Floating Point Optimization Using VHDL

Dr. Manal Hammadi Jassim*

Received on: 29/3/2009
Accepted on: 2/7/2009

Abstract

Due to inherent limitations of the fixed-point representation, it is sometimes desirable to perform arithmetic operations in the floating-point format. Although an established standard for floating-point arithmetic exists, optimal hardware implementations of algorithms require use of floating-point formats different from the ones specified in the standard. Hardware modules for floating-point format control, arithmetic operators and conversion to and from any fixed-point format are presented. Synthesis results for arithmetic operator modules in several floating-point formats, including the IEEE single precision format, Synthesis and processing results for both implementations are shown and compared.

تمثيل النقطه الطافيه باستخدام برمجه التوصيف العملي للدوائر الرقميه

الخلاصة

بسب محدودات استخدام حسابات النقطة الثابتة تستدعي الضرورة في بعض الأحيان أستخدام حسابات النقطه الطافيه. وبالرغم من وجود هيئه قياسية لتمثيل حسابات النقطة الطافية لكن عند أنجاز التمثيل العملي المثالي للوغاريتمات بعض الدوائر تستدعي الضرورة إلى استخدام هيئات تمثيل للأرقام الطافية غير تلك الهيئات القياسية المتواجدة والمتعارف عليها .في هذا العمل تم أنجاز موديلات عملية للسيطرة بالنقطة الطافية, وللمعاملات الحسابية , وللتحويل من والى أي تمثيل للنقطة الثابتة. وفي هذا العمل نتائج التحليل لموديلات المعاملات الحسابية لهيئات قياسية مختلفة متضمنا القياسات العلمية الدقيقة IEEE . نتائج التحليل والتركيب العملي لكلا الانجازين متواجدة في هذا العمل ومع المقارنة فيما بينهما .

Introduction

The hardware modules that constitute the parameterized library for floating-point arithmetic rely on the existence of some basic logic functions, such as multiplexers and fixed point adders. These basic building blocks that perform functions commonly encountered in logic design all share one characteristic - parameterization. All of them have parameterized functionality, so that they can be used to build higher level modules that are themselves parameterized. One of the most widely implemented formats for representing and storing numerical values in the

binary number system is the fixed-point format. Every numerical value is composed of an integer part and a

*Electrical and Electronic Engineering Department, University of Technology/ Baghdad

https://doi.org/10.30684/etj.27.16.11
2412-0758/University of Technology-Iraq, Baghdad, Iraq

fractional part and the delimitation between the two is referred to as the radix point. In the fixed-point format, the radix point is always implied to be in a predetermined position. The format thus gets its name from the fixed location of the radix point. Usually, the radix point is assumed to be located immediately to the right of the least significant digit, in which case only integer values are represented.

1-Related Work

One of the earliest investigations into using FPGAs to implement floating-point arithmetic was done by Fagin et al. [1], who in 1994 showed that implementing IEEE single precision operators was possible, but also impractical on then current FPGA technology. The circuits designed by the authors were an adder and a multiplier and both had full implementation of all four rounding modes specified by the IEEE 754 standard. This line of thought was expanded on by the significant work of Shirazi et al. [2] who suggested application-specific formats for image and DSP algorithms in widths of 16 (1-6-9) and 18 (1-7-10) bits, as opposed to the full 32 (1-8-23) bits in the IEEE standard. Another significant work came from Louca et al.[3] in which the authors, building on the work of Shirazi and others, abstract the normalization operation away from the actual arithmetic operators, in an effort to conserve area. In an effort to expand the capabilities of existing architectures, Ligon et al. [4] presented IEEE single precision adder and multiplier circuits on the then newly available Xilinx 4000 series FPGAs. Both circuits supported rounding to nearest, but did not use a separate normalizing unit. Similar work by Stamoulis et al. [5] presented IEEE single precision adder/subtractor, multiplier and division circuits. Two works by Sahin et al. [6] [7] present adder, subtractor, multiplier and accumulator circuits, but again only in IEEE single precision format. Also, rounding capability is not implemented. Recent work by Dido et al.[8] discusses optimization of datapaths, especially in image and video processing applications. This datapath optimization is achieved

by providing flexible floating-point formats that are optimal for every signal in the datapath. The floating-point formats in our work are a generalized superset of all these formats. It includes all the IEEE formats as particular instances of exponent and mantissa bitwidths, as well as the flexible floating-point format presented by Dido et al.[8] and the two formats by Shirazi et al.[2]. Also, we abstract normalization as well as rounding functionality into a separate unit with a choice of rounding to zero and rounding to nearest. In this way, as Dido et al. [8] explained

2-Floating-Point Format

The floating-point format is the most common way of representing real numbers in computer systems. The floating-point format is similar to the scientific notation of numbers, such as (-1.35×10^6) there are three fields in the representation of a value in the floating-point format: sign *s*, exponent *e* and fraction *f*. Thus, every floating-point value can be defined by

$$(-1)^s \times 1.f \times 2^{e-BIAS}$$

Depending on the bit width of the exponent field in the particular format, given that the exponent bit width is *exp_bits*, we can represent exponent values from $-2^{exp\_bit-1}+1$ to $2^{exp\_bits-1}-1$ by assigning the value $2^{exp\_bit-1}-1$ to the bias. It is worth noting that the bias value is not constant and changes with exponent bitwidth. The bitwidth alignment of the three fields is shown in Figure-1. The distinction between terms *fraction* and *mantissa* is that fraction represents only the portion of the mantissa to the right of the radix point (fractional part). Naturally, a tradeoff exists in total bitwidth between smaller width requiring less hardware resources and higher width providing better precision. Also, within a certain total bitwidth, it is possible to assign various combinations of values to the exponent and fraction fields (Figure-1), wider exponent fields brings higher range and wider fraction fields brings higher precision[9].

2-IEEE Standard and Other Formats

The Institute of Electrical and Electronics Engineers (IEEE) issued standard 754 in 1985, specifying details of implementing binary floating-point arithmetic. This standard details four floating-point formats - basic and extended each in single and double precision bitwidths. Most implementations of floating-point arithmetic adhere to one or more of these standard formats, though few follow the standard absolutely. However, optimal implementations of algorithms may not always require bitwidths as specified by the standard. In fact, it is often the case that much smaller bitwidths than those specified in the 754 standard are sufficient to provide desired precision and occupy less resource than the full standard bitwidth implementation [9].

