Arrays

Arrays are collections of objects of the same type. They can be one-dimensional (1D), two-dimensional (2D), or one-dimensional-by-one-dimensional (1Dx1D).

They can also be of higher dimensions, but then they are generally not synthesizable.

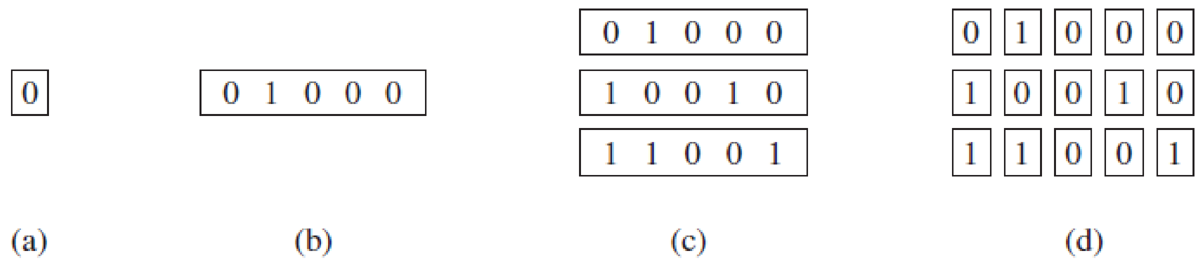


Illustration of (a) scalar, (b) 1D, (c) 1Dx1D, and (d) 2D data arrays.

TYPE type_name IS ARRAY (specification) OF data_type;

SIGNAL signal_name: type_name [:= initial_value];

Example:

TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D array

TYPE matrix IS ARRAY (0 TO 3) OF row; -- 1Dx1D array

SIGNAL x: matrix; -- 1Dx1D signal

Example: Another 1Dx1D array.

Another way of constructing the 1Dx1D array above would be the following:

TYPE matrix IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);

Example: 2D array.

The array below is truly two-dimensional. Notice that its construction is not based on vectors, but rather entirely on scalars.

TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;

-- 2D array

Example: Array initialization.

The initial value of a SIGNAL or VARIABLE is optional.

However, when initialization is required, it can be done as in the examples below.

... :="0001"; -- for 1D array

... :=('0','0','0','1') -- for 1D array

... :=(('0','1','1','1'), ('1','1','1','0')); -- for 1Dx1D or

-- 2D array

Example: Legal and illegal array assignments.

The assignments in this example are based on the following type definitions and signal declarations:

TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;

-- 1D array

TYPE array1 IS ARRAY (0 TO 3) OF row;

-- 1Dx1D array

TYPE array2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);

-- 1Dx1D

TYPE array3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;

```
-- 2D array
SIGNAL x: row;
SIGNAL y: array1;
SIGNAL v: array2;
SIGNAL w: array3;

--------- Legal scalar assignments: ---------------
-- The scalar (single bit) assignments below are all legal,
-- because the "base" (scalar) type is STD_LOGIC for all signals
-- (x,y,v,w).
x(0) <= y(1)(2); -- notice two pairs of parenthesis
-- (y is 1Dx1D)
x(1) <= v(2)(3); -- two pairs of parenthesis (v is 1Dx1D)
x(2) <= w(2,1); -- a single pair of parenthesis (w is 2D)
y(1)(1) <= x(6);
y(2)(0) <= v(0)(0);
y(0)(0) <= w(3,3);
w(1,1) <= x(7);
w(3,0) <= v(0)(3);
--------- Vector assignments: ---------------------
x <= y(0); -- legal (same data types: ROW)
x <= v(1); -- illegal (type mismatch: ROW x
-- STD_LOGIC_VECTOR)
x <= w(2); -- illegal (w must have 2D index)
x <= w(2, 2 DOWNTO 0); -- illegal (type mismatch: ROW x
-- STD_LOGIC)
v(0) <= w(2, 2 DOWNTO 0); -- illegal (mismatch: STD_LOGIC_VECTOR
-- x STD_LOGIC)
v(0) <= w(2); -- illegal (w must have 2D index)
y(1) <= v(3); -- illegal (type mismatch: ROW x
-- STD_LOGIC_VECTOR)
y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0); -- legal (same type,
-- same size)
v(1)(7 DOWNTO 3) <= v(2)(4 DOWNTO 0); -- legal (same type,
-- same size)
w(1, 5 DOWNTO 1) <= v(2)(4 DOWNTO 0); -- illegal (type mismatch)
```

