Draft Standard VHDL Synthesis Package

December 14, 1994 (date to be deleted from final version)

1. Overview

1.1 Scope

This standard defines standard practices for synthesizing binary digital electronic circuits from VHDL source code. It covers the following areas:

a) The hardware interpretation of values belonging to the BIT and BOOLEAN types defined by IEEE Std 1076-1993 and to the STD_ULOGIC type defined by IEEE Std 1164-1993.

b) The handling of initial value expressions for variables and default expressions for signals.

c) A function (STD_MATCH) that provides "don't care" or "wild card" testing of values based on the STD_ULOGIC type.

d) Standard functions for representing sensitivity to the edge of a signal.

e) Two packages that define vector types for representing signed and unsigned arithmetic values, and that define arithmetic, shift, and type conversion operations on those types.

RATIONALE: A paragraph that begins with the word "RATIONALE:" is not part of the standard but is included in the draft for convenience in revisiting decisions already made. Such paragraphs will be deleted from the published standard.

1.2 Terminology

The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*). The word *should* is used to indicate that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*). The word *may* indicates a course of action permissible within the limits of the standard (*may* equals *is permitted*).[1]

For the purposes of this standard, the *synthesis tool* is any system, process, or tool that interprets VHDL source code in accordance with the terms of this standard as a description of an electronic circuit and produces an alternative description or form of that circuit. The *user* is a person, system, process, or tool that generates the VHDL source code to be processed by the synthesis tool.

[1] The wording of this paragraph is adapted from the *IEEE Standards Style Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, c. 1992.

The synthesis tool *accepts* a VHDL construct if it allows that construct to be legal input; it *interprets* the construct (or provides an *interpretation* of the construct) by producing something that represents the construct. The synthesis tool is not required to provide an interpretation for every construct that it accepts, but only for those for which an interpretation is specified by this standard.

2. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision shall apply.

IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual.

IEEE Std 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_-logic_1164).

3. Definitions

3.1 argument: An expression occurring as the actual value in a function call or procedure call.

3.2 arithmetic operation: An operation for which the VHDL operator is +, -, *, /, **mod**, **rem**, **abs**, or **.

3.3 assignment reference: The occurrence of a literal or other expression as the waveform element of a signal assignment statement or as the right-hand side expression of a variable assignment statement.

3.4 don't care value: The enumeration literal `'-'` of the type STD_ULOGIC defined by IEEE Std 1164-1993.

3.5 equality relation: A VHDL relational expression in which the relational operator is =.

3.6 high-impedance value: The enumeration literal `'Z'` of the type STD_ULOGIC defined by IEEE Std 1164-1993.

3.7 inequality relation: A VHDL relational expression in which the relational operator is /=.

3.8 logical operation: An operation for which the VHDL operator is **and**, **or**, **nand**, **nor**, **xor**, **xnor**, or **not**.

3.9 metalogical value: One of the enumeration literals `'U'`, `'X'`, `'W'`, or `'-'` of the type STD_ULOGIC defined by IEEE Std 1164-1993.

3.10 ordering relation: A VHDL relational expression in which the relational operator is <, <=, >, or >=.

3.11 shift operation: An operation for which the VHDL operator is **sll**, **srl**, **sla**, **sra**, **rol**, or **ror**.

3.12 standard logic type: The type STD_ULOGIC defined by IEEE Std 1164-1993, or any type derived from it, including in particular one-dimensional arrays of STD_ULOGIC or of one of its subtypes.

3.13 synthesis tool: Any system, process, or tool that interprets VHDL source code as a description of an electronic circuit and derives an alternate description of that circuit.

3.14 well-defined: Containing no metalogical or high-impedance element values.

4. Interpretation of the standard logic types

This section defines how the synthesis tool shall interpret values of the standard logic types defined by IEEE Std 1164-1993 and of the BIT and BOOLEAN types defined by IEEE Std 1076-1993. Simulation tools, however, shall continue to interpret these values according to the standards in which they are defined.

4.1 The STD_LOGIC_1164 values

IEEE Std 1164-1993 defines the standard logic type:

```
type STD_ULOGIC is ( 'U', -- Uninitialized
                     'X', -- Forcing Unknown
                     '0', -- Forcing 0
                     '1', -- Forcing 1
                     'Z', -- High Impedance
                     'W', -- Weak Unknown
                     'L', -- Weak 0
                     'H', -- Weak 1
                     '-'  -- Don't care
                   );
```

The *logical values* `'1'`, `'H'`, `'0'`, and `'L'` are interpreted as representing one of two logic levels, where each logic level represents one of two distinct voltage ranges in the circuit to be synthesized.