3- Reconfigurable Hardware Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays (FPGAs) are integrated circuits with a flexible architecture, such that their structure can be programmed by the designer. FPGAs are composed of an array of hardware resources called configurable logic blocks (CLBs). The designer creates the functionality of the overall circuit by configuring CLBs to perform appropriate logic functions. Hence, FPGAs are a form of reconfigurable hardware, combining flexibility similar to software with the speed of specialized hardware. Designs mapped to FPGAs are usually described in hardware description languages, such as VHDL or Verilog. In this work, all hardware descriptions are written in VHDL. A number of software tools exists to aid the designer in mapping the high-level description of the design in VHDL to the logic level of each CLB. Such tools perform synthesis, mapping, placing and routing of the design to the hardware [1, 8].

4-Arithmetic Operator Hardware Modules

4-1Addition

Addition is one of the most computationally complex operations in floating-point arithmetic. The algorithm of the addition operation

for floating-point numbers is composed of four steps:
- ensure that the operand with larger magnitude is on input 1 (swap),
-align the mantissas (shift_adjust),
-add or subtract the mantissas (add_sub), and
-shift the result mantissa right by one bit and increment the exponent if addition overflow occurred.

Each of the four steps of the algorithm is implemented in a dedicated module, shown above in brackets. The four sub-modules are assembled into the overall fp_add module as shown in Figure- 2. The swap submodule compares the exponent and mantissa fields of the input variables. Based on these two comparisons, the two floating-point inputs are multiplexed to the outputs. If the exponent field of input A is larger, or the exponent fields are equal and the mantissa of input A is larger, input A is multiplexed to output large and input B to output small. Otherwise, the reverse mapping of inputs to outputs occurs.

Submodule shift_adjust is responsible for aligning the mantissas of the larger and smaller operands. It achieves this by shifting the smaller mantissa to the right as many bits as is the difference between the exponents. Another function of this module is to introduce the guard bit into the smaller operand's mantissa. Guard bits are introduced in the addition algorithm to provide rounding to nearest for the result. Hence, the mantissa of the sum is one bit wider than the mantissas of the inputs. Expansion of the mantissa fields happens during aligning of the mantissas, so that the extra information the guard bit carries can be saved when the smaller operand's mantissa is shifted right and some least significant bits may be lost. The guard bit is introduced into the larger operand's mantissa to the right of the least significant bit and always has value '0'. Once the mantissas are aligned, it is necessary to either add or subtract them, depending on the signs of the two operands. If the signs are the same, the addition operation is constructive and the mantissas are added. If the signs are opposite, however, the addition operation is destructive and the mantissas are subtracted. Sub-module add sub will perform this variable operation under the control of the op input, which is fed with the XOR of the input sign bits.

Outputs of the overall addition algorithm are controlled by the correction module. If an exception is indicated on the input, the exception is propagated to the output and the result output is set to all zeros. Otherwise, if the input values are detected to be of the same magnitude, but opposite sign, an exception is not generated on the output, but the result output is still blanked out, to indicate zero value, as $A + (-A) = 0$. Otherwise, if an overflow in the addition of the mantissas was detected, the result mantissa is shifted to the right by one bit, truncating the least significant bit, and the most significant bit is filled with '1'. Also, the exponent field is incremented by 1, to reflect the shift in the mantissa. These operations correct for the overflow in the addition of the mantissas. Finally, the floating-point value assembled from the sign, exponent and mantissa fields is presented on the output. Module fp_add is parameterized by the width of the exponent and mantissa fields of the floating-point format it operates on [6, 7, 10, and 11].

4-2Subtraction

The subtraction operation is similar to the addition operation, as

A - B = A + (-B)

Thus, we use a slightly modified addition module to perform subtraction. This is especially helped by the sign-magnitude form of the floating-point format. To invert a floating-point value, all that needs to be done is to invert the sign bit (most significant bit, MSB, of the floating-point signal).

There is only one, minor structural difference between the addition and subtraction modules: the inverter on the MSB of the second operand is not used on the input to the **parameterized_ comparator** module (Figure-2), but on the input to the swap module. That way, we invert the sign of the second operand to achieve subtraction. Also, the comparator now monitors input values equal in both sign and magnitude, since A - A = 0. Because the inverter is only moved from one location to another, module fp sub occupies the exact same area as the fp_add module. Similarly to the fp_add module, the fp_sub module is also parameterized by the width of the exponent and mantissa fields of the floating-point format it operates on [5, 6, 11, and 12].

4-3Multiplication

Unlike fixed-point arithmetic, in floating-point arithmetic, multiplication is a relatively straight-forward operation compared to addition. This is again due to the sign-magnitude nature of the floating-point format, because

$$((-1)^{s1} \times m_1 \times 2^{e1}) \times ((-1)^{s2} \times m_2 \times 2^{e2}) = (-1)^{s1 \oplus s2} \times (m_1 \times m_2) \times 2^{(e1+e2)}$$

From the above, it can be concluded that the three fields of the floating-point format do not interact during multiplication and can thus be processed at the same time, in parallel. The sign of the product is given as the exclusive OR (XOR) of the input value signs. Mantissa of the product is calculated by fixed-point multiplication of the input value mantissas, while the exponents of the input values are added to give the exponent of the product. The only further complication of the floating-point multiplication algorithm is the fact that the exponent fields are biased. When two biased exponent fields are added, the result contains the bias twice, one of which must be subtracted. If, using IEEE standard 754 notation, *E* is an unbiased exponent and *e* is a biased exponent, it stands that:

$$e_1 + e_2 = (E_1 + BIAS) + (E_2 + BIAS) = (E_1 + E_2) + 2 \times BIAS = E_p + 2 \times BIAS = e_P + BIAS$$

The structure of the floating-point multiplier is given in Figure-3. The fp_mul module is parameterized by the bitwidths of the exponent and mantissa fields of the floating-point format it processes. The bitwidth of the product is 1 + *exp_bits* + (2 X *man_bits*) . The mantissa field has twice the bitwidth of the input mantissas because it is their fixed-point product [5, 7, 11, and 12].

4-4Format Conversion Hardware Modules

Custom hardware architectures have the ability to perform some sections of the algorithm in fixed-point arithmetic and others in floating-point arithmetic, depending on the optimal representation of each variable in the algorithm. It is the goal of our library to provide all the hardware modules the designer needs to build such hybrid fixed and floating-point architectures. Hence, some of the most important modules are those that convert between fixed and floating-point representations of variables [7, 11, and 12].

