex multipliers are required.

Booth Algorithm
Multiplication is an important fundamental function in arithmetic operation. Signed multiplication is a careful process. With unsigned multiplication there is no need to take the sign of the number into consideration. However in signed multiplication the same process cannot be applied because the signed number is in a 2’s compliment form which would yield an incorrect result if multiplied in a similar fashion to unsigned multiplication. That’s where Booth’s algorithm comes in. Booth’s algorithm preserves the sign of the result.

Booth’s algorithm is a well known method for 2’s complement multiplication. It speeds up the process by analyzing multiple bits of multiplier at a time. This widely used scheme for two’s complement multiplication was designed by Andrew D. Booth in 1951. Booth algorithm is an elegant way for this type of multiplication which treats both positive and negative operands uniformly. It allows n bit multiplication to be done using fewer than n additions or subtractions, thereby making possible faster multiplication. Booth’s multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two’s complement notation.
There are 2 methods that you should know before attempting Booth’s algorithm. Right-shift circulant and right-shift arithmetic.

**Right-shift circulant (RSC)**, is simply shifting the bit, in a binary string, to the right 1 bit position and take the last bit in the string and append it to the beginning of the string.

Example:
10110
after right-shift circulant now equals – 01011

**Right-shift arithmetic (RSA)**, is where you add 2 binary number together and shift the result to the right 1 bit position.

Example:
0100
+0110
result = 1010
Now shift all bits right and put the first bit of the result at the beginning of the new string:
result 1010
shift 11010

According to Booth’s multiplication algorithm among the two input binary numbers the one with minimum number of bit changes is considered as multiplier and the other as a multiplicand in order to reduce the time taken for calculating the multiplication product.

**The steps for performing booth multiplication are as follows:**
Let the multiplicand be ‘B’ and multiplier be ‘Q’.
Assume initially value of ‘A’ and ‘Q-1’ is zero.
The main step is to check last two bits.
There will be iterations according to the number of multiplier.
For example, if the multiplier is of 2-bit then 2 iterations will be done, for 4-bit multiplier 4 iterations are done, and so on.
Now, the algorithm starts, first the last two digits are checked and if the two bits are “00” or “11” the only Arithmetic Right Shift is done.
And if the last two bits are “01”, then A is added with B, and result is stored into A.
If the last two bits are “10”, then A is subtracted from B, and result is stored into A.
Finally, the result obtained is coded in binary form which gives the desired output.
In this way multiplication of any two numbers is performed using booth algorithm.

**Example :-**
Multiply 14 * -5 using 5-bit numbers.
14 in binary: 01110
-14 in binary: 10010
5 in binary: 00101
-5 in binary: 11011
Result: -70 in binary: 11101 11010 (10-bit result).

**Module Designed**

**Floating Point Subtractor (Fps)**
The following flowchart shows the operation of FPS
Performing Floating Point Subtraction Result = X - Y = (Xm * 2^Xe) - (Ym * 2^Ye) involves the following steps:

1. **Align binary point**
   - Initial result exponent : the larger of Xe, Ye
   - Compute exponent difference : Ye - Xe
   - If Ye > Xe Right shift Xm that many positions to form Xm 2^Ye-Xe
   - If Xe > Ye Right shift Ym that many positions to form Ym 2^Xe-Ye

2. **Compute sum of aligned mantissas**
   i.e Xm*2^Ye-Ye - Ym or Xm-Xm*2^Ye-Xe

3. **If normalization of result is needed, then a normalization steps follows**
   - Left shift result, decrement result exponent (eg. if result is 0.001xx..) or
   - Right shift result, increment result exponent (eg. if result is 10.1xx..)
   Continue until MSB of data is 1

4. **Check result exponent**
   - If larger than maximum exponent allowed return exponent overflow
   - If smaller than minimum exponent allowed return exponent underflow
   - If result mantissa is 0, may need to set the exponent to zero by a special step to return a proper zero.