IEEE Std 1164-1993 also defines a resolution function named RESOLVED and a subtype STD_LOGIC that is derived from STD_ULOGIC by using RESOLVED. The resolution function RESOLVED treats the values `'0'` and `'1'` as *forcing values* that override the *weak values* `'L'` and `'H'` when multiple sources drive the same signal.

The values `'U'`, `'X'`, `'W'`, and `'-'` are *metalogical values*; they define the behavior of the model itself rather than the behavior of the hardware being synthesized. The value `'U'` represents the value of an object before it is explicitly assigned a value during simulation; the values `'X'` and `'W'` represent forcing and weak values, respectively, for which the model is not able to distinguish between logic levels.

The value `'-'` is also called the *don't care value*. This standard treats it in the same way as the other metalogical values except when it is furnished as an argument to the STD_MATCH functions in the IEEE.NUMERIC_STD package. The STD_MATCH functions use `'-'` to implement a "match all" or "wild card" matching.

The value `'Z'` is called the *high impedance value*, and represents the condition of a signal source when that source makes no effective contribution to the resolved value of the signal.

4.2 Statically-valued expressions

Wherever the synthesis tool accepts a reference to a statically-valued named constant, it shall treat that constant as the equivalent of the associated static expression.

> RATIONALE: We had considered saying that the synthesis tool replaces all statically-valued expressions (that it accepts) by the equivalent literals, but this goes beyond what any synthesis tool that we're aware of currently does, and it seems wise to limit our remarks to what synthesis tools actually do. The adjective "statically-valued" is included so as not to require replacement of named constants declared in subprograms, since they are not statically-valued.

4.3 Interpretation of logic values

This clause describes the interpretations of logic values occurring as literals (or in literals) after the synthesis tool has replaced named constants by their corresponding values.

4.3.1 Interpretation of the forcing and weak values ('0', '1', 'L', 'H', FALSE, TRUE)

The synthesis tool shall interpret the following values as representing a logic value 0:

- The BIT value `'0'`.
- The BOOLEAN value FALSE.
- The STD_ULOGIC values `'0'` and `'L'`.

It shall interpret the following values as representing a logic value 1:

- The BIT value `'1'`.
- The BOOLEAN value TRUE.
- The STD_ULOGIC value `'1'` and `'H'`.

The standard makes no restriction as to the interpretation of the relative strength of values.

> RATIONALE: Some current synthesis tools treat `'L'` and `'H'` as equivalent to `'0'` and `'1'`, but future tools may interpret them in a manner that recognizes that they have a different strength than the forcing values. Until a consensus emerges as to how to handle them, it is better for this standard not to restrict their usage.

4.3.2 Interpretation of the metalogical values ('U', 'W', 'X', '-')

4.3.2.1 Metalogical values in relational expressions

If the VHDL source code includes an equality relation (=) for which one operand is a static metalogical value and for which the other operand is not a static value, the synthesis tool shall interpret the equality relation as equivalent to the BOOLEAN value FALSE. If one operand of an equality relation is a vector, and one element of that vector is a static metalogical value, the synthesis tool shall interpret the entire equality relation as equivalent to the BOOLEAN value FALSE.

> RATIONALE: A non-static one-bit value is synthesized as some real value, and hence in a real circuit always has either a high or low value. Hence a comparison of a non-static value to a literal metalogical value would never be true in a real circuit. Such an equality test is generally used to check for an error or unexpected value, and hence should be treated as FALSE in synthesizing a real circuit.

> RATIONALE: The don't care value `'-'` is treated in the same way as a metalogical value, rather than as a "wild card," because the VHDL relational operators are implicitly defined by the STD_LOGIC_1164 package, and those definitions do not treat `'-'` as a wild card. Wild card behavior for `'-'` is provided by the STD_-MATCH function defined by this standard.

The synthesis tool shall interpret an inequality relation (/=) for which one operand is or contains a static metalogical value, and for which the other operand is not a static value, as equivalent to the BOOLEAN value TRUE.

The synthesis tool shall treat an ordering relation for which at least one operand is or contains a static metalogical value as an error.