4-5Conversion from Fixed-Point To Floating-Point

Module fix2float was designed to convert a given value from fixed to floating-point representation. Thus, its input is a fixed-point value and its output is the corresponding floating-point representation. Since fixed-point values can be in the unsigned or signed (two's complement) form, two versions of the fix2float module have been developed, The structure of the unsigned version is shown in Figure-4, while the structure of the signed version is shown in Figure-5. The signed version is more complex due to handling of the two's complement representations of the input and hence has a longer latency of 5 clock cycles, as opposed to 4 clock cycles for the unsigned version In the conversion from signed fixed-point numbers, it may be necessary to derive the two's complement of the input signal, while in the case of unsigned fixed-point numbers, no operation is necessary, as only non-negative values can be represented. This added operation results in the difference in latencies of the signed and unsigned module versions. The mantissa of the final result is produced by shifting left the absolute value of the input until its MSB is '1', while the exponent is derived from format constants and the number of shifts made to the mantissa. For example:

$\mathbf{01001011_2 = 75 = 01001011_2 \times 2^0 = 10010110_2 \times 2^{-1} = 1.0010110_2 \times 2^{7-1} = 1.0010110_2 \times 2^6 = 1.171875 \times 64 = 75 \rightarrow \boldsymbol{f} = 0010110_2 \rightarrow \boldsymbol{e} = 6+\boldsymbol{BIAS}}$

The value of the exponent field depends on the normalizing shift of the mantissa, *shift*, the bitwidth of the fixed-point input, *fix_ bits*, and the bias value, *BIAS*. Its final form is

$\boldsymbol{e = E + BIAS = ((fix\_bits - 1) - shift) + BIAS = (fix\_bit + BIAS - 1) - shift}$

The absolute value of the input is fed into a priority encoder, to determine the *shift* value. This constitutes the first clock cycle of the unsigned architecture and the second cycle of the signed one. Once the value of the normalizing shift is known, the exponent field is calculated by performing the subtraction (*fix_ bits* + *BIAS* - 1) - *shift*. The value of the signal const in Figures-4 and-5 is *fix_ bits* + *BIAS* - 1. Once the value of the normalizing shift is known, the mantissa is produced by shifting left the absolute value of the input and the exponent is calculated through subtraction. These operations happen in parallel, in the second clock cycle of the unsigned architecture and third of the signed one. After the shifting operation, the width of the mantissa field is that of the fixed-point input and may need to be reduced to the width specified by the floating-point format that is to appear on the output.

This reduction in bitwidth calls for rounding. Rounding to zero or nearest are both available through input round and happen in clock cycle three in the unsigned architecture and four in the signed one. The final clock cycle of both architectures is dedicated to determining the outputs of the circuit. The floating-point output is either the calculated value or all zeros. The latter option is multiplexed to the output in case of an exception being received at the input or encountered during processing, or a zero fixed-point input, which requires an all-zero floating-point output. Otherwise, the floating-point value calculated by the module is presented on the output. Both versions of the fix2float module are parameterized by three values: the width of the fixed-point input, the width of the exponent field and the width of the mantissa field of the floating-point output [6, 7, 11, and 12].

4-6Conversion from Floating-Point To Fixed-Point

Module float2fix implements the inverse function to that of the fix2float module: conversion from the floating-point representation of a value to its fixed-point representation. As before, two versions of the float2fix module exist: one for converting to signed and the other to unsigned fixed-point representation of the input floating-point value. The structure of the hardware for conversion to the unsigned fixed-point representation is shown in Figure-6, while Figure-7 shows the signed version. Due to the added complexity of handling two's complement representations of the output value, the signed version has a latency of 5 clock cycles, while the unsigned version has a latency of 4 clock cycles. The functioning of the float2fix module can easily produce exceptions because, in general, floating-point formats have a wider range than fixed-point formats.

For instance, all floating-point values that have negative exponents (magnitude less than 1) cannot be represented in integer fixed-point formats by values other than 0 or 1. Also, all floating-point values that exceed the largest representable value in the target fixed-point format produce an exception. In the unsigned version, another exception is caused by negative floating-point values appearing on the input, which instance, all floating-point values that have negative exponents (magnitude less than 1) cannot be represented in integer fixed-point formats by values other than 0 or 1.

Also, all floating-point values that exceed the largest representable value in the target fixed-point format produce an exception. In the unsigned version, another exception is caused by negative floating-point values appearing on the input, which can by definition not be represented in unsigned fixed-point format. These exceptions are trapped in the first clock cycle of both the signed and the unsigned architecture. Also in this clock cycle, the shift required to produce the fixed-point output from the mantissa value is calculated using the exponent field. This shift is simply the unbiased value of the exponent. For example:

$$\mathbf{1.01011_2 \times 2^6 = 1.34375 \times 64 = 86 = 1010110_2 \times 2^0 = 86}$$

The shift required is calculated by subtracting the bias value from the exponent field of the input. In the second clock cycle of both versions of the float2fix module, the absolute value of the fixed-point representation is obtained by shifting left the mantissa field of the input. In parallel with this, the exception signals obtained in the first clock cycle are combined into one exception signal. Because the fixed-point format on the output may specify a smaller bitwidth than the mantissa field of the input floating-point format, some least significant bits of the absolute value of the fixed-point representation, obtained by shifting the mantissa field, may need to be truncated. This truncation calls for rounding functionality, implemented in the third clock cycle of both the signed and the unsigned architecture. The prepared absolute value of the fixed-point representation, rounded to the required bitwidth, is ready for output in the unsigned version, while in the signed version, it may need to undergo a two's complement operation before being placed on the output. It is because of this extra step that the signed version of the module has the longer latency of 5 clock cycles. The two's complement of the absolute value is found by inverting all the bits and adding 1. The sign bit of the input floating-point value is used to select the correct form (positive or two's complement) of the fixed-point value, before it is passed to the next stage. The final stage of both the signed and the unsigned architectures is the output stage, where the computed fixed-point representation is placed on the output, unless the input was zero or an exception was encountered during operation or received at the input, in which case the output is set to all zeros. Module float2fix is parameterized by the bitwidths of the exponent and mantissa fields of the input floating-point signal, as well as the bitwidth of the fixed-point output [5, 7, 11, and 12].

5-Testing

All the hardware modules tested both in simulation and in hardware. The purpose of the two testing stages was to ensure the correct operation of the VHDL description of each module. A set of input vectors was developed for each module to test its operation with a range of inputs. Parameterization of each module was also tested to ensure correct operation of the module at various instances in the design space. The simulator used to test the VHDL descriptions was Xilinx ISE 9.2i . Iteration between modification of the VHDL description and analysis in the simulator continued until the correct operation of the module was achieved for all the test vectors. An example of a test vector used to test the IEEE single precision adder circuit given below.

