Enhancements to VHDL's Packages

Jim Lewis, Director of Training, SynthWorks Design Inc, Tigard, OR

Abstract

VHDL is a strongly typed language. Success in VHDL depends on understanding the types and overloading of operators provided in the packages std_logic_1164 (IEEE standard 1164) and Numeric_Std (IEEE standard 1076.3).

Currently, enhancements for both of these packages are being finalized for the next drafts of these standards. These enhancements include:

- *logical reduction functions,*
- *array/scalar logic operations,*
- *array/scalar addition operators,*
- *TO_X01, TO_X01Z, TO_UX01, IS_X for unsigned and signed,*
- *shift operators for std_logic_vector and std_ulogic_vector*
- *unsigned arithmetic for std_logic_vector and std_ulogic_vector (new package),*
- *textio for types in std_logic_1164 and numeric_std (two new packages),*
- *floating point arithmetic (new packages)*

This paper will provide details about the new features of std_logic_1164 and numeric_std and also provides some rules of thumb for remembering the overloading. This paper also contains a mini-tutorial on these packages to help those who are not currently using numeric_std to make the transition.

New Features for Std_Logic_1164 and Numeric_Std

This section contains proposals for additions to the packages std_logic_1164, numeric_std, and numeric_bit. Although numeric_bit is not mentioned in the discussion, everything that applies to numeric_std also applies to numeric_bit.

Logical reduction operators: std_logic_1164, numeric_std

Add reduction operators for std_logic_vector, std_ulogic_vector, unsigned and unsigned. Functions for **and_reduce**, **nand_reduce**, **or_reduce**, **nor_reduce**, **xor_reduce**, and **xnor_reduce** will be defined of the following form:

function **and_reduce** (arg : std_logic_vector) return std_ulogic;

This will allow expressions such as the following to be written:

Parity <= xor_reduce (Data) and ParityEnable ;

Array/scalar logic operators: std_logic_1164, numeric_std

Overload the logic operators to support mixing an array with a scalar for std_logic_vector, std_ulogic_vector, unsigned and signed. Functions for **and**, **nand**, **or**, **nor**, **xor**, and **xnor** will be defined in the following form:

```
function "and"  ( l : std_logic_vector;  r : std_ulogic ) return  std_logic_vector;
function "and"  ( l : std_ulogic; r : std_logic_vector  ) return  std_logic_vector;
```

This solves the following common design problem:

```
signal ASel, BSel, CSel, DSel : std_logic ;
signal Y, A, B, C, D : std_logic_vector(7 downto 0) ;
. . .
Y <= (A and ASel) or (B and BSel) or  (C and CSel) or (D and DSel) ;
```

Without these functions, a common issue is to write the above code as follows:

```
Y <=   A when ASel = '1' else
       B when BSel = '1' else
       C when CSel = '1' else
       D when DSel = '1' else
       (others => '0') ;
```

When the select signals (ASel, …) are mutually exclusive, this hardware functions correctly. However, this code implies priority select logic and is inefficient from a hardware area and timing perspective.

Array/scalar addition operators: numeric_std

Overload the addition operators to support mixing an array with a scalar for unsigned and signed. Functions for "+" and "-" will be defined in the following form:

```
function "+"(L: unsigned; R: std_ulogic) return unsigned;
function "+"(L: std_ulogic; R: unsigned) return unsigned;
```

These functions facilitate writing the following add with carry in:

```
signal Cin      : std_logic ;
signal A, B     : unsigned(7 downto 0) ;
signal Y        : unsigned(8 downto 0) ;
. . .
Y <= A + B + Cin ;
```

They also facilitate writing the following conditional incrementer:

```
signal IncEn                : std_logic ;
signal IncCur, IncNext      : unsigned(7 downto 0) ;
. . .
IncNext <= IncCur + IncEn ;
```

TO_X01, TO_X01Z, TO_UX01, IS_X: numeric_std

Add the strength reduction and 'X' detection operators for unsigned and signed. Currently these operators are only supported for std_logic_vector and std_ulogic_vector. Functions for TO_X01, TO_X01Z, TO_UX01, and IS_X will be defined in the following form:

```
function  To_X01 ( s : unsigned ) return unsigned;
function  Is_X     ( s : unsigned ) return boolean;
```

These functions are useful for testbenches to handle 'X's and resistive strength driving levels. It is also appropriate to use these functions in input pad cells of ASIC and FPGA libraries. In an RTL design, logic should only generate the values '0', '1', '-', and 'X'.

Shift operators: std_logic_1164, numeric_std?