> RATIONALE: An ordering relation in which both operands contain static metalogical values may actually be unambiguous. For example, for simulation purposes, the expression `"0X10" >= "00X0"` is definitely

TRUE while the expression `"01XX" < "00XX"` is definitely FALSE. However, neither expression has a clear purpose for synthesis, so this standard requires the synthesis tool to treat them as errors, just as in the case when only one operand contains a static metalogical value.

4.3.2.2 Metalogical values in a choice in a case statement

If a metalogical value occurs as a choice or as an element of a choice in a case statement that is interpreted by the synthesis tool, the synthesis tool shall interpret the choice as one that can never occur. That is, the interpretation that is generated is not required to contain any constructs corresponding to the presence or absence of the sequence of statements associated with the choice.

Whenever the synthesis tool interprets a case statement alternative that associates multiple choices with a single sequence of statements, it shall produce an interpretation consistent with associating the sequence of statements with each choice individually.

Whenever the synthesis tool interprets a selected signal assignment statement, it shall interpret the selected signal assignment statement as if it were the case statement in the equivalent process as defined by IEEE Std 1076-1993.

4.3.2.3 Metalogical values in logical, arithmetic, and shift operations

When a static metalogical value occurs as all of, or one element of, an operand to a logical, arithmetic, or shift operation, and when the other operand to the operation is not a static value, the synthesis tool shall treat the operation as an error.

4.3.2.4 Metalogical values in concatenate operations

If a static metalogical value occurs as all of, or as one element of, an operand to the concatenate (&) operator, the synthesis tool shall treat it as if it had occurred as the corresponding element of the expression formed by the concatenate operation.

4.3.2.5 Metalogical values in type conversion and sign-extension functions

If a static metalogical value occurs as all of, or as one element of, the value argument to a type conversion or sign-extension function, the synthesis tool shall treat it as if it had occurred as the corresponding element of the expression formed by the function call.

4.3.2.6 Metalogical values used in assignment references

The synthesis tool shall accept a static metalogical value used as all of, or as one element of, an assignment reference, but is not required to provide any particular interpretation of that metalogical value.

RATIONALE: The committee had considered requiring the synthesis tool to replace the metalogical value by either ’0’ or ’1’ where it occurred in the VHDL source code, but a counter example showed that this was not always desirable. In particular, consider a D flip-flop with both SET and RESET inputs. If both SET and RESET are active, the model may specify that the output Q should be a metalogical value, and hence an expression such as “Q xor not Q” is also unknown. If one replaces the metalogical value by either ’0’ or ’1’, the expression “Q xor not Q” has the value ’1’. This would require the synthesis tool to synthesize a circuit for which the Q and Q-bar outputs of a real flip-flop have opposing values even when SET and RESET are both active. But the user may wish instead to allow a flip-flop for which Q and Q-bar are not necessarily complementary under this condition. So it seems wiser not to require the synthesis tool to treat a metalogical value in an assignment reference as any particular value.

4.3.3 Interpretation of the high impedance value ('Z')

If the static value 'Z' occurs as an assignment reference in a signal assignment statement, the synthesis tool shall interpret the assignment as implying the equivalent of a three-state buffer that is disabled by the condition controlling the assignment. The output of the three-state buffer is the target of the assignment. The input of the three-state buffer is the logic network that represents the value of the target apart from any assignments to 'Z'.

If the 'Z' occurs as one or more elements of an assignment reference in a signal assignment statement, the synthesis tool shall interpret each such occurrence as implying the equivalent of a three-state buffer in the manner defined by the preceding paragraph.

This standard does not specify an interpretation when a static value 'Z' occurs as all of, or one bit of, an assignment reference in a variable assignment statement.

> RATIONALE: A variable, by definition, is never a resolved object, so it does not make sense to instantiate a three-state buffer. However, a particular synthesis tool may wish either to do so, or to save the value and propagate it to an assignment to a resolved signal. Therefore, this standard does not restrict the interpretation.

Whenever a static high-impedance value occurs in any context other than an assignment reference, the synthesis tool shall treat it as equivalent to a static metalogical value.

5. Initial value expressions and default expressions

The synthesis tool shall accept an initial value expression occurring in a variable declaration in a process declarative part, and shall also accept a default expression occurring in a signal declaration, but the synthesis tool is not required to provide an interpretation for either construct. The user should avoid having the desired behavior of the VHDL source code depend on such expressions.

> RATIONALE: Although a synthesis tool may use an initial value expression or default expression to infer a "power-on" or "start-up" reset, many target technologies do not support such resets. Nor is there a consensus that a synthesis tool should always interpret such expressions as a reset value even if the target technology allows it.

