Double Precision Floating Point Core Verilog

Introduction

This document describes the Verilog double precision floating point core, posted at www.opencores.org. The Verilog version of the code is in folder "fpu_double", and the VHDL version is in folder "double_fpu". There is a readme file in each folder, and a testbench file to simulate each core. These cores are designed to meet the IEEE 754 standard for double precision floating point arithmetic.

Double Precision Floating Point Numbers

The IEEE 754 standard defines how double precision floating point number are represented. 64 bits are used to represent a double precision floating point number.

Sign	Exponent	Mantissa
63	62……..52	51……0

The sign bit occupies bit 63. '1' signifies a negative number, and '0' is a positive number. The exponent field is 11 bits long, occupying bits 62-52. The value in this 11-bit field is offset by 1023, so the actual exponent used to calculate the value of the number is 2^(e-1023). The mantissa is 52 bits long and occupies bits 51-0. There is a leading '1' that is not included in the mantissa, but it is part of the value of the number for all double precision floating point numbers with a value in the exponent field greater than 0. A 0 in the exponent field corresponds to a denormalized number, which is explained in the next section. The actual value of the double precision floating point number is the following:

Value = -1^(sign bit) * 2^(exponent – 1023) * 1.(mantissa)

(1.mantissa) being a base 2 representation of a number between 1 and 2, with 1 followed by a decimal point and the 52 bits of the mantissa.

For an example, how would the number 3.5 be represented in a double precision floating point format? The sign bit 63 is 0 to represent a positive number. The exponent will be 1024. This is calculated by breaking down 3.5 as (1.75) * 2^(1). The exponent offset is 1023, so you add 1023 + 1 to calculate the value for the exponent field. Therefore, bits 62-52 will be "1000000000". The mantissa corresponds to the 1.75, which is multiplied by the power of 2 (2^1) to get 3.5. The leading '1' is implied in the mantissa but not actually included in the 64-bit format. So .75 is represented by the mantissa. Bit 51, the highest bit of the mantissa, corresponds to 2^(-1). Bit 50 corresponds to 2^(-2), and this continues down to Bit 0 which corresponds to 2^(-52). To represent .75, bits 51 and 50 are 1's, and the rest of the bits are 0's. So 3.5 as a double-precision floating point number is:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000000	1100000000000000000000000000000000000000000000000000

Denormalized Numbers

A denormalized number is any nonzero number with an exponent field of 0. The exponent offset is 1023, but for the case when the value in the exponent field is 0, then the offset is changed to 1022. The value multiplied by the mantissa is 2^(-1022). The exponent offset is changed to 1022 because for denormalized numbers, the implied leading '1' is no longer included. So to calculate the actual value of a denormalized number, you multiply 2^(-1022) by the mantissa without the leading '1'. The range of values that can be represented by a denormalized number is approximately 2.225e-308 to 4.94e-324.

For example, the number 2e-309, is represented in the double precision floating point format as:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	00000000000	0001011100000010101011100100110100011111101101011101

Special Cases

Positive Infinity is:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	11111111111	0000000000000000000000000000000000000000000000000000

Negative Infinity is:

Sign	Exponent	Mantissa
63	62……..52	51……..0
1	11111111111	0000000000000000000000000000000000000000000000000000

Quiet Not a Number (QNaN) is:

Sign	Exponent	Mantissa
63	62……..52	51……..0
x	11111111111	1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Signaling Not a Number (SNaN) is:

Sign	Exponent	Mantissa
63	62……..52	51……..0
x	11111111111	0xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

