



design & verification  
conference & exhibition

March 1-3, 2004

# VHDL 200X: The Future of VHDL

By Jim Lewis

SynthWorks VHDL Training

Team Leader VHDL-200X Fast Track

[jim@synthworks.com](mailto:jim@synthworks.com)

---

## VHDL-200X: The Future of VHDL

SynthWorks

- VHDL-200X is a multi-phased effort.
  - Covers the next several language revisions.
  - Fast Track = first set of enhancements. WG completion soon
- Goals:
  - Maintain VHDL style, nature, and backward compatibility
  - Leverage Industry Efforts (PSL, SystemVerilog, ...)

**Caution:** Changes presented here are work in progress

## VHDL-200X-FT: Proposals

- PSL
- IP Protection
- Type Generics
- Generics on Packages
- Context Unit
- Arrays with unconstrained arrays
- Records with unconstrained arrays
- Fixed Point Packages
- Floating Point Packages
- Process(all)
- Case Statement Updates
- Don't Care in a Case Statement
- Conditional Expression Updates
- Expressions in port maps
- Read out ports
- Stop and Finish
- Conditional and Selected assignment in sequential code
- If Expressions
- hwrite, owrite, ... hread, oread
- to\_string, to\_hstring, ...
- Sized bit string literals
- Unary Reduction Operators
- Array/Scalar Logic Operators
- Hierarchical references of signals
- Slices in array aggregates
- Std\_logic\_1164 Updates
- Numeric\_Std Updates

Much of VHDL's cumbersome syntax issues can be fixed

## PSL

- PSL will be incorporated directly into VHDL
- Implications
  - PSL Vunits are a VHDL Design Unit
  - PSL declarations (properties) can go in:
    - Packages
    - Declarative regions of entity, architecture, and block.
  - PSL directives (assert, cover, ...) are VHDL statements
    - Can be placed in any concurrent statement part.

**Note:** PSL code will not need to be placed in comments

# IP Protection and Encryption

- A pragma-based approach
- Allows IP authors to mark specific areas of VHDL code for encryption using standard algorithms.
- The proposal:
  - Defines constructs to demarcate protected envelopes in VHDL source code.
  - Defines keywords to specify algorithms and keys.
- Tools that work with encrypted IP must not reveal any details through any interface or output it generates.
  - For example, a synthesis tool should generate an encrypted netlist for any portion of a design that is encrypted.

---

# Type Generics + Generics on Packages

```
package MuxPkg is
  generic( type array_type) ;
  function Mux4 (
    Sel : std_logic_vector(1 downto 0);
    A   : array_type ;
    B   : array_type ;
    C   : array_type ;
    D   : array_type
  ) return array_type ;
end MuxPkg ;
package body MuxPkg is
  . . .
end MuxPkg ;
```

## Type Generics + Generics on Packages

- Making the Mux4 function available for both std\_logic\_vector and unsigned.

```

library ieee ;
package MuxPkg_slv is new work.MuxPkg
  Generic map (
    array_type => ieee.std_logic_1164.std_logic_vector
  ) ;

library ieee ;
package MuxPkg_unsigned is new work.MuxPkg
  Generic map (
    array_type => ieee.numeric_std.unsigned
  ) ;

```

## Context Unit = Primary Design Unit

- Allows a group of packages to be referenced by a single name

```

Context project1_Ctx is
  use std.textio.all ;
  library ieee ;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all ;
  use ieee.fixed_pkg.all ;
end ;

```

- Reference the named context unit

```

Library Lib_P1 ;
  context Lib_P1.project1_ctx ;

```

- Benefit increases as additional standard packages are created
  - Fixed Point, Floating Point, Assertion Libraries, . . .

## Arrays of Unconstrained Arrays

```
type std_logic_matrix is array (natural range <>)
  of std_logic_vector ;

-- constraining in declaration
signal A : std_logic_matrix(7 downto 0)(5 downto 0) ;

entity e is
port (
  A : std_logic_matrix(7 downto 0)(5 downto 0) ;
  . . .
) ;
```

---

## Records of Unconstrained Arrays

```
type complex is record
  a : std_logic ;
  re : signed ;
  im : signed ;
end record ;

-- constraining in declaration
signal B : complex (re(7 downto 0), im(7 downto 0)) ;
```

# Fixed Point Types

- Definitions in package, `ieee.fixed_pkg.all`

```
type ufixed is array (integer range <>) of std_logic;
type sfixed is array (integer range <>) of std_logic;
```

- For `downto` range, whole number is on the left and includes 0.

```
constant A : ufixed (3 downto -3) := "0011010000" ;

      3210 -3
      IIIIFFF
      0110100 = 0110.100 = 6.5
```

- Math is full precision math:

```
signal A, B : ufixed (3 downto -3) ;
signal Y    : ufixed (4 downto -3) ;
. . .
Y <= A + B ;
```