$41BA3C57_{16}$=010000011011101000111100010101112=+1.45496642 x2^4 =23.27946281

$4349C776_{16}$=01000011010010011100011101110110$_2$=+1.57639956 x2^7=201.77914429

Sum=225.05860710=+1.75827036x2^7 =01000011011000010000111100000001$_2$= 43610F01$_{16}$

6-Results

The results of synthesis experiments conducted on the floating-point operator modules *fp_add*, *fp_ sub* and *fp_mul presents, the aims of the experiments are to:*
-determine the area of the above modules in several floating-point formats,
-examine the relationship between the area and total bitwidth of the format, and
-estimate the number of modules that can realistically be used on a single FPGA.
The experiments were conducted by synthesizing the modules for specific floating-point formats Table- 2 shows results of the synthesis experiments on floating-point operator modules. The quantities for the area of each instance are expressed in Xilinx XCV1000 FPGA. Results for the fp_add module in Table-2 also represent the fp_sub module, which has the same amount of logic. Floating-point formats used in the experiments were chosen to represent the range of realistic floating-point formats from 8 to 32 bits in total bitwidth and include the IEEE single precision format E1 in Table-2. The number of operator cores per processing element, shown in the two rightmost columns, is based on a Xilinx XCV1000 FPGA, with a total of 12288 slices, with 85% area utilization. A realistic design cannot utilize all the resources on the FPGA because of routing overhead; a practical maximum is estimated at about 85%. Also included is an overhead allowance of approximately 1200 slices for necessary circuitry other than the operators themselves.

The results in Table-2 show growth in area with increasing total bitwidth, for all modules. This growth is represented graphically in Figure-8.

7-Conclusions

The library of parameterized hardware modules for floating point arithmetic has been created and modules for format control, arithmetic operators and conversion to and from any fixed point format. All the modules are parameterized to operate on any floating point format, with rounding to zero or nearest. Limited exception handling is implemented in all the modules, with the ability to propagate an error. Ready and done synchronization signals are provided in all modules. The library can be used to implement finely tuned datapaths, in both fixed and floating point arithmetic, to the exact bitwidths, ranges and precisions required by the signals in the algorithm. Also, library modules for format conversion enable creation of hybrid fixed and floating point design.

Synthesis results indicate design on a Xilinx XCV1000 FPGA may include up to 31 addition or 13 multiplication operators, complete with demoralizing, rounding and normalizing functionality each, for the IEEE single precision format. Similarly, a useful custom floating point format, with 5 exponent and 6 mantissa bits for example, may provide the designer with up to 113 addition or 85 multiplication modules, all also complete with full format handling functionalities, on the same FPGA

8-Future Work

Many of the approaches used by the synthesis framework presented in this work make use of simple methods and algorithms. However, this first version of the synthesis framework is dedicated to present the basic methodology for a new conceptual synthesis methodology. Optimizations of these methods are possible but have been left for future work.

9-Bibliography

[1] B. Fagin and C. Renard. Field Programmable Gate Arrays and Floating Point Arithmetic. *IEEE Transactions on VLSI Systems*, 2(3), September 1994.

[2] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, California, April 1995

[3] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998

[4] L. Louca, T. A. Cook, and W. H. Johnson. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In K. L. Pocek and J. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 107–116, April 1996.

[5] I. Stamoulis, M. White, and P. F. Lister. Pipelined Floating-Point Arithmetic Optimized for FPGA Architectures. In *9th International Workshop on Field Programmable Logic and Applications*, volume 1673 of *LNCS*, pages 365–370, August-September 1999.

[6] I. Sahin, C. S. Gloster, and C. Doss. Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems. In *2000 MAPLD International Conference*, 2000

[7] I. Sahin and C. S. Gloster. Floating-Point Modules Targeted for Use with RC Compilation Tools. http://www4.ncsu.edu:8030/~isahin/papers/DACPaper.pdf, (last visited January 2002), 2001

[8] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs. In *International Symposium on Field-Programmable Gate Arrays*, pages 50–55. ACM, ACM Press, February 2002.

[9]IEEE Standards Board and ANSI. IEEE Standard for Binary Floating point Arithmetic, 10985. IEEE Std 754-1985

[10]J. P. Hayes. Computer Architecture and Organization. McGraw Hill, Second edition, 1988.

[11]Annapolis Micro System INC. Floating Point Math Library. Technical Data Sheet Doc # 122763-0000 Rev 1.7.

[12]Z. Luo and M. Martonosi. Accelerating Pipelined Integer and floating point Accumulations in Configurable Hardware with Delayed Addition Techniques. In IEEE Transactions on Computer, volume 49, page 208-218, March 2000.

Table (1)

operand 1	Operand 2	Sum
41BA3C57	4349C776	43610F01

Table (2) Operator synthesis results

Format		Bitwidth		Area		Per IC	
	total	exponent	fraction	fp_add	fp_mul	fp_add	fp_mul
A0	8	2	5	39	46	236	200
A1	8	3	4	39	51	236	180
A2	8	4	3	32	36	288	256
B0	12	3	8	84	127	109	72
B1	12	4	7	80	140	115	65
B2	12	5	6	81	108	113	85
C0	16	4	11	121	208	76	44
C1	16	5	10	141	178	65	51
C2	16	6	9	113	150	81	61
D0	24	6	17	221	421	41	21
D1	24	8	15	216	431	42	21
D2	24	10	13	217	275	42	33
E0	32	5	26	328	766	28	12
E1	32	8	23	291	674	31	13
E2	32	11	20	284	536	32	17