This standard does not specify whether or not the synthesis tool should accept an initial value expression occurring in a variable declaration in a subprogram declarative part.

6. The STD_MATCH function

The NUMERIC_STD package defined by this standard defines functions named STD_MATCH to provide wild card matching for the don't care value. Whenever the STD_MATCH function compares two arguments which are STD_ULOGIC values, it returns TRUE if and only if

- both values are well-defined and the values are the same, or
- one value is '0' and the other is 'L', or
- one value is '1' and the other is 'H', or
- at least one of the values is the don't care value ('-').

Whenever the STD_MATCH function compares two arguments which are vectors of STD_ULOGIC values, it returns TRUE if and only if

a) the operands have the same length, and
b) STD_MATCH applied to each pair of matching elements returns TRUE.

When one of the arguments to the STD_MATCH function is a static value and the other is not, the synthesis tool shall interpret the call to the STD_MATCH function as equivalent to an equality test on matching elements of the arguments, excepting those elements of the static value which are equal to '-'.

NOTE—If any argument value passed to STD_MATCH is or contains a metalogical or high-impedance value other than '-', the function returns FALSE.

7. Signal edge detection

Wherever the synthesis tool interprets a particular expression as the edge of a signal, it shall also interpret the function RISING_EDGE as representing a rising edge and the function FALLING_EDGE as representing a falling edge, where RISING_EDGE and FALLING_EDGE are the functions declared either by the package STD_LOGIC_1164 of Std 1076-1993 or by the NUMERIC_BIT package of this standard.

8. Standard arithmetic packages

Two VHDL packages are defined by this standard. The NUMERIC_BIT package is based on type BIT, while the NUMERIC_STD package is based on type STD_ULOGIC. Simulations based on the subprograms of the NUMERIC_BIT package ordinarily require less execution time, because the subprograms do not have to deal with operands containing metalogical or high-impedance values. Use of the subprograms of the NUMERIC_STD package allow simulation to detect the propagation or generation of metalogical values.

To facilitate changing from one package to the other, most of the subprograms declared in one package are also declared for corresponding arguments in the other. The exceptions are

a) The NUMERIC_BIT package declares the functions RISING_EDGE and FALLING_EDGE; the corresponding functions for STD_ULOGIC are declared by the STD_LOGIC_1164 package.
b) The NUMERIC_STD package declares functions for the logical operators (**and**, **or**, **not**, **nand**, **nor**, **xor**, and **xnor**) applied to SIGNED and UNSIGNED types, whereas these operators are defined implicitly by VHDL for the SIGNED and UNSIGNED types in the NUMERIC_BIT package.
c) The NUMERIC_STD package declares the STD_MATCH functions which give special treatment to the don't-care value, whereas the BIT-based types of the NUMERIC_BIT package have no don't-care values.

Both packages interpret the type UNSIGNED as an unsigned binary integer with the most significant bit on the left, and the type SIGNED as a two's-complement binary integer with the most significant bit on the left. In particular, a one-element SIGNED value is interpreted as the integer values -1 and 0.

If a null array is furnished as an input argument to any subprogram declared by NUMERIC_BIT or NUMERIC_STD, the synthesis tool shall treat it as an error.

All vector return values which are not null array values are normalized so that the direction of the index range is **downto** and the right bound is 0. A vector return value which is a null array has the index range "0 **downto** 1".

The package declarations use the following format to declare each function:

```
-- Id: <id_nr>
function <designator> (<formal_parameter_list>) return <type_mark>;
-- Result Subtype: <subtype_indication>
-- Result: <description of function>
```

The elements of this format have the following meanings:

<id_nr>
A unique identifier of the form <?>. A corresponding identifier appears at the beginning of the corresponding function body in the package body for the same package.

<designator>
The function designator as defined by Std 1076-1993.

<formal_parameter_list>
The formal parameter list for the function as defined by Std 1076-1993.

<type_mark>
A type mark denoting the result subtype of the function as defined by Std 1076-1993.

<subtype_indication>
The subtype of the value returned by the function. If the result subtype of the function denotes an unconstrained vector subtype, <subtype_indication> also includes an index constraint defining the index range of the returned value in terms of sizes and values of the input parameters. <subtype_indication> is syntactically a subtype indication as defined by Std 1076-1993.

<description of function>
An English language description of the operation performed by the function.