Overload shift operators for std_logic_vector and std_ulogic_vector. Functions for sll, srl, sla, sra, rol, and ror will be defined in the following form:

```
function "sll" ( l : std_logic_vector;   r : integer )  return   std_logic_vector;
function "sll" ( l : std_ulogic_vector; r : integer )  return   std_ulogic_vector;
```

Numeric_std currently supports sll, srl, rol, and ror. Support is being considered for sla and sra.

Unsigned arithmetic for std_logic_vector and std_ulogic_vector: new package

Create a new package that implements unsigned arithmetic operators for std_logic_vector and std_ulogic_vector. Tenatively the package is named numeric_unsigned. It will include all functions included in numeric_std minus the ones that are in std_logic_1164 (or planned for std_logic_1164).

A testbench is one of the places that will benefit most. Testbenches often need to perform a numeric algorithm on an object that is not numeric in a formal sense. For example, the following code shows data being written to consecutive bits in a RAM with exactly one bit set in each data word.

```
constant  CHIP1_RAM_BASE : std_logic_vector(31 downto 0) := X"40000000" ;
constant  ZERO_DATA : std_logic_vector(31 downto 0) := (others => '0') ;
. . .
for i in 0 to 31 loop
    CpuWrite(CpuRec, CHIP1_RAM_BASE + i , ZERO_DATA + 2**i ); -- subprogram call
end loop ;
```

For RTL design the existence of this package permits one of three methodology variations:

1) Strict: Use only types unsigned and signed. Do not use numeric_unsigned.
2) Semi-Strict: Use unsigned and signed for all math operations except counters.
3) Flexible: Use std_logic_vector for any unsigned operation.
   Use signed for all signed operations.

Note, this proposal does not include a numeric_signed package. Use ieee.numeric_std.signed for signed operations.

Textio for std_logic_1164 and numeric_std types: (two new packages)

Overload read and write procedures to support std_ulogic, std_logic, std_ulogic_vector, std_logic_vector, unsigned, and signed. Functions for read and write will be defined in the following forms:

```
procedure READ( L: inout LINE; VALUE out std_logic; GOOD: out BOOLEAN);
procedure READ( L: inout LINE; VALUE out std_logic);
procedure WRITE( L: inout LINE; VALUE in std_logic; JUSTIFIED: in SIDE:= RIGHT;
                 FIELD: in WIDTH:= 0);
```

Overload read and write procedures to support base operations with std_ulogic_vector, std_logic_vector, unsigned, and signed. Functions for read and write will be defined in the following forms:

```
type REPRESENTATION is ( any, binary, octal, hexadecimal );
procedure READ( L: inout LINE; VALUE out std_logic;
                GOOD: out BOOLEAN; R: in REPRESENTATION := any );
procedure WRITE( L: inout LINE; VALUE in std_logic;
                 JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH:= 0;
                 R: in REPRESENTATION := any);
```

Floating point arithmetic

A family of VHDL packages are being introduced to implement floating point arithmetic. For more information, paper [1] in this conference. Also see http://www.eda.org/fphdl.

Strong Typing, Overloading, & Conversions

Strong Typing

VHDL is a strongly typeed language. Basically this means that every value returned by an expression must be an appropriate size and type for the context in which it is being used. If A, B, and Y are signals of type ieee.numeric_std.unsigned, then the following table applies:

Expression	Size of Y
Y <= A ;	A'Length
Y <= A and B ;	A'Length = B'Length
Y <= A > B ;	Boolean
Y <= A + B ;	Maximum (A'Length, B'Length)
Y <= A + 10 ;	A'Length
Y <= A * B ;	A'Length + B'Length

Strong typing provides a limited amount of error checking of expression construction. To see how this works, consider the following code.

```
signal A8, B8, Result8      :  unsigned (7 downto 0) ;
signal Result9              :  unsigned (8 downto 0) ;
signal Result7              :  unsigned (6 downto 0) ;
. . .
Result8 <= A8 + B8 ;
Result9 <= ('0' & A8) + ('0' & B8) ;
Result7 <= A(6 downto 0) + B(6 downto 0) ;
```

For each expression above to be correct, the expression must be correctly sized to match the size of the result. If this requirement is not met, the code will not compile. This leads to an observation: With strong typing, on a bad day, you will be abused by the compiler. However, without strong typing, on a bad day, you can code errors into your design that will require a good testbench and lots of time to find.

Overloading

Another great feature that strong typing facilitates is overloading of subprograms and operators. Overloading allows a subprogram name or operator symbol to be used more than once as long as there is a way to differentiate the calls. This means that when a new type, such as float, is added to a package, it can use the same set of operator symbols (+,-,*, . . .) that are used for all the other types.

In addition to understanding the previous set of expression rules, to be successful in VHDL, you must know about the overloading provided by the packages. The following tables summarize the overloading proposed for the enhanced versions of std_logic_1164, numeric_std, and numeric_unsigned.

