Floating point package user's guide

By David Bishop (dbishop@vhdl.org)

Floating-point numbers are the favorites of software people, and the least favorite of hardware people. The reason for this is because floating point takes up almost 3X the hardware of fixed-point math. The advantage of floating point is that precision is always maintained with a wide dynamic range, where fixed point numbers loose precision.

The floating point VHDL packages can be downloaded at:
http://www.vhdl.org/fphdl/vhdl.html

The files needed are "fixed_float_types.vhdl", "float_generic_pkg.vhdl", "float_generic_pkg-body.vhdl" and "float_pkg.vhdl". There is a dependency on "fixed_pkg.vhdl" and therefore "fixed_generic_pkg.vhdl", "fixed_generic_pkg-body.vhdl", so these files should be compiled first. These packages have been designed for use in VHDL-2008, where they will be part of the IEEE library. However, a version of the packages is provided that works great in VHDL-93, which is the version they were tested under. They also have no dependencies to the other new VHDL packages (with the exception of "fixed_pkg"). The compatibility package is synthesizable.

The files on the web page to download are:

fixed_float_types_c.vhdl	- Types used in the fixed point and floating point package
fixed_pkg_c.vhdl	- Fixed-point package (VHDL-93 compatibility version)
float_pkg_c.vhdl	- Floating-point package (VHDL-93 compatibility version)

These files should be compiled into a library called "ieee_proposed".

Floating-point numbers:

Floating-point numbers are well defined by IEEE-754 (32 and 64 bit) and IEEE-854 (variable width) specifications. Floating point has been used in processors and IP for years and is a well-understood format. This is a sign magnitude system, where the sign is processed differently from the magnitude.

There are many concepts in floating point that make it different from our common signed and unsigned number notations. These come from the definition of a floating-point number. Lets first take a look at a 32-bit floating-point number:

```
S   EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
31  30    25 24                    0
+/-     exp.  Fraction
```

Basically, a floating-point number comprises a sign bit (+ or–), a normalized exponent, and a fraction. To convert this number back into an integer, the following equation can be used:

```
S * 2 ** (exponent – exponent_base) * (1.0 + Fraction/fraction_base)
```

where the "exponent_base" is 2**((maximum exponent/2)–1), and "Fraction_base" the maximum possible fraction (unsigned) plus one. Thus, for a 32-bit floating-point an example would be:

```
0 10000001 10100000000000000000000
```

= +1 * 2** (129 – 127) * (1.0 + 10485760/16777216) = +1 * 4.0 * 1.625 = 6.5

There are also "denormal numbers", which are numbers smaller than can be represented with this structure. The tag for a denormal number is that the exponent is "0". This forces you to invoke another formula where you drop the "1.0 +":

```
1 00000000 10000000000000000000000
```

= -1 * 2** -126 * (8388608/16777216) = -1 * 2**-126 * 0.5 = -2**-127

Next, the "constants" that exist in the floating-point context are:

```
0 00000000 00000000000000000000000 = 0
1 00000000 00000000000000000000000 = -0 (which = 0)
0 11111111 00000000000000000000000 = positive infinity
1 11111111 00000000000000000000000 = negative infinity
```

If you get a number with an infinite (all "1"s) exponent and anything other than an all zero fraction, it is said to be a NAN, or "Not A Number". NANs come in two types, signaling and non-signaling. For the purposes of these packages, I chose a fraction with an MSB of "1" to be a signaling NAN and anything else to be a quiet NAN.

Thus, the following classes (or states) result, wherein each floating-point number can fall:

nan	Signaling NaN
quiet_nan	Quiet NaN
neg_inf	Negative infinity
neg_normal	Negative normalized nonzero
neg_denormal	Negative denormalized
neg_zero	-0
pos_zero	+0
pos_denormal	Positive denormalized
pos_normal	Positive normalized nonzero
pos_inf	Positive infinity

These states are used to examine and create numbers needed for floating-point operations. This defines the type "valid_fpstate". The constants zerofp, nanfp, qnanfp, pos_inffp, neginf_fp, neg_zerofp are also defined.

