ISSN No. 2455-5800
Journal of Scientific Research in Allied Sciences

Original Research Article

FUSED FLOATING POINT MAC (MULTIPLY AND ADD) UNIT WITH CONFIGURABLE ARCHITECTURE

Jyoti Singh Chouhan, Nitin Jain

Chouksey Engineering College, Bilaspur

Article history:
Submitted on: April 2016
Accepted on: May 2016
Email: info@jusres.com

ABSTRACT

Floating point unit is an integral part of next-generation processor. The fused Multiply Add (FMA) operation is used in many scientific and engineering applications. A key feature of floating point unit is, it greatly increases floating point performance and accuracy. Many floating point fused multiply add algorithm are used to reduce latency, the main objective of our work is to implement this algorithm with a little change to reduce the propagation of errors in an iterative process by using an additional error recovery circuit and to also increase the performance of block.

Keywords: Fused Multiply and Add (MAC) unit operation, floating point functional unit, FPGA design.

INTRODUCTION

Introduction: In computing, floating point describes a system for representing numbers that would be too large or too small to be represented as integers. Numbers are in general represented approximately to a fixed number of significant digits 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:

Significant digits × base^exponent

The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a computer

realization of scientific notation. Over the years, several different floating-point representations have been used in computers; however, for the last ten years the most commonly encountered representation is that defined by the IEEE 754 Standard.

The advantage of floating- point representation over fixed-point (and integer) representation is that it can support a much wider range of values. For example, a fixed-point representation that has seven decimal digits with two decimal places, can represent the numbers 12345.67, 123.45, 1.23 and so on, whereas a floating-point representation (such as the IEEE 754 decimal 32 format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, 12345670000000000, and so on. The floating-point format needs slightly more storage (to encode the position of the radix point), so when stored in the same space, floating-point numbers achieve their greater range at the expense of precision.

Floating point representation: Floating-point representation is similar in concept to scientific notation. Logically, a floating-point number consists of:

a. A signed digit string of a given base (or radix). This is known as the significand, or sometimes the mantissa or coefficient. The radix point is not explicitly included, but is implicitily assumed to always lie in a certain position within the significand-often just after or just before the most significant digit, or to the right of the rightmost digit. This article will generally follow the convention that the radix point is just after the most significant (leftmost) digit. The length of the significand determines the precision to which numbers can be represented.

b. A signed integer exponent, also referred to as the characteristic or scale, which modifies the magnitude of the number.

The significand is multiplied by the base raised to the power of the exponent, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent-to the right if the exponent is positive or to the left if the exponent is negative.

Symbolically, this final value is

$$S \times b^e$$

where s is the value of the significand (after taking into account the implied radix point), b is the base, and e is the exponent.

Equivalently, this is:

$$s/b^{p-1} \times b^e$$

Where s here means the integer value of the entire significand, ignoring any implied decimal point, and p is the precision-the number of digits in the significand.

Historically, different bases have been used for representing floating- point numbers, with base 2 (binary) being the most common, followed by base 10 (decimal), and other less common varieties such as base 16 (hexadecimal notation). Floating point numbers are rational numbers because they can be represented as one integer divided by another. The base however determines the fractions that can be represented exactly using a decimal base.

The way in which the significant, exponent and sign bits are internally stored on a computer is implementation-dependent. The common IEEE formats are described in detail later and elsewhere, but as an example, in the binary single-precision (32-bit) floating-point representation p=24 and so the significant is a string of 24 bits (1s and 0s). For instance, the number π's first 33 bits are 11001001 00001111 11011010 10100010 0. Rounding to 24 bits in binary mode means attributing the 24^{th} bit the value of the 25^{th} which yields 1101001 00001111 11011011. When this is stored using the IEEE 754 encoding, this becomes the significand s with e=1 (where s is assumed to have a binary point to the right of the first bit) after a left adjustment (or normalization) during which leading or padding zeros are are truncated should there be any. Note that they do not matter anyway. Then since the first bit of a non-zero binary significant is always 1 it need not be stored, giving an extra bit of precision. To calculate π the formula is:

$$(1+ {}^{p-1}\textstyle\sum_{n=1} bit_n \times 2_{-n}) \times 2^e = (1+1\times 2^{-1}+0\times 2^{-2}+1\times 2^{-4}+\ldots..2^{-23}) \times 2^1 = 1.5707964\times 2$$

Where n is the normalized significant's nth bit from the left. Normalization, which is reversed when 1 is being added above, can be thought of as a form of compression: it allows a binary significant to be compressed into a field one bit shorter than the maximum precision, at the expense of extra processing.