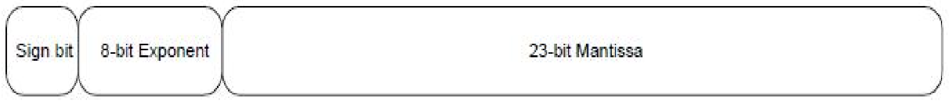

Figure (1) Alignment of fields in a floating-point format

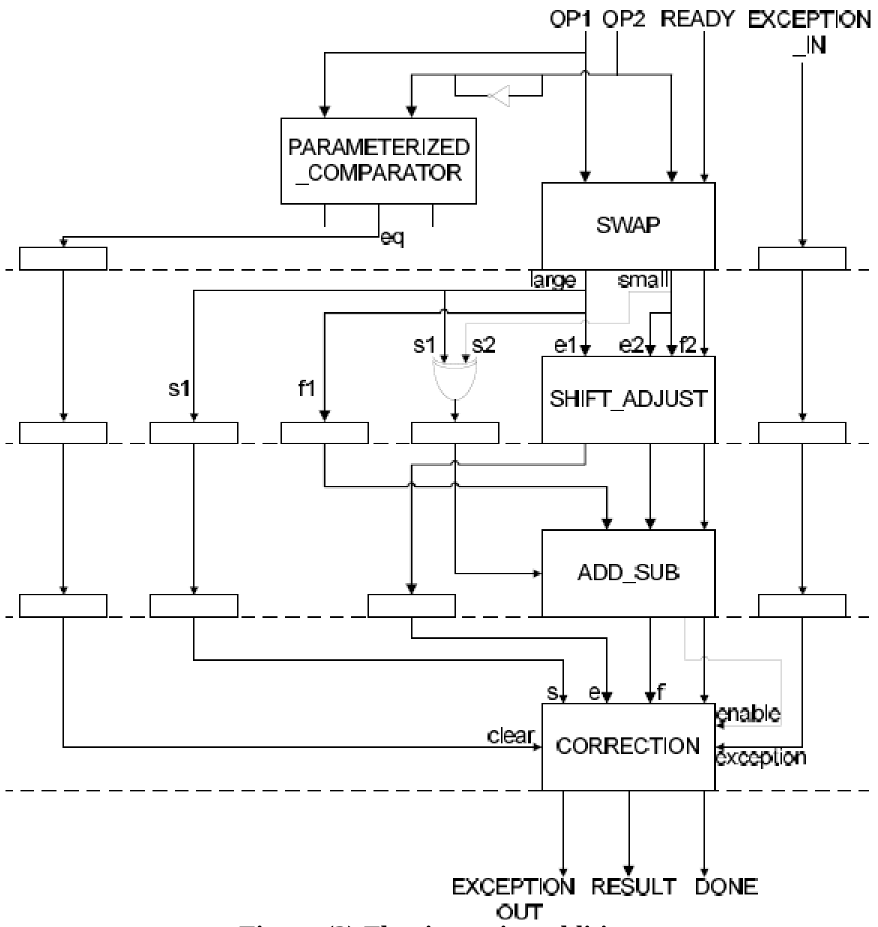

Figure (2) Floating-point addition

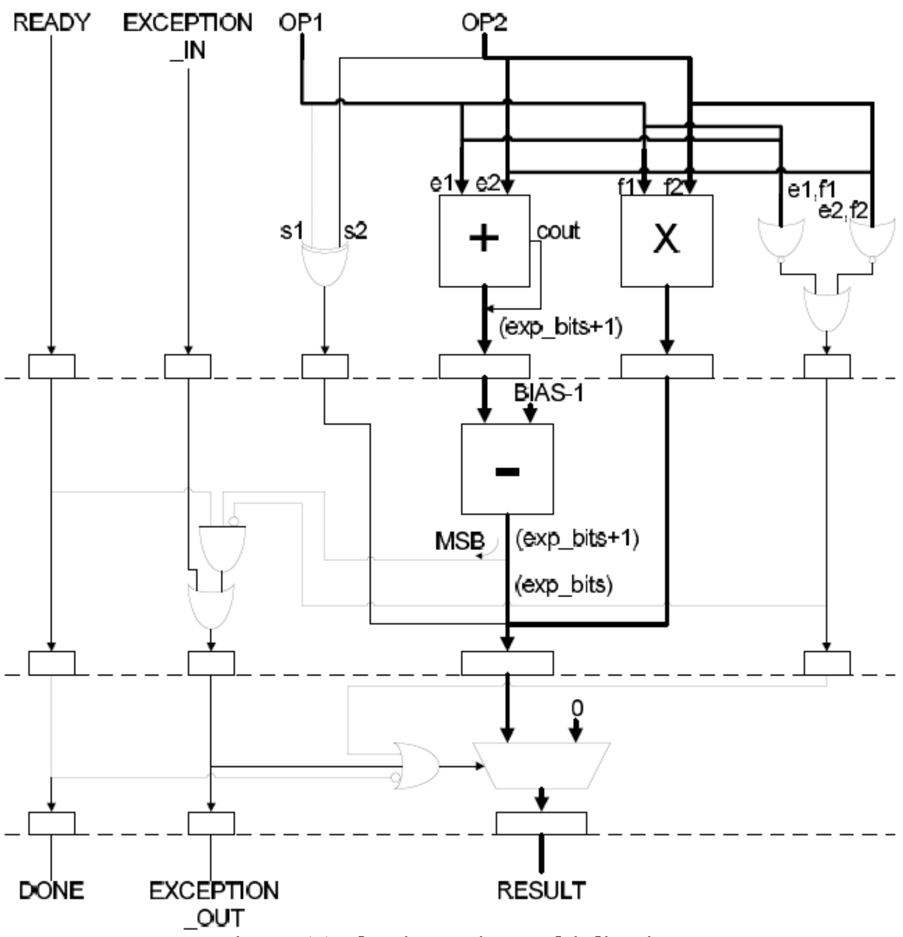

Figure (3) Floating-point multiplication

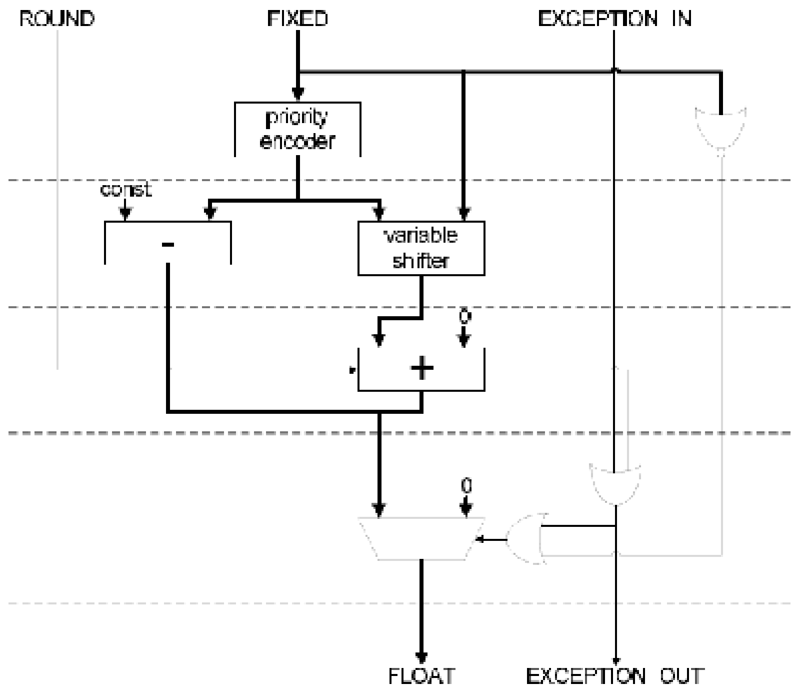

Figure (4) Conversion from unsigned fixed-point to floating-point representation

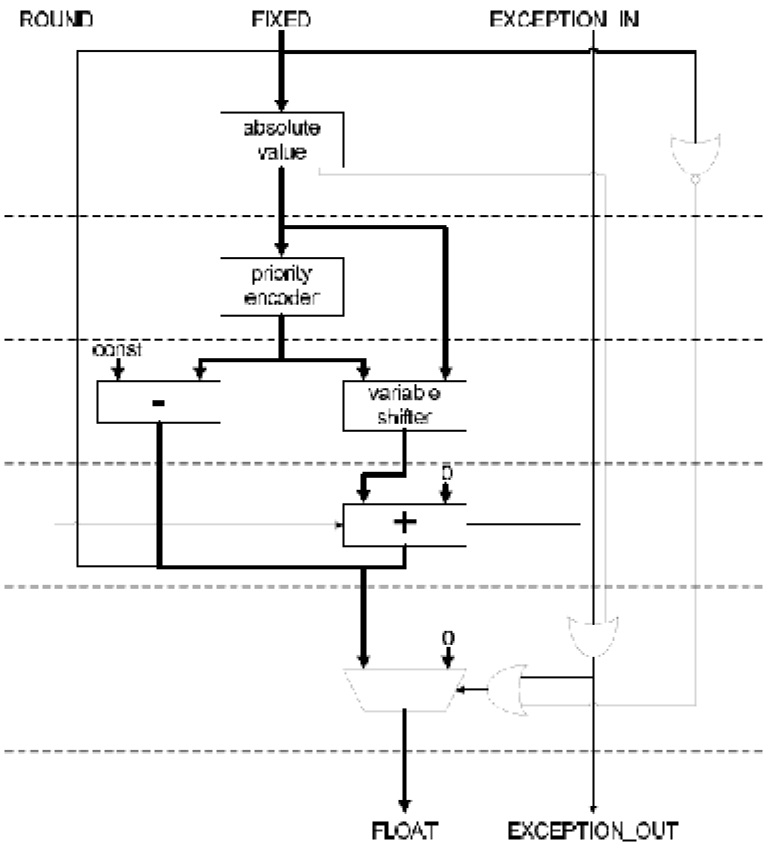

Figure (5) Conversion from signed fixed-point to floating-point representation

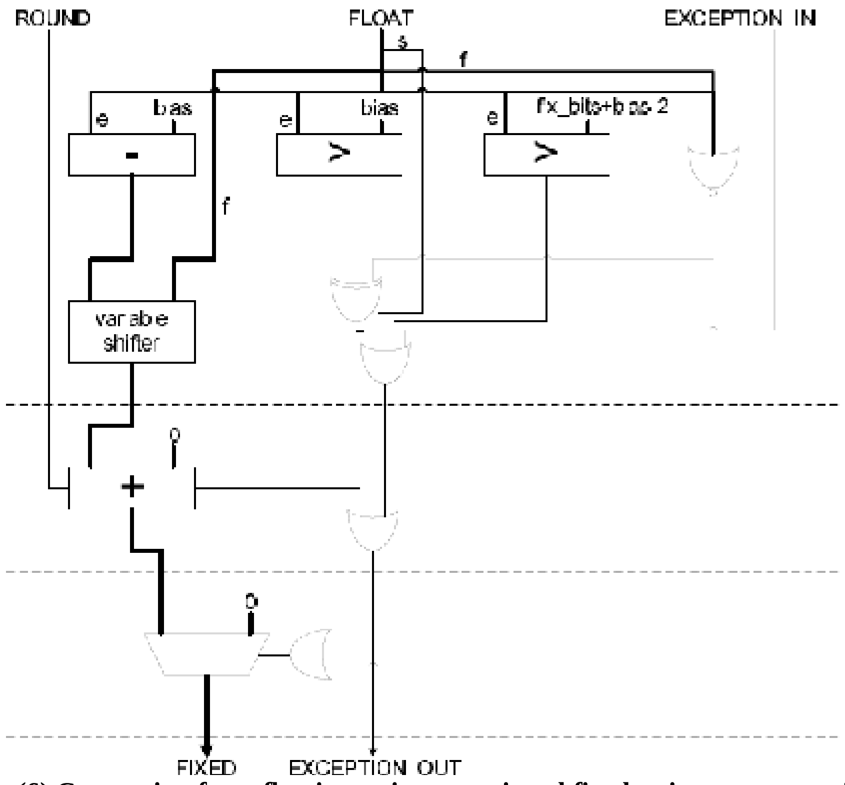

Figure (6) Conversion from floating-point to unsigned fixed-point representation

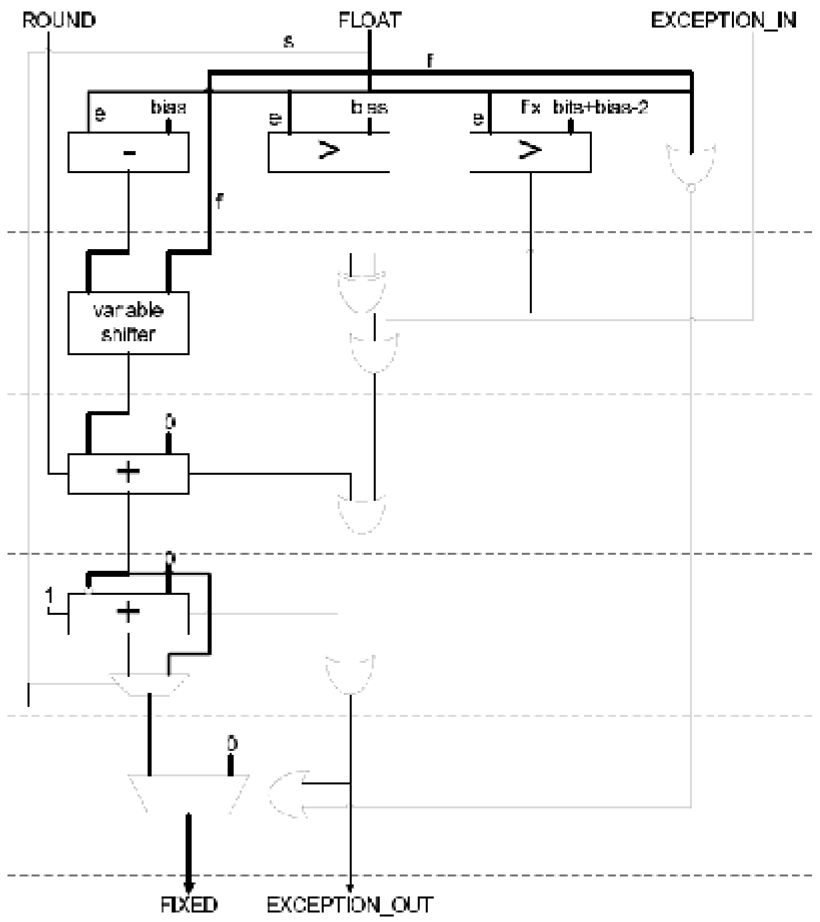

Figure (7) Conversion from floating-point to signed fixed-point representation

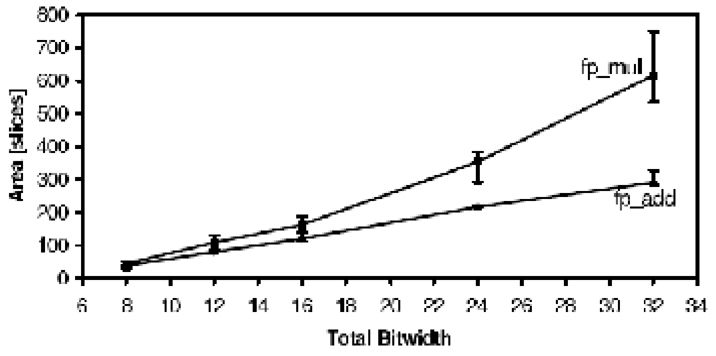

Figure (8) Growth of area with increasing bitwidth

VHDL Entities

```
Module:
parameterized_adder
Entity:
entity parameterized_adder is
generic
bits : integer := 0
);
port
(
--inputs
A : in std_logic_vector(bits-1 downto 0);
B : in std_logic_vector(bits-1 downto 0);
CIN : in std_logic;
--outputs
S : out std_logic_vector(bits-1 downto 0) := (others=>'0');
COUT : out std_logic := '0'
);
end parameterized_adder;
Module:
parameterized_subtractor
Entity:
entity parameterized_subtractor is
generic
(
bits : integer := 0
);
port
(
--inputs
A : in std_logic_vector(bits-1 downto 0);
B : in std_logic_vector(bits-1 downto 0);
--outputs
O : out std_logic_vector(bits-1 downto 0) := (others=>'0')
);
end parameterized_subtractor;
Module:
parameterized_multiplier
Entity:
entity parameterized_multiplier is
