Hardware Implementation of 64 Bit Floating Point Arithmetic Using VHDL

Vibha Mishra, Vinod Kapse

Abstract*— *Many of the scientific applications rely on floating point (FP) computation, often requiring the use of the 64 bit Floating Point format specified by the IEEE standard 754. The use of double precision (D.P.) data type improves the accuracy and dynamic range of the computation, but simultaneously it increases the complexity and performance of the arithmetical computation of the module. The design of high performance 64-Bit floating point units (FPUs) is thus of interest in this Document.

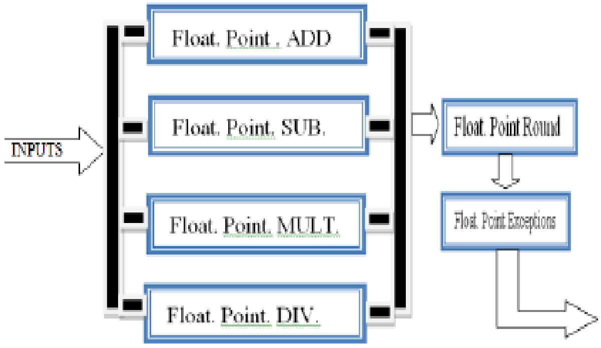
Keywords- IEEE, (D.P.) (FP).

I. INTRODUCTION

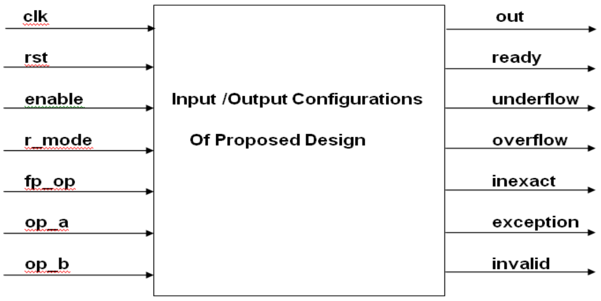
Floating Point Arithmetic are widely used in large set of scientific and signal processing computation. Hardware implementation of floating point arithmetic is more complex than for fixed point numbers, and this puts a performance limit on several of these applications. Several works also focused on their implementation on FPGA platforms.

Floating point arithmetic implementations involve processing separately the sign, exponent and mantissa parts, and then combining them after rounding and normalization. IEEE standard for floating point (IEEE-754) specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented.

Many of the scientific applications described above rely on floating point (FP) computation, often requiring the use of the double precision (D.P.) format specified by the IEEE standard 754. The use of D.P. data type improves the accuracy and dynamic range of the computation, but simultaneously it increases the complexity and performance of the arithmetical computation of the module. The design of high performance floating point units (FPUs) is thus of interest in this domain.



Hierarchy for the proposed Design of 64 bit Floating point arithmetic.



Combinations of Expected Inputs and Outputs for 64 bit Floating point arithmetic.

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

- The input signals to the top level module are the following:
- clk (global)
- rst (global)
- enable (set high to start operation)
- rmode (rounding mode, 2 bits, 00 = nearest, 01 = zero,10 = pos inf, 11 = neg inf)
- fpu_op (operation code, 3 bits, 000 = add, 001 = subtract,010 = multiply, 011 = divide, others are not used)
- opa, opb (input operands, 64 bits) The output signals from the module are the following:
- out_fp (output from operation, 64 bits)
- ready (goes high when output is available)
- underflow
- overflow
- Inexact
- Exception
- Invalid

Floating Point Addition/subtraction Module

Floating Point Addition is one of the complex unit in the floating point arithmetic operations. Addition / subtraction is the most basic arithmetic operation. The hardware implementation of this arithmetic for floating point numbers is a complicated operation due to the requirement of normalization. A proposed implementation method of 64-bit floating point adder/subtractor has been shown here. The flowchart for Floating Point adder/subtractor is shown in Fig.

1. Here the term *addition* is used to refer to both *addition* and *subtraction* as the same hardware is used in both cases. The steps for computing addition of two floating point numbers proceeds as follows,