Port Array

Since TYPE declarations are not allowed in an ENTITY, the solution is to declare user-defined data types in a PACKAGE, which will then be visible to the whole design (thus including the ENTITY).

```
------- Package: --------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------
PACKAGE my_data_types IS
TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
STD_LOGIC_VECTOR(7 DOWNTO 0);
END my_data_types;
--------------------------------------------
```

```
------- Main code: ------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_data_types.all; -- user-defined package
--------------------------
ENTITY mux IS
PORT (inp: IN VECTOR_ARRAY (0 TO 3);
... );
END mux;
... ;
------------------------------------------
```

A user-defined data type, called vector_array was created, which can contain an indefinite number of vectors of size eight bits each (NATURAL RANGE <> signifies that the range is not fixed, with the only restriction that it must fall within the NATURAL range, which goes from 0 to +2,147,483,647).

Signed and Unsigned Data Types

These types are defined in the std_logic_arith package of the ieee library.

Examples:

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

An UNSIGNED value is a number never lower than zero. For example, ‘‘0101’’ represents the decimal 5, while ‘‘1101’’ signifies 13. If type SIGNED is used instead, the value can be positive or negative (in two’s complement format). Therefore, ‘‘0101’’ would represent the decimal 5, while ‘‘1101’’ would mean -3.

SIGNED and UNSIGNED data types are intended mainly for arithmetic operations, that is, contrary to STD_LOGIC_VECTOR, they accept arithmetic operations. On the other hand, logical operations are not allowed.

Example: Legal and illegal operations with signed/unsigned data types.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; -- extra package necessary
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
...
v <= a + b; -- legal (arithmetic operation OK)
w <= a AND b; -- illegal (logical operation not OK)
```

Example: Legal and illegal operations with std_logic_vector.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all; -- no extra package required
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
...
v <= a + b; -- illegal (arithmetic operation not OK)
w <= a AND b; -- legal (logical operation OK)
```

The ieee library provides two packages, std_logic_signed and std_logic_unsigned, which allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED or UNSIGNED, respectively.

Example: Arithmetic operations with std_logic_vector.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all; -- extra package included
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b; -- legal (arithmetic operation OK), unsigned
w <= a AND b; -- legal (logical operation OK)
```

Data Conversion

```
TYPE long IS INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO 10;
SIGNAL x : short;
SIGNAL y : long;
...
y <= 2*x + 5; -- error, type mismatch
y <= long(2*x + 5); -- OK, result converted into type long
```

Several data conversion functions can be found in the std_logic_arith package of the ieee library. They are:

* conv_integer(p) : Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER value. Notice that STD_LOGIC_ VECTOR is not included.

*conv_unsigned(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED value with size b bits.

* conv_signed(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to a SIGNED value with size b bits.

* conv_std_logic_vector(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_LOGIC to a STD_LOGIC_VECTOR value with size b bits.

Example: Data conversion.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
y <= CONV_STD_LOGIC_VECTOR ((a+b), 8);
-- Legal operation: a+b is converted from UNSIGNED to an
-- 8-bit STD_LOGIC_VECTOR value, then assigned to y.
```

Example:

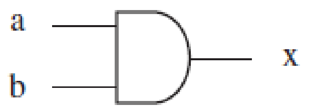
```
TYPE byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D array
TYPE mem1 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC; -- 2D array
TYPE mem2 IS ARRAY (0 TO 3) OF byte; -- 1Dx1D array
TYPE mem3 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(0 TO 7); -- 1Dx1D
-- array
SIGNAL a: STD_LOGIC; -- scalar signal
SIGNAL b: BIT; -- scalar signal
SIGNAL x: byte; -- 1D signal
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0); -- 1D signal
SIGNAL v: BIT_VECTOR (3 DOWNTO 0); -- 1D signal
SIGNAL z: STD_LOGIC_VECTOR (x'HIGH DOWNTO 0); -- 1D signal
SIGNAL w1: mem1; -- 2D signal
SIGNAL w2: mem2; -- 1Dx1D signal
SIGNAL w3: mem3; -- 1Dx1D signal
-------- Legal scalar assignments: ---------------------
x(2) <= a; -- same types (STD_LOGIC), correct indexing
y(0) <= x(0); -- same types (STD_LOGIC), correct indexing
z(7) <= x(5); -- same types (STD_LOGIC), correct indexing
b <= v(3); -- same types (BIT), correct indexing
```

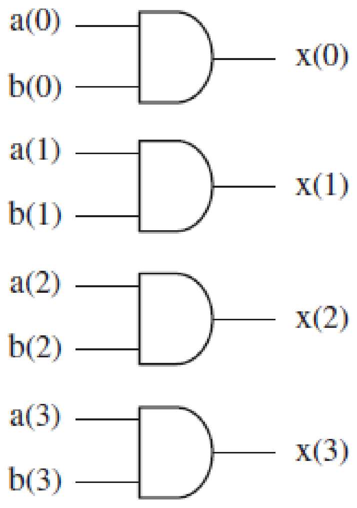
```
w1(0,0) <= x(3); -- same types (STD_LOGIC), correct indexing


w1(2,5) <= y(7); -- same types (STD_LOGIC), correct indexing
w2(0)(0) <= x(2); -- same types (STD_LOGIC), correct indexing
w2(2)(5) <= y(7); -- same types (STD_LOGIC), correct indexing
w1(2,5) <= w2(3)(7); -- same types (STD_LOGIC), correct indexing
------- Illegal scalar assignments: --------------------
b <= a; -- type mismatch (BIT x STD_LOGIC)
w1(0)(2) <= x(2); -- index of w1 must be 2D
w2(2,0) <= a; -- index of w2 must be 1Dx1D
------- Legal vector assignments: ----------------------
x <= "11111110";
y <= ('1','1','1','1','1','1','0','Z');
z <= "11111" & "000";
x <= (OTHERS => '1');
y <= (7 =>'0', 1 =>'0', OTHERS => '1');
z <= y;
y(2 DOWNTO 0) <= z(6 DOWNTO 4);
w2(0)(7 DOWNTO 0) <= "11110000";
w3(2) <= y;
z <= w3(1);
z(5 DOWNTO 0) <= w3(1)(2 TO 7);
w3(1) <= "00000000";
w3(1) <= (OTHERS => '0');
w2 <= ((OTHERS=>'0'),(OTHERS=>'0'),(OTHERS=>'0'),(OTHERS=>'0'));
```

```
w3 <= ("11111100", ('0','0','0','0','Z','Z','Z','Z',),
(OTHERS=>'0'), (OTHERS=>'0'));
w1 <= ((OTHERS=>'Z'), "11110000" ,"11110000", (OTHERS=>'0'));
------ Illegal array assignments: ----------------------
x <= y; -- type mismatch
y(5 TO 7) <= z(6 DOWNTO 0); -- wrong direction of y
w1 <= (OTHERS => '1'); -- w1 is a 2D array
w1(0, 7 DOWNTO 0) <="11111111"; -- w1 is a 2D array
w2 <= (OTHERS => 'Z'); -- w2 is a 1Dx1D array
w2(0, 7 DOWNTO 0) <= "11110000"; -- index should be 1Dx1D
```

Example:





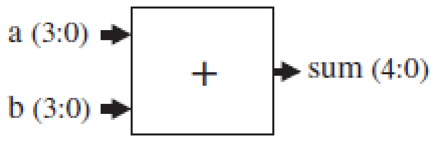
```
----------------------------------------------
ENTITY and2 IS
PORT (a, b: IN BIT;
x: OUT BIT);
END and2;
```

```
----------------------------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
x <= a AND b;
END and2;
----------------------------------------------
```

```
----------------------------------------------
ENTITY and2 IS
PORT (a, b: IN BIT_VECTOR (0 TO 3);
x: OUT BIT_VECTOR (0 TO 3));
END and2;
----------------------------------------------
```

```
----------------------------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
x <= a AND b;
END and2;
----------------------------------------------
```