generic
(
bits : integer := 0
);
port
(
--inputs
A : in std_logic_vector(bits-1 downto 0);
```

```
B : in std_logic_vector(bits-1 downto 0);
--outputs
S : out std_logic_vector((2*bits)-1 downto 0) := (others=>'0')
);
end parameterized_multiplier;
Module:
parameterized_variable_shifter
Entity:
entity parameterized_variable_shifter is
generic
(
bits : integer := 0;
select_bits : integer := 0;
direction : std_logic := '0' --0=right,1=left
);
port
(
--inputs
I : in std_logic_vector(bits-1 downto 0);
S : in std_logic_vector(select_bits-1 downto 0);
CLEAR : in std_logic;
--outputs
O : out std_logic_vector(bits-1 downto 0)
);
end parameterized_variable_shifter;
Module:
delay_block
Entity:
entity delay_block is
generic
(
bits : integer := 0;
delay : integer := 0
);
port
(
--inputs
A : in std_logic_vector(bits-1 downto 0);
CLK : in std_logic;
--outputs
A_DELAYED : out std_logic_vector(bits-1 downto 0) := (others=>'0')
);
end delay_block;
Module:
parameterized_absolute_value
Entity:
entity parameterized_absolute_value is
generic
```

```
(
bits : integer := 0
);
port
(
--inputs
IN1 : in std_logic_vector(bits-1 downto 0);
--outputs
EXC : out std_logic := '0';
OUT1 : out std_logic_vector(bits-1 downto 0) := (others=>'0')
);
end parameterized_absolute_value;
Module:
parameterized_priority_encoder
Entity:
entity parameterized_priority_encoder is
generic
(
man_bits : integer := 0;
shift_bits : integer := 0
);
port
(
--inputs
MAN_IN : in std_logic_vector(man_bits-1 downto 0);
--outputs
SHIFT : out std_logic_vector(shift_bits-1 downto 0) := (others=>'0');
EXCEPTION_OUT : out std_logic := '0'
