Ratneshwar Urman Hemantkumar, Bariya Rajendrasinh N., Prof. Vishal Mishra /
International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622
www.ijera.com Vol. 3, Issue 3, May-Jun 2013, pp.436-438

Implementation and Performance Comparison of Shift-add and Radix-4 Booth Multiplication for Single Precision Floating Point on Xilinx Using VHDL

Ratneshwar Urman Hemantkumar[1], Bariya Rajendrasinh N.[2], Prof. Vishal Mishra[3]

[1,2,3] Department of Electronics and Communication, Marwadi Education Foundation's Group of Institutions, Rajkot, Gujarat, India.

ABSTRACT

This paper presents implementation and comparison of two multiplication methods which are currently being used. The multiplication is carried out between single precision floating point numbers. There is significant reduction in number of intermediate computation using Radix-4 Booth multiplication algorithm. Comparison of both the methods is done on basis of number of registers and LUTs used for designing. The proposed design is implemented using VHDL on Xilinx ISE.

Keywords - component: *Floating-Point, Single Precision, Shift-add, Radix-4, Multiplication.*

I. INTRODUCTION

Real numbers are represented by different ways on computers. But IEEE Floating-point representation is so far most used representation. Single precision floating point numbers are vastly used in computers, it is known as float in C, C++, C#, Java and single in MATLAB. Floating point representation has several advantages over fixed point representation. Fixed point representation places a radix point anywhere in the middle of digits, which has no fixed position, which makes it more complex while designing arithmetic operations, whereas floating point representation doesn't have this irregularity. Floating point numbers supports wide range of values.

With recent advancements in graphic processors, robotic applications, audio signal processing and data processing multiplication is a very important component of computation. Robotic applications, where complex mathematical algorithms are required to be calculated such as inverse kinematics, interpolation, velocity computations, which are accomplished by repetitive use of multiplication and addition operations. The number of logic blocks required to design such system determines the cost of the system.

This paper concentrates on two multiplication methods used for binary number: Shift and add method and Radix-4 booth multiplication method. These methods are implemented on single precision floating point numbers. Comparison of the two methods is done on the basis of logic elements used to design the algorithm using VHDL.

II. SINGLE PRECISION FLOATING POINT

The single precision floating point representation is defined by IEEE 754. Single precision floating point numbers are represented by 32 bits per number. 32 bits are divided into three bit fields: sign bit (1-bit), biased exponent (8-bits) and mantissa (23-bits).

Fig. 1: Single precision representation

'1' for sign bit signifies negative sign and '0' signifies positive number. 23 fraction bits and one additional bit, left of the radix point in normalized floating point number, add additional one bit precision to the number making the number of 24 bits precision. Range of the biased exponent is 0 to 255 because of 8 bit width, but 0 and 255 are reserved for special cases so range used for normalized floating point numbers is 1 to 254. '127' is added to exponent of normalized floating point number for biased exponent field (B.E.) to accommodate negative values of the exponent.

$$\text{B.E.} = \text{Exponent}+127 \quad (1)$$

So range of exponent of normalized floating point number will be -126 to 127, which will be represented as 1 to 254 in B.E. field. At last 23 fraction bits, combination of these three fields will be used to represent the numbers in the range from $\pm 1.17549\ldots \times 10^{-38}$ to $\pm 3.40282\ldots \times 10^{38}$. The wide range of real numbers can be represented with single precision.

The actual value of the double precision floating point number is the following:

$$\left(1 + \sum_{k=1}^{n-1} \text{bit}_k \times 2^{-k}\right) \times 2^e \quad (2)$$

Where bit_k is the normalized significand's k-th bit from the left and 'e' is exponent.

Special cases:
Zero is represented as B.E.=0 and F=0, +0 and -0 both are possible as per sign bit.
Infinity is represented as B.E.=255 and F=0, +∞ and -∞ both are possible as per sign bit.
NaN(not a number) is represented as B.E.=255 and F≠0.

III. SHIFT AND ADD MULTIPLICATION

Shift-and-add multiplier is the basic binary multiplier and is used commonly in all applications. Shift-add-multiplication is the simplest way to perform multiplication. It is derived for binary n-bits x n-bits integer which can also be used for single precision floating point number with some minor changes. Number of intermediate additions operation for shift-and-add multiplication method is equal to number '1's in the multiplier number. Below are the steps for shift-and-add multiplication:

Step 1: Divide the multiplicand and multiplier in three fields and extract each field; those are sign, biased exponent and fraction.
Step 2: Sign of resultant multiplication will be XOR of the sign bits of the multiplier and the multiplicand.
Step 3: Biased exponent of the resultant will be addition of the biased exponents of the multiplier and the multiplicand and subtracting the bias (i.e. 127 for single precision).
Step 4: Starting from the LSB of the multiplier, add the multiplicand if there is '1' at the LSB of the multiplier, then shift multiplier 1 place to the right.
Step 5: Repeat step 4, until all the fraction bits from multiplier are considered shifting multiplicand to the right per every bit of multiplier, including the leading '1' of the normalized floating point number.
Step 6: Normalize the final number if necessary, shifting it right and incrementing the biased exponent.
Step 7: Round the fraction to the appropriate number of bits, and renormalize if rounding generates a carry.

If there are special cases, which means either of the numbers is ±0, ±∞ or NaN, than above algorithm is not applied, and following operations are performed:

Any_number × ±0 = ±0
±∞ × ±0 = ±0
NaN × ±0 = ±0
Any_number × ±∞ = ±∞
Any_number × NaN = NaN

There will be maximum of 24 addition operation per each multiplication. Most digital signal processing such as filtering, convolution, and various transforms uses multiplication. In addition robotic applications where mathematical equations are to be designed, multiplication is used more than once. So considering current scenario where multiplication is vastly used, logic elements required to design the multiplication should be as minimum as possible. So for the purpose of logic elements reduction, an algorithm where intermediate computations are less should be used. Such an algorithm is Radix-4 booth multiplication algorithm, which significantly reduces intermediate computations.

