Fixed point package user's guide

By David Bishop (dbishop@vhdl.org)

Fixed point is a step between integer math and floating point. This has the advantage of being almost as fast as numeric_std arithmetic, but able to represent numbers that are less than 1.0. A fixed-point number has an assigned width and an assigned location for the decimal point. As long as the number is big enough to provide enough precision, fixed point is fine for most DSP applications. Because it is based on integer math, it is extremely efficient – as long as the data does not vary too much in magnitude.

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

The files needed are "fixed_float_types.vhdl", "fixed_generic_pkg.vhdl", "fixed_generic_pkg-body.vhdl", and "fixed_pkg.vhdl". These packages have been designed for use in VHDL-2008, where they will be part of the new IEEE library. However, a compatibility version of the packages is provided that works great in VHDL-1993, which is the version of VHDL they were tested under. They also have no dependencies to the other new VHDL packages. 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 package
fixed_pkg_c.vhdl - Fixed-point package (VHDL-93 compatibility version)
This file should be compiled into a library called "ieee_proposed".

Overview

This package defines two new types: "ufixed" is the unsigned fixed point, and "sfixed" is the signed fixed point.

```
type ufixed is array (INTEGER range <>) of STD_LOGIC;
type sfixed is array (INTEGER range <>) of STD_LOGIC;
```

Usage model:

```
use ieee.fixed_float_types.all; -- ieee_proposed for VHDL-93 version
use ieee.fixed_pkg.all; -- ieee_proposed for compatibility version
....
  signal a, b : sfixed (7 downto -6);
  signal c: sfixed (8 downto -6);
begin
....
c <= a + b;
```

This data type shows you the location of the decimal point by using a negative index. The decimal point is assumed to be between the "0" and "—1" index. Thus, we can assume "signal y : ufixed (4 downto –5)" as the data type (unsigned fixed point, 10 bits wide, 5 bits of decimal), then y = 6.5 = "00110.10000", or simply:

```
y <= "0011010000";
```

Two other base data types are also defined in this package. They are "UNRESOLVED_UFIXED" (aliased to "U_UFIXED") and "UNRESOLVED_SFIXED" (aliased to "U_SFIXED") which are unresolved versions of the "UFIXED" and "SFIXED" types. The data types "UNRESOLVED_UFIXED" and "UFIXED" can be mixed freely as they are subtypes of each other. This is also true of the "UNRESOLVED_SFIXED" and "SFIXED" type.

Note that in the compatibility package “UNRESOLVED_UFIXED” is a subtype of “UFIXED” and “UNRESOLVED_SFIXED” is a subtype of “SFIXED”, both of which are in reality resolved types.

Literals & Type conversions:

Conversion functions have been created for INTEGER, REAL, SIGNED, and UNSIGNED types. These conversion functions can be called with two different sets of parameters:

```
a <= to_sfixed (-3.125, 7, -6);
b <= to_sfixed (inp1, b); -- returns “inp1” sized the same as “b”
```

You can also say:

```
y <= to_ufixed (6.5, 4, -5);
```

where “4” is the upper index, and “—5” is the lower index; so you could also say:

```
y <= to_ufixed (6.5, y'high, y'low);
```

further, there are also versions of these functions using the “size_res” parameter, so you can also say:

```
y <= to_ufixed (6.5, y);
```

The signed version uses a two’s complement to represent a negative number, just like the “numeric_std” package.

Any non-zero index range is valid. Thus:

```
signal z : ufixed (-2 downto -3);
z <= "11"; -- 0.011 = 0.375
signal x : sfixed (4 downto 1);
y <= "111";  -- 1110.0 = -2
```

to_signed and to_unsigned are also overloaded to take the “size_res” parameter as well as a “size”. Rounding and saturation rules also apply on these functions.

Sizing Rules

The data widths in the fixed-point package were designed so that there is no possibility of an overflow. This is a departure from the “numeric_std” model, which simply throws away underflow and overflow bits.