# Floating Point Types

- Definitions in package, `ieee.fphdl_pkg.all`

```
type fp is array (integer range <>) of std_logic;
```

- Format is Sign Bit, Exponent, Fraction

```
signal A, B, Y : fp (8 downto -23) ;

      8  76543210  12345678901234567890123
      S  EEEEEEEE  FFFFFFFFFFFFFFFFFFFFFFFF

E = Exponent has a bias of 127
F = Fraction has an implied 1 in leftmost bit

0  10000000  000000000000000000000000 = 2.0
0  10000001  101000000000000000000000 = 6.5
0  01111100  000000000000000000000000 = 0.125 = 1/8

Y <= A + B ; -- FP numbers must be same size
```

## Process (all)

- Creates a sensitivity list with all signals on sensitivity list

```
Mux3_proc : process(all)
begin
  case MuxSel is
    when "00" =>      Y <= A ;
    when "01" =>      Y <= B ;
    when "10" =>      Y <= C ;
    when others =>    Y <= 'X' ;
  end case ;
end process
```

- Benefit: Reduce mismatches between simulation and synthesis

## Simplified Case Statement

- Allow locally static expressions to contain:
  - implicitly defined operators that produce composite results
  - operators and functions defined in `std_logic_1164`, `numeric_std`, and `numeric_unsigned`.

```
constant ONE1      : unsigned := "11" ;
constant CHOICE2   : unsigned := "00" & ONE1 ;
signal A, B        : unsigned (3 downto 0) ;
. . .
process (A, B)
begin
  case A xor B is
    when "0000"      =>      Y <= "00" ;
    when CHOICE2     =>      Y <= "01" ;
    when "0110"      =>      Y <= "10" ;
    when ONE1 & "00" =>      Y <= "11" ;
    when others      =>      Y <= "XX" ;
  end case ;
end process ;
```

## Simplified Case Statement

- Although concatenation is specifically allowed, some cases will still require a type qualifier.

```

signal A, B, C, D : std_logic ;
. . .

process (A, B, C, D)
begin
  case std_logic_vector'(A & B & C & D) is
    when "0000" => Y <= "00" ;
    when "0011" => Y <= "01" ;
    when "0110" => Y <= "10" ;
    when "1100" => Y <= "11" ;
    when others => Y <= "XX" ;
  end case ;
end process ;

```

## Case With Don't Care

- Allow use of '-' in targets provided targets are non-overlapping

```

-- Priority Encoder
process (Request)
begin
  case? Request is
    when "1---" => Grant <= "1000" ;
    when "01--" => Grant <= "0100" ;
    when "001-" => Grant <= "0010" ;
    when "0001" => Grant <= "0001" ;
    when others => Grant <= "0000" ;
  end case ;
end process ;

```

**Note:** Only '-' in the case target is treated as a don't care.  
A '-' in the case? Expression will not be treated as a don't care



# Simplified Conditional Expressions

- Applies to conditional of: if, elsif, wait until, when, while

```
if (Cs1 and not nCs2 and Cs3 and Addr=X"A5") then
if (Cs1 and nCs2='0' and Cs3 and Addr=X"A5") then
if (not nWe) then
```

- Active low is represented with "not nCs2"

```
Sel <= Cs1 and not nCs2 and Cs3 ;
```

- Backward compatible with current VHDL syntax:

```
if nWe = '0' then
if (Cs1='1' and nCs2='0' and Addr=X"A5") then
if ((Cs1 and not nCs2)='1' and Addr=X"A5") then
```

# Simplified Conditional Expressions

- Implementation Part 1:

If top level of a conditional expression is either bit or std\_ulogic, then call the function condition? to do the conversion.

- This makes the following legal:

```
if (Cs1 and not nCs2 and Cs3) then
```

- Implementation Part 2:

Overload logic operators so if boolean used with bit/std\_ulogic, then the result will be bit/std\_ulogic

- Maintains accuracy by promoting true to '1' and false to '0'
- This makes the following legal:

```
if (Cs1 and not nCs2 and Cs3 and Addr=X"A5") then
DevSel1 <= Cs1 and not nCs2 and Cs3 and Addr=X"A5" ;
```

# Signal Expressions in Port Maps

```
U_UUT : UUT
  port map ( A, Y and C, B) ;
```

- Needed to avoid extra signal assignments with OVL
- If expression is not a single signal, constant, or does not qualify as a conversion function, then
  - convert it to an equivalent concurrent signal assignment
  - and it will incur a delta cycle delay

---

# Read Output Ports

- Read output ports
  - Value read will be locally driven value
- Assertions need to be able to read output ports

## Stop and Finish

- Add procedures STOP and FINISH:
  - STOP - Stops like breakpoint
  - FINISH - Terminate the simulation

```
procedure STOP;
procedure FINISH;
```

- Benefit:
  - Just stop. No annoying messages.

