fp_mult

Floating-Point Multiplier

January 1996, ver. 1 Functional Specification 4

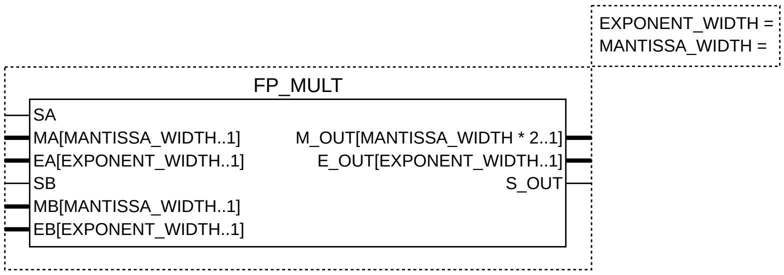
Features

- `fp_mult` reference design implementing a floating-point multiplier
- Parameterized mantissa and exponent bit widths
- Optimized for FLEX 10K and FLEX 8000 device families
- Supported by schematic and text design entry methods, including the Altera Hardware Description Language (AHDL), VHDL, and Verilog HDL
- Easily customized for particular applications
- Useful for a variety of applications, including video signal processing and scientific computation

General Description

The Altera `fp_mult` function implements a high-speed floating-point multiplier with parameterized input widths. This function uses sign-mantissa-exponent notation with parameterized mantissa and exponent bit widths. See Figure 1.

Figure 1. fp_mult Symbol



Function Prototype

The AHDL Function Prototype for the `fp_mult` function is shown below:

```
FUNCTION fp_mult (sa, ma[mantissa_width..1],
       ea[exponent_width..1], sb, mb[mantissa_width..1],
       eb[exponent_width..1])
  WITH (mantissa_width, exponent_width)
  RETURNS (m_out[(mantissa_width) * (2)..1],
       e_out[exponent_width..1], s_out);
```

Parameters

Parameters for the `fp_mult` function are provided in Table 1.

Table 1. fp_mult Parameters

Name	Default	Value	Description
`exponent_width`	7	Integers only	Width of all exponents (in bits)
`mantissa_width`	8	Integers only	Width of input mantissas (in bits)

Ports

Input and output ports for the `fp_mult` function are shown in Table 2.

Table 2. fp_mult Input & Output Ports

Port Type	Name	Description
Input	`sa`	Sign bit for the a input: 1 = positive, 0 = negative
Input	`ma[mantissa_width..1]`	Mantissa for the a input
Input	`ea[exponent_width..1]`	Exponent for the a input
Input	`sb`	Sign bit for the b input: 1 = positive, 0 = negative
Input	`mb[mantissa_width..1]`	Mantissa for the b input
Input	`eb[exponent_width..1]`	Exponent for the b input
Output	`m_out[(mantissa_width)*(2)..1]`	Mantissa for the output
Output	`e_out[exponent_width..1]`	Exponent for the output
Output	`s_out`	Sign bit for the output: 1 = positive, 0 = negative

Functional Description

The `fp_mult` reference design implements a fast, flexible floating-point multiplier that provides parameterized mantissa and exponent widths. The multiplier and the floating-point format can be easily customized for particular applications by modifying a copy of the AHDL Text Design File (**.tdf**).

In floating-point functions, the sign bit represents the sign of the mantissa: 1 for positive, 0 for negative. The mantissa is a positive number less than 1. A 0 is implied to the left of the binary point. After normalization, the most significant bit (MSB) is always 1. The exponent is represented in excess $2^{(n-1)}$ notation, where n is the number of bits in the exponent.

For example, the binary representation of the number 0.75×2^1 is shown below. This example assumes 8 bits for the mantissa (M) and 7 bits for the exponent (E). S represents the sign bit.

$S = 1, M = 11000000, E = 1000001$

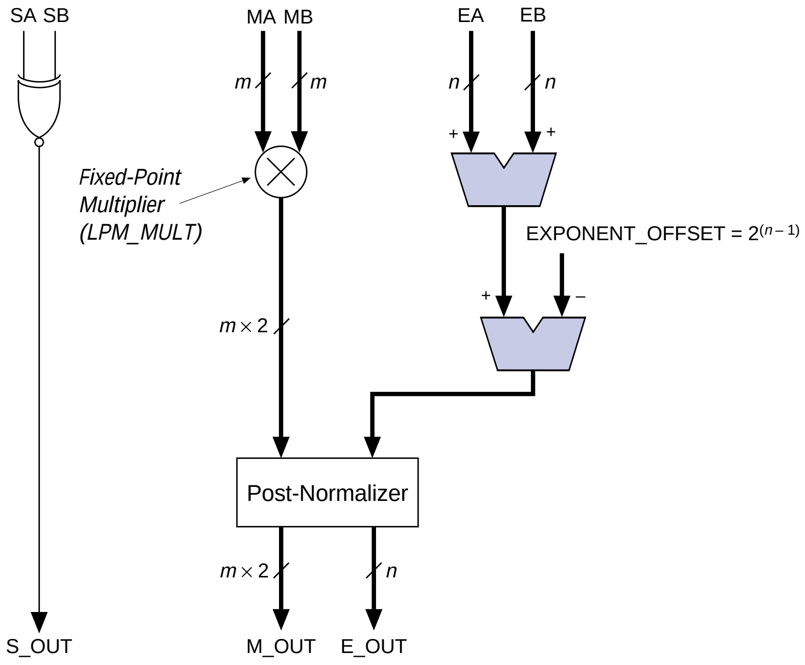
Similarly, the binary representation of the number 0.625×2^{-1} is:

$S = 1, M = 10100000, E = 0111111$

Figure 2 shows a block diagram of the `fp_mult` floating-point multiplier.

Figure 2. fp_mult Block Diagram

In this figure, m represents mantissa_width and n represents exponent_width.



Floating-Point Multiplication

To multiply floating-point numbers, the mantissas are first multiplied together with an unsigned integer multiplier. Then, the exponents are added, and the excess value (`exponent_offset`) $2^{(n-1)}$ is subtracted from the result. The sign of the output (`s_out`) is the `XNOR` of the signs of the inputs (`sa` and `sb`). After multiplication has taken place, the post-normalizer normalizes the result, if necessary, by adjusting the mantissa and exponent of the result to ensure that the MSB of the mantissa is 1.

Using the `fp_mult` function in a design produces a double precision output (i.e., the mantissa in the result has twice the number of bits of either input mantissa). Therefore, the result does not lose precision and does not require rounding.

The mantissa must be post-normalized whenever floating-point numbers are multiplied. As a double precision output is created, the implied denominator of the mantissa fraction is squared in value, from 8 to 16 bits. The denominator of the mantissa fraction is 65536 (= 2^{16}) in double precision format, and 256 (= 2^8) in single precision. To retain as many significant digits as possible for consequential floating point operations, the result must be normalized.

To normalize the mantissa, the mantissa is shifted left (i.e., the mantissa is multiplied by powers of 2). For each bit shifted left, the exponent must be reduced by 1. The following example shows a number before normalization and its normalized equivalent:

Unnormalized number:	$0.0001011001110001 \times 2^{50}$
Normalized equivalent:	$0.1011001110001000 \times 2^{47}$

Table 3 shows an example of floating-point multiplication. The subscript “d” indicates that the number is a decimal number.

Table 3. Floating-Point Multiplication

Operation	Decimal Equivalent (Exponent in Excess 0)	Binary (Exponent in Excess 64)
Multiplication	$(39 \times 2^{10d}) \times (203 \times 2^{6d})$	$(00100111.0 \times 2^{74}) \times (11001011.0 \times 2^{70})$
Normalization	$(0.609375 \times 2^{16d}) \times (0.79296875 \times 2^{14d})$	$(0.10011100 \times 2^{80}) \times (0.11001011 \times 2^{78})$
Result	7917×2^{16d}	$0.0001111011101101 \times 2^{80}$
Normalize	63336×2^{13d}	$0.1111011101101000 \times 2^{77}$
Decimal result	518,848,512	—

Floating-Point Representation

Floating-point numbers can be represented by many different notations. The `fp_mult` reference design uses an implied leading zero for the mantissa, with an unsigned m-bit mantissa, and n-bit exponent, where m = `mantissa_width` and n = `exponent_width`. A separate sign bit represents the sign of the mantissa.

The following examples of an 8-bit positive mantissa and a 7-bit exponent assume `mantissa_width` = 8 and `exponent_width` = 7. The numbers in Table 4 should be adjusted accordingly if different parameter values are used.

An 8-bit positive mantissa allows fractions with numerators ranging from 0 to 255. The implied leading zero limits the range of the mantissa from 0 to 0.9961, and the separate sign bit allows the mantissa to have a value from –0.9961 to +0.9961. Because the mantissa is in fractional form, having an additional number of bits in the mantissa does not result in a larger mantissa, but instead offers greater precision. Table 4 lists examples of 8-bit mantissas with implied leading zeros.

Table 4. 8-Bit Mantissas

Mantissa	Implied Zero	Binary Fraction	Decimal Fraction	Decimal
11001110	0.11001110	11001110 / 100000000	206 / 256	0.80469
00001100	0.00001100	00001100 / 100000000	12 / 256	0.04688
10100001	0.10100001	10100001 / 100000000	161 / 256	0.62891

A 7-bit exponent is represented in excess 64 format—i.e., for an *n*-bit exponent, the representation is excess $2^{(n-1)}$. Excess (or offset) format allows both negative and positive exponents to be represented with positive numbers, which results in simpler calculations for exponent handling. To represent an exponent in excess $2^{(n-1)}$ format, add $2^{(n-1)}$ to the value of the exponent. For example, to represent an exponent in excess 64 format, add 64 to the exponent; thus, the maximum value for the exponent is +63, and the minimum value is –64. In excess 64 format, an exponent of 10 is represented as 74, and an exponent of –10 is represented as 54. The exponent 0 is represented as 64.