For unsigned fixed point:

Operation	Result Range
A + B	Max(A'left, B'left)+1 downto Min(A'right, B'right)
A - B	Max(A'left, B'left)+1 downto Min(A'right, B'right)
A * B	A'left + B'left+1 downto A'right + B'right
A rem B	Min(A’left, B'left) downto Min(A’right, B'right)
Signed /	A'left - B'right+1 downto A'right - B'left
Signed A mod B	Min(A'left, B’left) downto Min(A'right, B’right)
Signed Reciprocal(A)	-A'right downto -A'left-1
Abs(A)	A’left +1 downto A’right
- A	A’left +1 downto A’right
Unsigned /	A'left - B'right downto A'right - B'left -1
Unsigned A mod B	B’left downto Min(A'right, B’right)
Unsigned Reciprocal(A)	-A'right +1 downto -A'left

Unsigned Example:

```
signal x : ufixed (7 downto –3);
signal y : ufixed (2 downto –9);
```
If we multiply x by y we would get a signal which would be:
```
x * y = ufixed (7+2+1 downto –3+(-9)) or ufixed (10 downto –12);
```

Signed Example:
```
signal x : sfixed (-1 downto –3);
signal y : sfixed (3 downto 1);
```
If we divide x by y we would get a signal which would be:
```
x/y = sfixed (-1-1+1 downto–3-3)
```
or
```
sfixed (-1 downto –6);
```

Don't worry, you do not need to memorize all of these sizes. You can use the "resize" function, or there are special functions called "ufixed_high", "ufixed_low", "sfixed_high", and "sfixed_low" designed to figure out the size for you. Example:
```
variable a : sfixed (5 downto –3);
variable b : sfixed (7 downto –9);
variable adivb : sfixed (sfixed_high (5, -3, '/', 7, -9)
                 downto sfixed_low (5, -3, '/', 7, -9));
Begin
adivb <= a / b; -- signed fixed point divide
```

You can also say:
```
Signal adivb:sfixed (sfixed_high (a'high, a'low, '/', b'high, b'low)
              downto sfixed_low (a'high, a'low, '/', b'high, b'low));
```
Or
```
Signal adivb:sfixed(sfixed_high (a,'/',b) downto sfixed_low (a,'/',b));
```

The "resize" function can be used to fix the size of the output. However, rounding and saturate rules are applied:

```
x <= resize (x * y, x'high, x'low);
```

What about an accumulator? An accumulator is a fixed-width number to which you can add continually. To implement an accumulator in the fixed-point packages, you can use the "resize" on the output of the add, or you can use the add_carry procedure as follows:

```
Signal ACC : ufixed (7 downto -3);
…..
add_carry ( L      => ACC,
            R      => X,
            C_in   => '0',
            Result => ACC,
            C_out  => open);
```

The divide function is defined as follows:
```
  function divide (
    l, r        : sfixed;
    round_style : BOOLEAN := fixed_round_style;
    guard_bits  : NATURAL := fixed_guard_bits)
    return sfixed;
```
The output is sized with the same rules as the "/" operator. The function allows you to override the number of guard bits and the rounding operation. Note that the output size is calculated so that overflow is not possible.

The reciprocal function is defined in a very similar manor to the divide function:

```
function reciprocal (
  arg         : ufixed;
  round_style : BOOLEAN := fixed_round_style;
  guard_bits  : NATURAL := fixed_guard_bits)
  return ufixed;
```

This function performs a “1/X” function, with the output vector following the sizing rules as noted above. This function is very useful for dividing by a constant, example:

```
A := B/Cons;
```

can be rewritten as:

```
A := B*reciprocal(Cons);
```

because a multiply uses less logic than a divide, this can save you significant hardware resources.

Rounding & Saturation:

```
X <= resize ( arg            => X + 1,
              left_index     => X'high,
              right_index    => X'low,
              round_style    => fixed_truncate,
              overflow_style => fixed_wrap);
```

"round_style" defaults to fixed_round (true) that turns on the rounding routines. If false (fixed_truncate), the number is truncated. Rounding is done by first looking to see if the MSB of the remainder is a “1”, AND the LSB of the unrounded result is a “1” or the lower bits of the remainder include a “1”, the result will be rounded. This is similar to the floating-point “round_nearest” style. The down side is that ALL of the bits are included in the decision to round.