Rounding can take four different forms:

round_nearest	Round nearest
round_inf	Round positive infinity
round_neginf	Round negative infinity
round_zero	Round zero (truncate)

"Round nearest" has the extra caveat that, if the remainder is exactly ½, you need to round so that the LSB of the number you get is a zero. The implementation of this feature requires two compare operations, but they can be consolidated. Round negative infinity rounds down, and round positive infinity always rounds up. Round zero is a mix of the two and has the effect of truncation (no rounding). A type called "round_type" is defined (in ieee.math_utility_pkg) for use on all of the rounding parameters.

Floating-point package:

In floating point there are several options that you need to choose from. Support for denormal numbers and a large precision can take up a great deal of hardware. These may be things you can live without. For that reason there is a version of these packages implemented with "package generics" so that you can change the default settings.

"float_generic_pkg" can not be used directly because of the way package generics works. An instance of "float_generic_pkg" called "float_pkg" is in the IEEE library. The data types in these packages also use a negative index.

The package generics uses in these packages are:

- Denormalize – Boolean – Thus is used to turn on and off denormal numbers. The default is true (allow denormal numbers)
- Round_style – round_type – This is used to specify the rounding style to be used. "Round_nearest" is the default, and takes the most hardware. Round_zero does a truncation, round_inf and round_neginf round up or down depending on weather the number is positive or negative

- Check_error – Boolean – Turns off NAN and infinity processing. These checks need to be at the beginning of every operation, and if you have already checked once you need not check again.
- Guard_bits – natural – This is the number of extra bits used on each operation to maintain precision. The default is 3. Note that if you take this down to zero, then rounding is automatically turned off.
- No_warning – Boolean – Allows you to turn off the “metavalue” warnings by setting this to “false”.
- Float_exponent_width – Default for conversion routines. Set by default to the size of a 32 bit floating point number (8).
- Float_fraction_width – Default for conversion routines. Set by default to the size of a 32 bit floating point number (23).

Use model:

```
use ieee.float_pkg.all; -- use “ieee_proposed” for VHDL-93 version
…
  variable x, y, z : float (5 downto -10);
begin
  y := to_float (3.1415, y); -- Uses “y” for the sizing only.
  z := “0011101010101010”; –- 1/3
  x := z + y;
```

The base package includes the definition for three floating-point types.
Float32 – 32-bit IEEE 754 single precision floating point
Float64 – 64-bit IEEE 754 double precision floating point
Float128 – 128-,bit IEEE 854 extended precision floating point
It also allows you to specify your own floating-point width, by bounding the “float” type as shown in the example above. The types “round_type”, defining the rounding style, and “valid_fpstate” (as noted above) are also defined here.

The actual floating-point type is defined as follows:

```
subtype float32 is float (8 downto –23);
```

A negative index is used to separate the fraction part of the floating-point number from the exponent. The top bit is the sign bit (‘high), the next bits are the exponent (‘high-1 downto 0), and the negative bits are the fraction (–1 downto ‘low). For a 32-bit representation, that specification makes the number look as follows:

```
0 00000000 0000000000000000000000
8 7      0 -1                 -23
+/-   exp.  fraction
```

where the sign is bit 8, the exponent is contained in bits 7–0 (8 bits) with bit 7 being the MSB, and the mantissa is contained in bits –1 -– –23 (32 – 8– 1 = 23 bits) where bit –1 is the MSB.

The negative index format turns out to be a very natural format for the floating-point number, as the fraction is always assumed to be a number between 1.0 and 2.0 (unless we are denormalized). Thus, the implied “1.0” can be assumed on the positive side of the index, and the negative side represents a fraction of less than one. The format here is similar to that used in the fixed point package, where everything to the right of the zero index is assumed to be less than 1.0.

Valid values for float_exponent_width and float_fraction_width are 3 and up. Thus, the smallest (width-wise) number that can be made is float (3 downto –3) or a 7-bit floating-point number.