Example:
X = 2345.125 = 100100101001.001 represented as:

| X | 10001010 | 00100101001001000000000 |

Y = 0.75 = 0.11 represented as:
Xe - Ye = 10001010 - 01111110 = 00000110 = 12\text{base 10}

Shift Ym 12\text{base 10} positions to the right to form Ym = 0.00000000000110000000000

Xm - Ym = 1.0010010100100100000000 - 0.00000000000110000000000

= 1.00100101001111000000000

Result is

- If the exponents differ by more than 24, the smaller number will be shifted right entirely out of the mantissa field, producing a zero mantissa.
  - The sum will then equal the larger number.
  - Such truncation errors occur when the numbers differ by a factor of more than \(2^{24}\), which is approximately \(1.6\times10^7\).
  - Thus, the precision of IEEE single precision floating point arithmetic is approximately 7 decimal digits.
- Negative mantissa are handled by first converting to 2’s complement and then performing the addition.
  - After the addition is performed, the result is converted back to sign-magnitude form.
- When adding numbers of opposite sign, cancellation may occur, resulting in a sum which is arbitrarily small, or even zero if the numbers are equal in magnitude.
  - Normalization in this case may require shifting by the total number of bits in the mantissa, resulting in a large loss of accuracy.

Booth multiplier_24bit

If 00 then P = Arithmetic Right Shift
If 11 then P = Arithmetic Right Shift
If 01 then P = P+S
If 10 then P = P+S
Example: 4*4

\[ m=4=0100 \quad m=4=1100 \]
\[ r=4=0100 \quad r=4=1100 \]
\[ A=0100\ 0000\ 0 \]
\[ S=1100\ 0000\ 1 \]
\[ P=0000\ 0100\ 0 \]

perform the loop 4 times:
\[ P=0000\ 0100\ 0 \]
\[ \text{Right shift } P=0000\ 0100\ 0 \]
\[ P=0000\ 0010\ 0 \]
\[ \text{Right shift } P=0000\ 0010\ 0 \]
\[ P=0000\ 0001\ 0 \]
\[ P=P+S=1100\ 0001\ 0 \]
\[ \text{Right shift } P=1100\ 0001\ 0 \]
\[ P=0110\ 0000\ 1 \]
\[ P=P+S=0010\ 0000\ 0 \]

Right shift P=0001\ 0000\ 0

The product is 0001\ 0000 which is 16

Floating Point Multiplier (Fpm)
The following flowchart shows the operation of FPM
Performing Floating Point Multiplication Result = X * Y = (-1)^Xs (Xm * 2^Xe) * (-1)^Ys(Ym * 2^Ye) involves the following steps:

1. If one or both operands is equal to zero, return the result as zero, otherwise:
2. Compute the sign of the result Xs XOR Ys
3. Compute the mantissa of the result:
   - Multiply the mantissas: Xm * Ym
   - Round the result to the allowed number of mantissa bits
4. Compute the exponent of the result:
   - Result Exponent = biased exponent (X) + biased exponent (Y) – bias
5. Normalize if needed, by shifting mantissa right, incrementing result exponent.
6. Check result exponent for overflow/underflow:
   - If larger than maximum exponent allowed return exponent overflow
   - If smaller than minimum exponent allowed return exponent underflow

**Example:**

X = 2345.125 = 100100101001.001 represented as:

Y = 0.75 = 0.11 represented as:

S = 0 XOR 0 = 0
Xe + Ye = 10001010 + 01111110 = 00001010
E = Xe + Ye – 127 = 00001010 – 01111111 = 10001001
M = Xm * Ym = 001001010010010000000000 * 100000000000000000000000
Rounding occurs in floating point multiplication when the mantissa of the product is reduced from 48 bits to 24 bits.

- The least significant 24 bits are discarded.

Overflow occurs when the sum of the exponents exceeds 127, the largest value which is defined in bias-127 exponent representation.

- When this occurs, the exponent is set to 128 (E=255) and the mantissa is set to zero indicating + or – infinity.

Underflow occurs when the sum of the exponents is more negative than -126, the smallest negative value which is defined in bias -127 exponent representation.

- When this occurs, the exponent is set to -127 (E=0).
- If M=0, the number is exactly zero.
- If M is not zero, then a denormalized number is indicated which has an exponent of -127 and a hidden bit of 0.
- The smallest such number which is not zero is $2^{-149}$. This number retains only a single bit of precision in the rightmost bit of the mantissa.

Fig: Simulation results of floating point multiplier

Conclusion
The modules designed are 24-bit Booth Multiplier for mantissa multiplication in Floating Point Multiplier, 32-bit Floating Point Subtractor and 32-bit Floating Point Multiplier. The Computational delay obtained by Floating Point Subtractor, booth multiplier and floating point multiplier is 16.180nsec, 33.159nsec and 18.623nsec respectively.

References
Conference on Communication, Computing and Virtualization 2016