"overflow_style" default to fixed_saturate (true) that returns the maximum possible number, if the number is too large to represent. You can also set this to “fixed_wrap” (false) a “wrap” routine is used that simply truncates the top bits. Unlike the way it is done in “numeric_std”, the sign bit is not preserved when wrapping. Thus, it is possible to get positive result when resizing a negative number in this mode.

Finally, “guard_bits” defaults to “fixed_guard_bits”, which defaults to 3. Guard bits are used in the rounding routines. If guard is set to 0, the rounding is automatically turned off. These extra bits are added to the end of the numbers in the division and “to_real” functions to make the numbers more accurate.

Note that the default for both overflow_style and round_style is “true.” This is true for all functions that use these parameters.

Overloading:

The following operations are defined for ufixed:

+, –, *, /, rem, mod, =, /=, <, >, >=, <=, sll, srl, rol, ror, sla, sra

The following operations are defined for sfixed:

+, –, *, /, rem, mod, =, /=, <, >, >=, <=, sll, srl, rol, ror, sla, sra, abs, - (unary)

All of the operators are overloaded for “real” and “integer” data types. In each case, the number is converted into fixed point before the operation is done. Thus, the fixed-point operand must be of a format large enough to accommodate the converted input or a “vector-truncated” warning is produced. In these functions, “overflow_style” is set to “fixed_saturate” or “true”.

Thus you can say:

```
Signal x : sfixed (4 downto –5);
Signal y : real;
…
z := x + y;
```

In the case where an operation is performed that includes both a fixed-point number and an integer or real, the sizing rules are modified. For a real number, the real is converted to a fixed-point number that is the same size as the fixed-point number that has been passed as the other argument. Thus, in the above example:

```
z := x + sfixed(y, 4, -5);
```

would be called, which would result in Z being an "sfixed (5 downto –5)" type. This is also true for integers.

Shift operators are functionally the same as the IEEE-1076-1993 shift operators with the exception of the arithmetic shift operations. An arithmetic shift ("sra", or "sla") on an unsigned number is the same as a logical shift. An arithmetic shift on a signed number is a logical shift, if you are shifting left, and an arithmetic shift (sign bit replicated), if you are shifting right.

The "scalb" function can be used to losslessly multiply or divide any number by a power of two. Example:

```
constant half : ufixed (2 downto –2) := "00010" – 000.10
variable two : ufixed (5 downto 0);
variable someval : ufixed (5 downto –5);
begin
two := scalb(half, 2); -- Will return "00010." Or 2.0
someval := resize (scalb (half, X), someval'high, someval'low);
```

All of the standard compare functions are implemented in these packages. =, /=, <, >, >=, <= perform in a similar way to the numeric_std functions. If values of two different lengths are given, then the inputs are resized and the compare is then made.

The "maximum" and "minimum" functions do a compare operation and return the appropriate value. These functions are not overloaded for integer and real inputs. The size of the inputs does not need to match.

The "find_leftmost" and "find_rightmost" functions are used to find the most significant bit or least significant bit of a fixed-point number. The function looks like the following:

```
function find_leftmost (arg : ufixed; y : STD_ULOGIC) return INTEGER;
```

In this case, "y" can be any "std_ulogic" value. These functions search for the first occurrence of "y" in the fixed-point number. "find_leftmost" starts at the MSB (arg'high) and goes down. "find_leftmost" starts at the LSB (arg'low) and goes up. If that value is not found in the "find_leftmost" function, "arg'low-1" is returned. If the value is not found in the "find_rightmost" function, "arg'high+1" is returned. Note that "find_leftmost (uf1, '1') will return the integer log (base 2) of "uf1".

"to_01" , "to_X01", "Is_X", "to_X01Z" and "to_UX01" are similar in function to the std_logic_1164 and numeric_std functions with the same name.

Most synthesis tools do not support any I/O format other than "std_logic_vector" and "std_logic". Thus, functions have been created to convert between std_logic_vector and ufixed or sfixed and vice versa:

```
uf7_3 <= to_ufixed (slv7, uf7_3'high, uf7_3'low);
```

and

```
slv7 <= to_slv (uf7_3);
```

One of the changes made to all packages in VHDL-2006 is that the read and write routines for all data types are now defined in the same package that defines that type. Thus, the READ, WRITE, HREAD,

HWRITE, OREAD, and OWRITE routines are defined for fixed-point data types. A "." `separator` is added between the integer part and the fractional part of the fixed-point number. Therefore the "6.5" example from above will write as "00110.10000", which you can also read back into that data type.