Operators for all of the standard math and compare operations are defined in this package. However, all operators use the full IEEE floating point. For most designs, full IEEE support is not necessary. Thus, functions have been created that allow you to parameterize your design. Example:

```
X <= add (l => z, r => y,
Denormalize => false, -- Trun off denormal numbers (default=true)
Check_error => false,-- turn off NAN and overflow checks (default=true)
Round_style => round_zero, -- truncate (default=round_nearest)
Guard_bits  => 0); -- Extra bits to maintain precision (default=3)
```

The add function performs just like the "+" operator; however, it allows the user the flexibility needed for hardware synthesis. Other similar functions are subtract "–" multiply "*", divide "/", modulo "mod", and remainder "rem". All of these operators and functions assume that both of the inputs are the same width. Other functions with similar parameters are "reciprocal" (1/x), and "dividebyp2" (divide by a power of 2). The "abs" and unary "–" operators need no parameters, as in floating-point sign operations, they only work on the MSB.

Compare operators work similarly, however there is only one extra parameter for these functions, which is the "check_error" parameter. These functions are called EQ "=", NE "/=", LT "<", GT ">", GE ">=", and LE "<=".

Conversion operators also work in a similar manor. Functions named "to_float" are available to convert the types real, integer, signed, unsigned, ufixed, and sfixed. All of these functions take as parameters either the exponent_width and fraction_width, or a "size_res" input, which will use the input variable for its size only. The functions "to_real", "to_integer", "to_sign", "to_unsigned", "to_ufixed", and "to_sfixed" are also overloaded in the base package with both size and size_res inputs. There is also a similar "resize" function to convert from one "float" size to another. Note that, like the fixed_pkg, and "to" direction on a float type, it is illegal.

Functions recommended by IEEE-854:
Copysign (x, y) – Returns x with the sign of y.
Scalb (y, N) – Returns y*(2**n) (where N is an integer or SIGNED) without computing 2**n.
Logb (x) – Returns the unbiased exponent of x
Nextafter(x,y) – Returns the next representable number after x in the direction of y.
Fininte(x) – Boolean, true if X is not positive or negative infinity
Isnan(x) – Boolean, true if X is a NAN or quiet NAN.
Unordered(x, y) – Boolean, returns true of either X or Y are some type of NAN.
Classfp(x) – valid_fpstate, returns the type of floating-point number (see valid_fpstate definition above)

Two extra functions named "break_number" and "normalize" are also provided. "break_number" takes a floating-point number and returns a "SIGNED" exponent (biased by –1) and an "ufixed" fixed-point number. "normalize" takes a SIGNED exponent and a fixed-point number and returns a floating-point number. These functions are useful for times when you want to operate on the fraction of a floating-point number without having to perform the shifts on every operation.

To_slv (aliased to to_std_logic_vector and to_StdLogicVector), as well as to_float(std_logic_vector), are used to convert between std_logic_vector and fp types. These should be used on the interface of your designs. The result of "to_slv" is a std_logic_vector with the length of the input fp type. The to_sulv (aliased to to_std_ulogic_vector and to_stdulogvector) works the same, but converts to std_ulogic_vector.

The procedures reading and writing floating-point numbers are also included in this package. Procedures read, write, oread, owrite (octal), bread, bwrite (binary), hread, and hwrite (hex) are defined. To_string, to_ostring, and to_hstring are also provided for string results. Floating-point numbers are written in the format "0:000:000" (for a 7-bit FP). They can be read as a simple string of bits, or with a "." Or ":" separator.

Base package use model:

```
entity xxx is port (
  a, b : in std_logic_vector (31 downto 0);
  Sum : out std_logic_vector (31 downto 0);
  Clk, reset : in std_ulogic);
end entity xxx;

library ieee_proposed;
use ieee_proposed.fixed_float_types.all; -- ieee in the release
use ieee_proposed.float_pkg.all; -- ieee.float_pkg.all; in the release

architecture RTL of xxx is
  Signal afp, bfp, Sumfp : float32; -- same as “float (8 downto –23)”
begin
  afp <= to_float (a, afp’high, -afp’low); -- SLV to float, with bounds
  bfp <= to_float (b, Bfp); -- SLV to float, using Bfp’range

Addreg : process (clk, reset) is
  begin
    if reset = ‘1’ then
      Sumfp <= (others => ‘0’);
    elsif rising_edge (clk) then
      Sumfp <= afp + bfp;
-- this is the same as saying:
-- Sumfp <= add (l => afp, r => bfp,
--   round_style => round_nearest, -- best, but most hardware
--    guard_bits => 3,     -- Use 3 guard bits, best for round_nearest
--   check_error => true,  –- NAN processing turned on
--   denormalize => true); -- Turn on denormal numbers
     end process addreg;
   Sum <= to_slv (Sumfp);
end process Addreg;
end architecture xxx;
```