1.3 Range of floating point numbers: By allowing the radix point to be adjustable, floating-point notation allows calculations over a wide range of magnitudes, using a fixed number of digits, while maintaining good precision. For example, in a decimal floating point system with three digits, the multiplication that humans would write as

$$0.12 \times 0.12 = 0.0144$$

Would be expressed as

$$(1.2 \times 10\text{^}{-1}) \times (1.2 \times 10\text{^}{-1}) = (1.44 \times 10\text{^}{-2}).$$

In a fixed-point system with the decimal point at the left, it would be

$$0.120 \times 0.120 = 0.014.$$

A digit of the result was lost because of the inability of the digits and decimal point to 'float' relative to each other within the digit string.

The range of floating-point numbers depends on the number of bits or digits used for

representation of the significand (the significant digits of the number) and for the exponent. On atypical computer system, a 'double precision' (64-bit) binary floating-point number has a coefficient of 53 bits (one of which is implied), an exponent of 11 bits, and one sign bit. Positive floating-point numbers in this format have an approximate range of 10^-308 to 10^308 (because 308 is approximately 1023 × log2, since the range of the exponent is [-1022, 1023]). The complete range of the format is from about -10^308 through +10^308 (see IEEE 754).

The number of normalized floating-point numbers in a system F(B,P,L,U) (where B is the base of the system, P s the precision of the system to p numbers, L is the smallest exponent representable in the system, and U is the largest exponent used in the system) is: 2*(B-1)*B^(P-1)*(U-L+1).

There is a largest floating-point number, Overflow level = OFL = B^(U+1)*(1-B^(-P)) which has B-1 as the value for each digit of the mantissa and the largest possible value for the exponent.

1.4 IEEE 754: Floating point in Modern Computers: The IEEE has standardized the computer representation for binary floating-point numbers in IEEE 754. This standard is followed by almost all modern machines. Notable exceptions include IBM mainframes, which support IBM's own format (in addition to the IEEE 754 binary and decimal formats), and Cray vector machines.

1.5 Basic Formats: The standard defines five basic formats, see table (1), the first three formats named and use 32,64 and 128 bits respectively. The last two formats are used for decimal floating point numbers and use 64 and 128 bits to encode them. All the basic formats may be available in both hardware and software implementations. This thesis concerns only on binary floating point with double precision format, so it will be discussed in more details.

Table 1 Five basic formats of IEEE Standard

Name	Common name	Base	Digits	Emin	Emax
Binary 32	Single precision	2	23+1	-126	+127
Binary 64	Double precision	2	52+1	-1022	+1023
Binary128	Quadruple precision	2	112+1	-16382	+16383
Decimal 64		10	16	-383	+384
Decimal 128		10	34	-6143	+6144

1.6 Rounding modes: rounding is used when the exact result of a floating-point operation (or a conversion to floating-point format) would need more digits than there are digits in the significand. There are several different rounding schemes (or rounding modes). IEEE 754

specifies the following rounding modes:

(a) Round to nearest, where ties round to the nearest even digit in the required position (the default and by far the most common mode).

(b) Round to nearest, where ties round away from zero (optional for binary floating-point and commonly used in decimal).

(c) Round up (toward +$; negative results thus round towards zero).

(d) Round down (toward -$; negative results thus round away from zero).

(e) Round towards zero (sometimes called "chop" mode; it is similar to the common behavior of float-to-integer conversions, ,which convert -3.9 to -3).

Floating Point Operations

2.1 Multiplication: Multiplication of two floating-point values follows basic algebraic concepts. A number x may be rewritten as shown in equation, where x_n is a normalized mantissa of x, base is the number base (10 for decimal and 2 for binary), and exp is the number of shifts the radix point was shifted to the left to normalize x_n* base^x_p. From equation it is clear that any IEEE Standard floating-point number may be written in this manner, and the format provides all the components directly. Given this notation, the product of two number x and y may be obtained by the following procedure:

$$\text{Product} = x*y$$

$$= x_n \times \text{base}^{exp1} * y_n \times \text{base}^{exp2}$$

$$= x_n * y_{-n} \times \text{base}^{exp1+\,exp2}$$

This splits the multiplication process into two parallel data paths. The first calculates the sum of the exponents, while the second calculates the product of the two mantissa. Because both data paths operate on standard integer values, they may be implemented using conventional hardware methods. When dealing with IEEE floating-point numbers, the multiplication involves several additional steps. In the unpack operands(UO) stage, the mantissa and exponent fields of each operand need to be evaluated in order to correctly generate the implied MSB of the mantissa. To reiterate, the MSB is implied to the one for all cases except when the exponent is zero. If the exponent is zero and the mantissa is greater than zero, the exponent is set to one since this indicates that the operand is a denormalized number.