New to vhdl-2006 are the functions "to_string", "to_ostring", and "to_hstring". These are very useful in "assert" statements. Example:

```
assert x=y
  report to_string(x) & " /= " & to_string(y) severity error;
```

Or, if you prefer to see the numbers as "real" numbers, you can use:

```
assert x=y
  report to_string(to_real(x)) & " /= " & to_string(to_real(y))
  severity error;
```

These days, MathWorks Simulink is the most common way to define a fixed-point DSP algorithm. Simulink seems to be a major step into the past because it is schematic based. In Simulink, an unsigned fixed-point number is described as ufix[14,10], which specifies a 14-bit long word with 10 bits after the fraction. This translates into "ufixed (3 downto –10)" in the unsigned fixed-point type. The Simulink "sfix" notation translates much better because of the extra sign bit that must be generated. Sfix(14, 10) will translate into "sfixed(3 downto –10) in the notation of the "fixed_pkg".

Package Generics:

These packages are done using something new in VHDL-200X called package generics.
"fixed_generic_pkg.vhd" contains the following:

```
use IEEE.fixed_float_types.all;
package fixed_generic_pkg is
  generic (
    -- Round style, fixed_round (do rounding), fixed_truncate (truncate)
    fixed_round_style    : fixed_round_style_type    := fixed_round;
    -- Overflow style, fixed_saturate (largest possible number),
    --   fixed_wrap (discard high bits)
    fixed_overflow_style : fixed_overflow_style_type := fixed_saturate;
    -- Guard bits are added to the bottom of some operation for rounding.
    -- any natural number (including 0) are valid.
    fixed_guard_bits     : NATURAL := 3;
    -- when "false" issue warnings, when "true" be silent.
    no_warning           : BOOLEAN := false
    );
```

Due to the way package generics work, "fixed_generic_pkg" can not be instantiated. However another package called "fixed_pkg" is provided. This package looks like the following:

```
package fixed_pkg is
  new work.fixed_generic_pkg
  generic map (
    fixed_round_style    => IEEE.fixed_float_types.fixed_round,
    fixed_overflow_style => IEEE.fixed_float_types.fixed_saturate,
    fixed_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 fixed point package if the defaults are not what they desire.

Example:
I don't want to round (takes up too much logic), I want to truncate numbers, not saturate them, I don't need guard bits on my division routines, and I want those silly "metavalue detected" warnings turned off. Easy:

```
package my_fixed_pkg is
  new ieee.fixed_generic_pkg
  generic map (
     -- Truncate, don't round
    fixed_round_style    => IEEE.fixed_float_types.fixed_truncate,
    -- wrap, don't saturate
    fixed_overflow_style => IEEE.fixed_float_types.fixed_wrap,
    fixed_guard_bits     => 0,     -- Don't need the extra guard bits
    no_warning           => true   -- turn warnings off
    );
```

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

```
use IEEE.fixed_pkg.all;
entity sin is
  port (
    arg      : in  ufixed (1 downto -16);
    clk, rst : in  std_ulogic;
    res      : out ufixed (1 downto -11));
end entity sin;
```

Architecture:

```
architecture structure of sin is
  component fixed_sin is
    port (
      arg      : in  work.my_fixed_pkg.ufixed (1 downto -16);
      clk, rst : in  STD_ULOGIC;
      res      : out work.my_fixed_pkg.ufixed (1 downto -11));
  end component fixed_sin;
  signal resx      : work.my_fixed_pkg.ufixed (1 downto -11);
begin
  U1: fixed_sin
    port map (
      arg => work.my_fixed_pkg.ufixed(arg),   -- convert "arg"
      clk => clk,
      rst => rst,
      res => resx);
  res <= ieee.fixed_pkg.ufixed (resx);
end architecture structure;
```

Issues:

The fixed-point math packages are based on the VHDL 1076.3 numeric_std package and use the signed and unsigned arithmetic from within that package. This makes them highly efficient because the numeric_std package is well supported by simulation and synthesis tools.

A negative or "to" index is flagged as an error by the fixed-point routines. Thus, if you define a number as "ufixed (1 to 5)" the routines will automatically error out.
String literals are also a problem. By default, if you do the following:

```
Z <= a + "011011";
```

the index of the fixed-point number is undefined. The VHDL compiler will assume that the range of this number has the range "integer'low to integer'low+5", making it a very very small number. To avoid crashing the simulator with a 32,000-bit-wide number, this will also automatically error out.

Another case to watch for is the following:

```
     signal a : sfixed (3 downto –3);
     signal b : sfixed (2 downto –4);
   begin
      b <= a;
```

In this case, you have two vectors of the same length. VHDL allows you to automatically go from one type to another as long as they are of the same base type and width. This causes problems for the fixed point routines. You can easily accidentaly turn 6.5 into 3.25 by doing this and not know that you have done it.

Index:

Operators:

"+" - Add two fixed-point numbers together, overloaded for real and integer. See output sizing rules.

"-" – Subtracts fixed-point numbers. Overloaded for real and integer. See output sizing rules. Unary version (- var1) version returns a value that is one bit larger than the input. Note that unary "-" is only implemented on "sfixed".

"*" – Multiply two fixed-point numbers together. Overloaded for real and integer. See output sizing rules.

"/" – Divides two fixed-point numbers. Overloaded for real and integer. See output sizing rules. Uses 3 guard bits and rounds the result by default. If this is not the desired functionality, then use the "divide" function, or modify the package generics.

"abs" – Absolute. Returns a result one bit larger than the input. There are two versions of this operator. Both take in an "sfixed".

"mod" – modulo. Returns the signed remainder. See sizing rules for the size of the result. Overloaded for real and integer.

"rem" – Remainder. Returns the unsigned remainder. See sizing rules for the size of the result. Overloaded for real and integer.

"sll" – Shift left logical. Left argument is ufixed or sfixed, right argument is integer. A negative right argument causes a logical right shift.

"srl" – Shift right logical. Left argument is ufixed or sfixed, right argument is integer. A negative right argument causes a logical left shift.

"rol" – Rotate logical left. Left argument is ufixed or sfixed, right argument is integer. A negative right argument causes a rotate right.

"ror" – Rotate logical right. Left argument is ufixed or sfixed, right argument is integer. A negative right argument causes a rotate left.

"sla" – Shift left arithmetic. Left argument is ufixed or sfixed, right argument is integer. A negative right argument causes right arithmetic shift. Note that a right arithmetic shift on an "sfixed" replicates the sign bit. A left shift does not replicate the LSB. Note that "x sla int" will multiply (or divide) X by a power of 2.

"sra" – Shift right arithmetic. Left argument is ufixed or sfixed, right argument is integer. A negative right argument causes left arithmetic shift. Note that a right arithmetic shift on an "sfixed" replicates the sign bit. A left shift does not replicate the LSB. Note that "x sra int" will divide (or multiply) X by a power of 2.

“=” – equal. Overloaded for real and integer. Returns a “false” if any “X” is found. Integers are converted to fixed point with to_fixed (arg, max(a’high+1, 0), 0), Reals are converted with to_fixed (arg, a’high+1, a’low) and rounded.

“/=”– not equal. Overloaded for real and integer. Returns a “true” if any “X” is found.. Integers are converted to fixed point with to_fixed (arg, max(a’high+1, 0), 0), Reals are converted with to_fixed (arg, a’high+1, a’low) and rounded.

“<”– less than. Overloaded for real and integer. Returns a “false” if any “X” is found . Integers are converted to fixed point with to_fixed (arg, max(a’high+1, 0), 0), Reals are converted with to_fixed (arg, a’high+1, a’low) and rounded.

“>”– greater than. Overloaded for real and integer. Returns a “false” if any “X” is found. . Integers are converted to fixed point with to_fixed (arg, max(a’high+1, 0), 0), Reals are converted with to_fixed (arg, a’high+1, a’low) and rounded.

“<=”– less than or equal to. Overloaded for real and integer. Returns a “false” if any “X” is found. Integers are converted to fixed point with to_fixed (arg, max(a’high+1, 0), 0), Reals are converted with to_fixed (arg, a’high+1, a’low) and rounded.