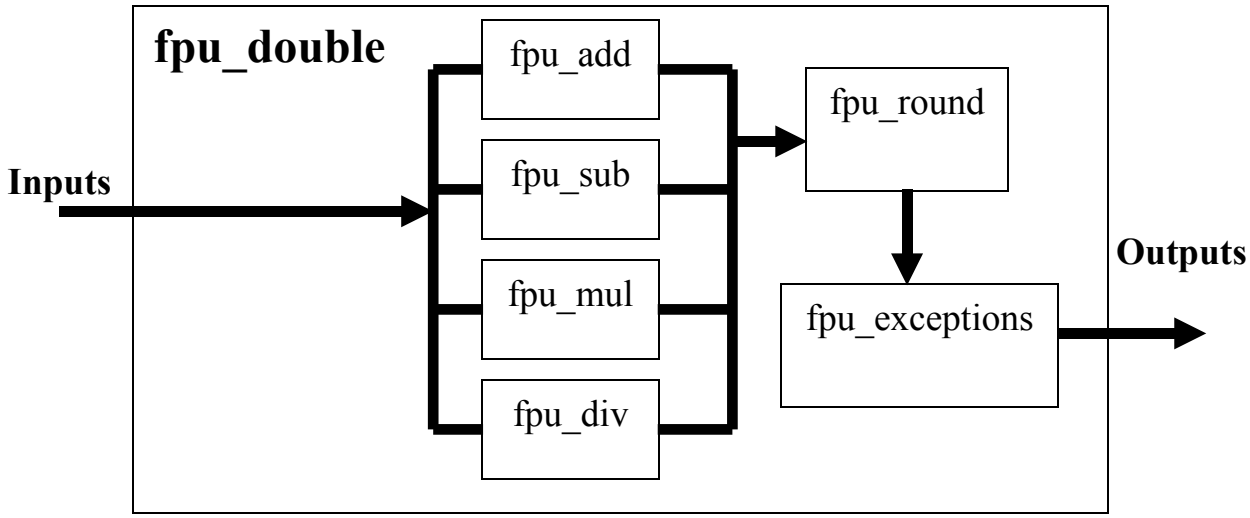
For the SNaN, at least one of the mantissa bits must be nonzero. Otherwise, it would be interpreted as Infinity. For both QNaNs and SNaNs, the sign bit can be 1 or 0, and the lower 51 bits of the mantissa can be any value, as long as it is nonzero for the SNaN.

Floating Point IP Core (Verilog)

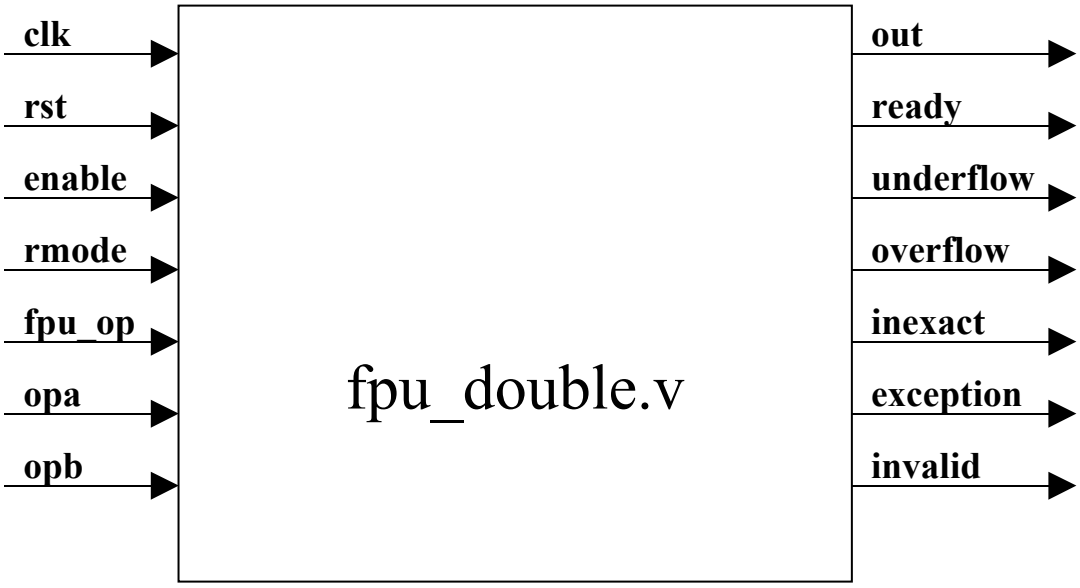
The floating point IP core is separated into 7 source files:

1. fpu_double.v (top level)
2. fpu_add.v
3. fpu_sub.v
4. fpu_mul.v
5. fpu_div.v
6. fpu_round.v
7. fpu_exceptions.v

Hierarchy



Top Level



The input signals to the top level module are the following:

1. clk (global)
2. rst (global)
2. enable (set high to start operation)
3. rmode (rounding mode, 2 bits, 00 = nearest, 01 = zero,
10 = pos inf, 11 = neg inf)
4. fpu_op (operation code, 3 bits, 000 = add, 001 = subtract,
010 = multiply, 011 = divide, others are not used)
5. opa, opb (input operands, 64 bits)

The output signals from the module are the following:

6. out_fp (output from operation, 64 bits)
7. ready (goes high when output is available)
8. underflow
9. overflow
10. inexact
11. exception
12. invalid

The top level, fpu_double, starts a counter (count_ready) one clock cycle after enable goes high. The counter (count_ready) counts up to the number of clock cycles required for the specific operation that is being performed. For addition, it counts to 20, for subtraction 21, for multiplication 24, and for division 71. Once count_ready reaches the specified final count, the ready signal goes high, and the output will be

valid for the operation being performed. fpu_double contains the instantiations of the other 6 modules, which are 6 separate source files of the 4 operations (add, subtract, multiply, divide) and the rounding module and exceptions module.