Another example package is “float_generic_pkg-body_real.vhdl”. This is an alternate body for “float_generic_pkg” which does all of the arithmetic using the type “real”. There are differences in rounding, however everything else works correctly. You can check the deferred constant “fphdlsynth_or_real” to find out which package body you are using (this will be “true” if you are using the synthesizable package and “false” for the real number package).

Package Generics

These packages are done using something new in VHDL-200X called package generics. “float_generic_pkg.vhdl” contains the following:

```
use ieee.fixed_float_types.all;
package float_generic_pkg is
  generic (
    -- Defaults for sizing routines, when you do a "to_float" this will
    -- the default size.  Example float32 would be 8 and 23 (8 downto -23)
    float_exponent_width : natural := 8;
    float_fraction_width : natural := 23;
    -- Rounding algorithm, "round_nearest" is default, other valid values
    -- are "round_zero" (truncation), "round_inf" (round up), and
    -- "round_neging" (round down)
    float_round_style    : float_round_type := round_nearest;
    -- Denormal numbers (very small numbers near zero) true or false
    float_denormalize    : BOOLEAN := true;
    -- Truns on NAN processing (invalud numbers and overflow) true of false
    float_check_error    : BOOLEAN := true;
    -- Guard bits are added to the bottom of every operation for rounding.
    -- any natural number (including 0) are valid.
    float_guard_bits     : NATURAL := 3;
    -- If TRUE, then turn off warnings on "X" propigation
    NO_WARNING           : BOOLEAN := false
    );
```

Due to the way package generics work, “float_generic_pkg” can not be used directly. However another package called “float_pkg” is provided. This package looks like the following:

```
use ieee.math_utility_pkg.all;
package float_pkg is
  new work.float_generic_pkg
  generic map (
    float_exponent_width => 8;    -- float32'high
    float_fraction_width => 23;   -- -float32'low
    float_round_style    => round_nearest; -- round nearest algorithm
    float_denormalize    => true; -- Use IEEE extended floating
    float_check_error => true; -- Turn on NAN and overflow processing
    float_guard_bits     => 3;     -- number of guard bits
    no_warning           => false  -- show warnings
  );
```

This is where the defaults get set. Note that the user can now create his/her own version of the floating point package if the defaults are not what they desire.

Example:
I don’t want any rounding (takes up too much logic), I want to default to a 17 bit floating point number with only 5 bits of exponent, I don’t want to worry about denormal numbers, I don’t need any NAN processing, and I want those silly “metavalue detected” warnings turned off. Easy:

```
use ieee.fixed_float_types.all;
package my_float_pkg is
  new ieee.float_generic_pkg
  generic map (
    float_exponent_width => 5;     -- 5 bits of exponent
    float_fraction_width => 11;    -- default will be float(5 dt –11)
    float_round_style    => round_zero; -- Truncate, don’t round
    float_denormalize    => false; -- no denormal numbers
    float_guard_bits     => 0;     -- Unused by round_zero, set to 0
    float_check_error    => false; -- Turn NAN and overflow off.
    no_warning           => true   -- turn warnings off
    );
```

Now you can compile this file into your code. You will have to do a “use work.my_float_pkg.all;” to make the floating point function visible. If you need to translate back to the IEEE “float_pkg” types, you can do that with type casting as follows:

```
use IEEE.float_pkg.all;
entity sin is
  port (
    arg      : in  float (5 downto -11);
    clk, rst : in  std_ulogic;
    res      : out float (5 downto -11));
end entity sin;
```