“>=”– greater than or equal to. Overloaded for real and integer. Returns a “false” if any “X” is found. Integers are converted to fixed point with to_fixed (arg, max(a’high+1, 0), 0), Reals are converted with to_fixed (arg, a’high+1, a’low) and rounded.

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

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

“?<” - Returns an “X” if a metavalue is in either number, a “1” if L is less than R, otherwise ‘0”. The VHDL-93 compatible version of this function is “\?<\”

“?<=” - Returns an “X” if a metavalue is in either number, a “1” if L is less than or equal to R, otherwise ‘0”. The VHDL-93 compatible version of this function is “\?<=\”

“?>” - Returns an “X” if a metavalue is in either number, a “1” if L is greater than R, otherwise ‘0”. The VHDL-93 compatible version of this function is “\?>\”

“?>=” - Returns an “X” if a metavalue is in either number, a “1” if L is greater than or equal to R, otherwise ‘0”. 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:
Find_leftmost – find most significant bit. Inputs: arg (ufixed or sfixed), y : std_ulogic. Returns the integer index of the first occurrence of "y" in the vector "arg" starting from the MSB. Arg'low-1 is returned if "y" is not found. Note that find_msb(x, '1') will return the integer log base 2 of the input "x".

Find_rightmost – find least significant bit. Inputs: arg (ufixed or sfixed), y : std_ulogic. Returns the integer index of the first occurrence of "y" in the vector "arg" starting from the LSB. Arg'high+1 is returned if "y" is not found.

Divide – arithmetic divide. Functionally identical to the "/" operator, but with two extra parameters. Inputs: l, r (both ufixed or sfixed), Parameters: guard_bits : natural, round_style : Boolean. See output sizing rules. "guard_bits" defaults to 3, these are extra bits that are added to the end of the divide routine to maintain precision when rounding. "round_style" (default true) is a Boolean that can be used to turn on or off rounding. If rounding is set to false, then the guard bits are ignored.

Reciprocal: Performs a 1/arg function. Inputs: arg (ufixed or sfixed), guard_bits : natural, round_style : Boolean. See output sizing rules. "guard_bits" defaults to 3, these are extra bits that are added to the end of the divide routine to maintain precision when rounding. "round_style" (default true) is a Boolean that can be used to turn on or off rounding. If rounding is set to false, then the guard bits are ignored.

Minimum – returns the minimum of the two inputs (both either ufixed or sfixed) by performing a ">" operation.

maximum– returns the maximum of the two inputs (both either ufixed or sfixed) by performing a ">" operation.

Std_match – Performs a "numeric_std.std_match" function (allows you to use "-" values on the inputs).

add_carry – This is a VHDL procedure which takes in two arguments (L and R) as well as a carry_in (C_IN) and return a carry out (C_OUT) as well as a result of the same length as the combined width of L and R. Note that this routine can be use as an accumulator.

Scalb – Inputs are ufixed or sfixed, with an integer or signed input. The Scalb function moved the index of the fixed point number, having the effect of multiplying or dividing by a power of two.

Conversion functions:
Resize – Changes the size of a "ufixed" or "sfixed" (larger or smaller). Inputs: arg (ufixed or sfixed), left_index and right_index (integer) OR size_res (same type as "arg").
Parameters: round_style : Boolean (true), saturate_style : Boolean (true). Output: resized "ufixed" or "sfixed".

To_ufixed – Converts to the "ufixed" type.

To_ufixed (std_logic_vector) Inputs: arg (std_logic_vector) , left_index and right_index (integer) OR size_res (ufixed). – This is a conversion function that will convert a std_logic_vector into a ufixed with the same width. A warning is produced if the width is incorrect.

To_ufixed (std_ulogic_vector) Inputs: arg (std_ulogic_vector) , left_index and right_index (integer) OR size_res (ufixed). – This is a conversion function that will convert a std_ulogic_vector into a ufixed with the same width. A warning is produced if the width is incorrect.

To_ufixed (unsigned) Inputs: arg (unsigned) , left_index and right_index (integer) OR size_res (ufixed). Parameters: round_style : Boolean, saturate_style – Converts an unsigned into a "ufixed"

To_ufixed (unsigned) Inputs: arg (unsigned) – Converts an unsigned into a ufixed of the same size with the left_index being arg'length-1 and the right_index being 0.