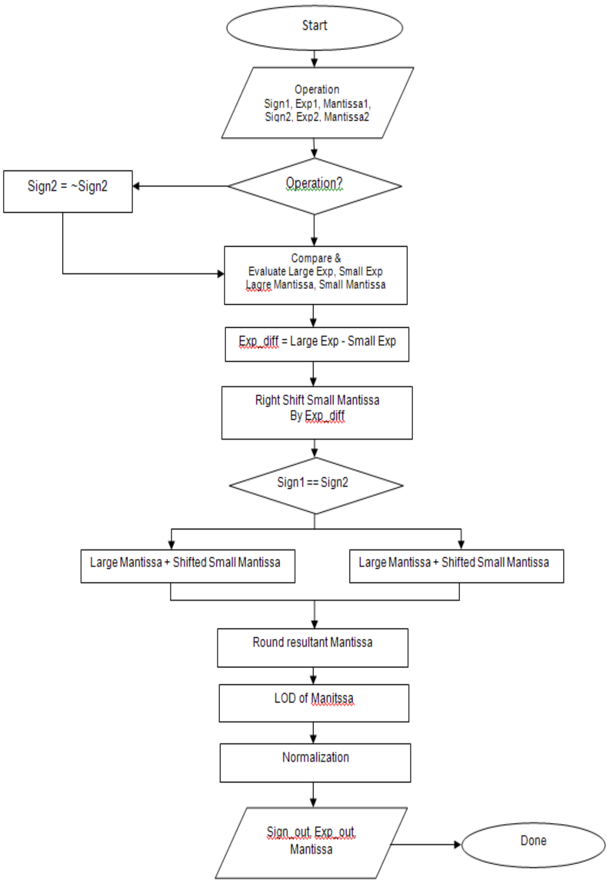
1. Compare exponents and mantissa of both numbers. Decide large exponent & mantissa and small exponent & mantissa.
2. Right shift the mantissa associated with the smaller exponent, by the difference of exponents.
3. Add both mantissa if signs are same else subtract smaller mantissa from large one.
4. Do the rounding of the result after mantissa addition.

Manuscript received February, 2014.

Vibha Mishra, M Tech (Embedded System & VLSI Design) GGITS, Jabalpur, India.

Vinod Kapse, Head (E &Communications) GGITS, Jabalpur, India.

5. If the subtraction results in loss of most significant bit (MSB), then the result must be normalized.
6. Do normalization and adjust large exponent accordingly.
7. Final result includes sign of larger number, normalized exponent and mantissa.

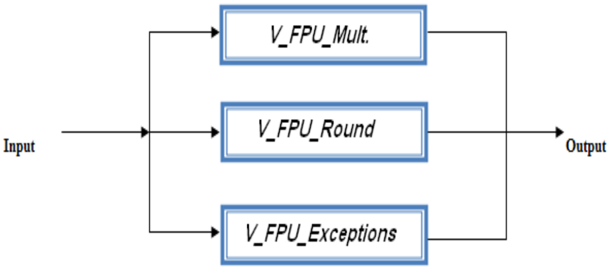


Implementation of Floating Point Multiplier

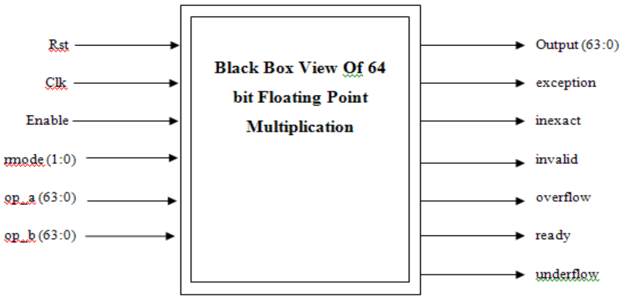
Floating Point Multiplication Algorithm

Multiplying two numbers in floating point format is done as follows.

Flow Chart Of Proposed ADDER/SUBTRACTOR Design



Multiplier structure with rounding and exceptions



Black box view of 64 bit Floating point Multiplier with rounding and exceptions

Multiplier Operation

The multiplication operation is performed in the module (V_Fpu_Mul).

1. The mantissa of operand A and the leading _1‘ (for normalized numbers) are stored in the 53-bit register (mul_a).
2. The mantissa of operand B and the leading _1‘ (for normalized numbers) are stored in the 53-bit register (mul_b).
3. Multiplying all 53 bits of mul_a by 53 bits of mul_b would result in a 106-bit product.
4. 53 bit by 53 bit multipliers are not available in the most popular 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.

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]

7. 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. Similar offsets are used for each product (c-j) when adding them together.
8. 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.
9. The final 106-bit product is stored in register (product).
10. 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).
11. 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.
12. 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.
13. 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.
14. 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.
15. 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 fromthe 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.

1 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.
2 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).
3 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.
4 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.
5 The exponent for the divide operation is calculated from the exponent fields of operands A and B.
6 The exponent of operand A is added to 1023, and then the exponent of operand B is subtracted from this sum.
7 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.
8 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.
9 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).
10 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.
11 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 extrabit is saved and passed on to the rounding stage.
12 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 and Exceptions

- The IEEE standard specifies four rounding modes
- Round to nearest, Round to zero,
- Round to positive infinity, and
- Round to negative infinity.

The rounding operation is performed in the module (fpu_round) in the source file, (fpu_round.vhd). 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.