Operator	**Right**	**Result**	**Package**
Logic reduction	Std_logic_vector	Std_ulogic	Std_logic_1164
	Std_ulogic_vector	Std_ulogic	
	Unsigned	Std_ulogic	
	Signed	Std_ulogic	

Operator	**Left**	**Right**	**Result**	**Package**
Logic	Std_ulogic	Std_ulogic	Std_ulogic	Std_logic_1164
	Std_logic_vector	Std_logic_vector	Std_logic_vector	
	Std_logic_vector	Std_ulogic	Std_logic_vector	
	Std_ulogic	Std_logic_vector	Std_logic_vector	
	Std_ulogic_vector	Std_ulogic_vector	Std_ulogic_vector	
	Std_ulogic_vector	Std_ulogic	Std_ulogic_vector	
	Std_ulogic	Std_ulogic_vector	Std_ulogic_vector	
	Unsigned	Unsigned	Unsigned	Numeric_std
	Unsigned	Std_ulogic	Unsigned	
	Std_ulogic	Unsigned	Unsigned	
	Signed	Signed	Signed	
	Signed	Std_ulogic	Signed	
	Std_ulogic	Signed	Signed	

Operator	**Left**	**Right**	**Result**	**Package**
Comparison	Std_ulogic	Std_ulogic	boolean	*
	Std_logic_vector	Std_logic_vector	boolean	Numeric_unsigned, *
	Std_logic_vector	Integer	boolean	Numeric_unsigned
	Integer	Std_logic_vector	boolean	Numeric_unsigned
	Std_ulogic_vector	Std_ulogic_vector	boolean	Numeric_unsigned, *
	Std_ulogic_vector	Integer	boolean	Numeric_unsigned
	Integer	Std_ulogic_vector	boolean	Numeric_unsigned
	Unsigned	Unsigned	Boolean	Numeric_std
	Unsigned	Natural	boolean	Numeric_std
	Natural	Unsigned	boolean	Numeric_std
	Signed	Signed	Boolean	Numeric_std
	Signed	Integer	boolean	Numeric_std
	Integer	Signed	boolean	Numeric_std
Notes: * Implicitly created comparison operators				

Operator	Left	Right	Result	Package
Addition	Std_logic_vector	Std_logic_vector	Std_logic_vector	Numeric_unsigned
	Std_logic_vector	Integer	Std_logic_vector	Numeric_unsigned
	Integer	Std_logic_vector	Std_logic_vector	Numeric_unsigned
	Std_logic_vector	Std_ulogic	Std_logic_vector	Numeric_unsigned
	Std_ulogic	Std_logic_vector	Std_logic_vector	Numeric_unsigned
	Std_ulogic_vector	Std_ulogic_vector	Std_ulogic_vector	Numeric_unsigned
	Std_ulogic_vector	Integer	Std_ulogic_vector	Numeric_unsigned
	Integer	Std_ulogic_vector	Std_ulogic_vector	Numeric_unsigned
	Std_ulogic_vector	Std_ulogic	Std_ulogic_vector	Numeric_unsigned
	Std_ulogic	Std_ulogic_vector	Std_ulogic_vector	Numeric_unsigned
	Unsigned	Unsigned	Unsigned	Numeric_std
	Unsigned	Natural	Unsigned	Numeric_std
	Natural	Unsigned	Unsigned	Numeric_std
	Unsigned	Std_ulogic	Unsigned	Numeric_std
	Std_ulogic	Unsigned	Unsigned	Numeric_std
	Signed	Signed	Signed	Numeric_std
	Signed	Integer	Signed	Numeric_std
	Integer	Signed	Signed	Numeric_std
	Signed	Std_ulogic	Signed	Numeric_std
	Std_ulogic	Signed	Signed	Numeric_std

Operator	Left	Right	Result	Package
Multiplication	Std_logic_vector	Std_logic_vector	Std_logic_vector	Numeric_unsigned
	Std_logic_vector	Integer	Std_logic_vector	Numeric_unsigned
	Integer	Std_logic_vector	Std_logic_vector	Numeric_unsigned
	Std_ulogic_vector	Std_ulogic_vector	Std_ulogic_vector	Numeric_unsigned
	Std_ulogic_vector	Integer	Std_ulogic_vector	Numeric_unsigned
	Integer	Std_ulogic_vector	Std_ulogic_vector	Numeric_unsigned
	Unsigned	Unsigned	Unsigned	Numeric_std
	Unsigned	Natural	Unsigned	Numeric_std
	Natural	Unsigned	Unsigned	Numeric_std
	Signed	Signed	Signed	Numeric_std
	Signed	Integer	Signed	Numeric_std
	Integer	Signed	Signed	Numeric_std

The following is a generalization of the previous tables.