## Allow Conditional Assignments for Signals and Variables in Sequential Code

- Statemachine code:

```
if (FP = '1') then
    nextState <= FLASH ;
else
    nextState <= IDLE ;
end if ;
```

- Simplification (new part is that this is in a process):

```
nextState <= FLASH when (FP = '1') else IDLE ;
```

- Also support conditional variable assignment:

```
nextState := FLASH when (FP = '1') else IDLE ;
```

# Allow Selected Assignments for Signals and Variables in Sequential Code

```

signal A, B, C, D, Y : std_logic ;
signal MuxSel : std_logic_vector(1 downto 0) ;
. . .

Process(clk)
begin
  wait until Clk = '1' ;
  with MuxSel select
    Mux :=
      A when "00",
      B when "01",
      C when "10",
      D when "11",
      'X' when others ;

  Yreg <= nReset and Mux ;
end process ;

```

## IF Expressions

- Similar to conditional signal assignment ...

```

Y <= A and B if S = '1', C and D ;
Y <= (A and B) if S = '1', (C and D) ;

```

- ... except it is an expression:

```

Y <= A and (B if S = '1', C) and D ;

```

- And it can be used anywhere an expression can:

```

Signal A : integer := 7 if GEN_VAL = 1, 15 ;

with MuxSel select
  Y <= (A if Asel='1', B) when '0',
      (C if Csel='1', D) when '1',
      'X' when others ;

```

## Hwrite, Hread, Owrite, Oread

- Support Hex and Octal read & write for all bit based array types

```

procedure hwrite (
    Buf           : inout Line ;
    VALUE         : in integer;
    JUSTIFIED     : in SIDE := RIGHT;
    FIELD        : in WIDTH := 0
) ;

procedure hread (
    Buf           : inout Line ;
    VALUE         : out integer;
    Good         : out boolean
) ;

procedure oread ( . . . ) ;
procedure owrite ( . . . ) ;

```

- No new packages. Supported in base package
  - For backward compatibility std\_logic\_textio will be empty

## To String, To HString, To OString

- Create to\_string for all types.
- Create hex and octal functions for all bit based array types

```

function to_string (
    VALUE         : in std_logic_vector;
    JUSTIFIED     : in SIDE := RIGHT;
    FIELD        : in WIDTH := 0
) return string ;

function to_hstring ( . . . ) return string ;
function to_ostring ( . . . ) return string ;

```

- Formatting Output with Write (not write from TextIO):

```

write(Output, "%%ERROR data value miscompare." &
  NL & " Actual value = " & to_hstring (Data) &
  NL & " Expected value = " & to_hstring (ExpData) &
  NL & " at time: " & to_string (now, right, 12)) ;

```

## Sized Bit String Literals

- Currently hex bit string literals are a multiple of 4 in size

```
X"AA" = "10101010"
```

- Allow specification of size (and decimal bit string literals):

```
7X"7F" = "11111111"
7D"127" = "11111111"
```

- Allow specification of signed vs unsigned (extension of value):

```
9UX"F" = "000001111"   Unsigned 0 fill
9SX"F" = "111111111"   Signed: left bit = sign
9X"F"  = "000001111"   Defaults to unsigned
```

- Allow Replication of X and Z

```
7X"XX" = "XXXXXXXX"
7X"ZZ" = "ZZZZZZZ"
```

## Unary Reduction Operators

- Define unary AND, OR, XOR, NAND, NOR, XNOR

```
function "and" ( anonymous: BIT_VECTOR) return BIT;
function "or"  ( anonymous: BIT_VECTOR) return BIT;
function "nand" ( anonymous: BIT_VECTOR) return BIT;
function "nor"  ( anonymous: BIT_VECTOR) return BIT;
function "xor"  ( anonymous: BIT_VECTOR) return BIT;
function "xnor" ( anonymous: BIT_VECTOR) return BIT;
```

- Calculating Parity with reduction operators:

```
Parity <= xor Data ;
```

- Calculating Parity without reduction operators:

```
Parity <= Data(7) xor Data(6) xor Data(5) xor
          Data(4) xor Data(3) xor Data(2) xor
          Data(1) xor Data(0) ;
```

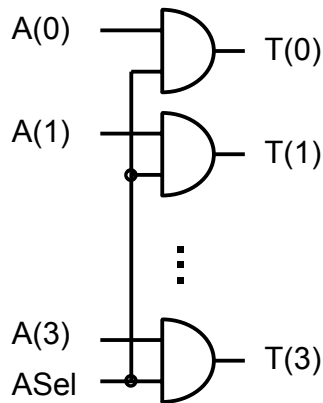
## Array / Scalar Logic Operators

- Overload logic operators to allow:

```

signal ASel : std_logic ;
signal T, A : std_logic_vector(3 downto 0) ;
. . .
T <= (A and ASel) ;

```



The value of ASel will replicated to form an array.