2.2 Basic Algorithm: Let A,B and C be the operands represented by (M_a,E_a), (M_b,E_b), and (M_c,E_c) respectively. The significand are signed and normalized, and the result W is given by:

$$W=A+(B\times C)$$

Where W is represented by (M_w,E_w), where Mw is also signed and normalized. The high level description of this operation is composed of the following five steps:

(a)Multiply significand M_b and M_c, add exponents Eb and Ec, and determine the alignment shift and shift Ma, produce the intermediate result exponent Ew = max (E_a,E_b+E_c).

(b) Add the product and aligned Ma.

(c) Normalize the adder output and update the result exponent.

(d) Round.

(e)Determine the exception flags and special values.

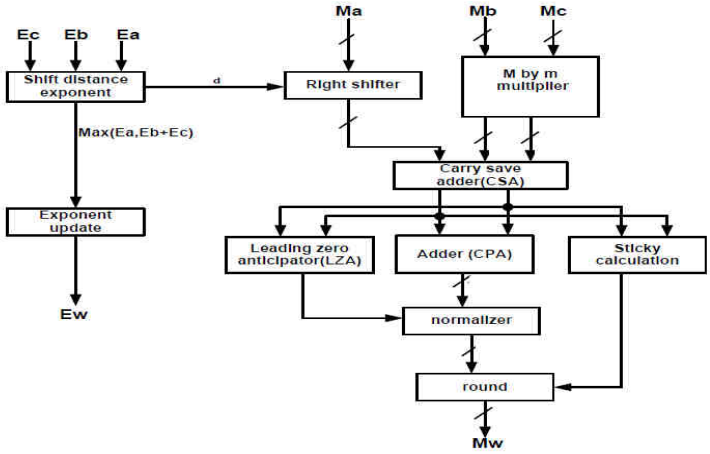

Figure 1 Basic implementation of FMA operation

2.3 Algorithm Description:

(a) Sticky bit: Inexact sticky bit (or just sticky-bit) calculation is closely related to the alignment phase of floating-point addition. The sticky bit is required to guarantee correct rounding in the final stage of floating-point arithmetic. In a typical FMA path, inexactness can occur in two process when bits are shifted out of range during alignment of C, and when intermediate result is scaled back to register size after normalization. During alignment, the sticky bit calculation can be thought of as an extra bit position behind the shifter that records all bits that pass through it while being shifted out of range.

1.0010011010001000

1.00100110100010000

1.00100110101001000

If a1 passes through, the sticky-bit will remember this and stay 1 regardless of other bits that pass through. This cannot directly be implemented in hardware, because most shifters do not shift bit-after-bit. The sticky bit can however be found by OR'ing all bits beyond the LSB into a sticky bit.

(b) Multiplication: Fundamentally, a multiplication is a series of additions. A binary n-bit multiplication can be realized by performing a series of n-shifts and additions.

(b.1) Both encoding: booth's algorithm serves two purposes. First it enables us to multiply signed (two's complement) numbers and secondly it helps reduce the number of partial products. To understand Booth multipliers we first have to recapitulate the basics of binary multiplication. Several important observations can be made in binary multiplication: In multiplications (A*B) we have a multiplier (A) and a multiplicand (B). for each digit in the multiplier, a partial product is generated. If the multiplier bit is 0, the partial product is zero, otherwise the partial product is the multiplicand. The final product is produced by repeatedly shifting a partial product to the left and adding it to the preceding partial product. A small example of binary multiplication is shown below:

```
      101110   Multiplicand
      010011   Multiplier
------------------------------
      101110   Partial products
     101110
    000000
   000000
  101110
 00000
------------------------------
001101101010
```

(b.2) Carry save adders; The adder is also one of the oldest and most widely used arithmetic components in digital processors. Its purpose is to add two operands A and B. in its simplest form a binary adder adds two bits. Such a combinational circuit is called a half adder. The elementary operations of a half adder. The elementary operations of a half adder are 0+0=0, 0+1=1, 1+0=1, 1+1=10. When both the augends and addend are 1, the output consists of two bits. Because of this, the output of adder is always represented by two bits, the sum and the carry. If n-bits operands are added, the carry of bits in position i-1 ($i \leq n$) is added to the next higher order pair of bits i. this requires a combinatorial circuit that can perform addition on three bits: A,B and carry in. Such a circuit can be constructed from two half adders combined with an OR-gate and is called a full adder. To add two n-bit operands, a chain of n full adders

can be used in cascade, with their carry out from the full adder connected to the carry in of the next full adder, as shown in figure.

The carry save adder (CSA) is a type of adder that computes the sum of three or more binary inputs. It differs from other binary adders in that it outputs two numbers of the same dimensions as the inputs, one which is a sequence of partial sum bits and other which is a sequence of carry bits. Due to this redundant form of output, carry propagation is completely eliminated. A single bit, three bit input CSA is shown in figure, compared to a full adder.