Operators	**Left**	**Right**	**Result**	**Notes:**
Logic	TypeA	TypeA	TypeA	Array = unsigned, signed, std_ulogic_vector, std_logic_vector TypeA = boolean, std_logic, std_ulogic, Array * for comparison operators the result is boolean
Numeric	Array	Array	Array, *	
	Array	Integer	Array, *	
	Integer	Array	Array, *	
Logic, Addition	Array	Std_ulogic	Array	
	Std_ulogic	Array	Array	
Logic Reduction		Array	Std_ulogic	

Type Conversions

In addition to understanding the available types and overloading, it is also important to know how to convert between std_ulogic, std_logic, unsigned, signed, std_logic_vector and integer. Some of these can be accomplished by built in language features and some by conversion fuctions in numeric_std.

Conversion between std_logic and std_ulogic occurs automatically. In VHDL, two types convert automatically when they are both subtypes of the same type. Std_logic is a subtype of std_ulogic and std_ulogic is a subtype of itself. Elements of signed, unsigned, and std_logic_vector are all of the type std_logic and convert automatically to std_ulogic. As a result, in the example below, only the **"AND"** function with std_ulogic arguments is required.

```
signal A_sl : std_logic ;
signal B_slv : std_logic_vector(7 downto 0) ;
signal C_uv : unsigned (7 downto 0) ;
signal D_sv : signed (7 downto 0) ;
signal Y_sul : std_ulogic ;

Y_sul <= A_sl and B_slv(0) and C_uv(1) and D_sv(2)  ;
```

Conversion between std_logic_vector, signed, and unsigned can be accomplished by type casting. In VHDL, type casting can be used to convert two equivalent sized arrays when they both have a common base type (std_logic) and their indicies have a common base type (natural). Hence, to subtract two unsigned numbers and get a signed result:

```
signal A_uv, B_uv : unsigned (7 downto 0) ;
signal Y_sv : signed (8 downto 0) ;
. . .
Y_sv <= signed('0' & A_uv) - signed('0' & B_uv) ;
```

To convert from unsigned and signed to integer use the to_integer function from numeric_std as shown in the example below:

```
signal A_uv, C_uv     : unsigned (7 downto 0) ;
signal Unsigned_int   : integer range 0 to 255 ;
signal B_sv, D_sv     : signed( 7 downto 0) ;
signal Signed_int     : integer range -128 to 127;

Unsigned_int          <=  TO_INTEGER ( A_uv ) ;
Signed_int            <=  TO_INTEGER ( B_sv ) ;
```

To convert from integer to unsigned and signed use the to_unsigned and to_signed functions from numeric_std as shown in the example below. Note the value 8 specifies the width of the array.

```
C_uv  <=  TO_UNSIGNED ( Unsigned_int, 8) ;
D_sv  <=  TO_SIGNED ( Signed_int, 8 ) ;
```

To convert between std_logic_vector and integer, use type casting plus type conversion fuctions.

```
signal E_slv   : std_logic_vector (7 downto 0) ;
signal Unsigned_int : integer range 0 to 255 ;
. . .
E_slv <= std_logic_vector( to_unsigned(Unsigned_int, E_slv'length)) ;
```

Participating In Standards

VHDL standards are IEEE standards. As a VHDL community member it is both your right and responsibility to join IEEE committees and participate in VHDL standards. If you don't participate, the changes you envision and wish for (no matter how simple or obvious) will not happen. To find out more about participating in VHDL standards go the the web links, http://www.eda.org and http://www.SynthWorks.com/VhdlLinks.htm.

Acknowlegements

This paper documents work which is the collaboration of the participants of the IEEE 1076.3 and IEEE 1164 reflectors.

References

1. Bishop, David "Floating Point for VHDL and Verilog", to be published in DVCon 2003 proceedings.
2. Lewis, Jim "Comprehensive VHDL Introduction" copyright by SynthWorks Design Inc, 1999 through 2003. Numerous examples used by permission.

About SynthWorks VHDL Training

Comprehensive VHDL Introduction 4 Days
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
Engineers learn VHDL Syntax plus basic RTL coding styles and simple procedure-based, transaction testbenches. Our designer focus ensures that your engineers will be productive in a VHDL design environment.

VHDL Coding Styles for Synthesis 4 Days
http://www.synthworks.com/vhdl_rtl_synthesis.htm
Engineers learn RTL (hardware) coding styles that produce better, faster, and smaller logic.

VHDL Testbenches and Verification 3 days
http://www.synthworks.com/vhdl_testbench_verification.htm
Engineers learn how create a transaction-based verification environment based on bus functional models.

For additional courses see: http://www.synthworks.com