To_ufixed (real) Inputs: arg (real) , left_index and right_index (integer) OR size_res (ufixed). Parameters: round_style : Boolean, saturate_style – Converts a real into a "ufixed". If the input is negative then and error is produced and 0 is returned.

To_ufixed (integer) Inputs: arg (natural) , left_index and right_index (integer) OR size_res (ufixed). Parameters: round_style : Boolean, saturate_style – Converts a integer into a "ufixed"

To_ufixed (sfixed) Inputs: arg (sfixed) - Converts and sfixed into a ufixed of the same size. Performs an "abs" function if the input is negative.

To_sfixed – Converts to the "sfixed" type.

To_sfixed (std_logic_vector) Inputs: arg (std_logic_vector) , left_index and right_index (integer) OR size_res (sfixed). Parameters: round_style : Boolean, saturate_style – This is a conversion function which will convert a std_logic_vector into a sfixed with the same width. A warning is produced if the width is incorrect.

To_sfixed (std_ulogic_vector) Inputs: arg (std_ulogic_vector) , left_index and right_index (integer) OR size_res (sfixed). Parameters: round_style : Boolean, saturate_style – This is a conversion function which will convert a std_ulogic_vector into a sfixed with the same width. A warning is produced if the width is incorrect.

To_sfixed (signed) Inputs: arg (signed) , left_index and right_index (integer) OR size_res (ufixed). Parameters: round_style : Boolean, saturate_style – Converts a signed into a "sfixed"

To_sfixed (signed) Inputs: arg (signed) – Converts a signed into a sfixed of the same size with the left_index being arg'length-1 and the right_index being 0.

To_sfixed (real) Inputs: arg (real) , left_index and right_index (integer) OR size_res (ufixed). Parameters: round_style : Boolean, saturate_style – Converts a real into a “sfixed”.

To_sfixed (integer) Inputs: arg (integer) , left_index and right_index (integer) OR size_res (ufixed). Parameters: round_style : Boolean, saturate_style – Converts a integer into a “sfixed”

To_sfixed (ufixed) Inputs: arg (ufixed) – Converts a “ufixed” into an “sfixed”. The result grows by one bit, so the resulting vector will be sfixed (arg’high+1 downto arg’low).

To_unsigned - Inputs: arg (ufixed) and size (natural) OR size_res (unsigned). Parameters: round_style : Boolean, saturate_style – Converts a ufixed into an unsigned. This does not produce a “vector truncated” warning as the ieee.numeric_std functions do.

To_signed - Inputs: arg (sfixed) and size (natural) OR size_res (signed). Parameters: round_style : Boolean, saturate_style – Converts a sfixed into an signed. This does not produce a “vector truncated” warning as the ieee.numeric_std functions do.

To_real – inputs: arg (ufixed or sfixed). Converts a fixed-point number into a real number.

To_integer – inputs: arg (ufixed or sfixed), Parameters: round_style and saturate_style : Boolean. Converts a fixed-point number into an integer

Add_sign – Inputs: arg(ufixed). Converts an unsigned fixed-point number (ufixed) into a signed fixed-point number (sfixed). This function grows the result by 1. There is an “abs” function that converts a sfixed back to a ufixed.

To_slv – Inputs: arg (ufixed or sfixed). Converts a fixed-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 (ufixed or sfixed). Converts a fixed-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_01 – Inputs (s: ufixed or sfixed). Parameters: XMAP : std_ulogic. Converts metavalues in the vector S to the XMAP state (defaults to 0).

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

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

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

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

Sizing functions:

Each of these functions take as a parameter a character that is used to describe the operation to be performed. They work as follows (table 1):

operation	Operator assumed
'+'	"+"
'-'	"-"
'*'	"*"
'/'	"/", divide
'1'	reciprocal
'M' or 'm'	"mod", modulo
'R' or 'r'	"rem", remainder
'A' or 'a'	abs
'N' or 'n'	"-" - unary
others	index

Ufixed_high – Inputs: left_index, right_index: integer (bounds of the left argument) or size_res: ufixed, operation : character, left_index2, right_index2 : integer (bounds of the left argument) or size_res2: ufixed. This function us used to compute the width of an operation. If "operation" is set to "+", then the result of this function will be the 'high value of the computed result. Any values for "operation" other than those defined in table 1 cause the "left_index" to be returned.