If the fpu operation is addition, and one operand is positive and the other is negative, the fpu_double module will route the operation to the subtraction module. Likewise, if the operation called for is subtraction, and the A operand is positive and the B operand is negative, or if the A operand is negative and the B operand is positive, the fpu_double module will route the operation to the addition module. The sign will also be adjusted to the correct value depending on the specific case.

Addition

The addition operation is performed in the source file, (fpu_add.v). The input operands are separated into their mantissa and exponent components, and the larger operand goes into "mantissa_large" and "exponent_large", with the smaller operand populating "mantissa_small" and "exponent_small". The comparison of the operands to determine which is larger only compares the exponents of the two operands, so in fact, if the exponents are equal, the smaller operand might populate the "mantissa_large" and "exponent_large" registers. This is not an issue because the reason the operands are compared is to find the operand with the larger exponent, so that the mantissa of the operand with the smaller exponent can be right shifted before performing the addition. If the exponents are equal, the mantissa's are added without shifting.

For an example of the addition operation, suppose you are adding 93 and .07. Operand A (93) is represented in the double precision floating point format as:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000101	0111010000000000000000000000000000000000000000000000

Operand B (.07) is represented as:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	01111111011	0001111010111000010100011110101110000101000111101100

The exponent value for A (93) is 1029. For B (.07), the exponent is 1019. The difference between the exponents is 10. So the mantissa of B will be shifted to the right by 10 bits, then added to the mantissa of A. Also, because both numbers are normalized, meaning their exponents are greater than 0, the leading '1' needs to be included in the addition.

	Mantissa Large	1011101000000000000000000000000000000000000000000000
+	Mantissa Small	0000000000100011110101110000101000111101011100001010 0
=	Result	1011101000100011110101110000101000111101011100001010 0

The resultant mantissa will be the 52 bits following the leading 1 of the result above. The resultant exponent is equal to the exponent of the larger operand, in this case (93), so the final exponent is 1029. So the output of the addition operation of 93 + .07 is the following double precision floating point number, corresponding to decimal number (93.07):

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000101	0111010001000111101011100001010001111010111000010100

What if the exponents of the two operands are equal? Let's use 3.9 + 3.8 as an example. 3.9 is represented as the following double precision floating point number:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000000	1111001100110011001100110011001100110011001100110011001100110011

3.8 is represented as the following double precision floating point number:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000000	1110011001100110011001100110011001100110011001100110011001100110

We need to put the leading '1' in front of each mantissa, and also include an extra '0' bit in front in case the addition overflows. This was not shown in the previous example because 93 + .07 did not overflow to an extra bit. When an overflow occurs, the exponent must be increased by one. Here is how the addition is performed:

	Mantissa Large (3.9)	0111110011001100110011001100110011001100110011001100110011001100110011
+	Mantissa Small (3.8)	0111100110011001100110011001100110011001100110011001100110011001100110
=	Result	1111011001100110011001100110011001100110011001100110011001100110011001

The leftmost '1' in the result becomes the leading '1' of the mantissa, and then the next 52 bits are the actual mantissa. There is a '1' as the 54th bit that gets shifted out of the mantissa. That will be saved for rounding purposes and also will cause the "inexact" signal to be asserted in the exceptions module, but it is not included in the mantissa. And because there is an overflow '1' in the result, the exponent of the larger operand (1024) is increased by one. The result of the addition is the following double precision floating point number, corresponding to the decimal number (7.7):

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000001	1110110011001100110011001100110011001100110011001100

In the source file, fpu_add.v, the module (fpu_add) output the result in floating point format, because the result of the addition operation still needs to go through the rounding stage and then the exceptions stage. So the signals that are passed to the rounding stage are the three components that make up the floating point number: sign, exponent, and mantissa, and these three are named, respectively, as sign, exponent_2, and sum_2. sum_2 is a 56-bit register, with an extra bit at the front of the mantissa for possible overflow in the rounding stage, the leading implied '1', the 52 bits of the mantissa, and 2 extra bits which will determine both how the addition result gets rounded and if the "inexact" signal is asserted in the exceptions module. The 2 extra bits are the result of extending the addition operation to 2 extra bits on the low end of the mantissa. The 2nd extra bit also is the result of an OR on any leftover bits that might have been shifted out of the mantissa as caused by a difference in the exponents of the two operands. These 2 extra bits will come into play in the rounding stage, and they will be explained in more detail in the rounding section of this document.