Example:



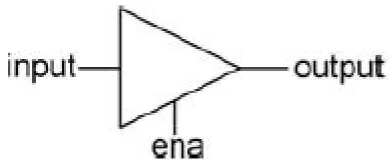
```
----- Solution 1: in/out=SIGNED ----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-----------------------------------------
ENTITY adder1 IS
PORT ( a, b : IN SIGNED (3 DOWNTO 0);
sum : OUT SIGNED (4 DOWNTO 0));
END adder1;
-----------------------------------------
ARCHITECTURE adder1 OF adder1 IS
BEGIN
sum <= a + b;
END adder1;
-----------------------------------------
```

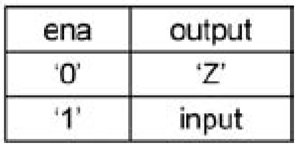
```
------ Solution 2: out=INTEGER -----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-----------------------------------------
ENTITY adder2 IS
PORT ( a, b : IN SIGNED (3 DOWNTO 0);
sum : OUT INTEGER RANGE -16 TO 15);
END adder2;
```

```
10 ----------------------------------------
11 ARCHITECTURE adder2 OF adder2 IS
12 BEGIN
13 sum <= CONV_INTEGER(a + b);
14 END adder2;
15 ----------------------------------------
```

Notice also the inclusion of the std_logic_arith package (line 4 of each solution), which specifies the SIGNED data type. Recall that a SIGNED value is represented like a vector; that is, similar to STD_LOGIC_VECTOR, not like an INTEGER.

Example: Tri-state Buffer



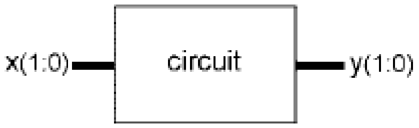


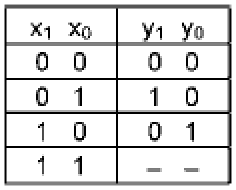
ena	output
'0'	'Z'
'1'	input

```
----------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------------------
ENTITY tri_state IS
PORT (input, ena: IN STD_LOGIC;
         output: OUT STD_LOGIC);
END ENTITY;
----------------------------------------
ARCHITECTURE behavior_of_tri_state OF tri_state IS
BEGIN
```

```
        output <= input WHEN ena='1' ELSE 'Z';
 END ARCHITECTURE;
 ----------------------------------------
```

Example:





x_1 x_0	y_1 y_0
0 0	0 0
0 1	1 0
1 0	0 1
1 1	– –

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY circuit IS
        PORT (x: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        y: OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END ENTITY;

ARCHITECTURE circuit OF circuit IS
BEGIN
        y <= "00"  WHEN x="00" ELSE
                "01" WHEN x="10" ELSE
                "10" WHEN x="01" ELSE
                "--";
END ARCHITECTURE;
```

Fixed- and Floating-Point Types

Fixed-Point Types:

The main fixed-point types are the following:

```
TYPE UFIXED IS ARRAY (INTEGER RANGE <>) OF STD_LOGIC; --unsigned
TYPE SFIXED IS ARRAY (INTEGER RANGE <>) OF STD_LOGIC; --signed
```

The files needed to use such types in VHDL 2008 are (see IEEE 1076-2008 standard):

fixed_ pkg.vhdl (contains the package fixed_ pkg)

fixed_generic_ pkg.vhdl (contains the package fixed_generic_ pkg)

fixed_generic_ pkg-body.vhdl (contains the package body of fixed_generic_ pkg)

fixed_ float_types.vhdl (contains the package fixed_ float_types)

Examples with unsigned fixed-point numbers:

```
x: SIGNAL UFIXED(2 DOWNTO -3); --this is "xxx.xxx"
y: SIGNAL UFIXED(4 DOWNTO -1); --this is "yyyyy.y"
z: SIGNAL UFIXED(-2 DOWNTO -3); --this is "0.0zz"
...
x <= "100011"; --1x22+0x21+0x20+0x2-1+1x2- 2+1x2- 3=4.375
y <= "100011"; --1x24+0x23+0x22+0x21+1x20+1x2-1=17.5
z <= "10"; --1x2- 2+0x2-3=0.25
```

x: range from 0 to 7.875 in steps of 0.125 (a total of $2^{bits} = 2^6 = 64$ values).

y: range from 0 to 31.5 in steps of 0.5 (a total of $2^{bits} = 2^6 = 64$ values).

z: range from 0 to 0.375 in steps of 0.125 (a total of $2^{bits} = 2^2 = 4$ values).

Examples with signed fixed-point numbers:

```
x: SIGNAL SFIXED(2 DOWNTO -3); --this is "xxx.xxx"
y: SIGNAL SFIXED(4 DOWNTO -1); --this is "yyyyy.y"
z: SIGNAL SFIXED(-2 DOWNTO -3); --this is "0.0zz"
...
x <= "100011"; --100.011 -> 2's compl=011.101 -> -3.625 (or 4.375-8=-3.625)
y <= "100011"; --10001.1 -> 2's compl=01110.1 -> -14.5 (or 17.5-32=-14.5)
z <= "10"; --0.010 = +0.25
```

A large set of operators (details about operators are given in chapter 4) and type conversion functions are defined for these new types. A simplified list is presented below.

a) Operators:

Logical: NOT, AND, NAND, OR, NOR, XOR, XNOR

Arithmetic: +, -, *, /, ABS, REM, MOD, ADD_CARRY, etc.

Comparison: =, /=, >, <, <=, >=, MAXIMUM, MINIMUM

Shift: SLL, SRL, SLA, SRA, ROR, ROL, SHIFT_LEFT, SHIFT_RIGHT

and others…..

b) Type-conversion functions:

TO_UFIXED, TO_SFIXED, TO_UNSIGNED, TO_SIGNED, TO_SLV (same as TO_STDLOGICVECTOR), TO_INTEGER, TO_REAL, TO_STRING, etc.

Example:

```
x: SIGNAL UFIXED(4 DOWNTO -3); --"xxxxx.xxx"
y: SIGNAL SFIXED(4 DOWNTO -3); --"yyyyy.yyy"
z: SIGNAL SFIXED(5 DOWNTO -3); --"zzzzzz.zzz"
...
x <= TO_UFIXED(17.5, 4, -3); --converts 17.5 to UFIXED; result="10001100"
z <= -y; --unary "-" (only for signed); result=2's compl. of y
```

Example:

```
 LIBRARY ieee_proposed;
 USE ieee_proposed.fixed_pkg.all;
 ENTITY fixed IS
           PORT (a, b: IN SFIXED(3 DOWNTO -3);
          x: OUT SFIXED(4 DOWNTO -3);
          y: OUT SFIXED(7 DOWNTO -6));
 END ENTITY;