);
end parameterized_priority_encoder;
Module:
parameterized_mux
Entity:
entity parameterized_mux is
generic
(
bits : integer := 0
);
port
(
--inputs
A : in std_logic_vector(bits-1 downto 0);
B : in std_logic_vector(bits-1 downto 0);
S : in std_logic;
--outputs
O : out std_logic_vector(bits-1 downto 0) := (others=>'0')
);
end parameterized_mux;
```

Module:
parameterized_comparator
Entity:

```
entity parameterized_comparator is
generic
(
bits : integer := 0
);
Port
(
--inputs
A : in std_logic_vector(bits-1 downto 0);
B : in std_logic_vector(bits-1 downto 0);
--outputs
A_GT_B : out std_logic := '0';
A_EQ_B : out std_logic := '0';
A_LT_B : out std_logic := '0'
);
end parameterized_comparator;
```

Module:
denorm
Entity:

```
entity denorm is
generic
(
exp_bits : integer := 0;
man_bits : integer := 0
);
port
(
--inputs
IN1 : in std_logic_vector(exp_bits+man_bits downto 0);
READY : in std_logic;
EXCEPTION_IN : in std_logic;
--outputs
OUT1 : out std_logic_vector(exp_bits+man_bits+1 downto 0) := (others=>'0');
DONE : out std_logic := '0';
EXCEPTION_OUT : out std_logic := '0'
);
end denorm;
```

Module:
rnd_norm
Entity:

```
entity rnd_norm is
generic
(
exp_bits : integer := 0;
man_bits_in : integer := 0;
man_bits_out : integer := 0
);
port
(
--inputs
IN1 : in std_logic_vector((exp_bits+man_bits_in) downto 0);
READY : in std_logic;
CLK : in std_logic;
ROUND : in std_logic;
EXCEPTION_IN : in std_logic;
--outputs
OUT1 : out std_logic_vector((exp_bits+man_bits_out) downto 0) := (others=>'0');
DONE : out std_logic := '0';
EXCEPTION_OUT : out std_logic := '0'
);
end rnd_norm;
```