Examples of floating-point multiplication for 8-bit mantissa, 7-bit exponent floating-point numbers are provided below. The subscripts "b" and "d" indicate that the number is a binary or a decimal number, respectively.

Example 1: Largest Positive Number

$+11111111_b\ 1111111_b$

$= +0.11111111_b \times 2^{(1111111_b - 1000000_b)}$

$= +0.11111111_b \times 2^{0111111_b}$

$= +0.11111111_b \times 2^{63_d}$

$= +11111111.0_b \times 2^{55_d}$

$= +255_d \times 2^{55_d}$

$= +9.187343239836_d \times 10^{18_d}$

Example 2: Largest Negative Number

$-11111111_b\ 1111111_b$

$= -9.187343239836_d \times 10^{18_d}$

Example 3: Smallest Number (Closest to Zero)

$\pm 10000000_b\ 0000000_b$

$= \pm 0.10000000_b \times 2^{(0000000_b - 1000000_b)}$

$= \pm 0.10000000_b \times 2^{(0_d - 64_d)}$

$= \pm 0.10000000_b \times 2^{-64_d}$

$= \pm 10000000.0_b \times 2^{-72_d}$

$= \pm 128_d \times 2^{-72_d}$

$= \pm 2.710505431214_d \times 10^{-20_d}$

Example 4: Typical Value

$-11000111_b\ 1001001_b$

$= -0.11000111_b \times 2^{(1001001_b - 1000000_b)}$

$= -0.11000111_b \times 2^{1101_b}$

$= -0.11000111_b \times 2^{9_d}$

$= -11000111.0_b \times 2^{1_d}$

$= -199_d \times 2^d$

$= -398_d$

Floating-Point Error Detection

The `fp_mult` function does not check for error conditions such as overflow and underflow because implementing error detection and correction in a design can add significant delays to the circuit. Although error detection can be implemented quickly, error correction causes long delays because it requires the exponent and/or mantissa values to be modified. Boundary conditions for the mantissa and exponent can also be added by the designer. Error detection and correction, and boundary conditions can be added to `fp_mult` by modifying the Logic Section of a copy of the **fp_mult.tdf** file.

Go to MAX+PLUS II Help for more information about programming with AHDL.

Overflow & Underflow Error Conditions

The most common floating-point errors are overflow and underflow. Overflow occurs when the resultant exponent has a value greater than the number of bits in `exponent_width`. For example, overflow occurs with a 7-bit exponent when the resultant exponent is greater than 63.

To detect overflow in a design, an extra bit of precision in the exponent must be carried when calculating the exponent of the output. With a 7-bit exponent, overflow may have occurred if the MSB of the exponent is 1 after the exponents are added and 64 is subtracted from the result. Overflow has occurred if the MSB of the exponent is 1 after the post-normalization correction. Therefore, the general AHDL equation for overflow detection is as follows:

```
Overflow = <MSB of exponent after addition>
           AND <MSB after excess 64 subtraction>
           AND <MSB after post-normalization correction>;
```

Underflow occurs when the MSB of the exponent is equal to 0 after the exponents are added and 64 is subtracted from the result, but equal to 1 after the post-normalization correction. With a 7-bit exponent, underflow occurs when the exponent is less than –64. The general AHDL equation for underflow is as follows:

```
Underflow = !<MSB after addition>
             AND <MSB after post-normalization correction>;
```

Boundary Conditions

The values of the mantissa and/or exponent can be used to set boundary conditions or to indicate error conditions such as not-a-number (NaN). The NaN condition occurs as a result of invalid operations, such as zero × infinity. Infinity can be represented in several ways, such as setting the exponent to its largest value. The largest exponent cannot be used to represent other numbers if it is used to represent infinity. Therefore, some dynamic range is lost.

Table 5 shows the mantissa and exponent values used to set boundary conditions and the NaN error condition.

Table 5. Boundary & NaN Error Conditions

Condition Type	Name	Sign	Mantissa	Exponent
Boundary	Zero	0	0	0
Boundary	–Zero	1	0	0
Boundary	Infinity	0	0	$2^{(n-1)}$
Boundary	–Infinity	1	0	$2^{(n-1)}$
Error	NaN	–	$2^{(m-1)}$	Anything except 0

The `fp_mult` reference design does not define boundary conditions or check for the NaN error condition. Like the overflow and underflow error conditions, boundary conditions and NaN detection can be added to `fp_mult` by modifying the Logic Section of a copy of the **fp_mult.tdf** file.

2610 Orchard Parkway
San Jose, CA 95134-2020
(408) 894-7000
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 894-7104
Literature Services:
(408) 894-7144