Ufixed_low– Inputs: left_index, right_index: integer (bounds of the left argument) or size_res: ufixed, operation : character, left_index2, right_index2 : integer (bounds of the left argument) or size_res2: ufixed. This function us used to compute the width of an operation. If "operation" is set to "+", then the result of this function will be the 'low value of the computed result. Any values for "operation" other than those defined in table 1 cause the "right_index" to be returned.

Sfixed_high– Inputs: left_index, right_index: integer (bounds of the left argument) or size_res: sfixed, operation : character, left_index2, right_index2 : integer (bounds of the left argument) or size_res2: sfixed. This function us used to compute the width of an operation. If "operation" is set to "+", then the result of this function will be the 'high value of the computed result. Any values for "operation" other than those defined in table 1 cause the "left_index" to be returned.

Sfixed_low– Inputs: left_index, right_index: integer (bounds of the left argument) or size_res: sfixed, operation : character, left_index2, right_index2 : integer (bounds of the left argument) or size_res2: sfixed. This function us used to compute the width of an operation. If "operation" is set to "+", then the result of this function will be the 'low value of the computed result. Any values for "operation" other than those defined in table 1 cause the "right_index" to be returned.

To_ufix – Similar to "to_ufixed", but with natural arguments. Thus to_ufix ("00100", 5, 3); = "00.100", or 0.5.
To_sfix – Similar to "to_sfixed", but with natural arguments. The sign bit is understood, thus: to_sfix("00100", 4, 3) = "00.100" or 0.5.
Ufix_high – Similar to "ufixed_high", but with natural numbers.
Ufix_low – Similar to "ufixed_low", but with natural numbers.
Sfix_high – Similar to "sfixed_high", but with signed numbers.
Sfix_low – Similar to "sfixed_low", but with signed numbers.

Textio Functions:

Write – Similar to the textio “write” procedure. Automatically puts in a decimal point where needed. If the range of the input number does not include the “0” index then the number is extended until it does before writing.

Read – Similar to the textio “read” procedure. If a decimal point 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. In this case the number is treated as two different numbers with a decimal point between them. Both sides of the number are padded so that the number winds up with a range that is divisible by 3 on both sides. A “.” is written out to separate the integer portion from the fraction.

Oread – Octal read. The number being red is treated as two different numbers with an optional decimal point between them. If the number being red does not have a size divisible by 3 on both sides of the decimal place, then both sides are padded until it does. If a “.” is found in the input string then the index is checked to make sure that it is in the correct place.

Hwrite - Hex write. In this case the number is treated as two different numbers with a decimal point between them. Both sides of the number are padded so that the number winds up with a range that is divisible by 4 on both sides. A “.” is written out to separate the integer portion from the fraction.

Hread– hex read. The number being red is treated as two different numbers with an optional decimal point between them. If the number being red does not have a size divisible by 4 on both sides of the decimal place, then both sides are padded until it does. If a “.” is found in the input string then the index is checked to make sure that it is in the correct place.

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 an octal value with a decimal point. The padding rules of the owrite procedure apply to this function.

To_hstring – Similar to to_string, but returns a hex value with a decimal point. The padding fules of the hwrite procedure apply to this function.

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

```
  Signal a : ufixed (3 downto –3);
Begin
   A <= from_string (“0000.000”, a’high, a’low);
   A <= from_string (“0001.000”, a);
   A <= from_string (“0000.100”); -- Works only if size is exact.
```

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. The “oread” padding rules apply in this function.

From_hstring – Same as “from_string”, but uses hex numbers. The “hread” padding fules apply in this function.