Architecture:

```
architecture structure of sin is
  component float_sin is
    port (
      arg      : in  work.my_float_pkg.float (5 downto -11);
      clk, rst : in  STD_ULOGIC;
      res      : out work.my_float_pkg.ufixed (5 downto -11));
  end component fixed_sin;
  signal resx      : work.my_fixed_pkg.ufixed (5 downto -11);
begin
  U1: float_sin
    port map (
      arg => work.my_float_pkg.float(arg),   -- convert “arg”
      clk => clk,
      rst => rst,
      res => resx);
  res <= ieee.float_pkg.float (resx);
end architecture structure;
```

Challenges for synthesis vendors:

Now that we are bringing numbers that are less than 1.0 into the realm of synthesis, the type “REAL” becomes meaningful. To_float (MATH_PI) will now evaluate to a string of bits. This means that synthesis vendors will now have to not only understand the “real” type, but the functions in the “math_real” IEEE package as well.

Both of these packages depend upon a negative index. Basically, everything that is at an index that is less than zero is assumed to be to the right of the decimal point. By doing this, we were able to avoid using record types. This also represents a challenge for some synthesis vendors, but it makes these functions portable to Verilog.

Index:

Operators:
“+” - Add two floating-point numbers together, overloaded for real and integer. By default Rounding is set to “round_nearest”, guard bits set to 3, and denormal number and NANs are turned on. If this is not the desired functionality, use the “add” function. Will accept floating-point numbers of any valid width on either input.

“-” – Subtracts floating-point numbers. Overloaded for real and integer. By default Rounding is set to “round_nearest”, guard bits set to 3, and denormal number and NANs are turned on. If this is not the desired functionality, use the “subtract” function. Will accept floating-point numbers of any valid width on either input.

“*” – Multiply two floating-point numbers together. Overloaded for real and integer. By default Rounding is set to “round_nearest”, guard bits set to 3, and denormal number and NANs are turned on. If this is not the desired functionality, use the “multiply” function. Will accept floating-point numbers of any valid width on either input.

“/” – Divides two floating-point numbers. Overloaded for real and integer. By default Rounding is set to “round_nearest”, guard bits set to 3, and denormal number and NANs are turned on. If this is not the desired functionality, then use the “divide” function. Will accept floating-point numbers of any valid width on either input.

“abs” – Absolute value. Changes only the sign bit.

“-“ – Unary minus. Changes only the sign bit.

“mod” – modulo. Overloaded for real and integer. By default Rounding is set to “round_nearest”, guard bits set to 3, and denormal number and NANs are turned on. If this is not the desired functionality, then use the “modulo” function. Will accept floating-point numbers of any valid width on either input.

“rem” – Remainder. Overloaded for real and integer. By default Rounding is set to “round_nearest”, guard bits set to 3, and denormal number and NANs are turned on. If this is not the desired functionality, then use the “modulo” function. Will accept floating-point numbers of any valid width on either input.

“=” – equal. Overloaded for real and integer. By default NAN processing is turned on. If this is not the desired functionality, then use the “eq” function.

“/=”– not equal. Overloaded for real and integer. If this is not the desired functionality, then use the “ne” function.

“<”– less than. Overloaded for real and integer. If this is not the desired functionality, then use the “lt” function.

“>”– greater than. Overloaded for real and integer. If this is not the desired functionality, then use the “gt” function.

“<=”– less than or equal to. Overloaded for real and integer. If this is not the desired functionality, then use the “le” function.

“>=”– greater than or equal to. Overloaded for real and integer. If this is not the desired functionality, then use the “ge” function.

“?=” - Performs an operation similar to the “std_match” function, but returns a std_ulogic value. The values returned from this function are “U”, “X”, “0”, and “1”. The VHDL-93 compatible version of this function is “\?=\”

“?/=” - Performs an operation similar to the inverse of the “std_match” function, but returns a std_ulogic value. The VHDL-93 compatible version of this function is “\?/=\”.

“?<” - Performs a “<” function, but returns a std_ulogic value. In this function all metavalues are mapped to an “X”. The VHDL-93 compatible version of this function is “\?<\”.