The order of the function declarations within the package declarations is:

NUMERIC_BIT	**NUMERIC_STD**
abs, unary -	**abs**, unary -
binary +, -, *, /, **rem**, **mod**	binary +, -, *, /, **rem**, **mod**
>, <, <=, >=, =, /=	>, <, <=, >=, =, /=
SHIFT_LEFT and SHIFT_RIGHT	SHIFT_LEFT and SHIFT_RIGHT
ROTATE_LEFT and ROTATE_RIGHT	ROTATE_LEFT and ROTATE_RIGHT
RESIZE	RESIZE
TO_INTEGER, TO_UNSIGNED, TO_SIGNED, TO_BITVECTOR	TO_INTEGER, TO_UNSIGNED, TO_SIGNED, TO_STDLOGICVECTOR
	not, **and**, **or**, **nand**, **nor**, **xor**, **xnor**
RISING_EDGE, FALLING_EDGE	STD_MATCH

RATIONALE: The packages do not provide implementations of the VHDL ’93 shift operators (**sll**, **srl**, **sla**, **sra**, **rol**, and **ror**) for the SIGNED and UNSIGNED types. The reason for this is that the VHDL ’93 arithmetic shift functions do not interpret the arithmetic shift operators in a manner consistent with interpreting the operations as multiplying or dividing by powers of two, as do the SHIFT_LEFT and SHIFT_RIGHT functions. The names differ in order to emphasize the distinction.

RATIONALE: The names of the ROTATE_LEFT and ROTATE_RIGHT functions were chosen to be consistent with the names SHIFT_LEFT and SHIFT_RIGHT.

Both packages shall be analyzed into the library symbolically named IEEE.

8.1 Allowable modifications

Vendors of tools conforming to this standard shall not modify the package declarations for NUMERIC_BIT or NUMERIC_STD. However, a vendor may provide package bodies for either or both packages in which subprograms are rewritten for more efficient simulation or synthesis, provided that the behavior of the rewritten subprograms remains the same under simulation. The behavior of the original and rewritten subprograms are the same if for any combination of input values they return the same return values. The text of messages associated with assertions may differ in the rewritten subprogram.

The package bodies for both packages declare a constant named NO_WARNING that has the value FALSE. The user may set NO_WARNING to TRUE and reanalyze the package body to suppress warning messages generated by calls to the functions in these packages. For this reason,

- A tool vendor who rewrites the package body shall preserve the declaration of the NO_WARNING constant to allow the user to suppress warnings by editing and reanalyzing the package body.
- A simulation tool vendor who provides a preanalyzed version of the package body should also provide a mechanism for suppressing warning messages generated by the package functions.

8.2 The NUMERIC_BIT package

(Insert NUMERIC_BIT package here)

8.3 The NUMERIC_STD package

(Insert NUMERIC_STD package here.)

Annex A
Notes on the Package Functions

(informative)

This annex provides notes on functions included in the NUMERIC_BIT and NUMERIC_STD packages.

A.1 Arithmetic operator functions

A.1.1: When the operand to an **abs** or unary - function from either package has the maximum negative value, the result is the maximum negative value of the same size. This means, for example, that

```
- "1000" -> "1000"
```

A.1.2: When both operands to a binary arithmetic functions + or - are either SIGNED or UNSIGNED, the function returns a value with the same number of elements (bits) as the larger of the two operands. If one operand is SIGNED or UNSIGNED and the other is INTEGER or NATURAL, the function returns a value with the same number of elements as the vector operand. Thus, these functions do not return an extra bit to represent a carry, borrow, or overflow value, nor do they generate a warning if a carry, borrow, or overflow occurs.

The choice not to generate a carry or borrow (and not to generate a warning) makes it easier to represent counter operations in the VHDL source code via assignments such as

```
A := A + 1;
```

or

```
B <= B - "1";
```

A.1.3: If an operand to a NUMERIC_STD arithmetic function contains a metalogical or high-impedance value, the function returns a vector in which every element has the value `'X'`. The function does not report a warning or error.

A.2 Relational operator functions

A.2.1: The relational operator functions for the SIGNED and UNSIGNED types in both packages have a behavior different from the default behavior defined by Std 1076-1993 for vector types. The default behavior compares the vector elements left to right after the operands are left-justified, whereas the relational operator functions for SIGNED and UNSIGNED treat their operands as representing binary integers.

The following table compares results for relational operators when both operands are BIT_VECTORs, SIGNED values, or UNSIGNED values.