IV.RADIX-4 BOOTH MULTIPLICATION

One of the solutions of reducing number of logic elements used for designing is reduction of intermediate calculation stages. Unlike shift-and-add multiplication method, radix-4 booth multiplication algorithm considers three bits of the multiplier at a time, hence reducing number addition stages required. It is a technique that allows for faster and smaller multiplication circuit by recoding the multiplier and multiplicand, which is to be multiplied. Radix-4 booth multiplication reduces the number of addition stages by half. Unlike shift-and-add multiplication, considering three bits of multiplier at a time and multiply multiplicand by 0, ±1 or ±2, according to the three bits under consideration. Three bits of the multiplier are considered such that each block overlaps the previous by one bit. Grouping starts from least significant bit and two bits from right and a '0' is considered as the first group. For example, consider 1011010001, than grouping is performed as shown below:

1 0 1 1 0 1 0 0 0 1 **0**

As we can see a '0' is added after LSB for the grouping and every group overlap the previous one by one bit. According to the groups starting from LSB, recoding and then further procedure is followed. Below is the recoding table:

TABLE 1: Radix-4 Booth Recoding [2]

Group	Partial Product
000,111	0
001,010	1*Multiplicand
011	2*Multiplicand
100	-2*Multiplicand
101,110	-1*Multiplicand

Here it should be noted that multiplying any number by two in binary is performed by shifting the number one bit to the right and negative numbers are represented by 2's complement. Example of radix-4 multiplication is shown below:

```
Multiplicand                0101000001
Multiplier          ×       0101101110
Recoding                  +1 +2 -1 +0 -2
                    11111110101111110
                    00000000000000000
                    11111010111110000
                    00101000010000000
                    01010000100000000
                    01100101011101110
```

So now by using Radix-4 booth multiplication algorithm, steps for single precision floating point multiplication is shown below:

Step 1: Divide the multiplicand and multiplier in three fields and extract each field; those are sign, biased exponent and fraction.
Step 2: Sign of resultant multiplication will be XOR of the sign bits of the multiplier and the multiplicand.
Step 3: Biased exponent of the resultant will be addition of the biased exponents of the multiplier and the multiplicand and subtracting the bias (i.e. 127 for single precision).
Step 4: Starting from the LSB of the multiplier, make groups of three bits as discussed earlier.
Step 5: Now according to the groups made by step 4, recode the multiplicand for first group.
Step 6: Then shift two bits to the left and recode the multiplicand for the second group and add it to the previous recoded number.
Step 7: Repeat step 6 for all the groups.
Step 8: Normalize the final number if necessary, shifting it right and incrementing the biased exponent.
Step 9: Round the fraction to the appropriate number of bits, and renormalize if rounding generates a carry.

If there are special cases, which means either of the numbers is ±0, ±∞ or NaN, than above algorithm is not applied, and following operations are performed:

```
Any_number × ±0    = ±0
±∞         × ±0    = ±0
NaN        × ±0    = ±0
Any_number × ±∞    = ±∞
Any_number × NaN = NaN
```

There will be maximum of 12 addition operations per each multiplication, which is half the number than shift-and-add operation. It will reduce the addition operations if there is a long sequence of '1's in the multiplier fraction field. Reduction in computation will result in reduction of number of logic elements required. It will also reduce the computation time, which is very important for real time applications.

V. SIMULATION RESULTS & PERFORMANCE ANALYSIS

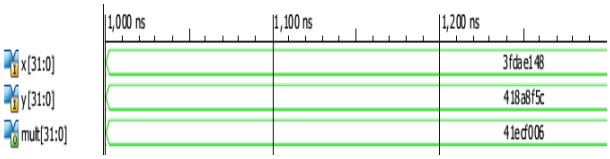

Fig. 2: Simulation result of radix-4 booth single precision floating point multiplication

Figure shows the simulation result of radix-4 booth multiplication for single precision multiplication. The inputs *x* (multiplicand) and *y* (multiplier) are shown in hexadecimal and so is the output *mult*. Here first input *x* is real number 1.71 and second input *y* is 17.32, which are represented as single precision floating point numbers in hexadecimal format. Output *mult* is 29.6172 in single precision floating point number hexadecimal format. Multiplication carried out by both the methods shift-and-add and radix-4 booth multiplication algorithm gives exact result but with vast difference in the number logic elements used while designing. Comparison of performance analysis for both the methods is shown below:

TABLE 2: Performance Analysis

	Shift-add	Radix 4
Slice Registers	69	29
Slice LUTs	1898	1886
Occupied slices	502	484

VI. CONCLUSION

By looking at the performance analysis, it can be concluded that radix-4 booth algorithm uses less hardware than shift-and-add algorithm. There is significant amount of reduction in logic elements used in the radix-4 booth algorithm. Speed of the radix-4 booth algorithm will be more than shift-and-add algorithm because of the reduction in hardware.

REFERENCES

[1] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754, 1985.
[2] K. Babulu,G.Parasuram "FPGA Realization of Radix-4 Booth Multiplication Algorithm for High Speed Arithmetic Logics", (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (5) , 2011, 2102-2107.
[3] S.Jagadeesh, S.Venkata Chary "Design of Parallel Multiplier–Accumulator Based on Radix-4 Modified Booth Algorithm with SPST", International Journal Of Engineering Research And Applications (IJERA) ISSN: 2248-9622, Vol. 2, issue 5, September-October 2012, pp.425-431.