Module:
fp_add
Entity:

```
entity fp_add is
generic
(
exp_bits : integer := 0;
man_bits : integer := 0
);
port
(
--inputs
OP1 : in std_logic_vector(man_bits+exp_bits downto 0);
OP2 : in std_logic_vector(man_bits+exp_bits downto 0);
READY : in std_logic;
EXCEPTION_IN : in std_logic;
CLK : in std_logic;
--outputs
RESULT : out std_logic_vector(man_bits+exp_bits+1 downto 0) := (others=>'0');
EXCEPTION_OUT : out std_logic := '0';
DONE : out std_logic := '0'
);
end fp_add;
```

Module:
fp_sub
Entity:

```
entity fp_sub is
generic
(
exp_bits : integer := 0;
man_bits : integer := 0
);
```

```
port
(
--inputs
OP1 : in std_logic_vector(man_bits+exp_bits downto 0);
OP2 : in std_logic_vector(man_bits+exp_bits downto 0);
READY : in std_logic;
EXCEPTION_IN : in std_logic;
CLK : in std_logic;
--outputs
RESULT : out std_logic_vector(man_bits+exp_bits+1 downto 0) := (others=>'0');
EXCEPTION_OUT : out std_logic := '0';
DONE : out std_logic := '0'
);
end fp_sub;
Module:
fp_mul
Entity:
entity fp_mul is
generic
(
exp_bits : integer := 0;
man_bits : integer := 0
);
port
(
--inputs
OP1 : in std_logic_vector(exp_bits+man_bits downto 0);
OP2 : in std_logic_vector(exp_bits+man_bits downto 0);
READY : in std_logic;
EXCEPTION_IN : in std_logic;
CLK : in std_logic;
--outputs
RESULT : out std_logic_vector(exp_bits+(2*man_bits) downto 0) := (others=>'0');
EXCEPTION_OUT : out std_logic := '0';
DONE : out std_logic := '0'
);
end entity;
Module:
fix2float
Entity:
entity fix2float is
generic
(
fix_bits : integer := 0;
exp_bits : integer := 0;
man_bits : integer := 0
);
port
```

```
(
--inputs
FIXED : in std_logic_vector(fix_bits-1 downto 0);
ROUND : in std_logic;
EXCEPTION_IN : in std_logic;
CLK : in std_logic;
READY : in std_logic;
--outputs
FLOAT : out std_logic_vector(exp_bits+man_bits downto 0) := (others=>'0');
EXCEPTION_OUT : out std_logic := '0';
DONE : out std_logic := '0'
);
end fix2float;
Module:
float2fix
Entity:
entity float2fix is
generic
(
fix_bits : integer := 0;
exp_bits : integer := 0;
man_bits : integer := 0
);
port
(
--inputs
FLOAT : in std_logic_vector(exp_bits+man_bits downto 0);
ROUND : in std_logic;
EXCEPTION_IN : in std_logic;
CLK : in std_logic;
READY : in std_logic;
--outputs
FIXED : out std_logic_vector(fix_bits-1 downto 0) := (others=>'0');
EXCEPTION_OUT : out std_logic := '0';
DONE : out std_logic := '0'
);
end float2fix;
```

VHDL Description of the IEEE Single Precision Adder

```
--================================================--
-- LIBRARIES --
--================================================--
-- IEEE Libraries --
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
-- float
library PEX_Lib;
```

```
use PEX_Lib.float_pkg.all;
-----------------------------------------------------------
-- IEEE Single Precision Adder --
-----------------------------------------------------------
entity single_precision_adder is
port
(
--inputs
IN1 : in std_logic_vector(31 downto 0);
IN2 : in std_logic_vector(31 downto 0);
READY : in std_logic;
EXCEPTION_IN : in std_logic;
ROUND : in std_logic;
CLK : in std_logic;
--outputs
OUT1 : out std_logic_vector(31 downto 0) := (others=>'0');
EXCEPTION_OUT : out std_logic := '0';
DONE : out std_logic := '0'
);
end single_precision_adder;
-----------------------------------------------------------
-- IEEE Single Precision Adder --
-----------------------------------------------------------
architecture single_precision_adder_arch of single_precision_adder is
signal rd1 : std_logic := '0';
signal rd2 : std_logic := '0';
signal rd3 : std_logic := '0';
signal rd4 : std_logic := '0';
signal exc1 : std_logic := '0';
signal exc2 : std_logic := '0';
signal exc3 : std_logic := '0';
signal exc4 : std_logic := '0';
signal rnd1 : std_logic := '0';
signal rnd2 : std_logic := '0';
signal rnd3 : std_logic := '0';
signal rnd4 : std_logic := '0';
signal op1 : std_logic_vector(32 downto 0) := (others=>'0');
signal op2 : std_logic_vector(32 downto 0) := (others=>'0');
signal sum : std_logic_vector(33 downto 0) := (others=>'0');
begin
--instances of components
denorm1: denorm
generic map
(
exp_bits => 8,
man_bits => 23
)
port map
```

```
(
--inputs
IN1 => IN1,
READY => READY,
EXCEPTION_IN => EXCEPTION_IN,
--outputs
OUT1 => op1,
DONE => rd1,
EXCEPTION_OUT => exc1
);
denorm2: denorm
generic map
(
exp_bits => 8,
man_bits => 23
)
port map
(
--inputs
IN1 => IN2,
READY => READY,
EXCEPTION_IN => EXCEPTION_IN,
--outputs
OUT1 => op2,
DONE => rd2,
EXCEPTION_OUT => exc2
);
adder: fp_add
generic map
(
exp_bits => 8,
man_bits => 24
)
port map
(
--inputs
OP1 => op1,
OP2 => op2,
READY => rd3,
EXCEPTION_IN => exc3,
CLK => CLK,
--outputs
RESULT => sum,
EXCEPTION_OUT => exc4,
DONE => rd4
);
rnd_norm1: rnd_norm
generic map
```

```
(
exp_bits => 8,
man_bits_in => 25,
man_bits_out => 23
)
port map
(
--inputs
IN1 => sum,
READY => rd4,
CLK => CLK,
ROUND => rnd4,
EXCEPTION_IN => exc4,
--outputs
OUT1 => OUT1,
DONE => DONE,
EXCEPTION_OUT => EXCEPTION_OUT
);
rd3 <= rd1 AND rd2;
exc3 <= exc1 OR exc2;
main: process (CLK)
begin
if(rising_edge(CLK)) then
rnd4 <= rnd3;
rnd3 <= rnd2;
rnd2 <= rnd1;
rnd1 <= ROUND;
end if;--CLK
end process;--main
end single_precision_adder_arch;--end of architecture
```