Expression	**Value when both operands are ...**		
	BIT_VECTOR	UNSIGNED	SIGNED
"001" = "00001"	FALSE	TRUE	TRUE
"001" > "00001"	TRUE	FALSE	FALSE
"100" < "01000"	FALSE	TRUE	TRUE
"010" < "10000"	TRUE	TRUE	FALSE
"100" < "00100"	FALSE	FALSE	TRUE

A.2.2: When a relational operator compares a SIGNED or UNSIGNED argument value with an INTEGER or NATURAL value, the function has the effect of converting the SIGNED or UNSIGNED argument to its equivalent universal integer value and then doing the corresponding comparison of integer values. For example,

```
(SIGNED'("111") > -8) = TRUE
```

and

```
(UNSIGNED'("111") < 8) = TRUE
```

That is, the INTEGER value may be larger in magnitude than any value that can be represented by the number of elements in the SIGNED or UNSIGNED value.

A.2.3: If an operand to any of the NUMERIC_STD relational operator functions for =, <, <=, >, or >= contains a metalogical or high-impedance value, the function returns the value FALSE. If an operand to the NUMERIC_STD relational operator function /= contains a metalogical or high-impedance value, the function returns the value TRUE.

A.3 Shift functions

A.3.1: The SHIFT_LEFT function for an UNSIGNED argument provides for multiplication by a power of 2 remaindered by the maximum size of the vector argument. In particular, if ARG is UNSIGNED and is well-defined, and if the integer values fall within the range allowed for INTEGERs,

```
TO_INTEGER(SHIFT_LEFT(ARG, COUNT)) =
  TO_INTEGER(ARG) * (2 ** COUNT) rem (2 ** ARG'LENGTH)
```

A.3.2: The SHIFT_RIGHT function for an UNSIGNED argument provides for division by a power of 2. That is, if ARG is UNSIGNED and is well-defined, and if the integer values fall with the range allowed for INTEGERs,

```
TO_INTEGER(SHIFT_RIGHT(ARG, COUNT) = TO_INTEGER(ARG) / (2 ** COUNT)
```

A.4 Type conversion functions

A.4.1: The TO_INTEGER function does not contain code to check that the SIGNED or UNSIGNED argument has an equivalent universal integer value that belongs to the range defined for the INTEGER or NATURAL subtypes. If TO_INTEGER is called with an argument value that is too large, the simulation tool may therefore detect an overflow. The user should avoid applying TO_INTEGER to argument subtypes for which the number of elements is greater than the number of bits used to represent INTEGERs in the user's simulation and synthesis tools.

A.4.2: The packages do not provide functions for converting directly between the SIGNED and UNSIGNED types. Such conversions must be performed by the user. There are several ways to convert between SIGNED and UNSIGNED types. n performing such conversions, the user must determine how to handle any possible differences in the ranges supported by SIGNED and UNSIGNED objects having the same number of elements. For example, suppose the VHDL source code contains the following declarations:

```
signal S: SIGNED(3 downto 0);
signal BIG_S: SIGNED(4 downto 0);
signal U: UNSIGNED(3 downto 0);
constant S1: SIGNED(3 downto 0) := "1000";  -- equivalent to -8
constant U1: UNSIGNED(3 downto 0) := "1100";  -- equivalent to +10
```

a) The user can use a VHDL type conversion to convert one form to another:

```
S <= SIGNED(U1);  -- U1 (= +10) gets converted to S (= -4)
U <= UNSIGNED(S1);  -- S1 (= -8) gets converted to U (= +8)
```

b) The user can add an extra bit to represent the sign when converting from UNSIGNED to SIGNED:

```
BIG_S <= SIGNED('0' & U1); -- U1 (= +10) gets converted to BIG_S (= +10)
```

c) Finally, the user can generate an error or warning when the value of one cannot be represented in the number of elements available in the other.

```
assert S >= "0000" report "Cannot convert negative value."
  severity WARNING;
U <= UNSIGNED(S);
```

A.5 Logical operator functions

A.5.1: The functions that define the application of the logical operators **and**, **or**, **nand**, **nor**, **xor**, and **xnor** to SIGNED and UNSIGNED argument values are equivalent to functions that apply the same logical operators to STD_LOGIC_VECTOR (or STD_ULOGIC_VECTOR) arguments. This equivalence includes the handling of metalogical and high-impedance element values. That is, for example, if S1 and S2 are SIGNED values of equal length,

```
S1 nand S2 = SIGNED(STD_LOGIC_VECTOR(S1) nand STD_LOGIC_VECTOR(S2))
```