Subtraction

The subtraction operation is performed in the source file, (fpu_sub.v). The input operands are separated into their mantissa and exponent components, and the larger operand goes into "mantissa_large" and "exponent_large", with the smaller operand populating "mantissa_small" and "exponent_small". Subtraction is similar to addition in that you need to calculate the difference in the exponents between the two operands, and then shift the mantissa of the smaller exponent to the right before subtracting. The definition of the subtraction operation is to take the number in operand B and subtract it from operand A. However, to make the operation easier, the smaller number will be subtracted from the larger number, and if A is the smaller number, it will be subtracted from B and then the sign will be inverted of the result. Take for example the operation 5- 20. You can take the smaller number (5) and subtract it from the larger number (20), and get 15 as the answer. Then invert the sign of the answer to get –15. This is how the subtraction operation is performed in the module fpu_sub.

As an example, consider 45.8 – 45.795. 45.8 is represented in the double precision floating point format as:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000100	0110111001100110011001100110011001100110011001100110

45.795 is represented as:

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	10000000100	0110111001011100001010001111010111000010100011110110

There is no shifting of the smaller number's mantissa because the exponents are equal. So, putting the implied '1' in front of the mantissa, and performing the subtraction is as follows:

	Mantissa Large (45.8)	1011011100110011001100110011001100110011001100110011001100110
-	Mantissa Small (45.795)	1011011100101110000101000111101011100001010001111010110
=	Result	0000000000000101000111101011100001010001111010111000

The result is stored in register "diff", and the signal "diff_shift" counts the number of 0's in "diff" before the leftmost '1'. "diff_shift" in this case is 13, and the exponent of the larger operand is reduced by 13, from 1028 to 1015, and the result is shifted 13 bits to the left to form the mantissa of the answer. The leftmost '1' of "diff" becomes the leading '1' in front of the mantissa. The result of the subtraction operation is the following double precision floating point number :

Sign	Exponent	Mantissa
63	62……..52	51……..0
0	01111110111	0100011110101110000101000111101011100000000000000000

As with the fpu_add module, the fpu_sub module passes on the sign, exponent, and mantissa signals to the rounding module. There are 2 extra remainder bits at the end of the mantissa that determine if rounding will be performed, with the least significant remainder bit calculated by performing an OR on any bits that were shifted out of the mantissa due to the difference in exponents.

Multiply

The multiplication operation is performed in the module (fpu_mul) in the source file, (fpu_mul.v). The mantissa of operand A and the leading '1' (for normalized numbers) are stored in the 53-bit register (mul_a). The mantissa of operand B and the leading '1' (for normalized numbers) are stored in the 53-bit register (mul_b). Multiplying all 53 bits of mul_a by 53 bits of mul_b would result in a 106-bit product. Depending on the synthesis tool used, this might be synthesized in different ways that would not take efficient advantage of the multiplier resources in the target device. 53 bit by 53 bit multipliers are not available in the most popular Xilinx® and Altera® FPGAs, so the multiply would be broken down into smaller multiplies and the results would be added together to give the final 106-bit product. Instead of relying on the synthesis tool to break down the multiply, which might result in a slow and inefficient layout of FPGA resources, the module (fpu_mul) breaks up the multiply into smaller 24-bit by 17-bit multiplies. The Xilinx Virtex5® device contains DSP48E slices with 25 by 18 twos complement multipliers, which can perform a 24-bit by 17-bit unsigned multiply.