The below table shows the rounding modes selected for various bit combinations of rmode.

S. No.	**Bit**	**Rounding Mode**
1	00	round_nearest_even
2	01	round_to_zero
3	10	round_up
4	11	round_down

Rounding modes for various bit combinations of rmode

1. 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 levelmodule, fpu_double.
2. 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.
3. 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.
4. Likewise, for round to negative infinity mode, the two extra remainder bits are checked, and if there is a1‘ 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.

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.

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. multiply 0 by infinity - result is SNaN, and the invalid signal will be asserted
5. add, subtract, multiply, or divide overflow – result is infinity, and the overflow signal will be asserted
6. add, subtract, multiply, or divide underflow – result is 0, and the underflow signal will be asserted
7. add positive infinity with negative infinity - result is SNaN, and the invalid signal will be asserted

8. subtract positive infinity from positive infinity - result is SNaN, and the invalid signal will be asserted
9. subtract negative infinity from negative infinity - result is SNaN, and the invalid signal will be asserted
10. divide by infinity – result is 0, positive or negative, depending on the sign of operand A the underflow 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 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.aN 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.

II. RESULTS

The 64 Bit floating point adder/subtractor, multiplier and divisor designs were simulated and synthesized in Altera‘s Quartus-II 11. which are mapped on to Cyclone IV GX FPGA. The simulation results of 64-bit floating point double precision adder /subtractor multiplier and divisor are shown in Figure and respectively. The ‗opa‘ and ‗opb‘ are the inputs and ‗out‘ is the output. Table 1 gives the device utilization for implementing the circuits on Cyclone IV GX FPGA. Table 2 shows the timing summary of 64 Bit floating point adder/subtractor multiplier and divisor.

In case of Xilinx core, it occupies an area of 1266 slices and its operating frequency 284 MHz respectively. Hence the present design provides high operating frequency.

Table 1. Device utilization summary of 64 Bit floating point adder/subtractor, multiplier and divisor.

S.No.	Slice Logic Utilization	Adder	Subtractor	Multiplier	Divisor
1	Minimum Period (ns)	2.749	2.64	2.411	2.46
2	Maximum Frequency (MHz)	363.76	354.76	414.714	394.714

Table2 Timing summary of 64 Bit floating point adder/ subtractor ,multiplier and divisor.

S.No.	Slice Logic Utilization	Adder	Subtractor	Multiplier	Divisor
1	Number of logic Registers	672	689	1741	994
2	Number of Combinational logics	965	888	6,860	1,633

CONCLUSION

The double precision floating point adder/subtractor, multiplier and divisor supports the IEEE-754 binary interchange format, targeted on a Altera‘s Cyclone IV GX EP4CGX30CF23C6 FPGA. The designs achieved the operating frequencies of 363.76 MHz and 414.714 MHz. FLOPs with an area of 660 and 648 slices respectively. The adder/subtractor design operates at a frequency which is 3% and 28% more compared to [6] and Xilinx core respectively. As compared to the single precision floating point multiplier [12] and Xilinx core, the multiplier design supports double precision, provides high speed and gives more accuracy. These designs handles the overflow, underflow, rounding mode and various exception conditions.

REFERENCES

[1] P. Belanovic and M. Leeser, ―A Library of Parameterized Floating-Point Modules and Their Use‖, in 12th International Conference on Field-Programmable Logic and Applications (FPL- 02). London, UK: Springer-Verlag, (2002) September, pp. 657–666.

[2] K. Hemmert and K. Underwood, ―Open Source High Performance Floating-Point Modules‖, in 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM-06), (2006) April, pp. 349–350.

[3] A. Malik and S. -B. Ko, ―A Study on the Floating-Point Adder in FPGAs‖, in Canadian Conference on Electrical and Computer Engineering (CCECE-06), (2006) May, pp. 86–89.

[4] D. Sangwan and M. K. Yadav, ―Design and Implementation of Adder/Subtractor and Multiplication Units for Floating-Point Arithmetic‖, in International Journal of Electronics Engineering, (2010), pp. 197-203.

[5] M. K. Jaiswal and R. C. C. Cheung, ―High Performance FPGA Implementation of Double Precision Floating Point Adder/Subtractor‖, in International Journal of Hybrid Information Technology, vol. 4, no. 4, (2011) October.

[6] B. Lee and N. Burgess, ―Parameterisable Floating-point Operations on FPGA‖, Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, (2002).

[7] M. Al-Ashrafy, A. Salem, W. Anis, ―An Efficient Implementation of Floating Point Multiplier‖, Saudi International Electronics, Communications and Photonics Conference (SIECPC), (2011) April 24-26, pp. 1-5.