“?<=” - Performs a “<=” function, but returns a std_ulogic value. In this function all metavalues are mapped to an “X”. The VHDL-93 compatible version of this function is “\?<=\”

“?>” - Performs a “>” function, but returns a std_ulogic value. In this function all metavalues are mapped to an “X”. The VHDL-93 compatible version of this function is “\?>\”

“?>=” - Performs a “>=” function, but returns a std_ulogic value. In this function all metavalues are mapped to an “X”. The VHDL-93 compatible version of this function is “\?>=\”

“not” – Logical not

“and” –logical and. There are 3 versions of this operator. “vector op vector”, “vector op std_ulogic”, and “op vector”. The “vector op vector” version operates on each bit of the vector independently. “vector op std_ulogic” performs the operation on every bit of the vector with the “std_ulogic”. The “op vector” version performs a reduction operation and returns a single bit. The vhdl-2002 version of this function is “and_reduce”. An “op vector” with a null array returns a “1”.

“nand” –logical nand. There are 3 versions of this operator. “vector op vector”, “vector op std_ulogic”, and “op vector”. The “vector op vector” version operates on each bit of the vector independently. “vector op std_ulogic” performs the operation on every bit of the vector with the “std_ulogic”. The “op vector” version performs a reduction operation and returns a single bit. The vhdl-2002 version of this function is “nand_reduce”. An “op vector” with a null array returns a “0”.

“or” –logical or. There are 3 versions of this operator. “vector op vector”, “vector op std_ulogic”, and “op vector”. The “vector op vector” version operates on each bit of the vector independently. “vector op std_ulogic” performs the operation on every bit of the vector with the “std_ulogic”. The “op vector” version performs a reduction operation and returns a single bit. The vhdl-2002 version of this function is “or_reduce”. An “op vector” with a null array returns a “0”.

“nor” –logical nor. There are 3 versions of this operator. “vector op vector”, “vector op std_ulogic”, and “op vector”. The “vector op vector” version operates on each bit of the vector independently. “vector op std_ulogic” performs the operation on every bit of the vector with the “std_ulogic”. The “op vector” version performs a reduction operation and returns a single bit. The vhdl-2002 version of this function is “nor_reduce”. An “op vector” with a null array returns a “0”.

“xor” –logical exclusive or. There are 3 versions of this operator. “vector op vector”, “vector op std_ulogic”, and “op vector”. The “vector op vector” version operates on each bit of the vector independently. “vector op std_ulogic” performs the operation on every bit of the vector with the “std_ulogic”. The “op vector” version performs a reduction operation and returns a single bit. The vhdl-2002 version of this function is “xor_reduce”. An “op vector” with a null array returns a “0”.

“xnor” –logical exclusive nor. There are 3 versions of this operator. “vector op vector”, “vector op std_ulogic”, and “op vector”. The “vector op vector” version operates on each bit of the vector independently. “vector op std_ulogic” performs the operation on every bit of the vector with the “std_ulogic”. The “op vector” version performs a reduction operation and returns a single bit. The vhdl-2002 version of this function is “xnor_reduce”. An “op vector” with a null array returns a “1”.

Functions:
Class – Find the classification of a floating-point number. Inputs: arg (float). Returns a value of the type “valid_fpstate”.

Add - see “+” operator

Subtract – see “-“ operator.

Multiply – see “*” operator.

Divide – see “/” operator.

Remainder – see the “rem” operator.

Modulo – see the “mod” operator.

Reciprocal – returns 1/arg. Inputs: l, r: float, round_style : round_type, guard : natural, check_error: Boolean, denormalize: Boolean. Works similar to the divide function

Dividebyp2 – divide by a power of two. Inputs: l, r: float, round_style : round_type, guard : natural, check_error: Boolean, denormalize: Boolean. Takes the exponent from R and multiplies L by that amount. Returns an error if R is not a power of 2.

Mac – Multiply accumulate function - This is a floating point multiplier followed by an addition operation.

Eq – see “=” operator.

Ne – see “/=” operator.

Lt – see “<” operator.

Gt – see “>” operator.

Le – see “<” operator.

Ge – see “>” operator.

Std_match – Same as the numeric_std “std_match” function. Overloaded for type “float”.

Maximum – returns the larger of two numbers. Inputs: l, r, check_error : Boolean, denormalize : Boolean.

Minimum – returns the smaller of two numbers. Inputs: l, r, check_error : Boolean, denormalize : Boolean.