The breakdown of the 53-bit by 53-bit floating point multiply into smaller components is described on pages 77-78 of the Xilinx User Guide Document, "Virtex-5 FPGA XtremeDSP Design Considerations", also known as UG193. You can find this document at www.xilinx.com by searching for "UG193". The breakdown of the multiply in module (fpu_mul) is similar to the approach described in this document, though not exactly the same. The multiply is broken up as follows:

product_a = mul_a[23:0] * mul_b[16:0]
product_b = mul_a[23:0] * mul_b[33:17]
product_c = mul_a[23:0] * mul_b[50:34]
product_d = mul_a[23:0] * mul_b[52:51]
product_e = mul_a[40:24] * mul_b[16:0]
product_f = mul_a[40:24] * mul_b[33:17]
product_g = mul_a[40:24] * mul_b[52:34]
product_h = mul_a[52:41] * mul_b[16:0]
product_i = mul_a[52:41] * mul_b[33:17]
product_j = mul_a[52:41] * mul_b[52:34]

The products (a-j) are added together, with the appropriate offsets based on which part of the mul_a and mul_b arrays they are multiplying. For example, product_b is offset by 17 bits from product_a when adding product_a and product_b together. Similar offsets are used for each product (c-j) when adding them together. The summation of the products is accomplished by adding one product result to the previous product result instead of adding all 10 products (a-j) together in one summation. The goal is to take advantage of the adders in the Virtex5 DSP48E slices that follow each 24 by 17 multiply block. If you are not targeting the Virtex5, you will most likely want to modify the way the multiply is broken up into smaller multiply blocks to match your target multiply resources.

The final 106-bit product is stored in register (product). The output will be left-shifted if there is not a '1' in the MSB of product. The number of leading zeros in register (product) is counted by signal (product_shift). The output exponent will also be reduced by (product_shift).

The exponent fields of operands A and B are added together and then the value (1022) is subtracted from the sum of A and B. If the resultant exponent is less than 0, than the (product) register needs to be right-shifted by the amount. This value is stored in register (exponent_under). The final exponent of the output operand will be 0 in this case, and the result will be a denormalized number. If exponent_under is greater than 52, than the mantissa will be shifted out of the product register, and the output will be 0, and the "underflow" signal will be asserted.

The mantissa output from the (fpu_mul) module is in 56-bit register (product_7). The MSB is a leading '0' to allow for a potential overflow in the rounding module. The first bit '0' is followed by the leading '1' for normalized numbers, or '0' for denormalized numbers. Then the 52 bits of the mantissa follow. Two extra bits follow the mantissa, and are used for rounding purposes. The first extra bit is taken from the next bit after the mantissa in the 106-bit product result of the multiply. The second extra bit is an OR of the 52 LSB's of the 106-bit product.

Divide

The divide operation is performed in the module (fpu_div) in the source file, (fpu_div.v). The leading '1' (if normalized) and mantissa of operand A is the dividend, and the leading '1' (if normalized) and mantissa of operand B is the divisor. The divide is executed long hand style, with one bit of the quotient calculated each clock cycle based on a comparison between the dividend register (dividend_reg) and the divisor register (divisor_reg). If the dividend is greater than the divisor, the quotient bit is '1', and then

the divisor is subtracted from the dividend, this difference is shifted one bit to the left, and it becomes the dividend for the next clock cycle. If the dividend is less than the divisor, the dividend is shifted one bit to the left, and then this shifted value becomes the dividend for the next clock cycle.

The exponent for the divide operation is calculated from the exponent fields of operands A and B. The exponent of operand A is added to 1023, and then the exponent of operand B is subtracted from this sum. The result is the exponent value of the output of the divide operation. If the result is less than 0, the quotient will be right shifted by the amount.

The divide operation takes 54 clock cycles to complete, as it takes 1 clock cycle to calculate each of the 54 bits of the quotient. The register (count_out) counts down from 53 to 0, and when it reaches 0, the 54-bit quotient register has its final value. The value that is passed on to the rounding module is stored in the 56-bit register (mantissa_7). The first most significant bit is a '0' to hold a value in case of overflow in the rounding stage, the next bit is the leading '1' for normalized numbers, and the next 52 bits are the mantissa bits. The remaining 2 bits are extra bits for rounding purposes. The first extra bit is the last bit that was calculated in the quotient. The quotient has 54 bits, while the mantissa and leading '1' are only 53 bits, so the extra bit is saved and passed on to the rounding stage. The second extra bit is calculated by performing an OR on all of the remainder bits that were leftover after the last compare between the dividend and divisor registers.