 ARCHITECTURE fixed OF fixed IS
 BEGIN
          x <= a + b;
           y <= a * b;
 END ARCHITECTURE;
```

The arithmetic operations are constructed with vector sizes such that overflow is always prevented. Some examples are shown below, where a and b are the inputs

a + b, a - b: Range is max(a'LEFT, b'LEFT) + 1 DOWNTO min(a'RIGHT, b'RIGHT)

a*b: Range is a'LEFT + b'LEFT + 1 DOWNTO a'RIGHT + b'RIGHT

a/b unsigned: Range is a'LEFT - b'RIGHT DOWNTO a'RIGHT + b'LEFT - 1

a/b signed: Range is a'LEFT - b'RIGHT + 1 DOWNTO a'RIGHT + b'LEFT

-a (unary "-", for signed only): Range is a'LEFT + 1 DOWNTO a'RIGHT

Floating-Point Types

The main floating-point type and subtypes are the following:

TYPE FLOAT IS ARRAY (INTEGER RANGE <>) OF STD_LOGIC; --generic length

SUBTYPE FLOAT32 IS FLOAT(8 DOWNTO -23); --32-bit FP of IEEE 754

SUBTYPE FLOAT64 IS FLOAT(11 DOWNTO -52); --64-bit FP of IEEE 754

SUBTYPE FLOAT128 IS FLOAT(15 DOWNTO -112); --128-bit FP of IEEE 754

The files needed to use such types in VHDL 2008 are (see IEEE 1076-2008 Standard):

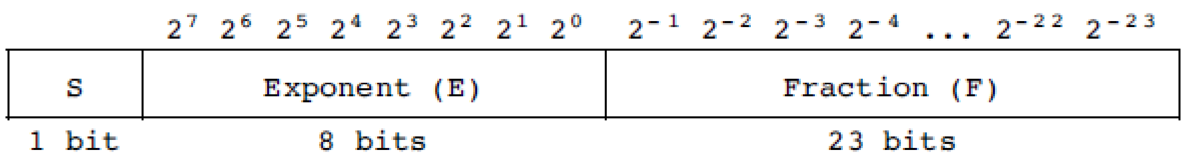
- float_ pkg.vhdl (contains the package float_ pkg)

- float_generic_ pkg.vhdl (contains the package float_generic_ pkg)

- float_generic_ pkg-body.vhdl (contains the package body of float_generic_ pkg)

- fixed_ float_types.vhdl. (already mentioned)

The representation of floating-point numbers in the IEEE 754 standard obeys the structure illustrated in the figure below (for the 32-bit case):



Calling x the stored number, its value is given by $x = (-1)^S(1+F)2^{E-N}$, where S is the sign (0 when positive, 1 when negative), F is the fraction, E is the exponent, and N is a normalization factor given by $N = (E_{max}+1)/2 - 1$ (for example, $N = 127$ when the exponent has 8 bits or $N = 1023$ when it has 11 bits).

Example (with 3-bit exponent and 4-bit fraction, hence a total of 8 bits):

x <= "10010110"; --(1)(001)(0110) = -(1+0.375)21 - 3 = -0.34375

x <= "01101000"; --(0)(110)(1000) = +(1+0.5)26 -3 = 12.0

In VHDL, the minimum length of a floating-point number is 7 bits, with the following distribution: 1 bit for the sign, 3 bits for the exponent, and 3 bits for the fraction.

As with fixed-point, a large set of operators and type-conversion functions are defined for the floating-point types. A simplified list is presented below.

a) Operators:

Logical: NOT, AND, NAND, OR, NOR, XOR, XNOR

Arithmetic: +, -, *, /, ABS, REM, MOD, ADD_CARRY, etc.

Comparison: =, /=, >, <, <=, >=, MAXIMUM, MINIMUM

others…..

b) Type-conversion functions:

TO_FLOAT, TO_FLOAT32, TO_FLOAT64, TO_FLOAT128, TO_UNSIGNED,

TO_SIGNED, TO_SLV, etc.

Example:

```
SIGNAL x: FLOAT(3 DOWNTO -4); --(S)(EEE)(FFFF)
SIGNAL y: FLOAT(3 DOWNTO -4);
SIGNAL z: STD_LOGIC_VECTOR(7 DOWNTO 0);
...
--Convert 12.0 to float with format (S)(EEE)(FFFF):
x <= TO_FLOAT(12.0, 3, 4); --result="01101000"
x <= TO_FLOAT(12.0, x); --same as above
--Convert std_logic_vector to float with format (S)(EEE)(FFFF):
y <= TO_FLOAT(z, 3, 4);
```

Example:

```
 LIBRARY ieee_proposed;
 USE ieee_proposed.float_pkg.all;

 ENTITY floating IS
          PORT (a, b: IN FLOAT(3 DOWNTO -4);
          x, y: OUT FLOAT(3 DOWNTO -4));
 END ENTITY;

 ARCHITECTURE floating OF floating IS
 BEGIN
          x <= TO_FLOAT(0.34375, 3, 4) + a + b;
         y <= a * b;
 END ARCHITECTURE;
```