Conversion functions:
Resize – Changes the size of a float (larger or smaller). Inputs: arg (float), exponent_width and fraction_width (natural) OR size_res, round_style : round_type, Check_error : Boolean, denormalize_in : Boolean, denormalize : Boolean. In this function “denormalize_in” is true if the input number can be denormal. “denormalize” is true if the output number can be denormal.

To_slv – Inputs: arg (float). Converts a floating-point number into a std_logic_vector of the same length.
To_std_logic_vector – alias of to_slv.
To_stdlogicvector – alias of to_slv

To_sulv – Inputs: arg (float). Converts a floating-point number into a std_ulogic_vector of the same length.
To_std_ulogic_vector – alias of to_sulv.
To_stdulogicvector – alias of to_sulv

To_float – Converts to the “float” type. The default size returned by these functions is set by “float_exponent_width” and “float_fraction_width”.

> To_float (std_logic_vector) – “std_logic_vector” to “float”. Inputs: arg (std_logic_vector) , exponent_width and fraction_width (natural) OR size_res (float).
>
> To_float (std_ulogic_vector) – “std_ulogic_vector” to “float”. Inputs: arg (std_ulogic_vector) , exponent_width and fraction_width (natural) OR size_res (float).

To_float (integer) – “integer” to “float”. Inputs: arg (integer) exponent_width and fraction_width (natural) OR size_res (float), round_style : round_type.

To_float (real) – “real” to “float”. Inputs: arg (real) exponent_width and fraction_width (natural) OR size_res (float), round_style : round_type, denormalize : boolean.

To_float(ufixed) – “ufixed” to “float”. Inputs: arg(ufixed) exponent_width and fraction_width (natural) OR size_res (float), round_style : round_type, denormalize : boolean.

To_float(sfixed) – “sfixed” to “float”. Inputs: arg(sfixed) exponent_width and fraction_width (natural) OR size_res (float), round_style : round_type, denormalize : boolean.

To_float (signed) – “signed” to “float”. Inputs: arg (signed) , exponent_width and fraction_width (natural) OR size_res (float), round_style : round_type.

To_float (unsigned) – “unsigned” to “float”. Inputs: arg (signed) , exponent_width and fraction_width (natural) OR size_res (float), round_style : round_type.

To_unsigned – “float” to “unsigned”. Inputs: arg (float) , size :natural. Parameters: round_style : round_type, check_error : boolean. This does not produce a “vector truncated” warning as the ieee.numeric_std functions do. Returns a zero if the number is negative. Returns a saturated value if the input is too big.

To_signed – “float” to “signed”. Inputs: arg (float) , size :natural. Parameters: round_style : round_type, check_error : boolean. This does not produce a “vector truncated” warning as the ieee.numeric_std functions do. Returns a saturated value if the number is too big.

To_ufixed – “float” to “ufixed”. Inputs: arg (float), left_index and right_index (natural) OR size_res (ufixed). Parameters round_style : Boolean (true), overflow_style : Boolean (true), check_error : Boolean (true), and denormalize : Boolean (true).

To_sfixed – “float” to “sfixed”. Inputs: arg (float), left_index and right_index (natural) OR size_res (ufixed). Parameters round_style : Boolean (true), overflow_style : Boolean (true), check_error : Boolean (true), and denormalize : Boolean (true).

To_real – “float” to “real”. inputs: arg (float). Parameters: check_error : Boolean, denormalize : Boolean.

To_integer – “float” to “integer”. inputs: arg (float), Parameters: round_style : round_type, check_error : Boolean.

Is_Negative – Input (arg: float). Returns a “true” if the number is negative, otherwise returns false.

To_01 – Inputs (arg: float). Parameters: XMAP : std_ulogic. Converts metavalues in the vector ARG to the XMAP state (defaults to 0).

Is_X – Inputs (arg: float) – returns a Boolean which is “true” if there are any metavalues in the vector “arg”.

To_x01 – Inputs (arg: float) – Converts any metavalues found in the vector “arg” to be “X” , “0”, or “1”.

To_ux01 – Inputs (arg: float) – Converts any metavalues found in the vector “arg” to be “U”, “X” , “0”, or “1”.