Rounding

The rounding operation is performed in the module (fpu_round) in the source file, (fpu_round.v). The inputs to the (fpu_round) module from the previous stage (addition, subtraction, multiply, or divide) are sign (1 bit), mantissa_term (56 bits), and exponent_term (12 bits). The mantissa_term includes an extra '0' bit as the MSB, and two extra remainder bits as LSB's, and in the middle are the leading '1' and 52 mantissa bits. The exponent_term has an extra '0' bit as the MSB so that an overflow from the highest exponent (2047) will be caught; if there were only 11 bits in the register, a rollover would result in a value of 0 in the exponent field, and the final result of the fpu operation would be incorrect.

There are 4 possible rounding modes. Round to nearest (code = 00), round to zero (code = 01), round to positive infinity (code = 10), and round to negative infinity (code = 11).

For round to nearest mode, if the first extra remainder bit is a '1', and the LSB of the mantissa is a '1', then this will trigger rounding. To perform rounding, the mantissa_term is added to the signal (rounding_amount). The signal rounding_amount has a '1' in the bit space that lines up with the LSB of the 52-bit mantissa field. This '1' in rounding_amount lines up with the 2 bit of the register (mantissa_term); mantissa_term has bits numbered 55 to 0. Bits 1 and 0 of the register (mantissa_term) are the extra remainder bits, and these don't appear in the final mantissa that is output from the top level module, fpu_double.

For round to zero mode, no rounding is performed, unless the output is positive or negative infinity. This is due to how each operation is performed. For multiply and divide, the remainder is left off of the mantissa, and so in essence, the operation is already rounding to zero even before the result of the operation is passed to the rounding module. The same occurs with add and subtract, in that any leftover

bits that form the remainder are left out of the mantissa. If the output is positive or negative infinity, then in round to zero mode, the final output will be the largest positive or negative number, respectively.

For round to positive infinity mode, the two extra remainder bits are checked, and if there is a '1' in either bit, and the sign bit is '0', then the rounding amount will be added to the mantissa_term, and this new amount will be the final mantissa.

Likewise, for round to negative infinity mode, the two extra remainder bits are checked, and if there is a '1' in either bit, and the sign bit is '1', then the rounding amount will be added to the mantissa_term, and this new amount will be the final mantissa.

The output from the fpu_round module is a 64-bit value in the round_out register. This is passed to the exceptions module.

Exceptions

In the exceptions module, all of the special cases are checked for, and if they are found, the appropriate output is created, and the individual output signals of underflow, overflow, inexact, exception, and invalid will be asserted if the conditions for each case exist. The special cases are:

1. divide by 0 – result is infinity, positive or negative, depending on the sign of operand A
2. divide 0 by 0 – result is SNaN, and the invalid signal will be asserted
3. divide infinity by infinity - result is SNaN, and the invalid signal will be asserted
4. divide by infinity – result is 0, positive or negative, depending on the sign of operand A
   the underflow signal will be asserted
5. multiply 0 by infinity - result is SNaN, and the invalid signal will be asserted
6. add, subtract, multiply, or divide overflow – result is infinity, and the overflow signal will be asserted
7. add, subtract, multiply, or divide underflow – result is 0, and the underflow signal will be asserted
8. add positive infinity with negative infinity - result is SNaN, and the invalid signal will be asserted
9. subtract positive infinity from positive infinity - result is SNaN, and the invalid signal will be asserted
10. subtract negative infinity from negative infinity - result is SNaN, and the invalid signal will be asserted
11. one or both inputs are QNaN – output is QNaN
12. one or both inputs are SNaN – output is QNaN, and the invalid signal will be asserted
13. if either of the two remainder bits is '1' – inexact signal is asserted

If any of the above cases occurs, the exception signal will be asserted.

If the output is positive infinity, and the rounding mode is round to zero or round to negative infinity, then the output will be rounded down to the largest positive number (exponent = 2046 and mantissa is all 1's). Likewise, if the output is negative infinity, and the rounding mode is round to zero or round to positive infinity, then the output will be rounded down to the largest negative number. The rounding of infinity occurs in the exceptions module, not in the rounding module.

QNaN is defined as Quiet Not a Number. SNaN is defined as Signaling Not a Number. If either input is a SNaN, then the operation is invalid. The output in that case will be a QNaN. For all other invalid operations, the output will be a SNaN. If either input is a QNaN, the operation will not be performed, and the output will be a QNaN. The output in that case will be the same QNaN as the input QNaN. If both inputs are QNaNs, the output will be the QNaN in operand A. The use of Not a Number is consistent with the IEEE 754 standard.