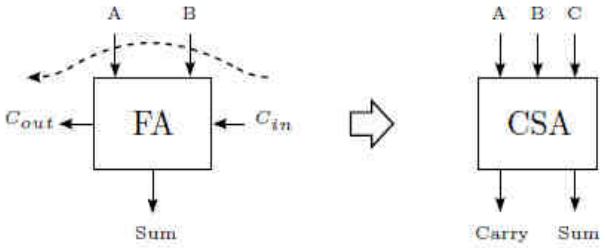

Figure 2 Full adder and carry save adder

(b.3) Addition: So far we have seen that C can efficiently be aligned to the product in parallel with the multiplication of A and B itself. Even adding C to the product does not require many sources or additional delay. However, we have not yet mentioned anything about negative numbers. A problem arises when the signs of A*B and C are different or when we simply want to subtract C from A*B. eight different scenarios can be distinguished when performing signed addition/ subtraction:

A + B

-A + -B

A + -B (A>B)

A + -B (A<B)

A – B (A>B)

A – B (A<B)

-A - -B (A>B)

-A - -B (A<B)

(b.4) Normalization: In the final stage, after having multiplied A with B and after having added c, the result will mostly likely have to be normalized and rounded. Normally, these operations contribute to the critical path. We can improve the situation by using a technique called LZA. A prediction of the number of leading zeros that can be performed in parallel

with addition.

(b.5) Leading Zero Anticipation: All normalized floating point numbers must have a '1' as leading significand bit. In most cases, an intermediate result is not normalized. Hence, a floating-point unit must be able to normalize its results. The normalization process involves detection of leading zeroes. LZAs make use of a propagate (T), a generate (G), and a kill (Z) function . These functions are defined as the names already suggest, these functions look for carry generation, carry termination and carry propagation. They help find patterns in the input (tuples (Ai-1,Bi-1), (Ai,Bi), (Ai+1,Bi+1) per indicator i) that generate the appropriate indicator bit for position i. if the adder is able to perform addition on signed numbers, the LZA algorithm becomes a bit more complicated. It should not only be able to detect leading zeroes, but also leading ones, several cases can be distinguished in which different patterns have to be matched to find the leading one.

A	0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 0
B	0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 0 1 0
LZA pattern	Z Z Z Z T T T T Z Z T G G G G Z T Z
A + B	0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 1 0
Leading one	- - - - X - - - - - - - - - - - - -

Figure 3 Leading zero anticipation example

CONCLUSION

The precise fused floating-point MAC unit with configurable architecture is very efficient. The MAC unit uses only one rounding technique that is rounding towards zero. The earlier methods uses more hardware but the MAC unit uses very less hardware. The normal adder used in earlier methods while representing the bit, the LSB there corresponds to half adder and rest of bits including MSB corresponds to full adder. While in MAC unit all the bits from LSB to MSB totally corresponds to full adder. The earlier methods does not introduce any error recovery circuit to avoid the errors while the MAC unit introduces error recovery circuit to avoid the errors.

REFERENCES

[1] Akkas, A.; Schulte, M.J., "A decimal floating-point fused multiply-add unit with a novel decimal leading-zero anticipator," *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on* , vol., no., pp.43,50, 11-14 Sept. 2011

[2] Swartzlander, E.E.; Saleh, H.H., "FFT Implementation with Fused Floating-Point Operations," *Computers, IEEE Transactions on* , vol.61, no.2, pp.284,288, Feb. 2012

[3] Chi Wai Yu; Smith, A.M.; Luk, W.; Leong, P.H.-W.; Wilton, S. J E, "Optimizing Floating Point Units in Hybrid FPGAs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.20, no.7, pp.1295,1303, July 2012

[4] Janhunen, J.; Salmela, P.; Silven, O.; Juntti, M., "Fixed- versus floating-point implementation of MIMO-OFDM detector," *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on* , vol., no., pp.3276,3279, 22-27 May 2011.

[5] Zaki, A.M.; Bahaa-Eldin, A.M.; El-Shafey, M.H.; Aly, G.M., "Accurate floating-point operation using controlled floating-point precision," *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on* , vol., no., pp.696,701, 23-26 Aug. 2011

[6] Jongwook Sohn; Swartzlander, E.E., "Improved Architectures for a Fused Floating-Point Add-Subtract Unit," *Circuits and Systems I: Regular Papers, IEEE Transactions on* , vol.59, no.10, pp.2285,2291, Oct. 2012

[7] Gilani, S.Z.; Nam Sung Kim; Schulte, M., "Energy-efficient floating-point arithmetic for digital signal processors," *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on* , vol., no., pp.1823,1827, 6-9 Nov. 2011.

[8] Krasniewski, A., "Low-Cost Concurrent Error Detection for FSMs Implemented Using Embedded Memory Blocks of FPGAs," *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE* , vol., no., pp.178,183, 18-21 April 2006