To_x01z – Inputs (arg: float) – Converts any metavalues found in the vector “arg” to be “X” , “0”, “1” or “Z”.

Break_number – Procedure to break a floating point number into its parts. Inputs: arg : float, denormalize : Boolean (true), check_error : Boolean (true). Output: fract : unsigned or ufixed (with a "1" in the MSB). Expon – exponent (biased by –1, so add "1" to get the true exponent. Sign – sign bit.

Normalize – Function to take a fixed-point number and an exponent and return a floating-point number. Inputs : fract (ufixed) or unsigned, expon : signed (assumed to be biased by –1), sign : std_ulogic. Parameters : exponent_width and fraction_width (natural) or size_res (float), round_style : round_type (round_nearest), denormalize : Boolean (true), nguard : natural (3), sticky : std_ulogic. The input fraction needs to be at least as big as the fraction of the output floating point number. The sticky bit is used to preserve precision in the rounding routines.

Copysign (x, y) – Returns x with the sign of y.

Scalb (y, N) – Returns y*(2**n) (where N is an integer or SIGNED) without computing 2**n.

Logb (x) – Returns the unbiased exponent of x

Nextafter(x,y) – Returns the next representable number after x in the direction of y.

Fininte(x) – Boolean, true if X is not positive or negative infinity

Isnan(x) – Boolean, true if X is a NAN or quiet NAN.

Unordered(x, y) – Boolean, returns true of either X or Y are some type of NAN.

Classfp(x) – returns the "valid_fpstate" type, tells you what type of floating point number was entered.

Zerofp – Returns a floating point zero. Parameters are "exponent_width" and "fraction_width" or "size_res". The default size is set by "float_exponent_width" and "float_fraction_width".

nanfp – Returns a floating point signaling NAN. Parameters are "exponent_width" and "fraction_width" or "size_res". The default size is set by "float_exponent_width" and "float_fraction_width".

qnanfp – Returns a floating point quiet NAN. Parameters are "exponent_width" and "fraction_width" or "size_res". The default size is set by "float_exponent_width" and "float_fraction_width".

Pos_inffp – Returns a floating point positive infinity. Parameters are "exponent_width" and "fraction_width" or "size_res". The default size is set by "float_exponent_width" and "float_fraction_width".

Neg_inffp – Returns a floating point negative infinity. Parameters are "exponent_width" and "fraction_width" or "size_res". The default size is set by "float_exponent_width" and "float_fraction_width".

Neg_zerofp – Returns a floating point negative zero (which by definition is equal to a floating point zero). Parameters are "exponent_width" and "fraction_width" or "size_res". The default size is set by "float_exponent_width" and "float_fraction_width".

Textio Functions:

Write – Similar to the textio "write" procedure. Automatically puts in a ";" after the sign and the exponent.

Read – Similar to the textio "read" procedure. If a decimal point or colon is encountered then it is tested to be sure that it is in the correct place.

Bwrite – Alias to "write"

Bread – Alias to "read"

Owrite – Octal write. If the range is not divisible by 3 then the number is padded until it does.

Oread – Octal read. If the number you are reading does not have a range divisible by 3 then the number is resized to fit.

Hwrite - Hex write. If the range is not divisible by 4 then the number is padded until it does.

Hread– hex read. If the number you are reading does not have a range divisible by 4 then the number is resized to fit.

To_string – Returns a string that can be padded and left or right justified. Example:
Assert (a = 1.5) report "Result was " & to_string (a) severity error;

To_bstring – Alias to "to_string".

To_ostring – Similar to to_string, but returns a padded octal value.

To_hstring – Similar to to_string, but returns a padded hex value.

From_string – Allows you to translate a string (with a decimal point in it) into a floating-point number.
Examples:

```
  Signal a : float (3 downto –3);
Begin
   A <= from_string ("0000.000", a'high, -a'low);
   A <= from_string ("0001.000", a);
```

Note that this is typically not synthesizable (as it uses "string" type). However you can still do "A <= "0000000";" which will synthesize.

From_ostring – Same as "from_string", but uses octal numbers.

From_hstring – Same as "from_string", but uses hex numbers.