When ASel = '0', value expands to "0000"

When ASel = '1', value expands to "1111"

## Array / Scalar Logic Operators

- A common application is to data read back logic

```

signal Sel1, Sel2, Sel3, Sel4 : std_logic ;
signal DO, Reg1, Reg2, Reg3, Reg4
      : std_logic_vector(3 downto 0) ;
. . .
DO <= (Reg1 and Sel1) or (Reg2 and Sel1) or
      (Sel3 and Reg3) or (Sel4 and Reg4) ;

```

## Hierarchical Reference \*

- Near Term Alternative, Package based approach

```

Procedure Drive(
  source_signal      : IN STRING ;
  destination_signal : IN STRING ;
  delay              : IN TIME := 0 ns ;
  delay_mode         : IN delay_mode_type := DEPOSIT;
  verbose            : IN integer ) ;

Procedure Probe(
  source_signal      : IN STRING ;
  destination_signal : IN STRING ;
  verbose            : IN integer ) ;

```

- \* Inspired by donations from Mentor (Signal Spy) and Cadence (NCMirror)

## Hierarchical Reference\*

- Temporary read / write signal with procedures

```

procedure signal_force (
  destination_signal : IN STRING;
  force_value        : IN STRING;
  delay              : IN TIME := 0 ns ;
  delay_mode         : IN delay_mode_type := DEPOSIT;
  cancel_period      : IN DELAY_LENGTH := 0 ns ;
  delta_event        : IN BOOLEAN := FALSE ) ;

procedure signal_release (
  destination_signal : IN STRING;
  verbose            : IN integer) ;

get_value(
  destination_signal : IN STRING;
  verbose            : IN integer) ;

```

- \*Work inspired by donations from both Mentor and Cadence



## Slices in Array Aggregates

- Allow slices in an Array Aggregate

```

Signal A, B, Y      : unsigned (7 downto 0) ;
signal CarryOut    : std_logic ;

. . .

(CarryOut, Y) <= ('0' & A) + ('0' & B) ;

```

- Currently this would have to be written as:

```

(CarryOut, Y(7), Y(6), Y(5), Y(4), Y(3), Y(2), Y(1), Y(0))
  <= ('0' & A) + ('0' & B) ;

```

## Std Logic 1164 Updates

- Goals: Enhance current std\_logic\_1164 package
- A few items on the list are:
  - Uncomment xnor operators
  - Add shift operators for vector types
  - Add logical reduction operators
  - Add array/scalar logical operators
  - Match Function
  - Provide text I/O package for standard logic (similar to Synopsys' std\_logic\_textio)
- See also DVCon 2003 paper, "Enhancements to VHDL's Packages" which is available at:
  - <http://www.synthworks.com/papers>

# Numeric Std Updates

- Goals:
  - Enhance current numeric\_std package.
  - Unsigned math with std\_logic\_vector/std\_ulogic\_vector
- A few items on the numeric\_std list are:
  - Logic reduction operators
  - Array / scalar logic operators
  - Array / scalar addition operators
  - TO\_X01, IS\_X for unsigned and signed
  - TextIO for numeric\_std

# Resulting Operator Overloading

<u>Operator</u>	<u>Left</u>	<u>Right</u>	<u>Result</u>
Logic	TypeA	TypeA	TypeA
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
<u>Notes:</u>			
Array = std_ulogic_vector, std_logic_vector, bit_vector unsigned, signed,			
TypeA = boolean, std_logic, std_ulogic, Array			
For Array and TypeA, arguments must be the same.			
* for comparison operators the result is boolean			

## VHDL-200X, Summary

- Fast Track work is the first phase of VHDL-200X
  - Expected completion of current work is mid-2005.
- Next Steps are to add:
  - Verification Data Structures such as
    - associative arrays, queues, FIFOs, and memories
  - Object Orientation
  - Constrained Random Stimulus Generation
  - Direct C and Verilog/SystemVerilog Calls
- Goal = HDVL: Hardware Description and Verification Language
  - Full verification capabilities in one consistent language
- VHDL-200X = 200 X better than ... cont

## VHDL-200X: Summary

- We need your help:
  - Participate! Don't sit on the bench and wait and watch.
  - Tell your vendors about features you want supported
    - Be specific and prioritize your requests
- We need your financial support (particularly corporate)
  - Most work is voluntary.
  - LRM writing is work for hire.

# SynthWorks & VHDL Standards

- At SynthWorks, we are committed to see that VHDL is updated to incorporate the good features/concepts from other HDL/HVL languages such as SystemVerilog, E (specman), and Vera.
- At SynthWorks, we invest 100's of hours each year working on VHDL's standards
- Support VHDL's standards efforts by:
  - Encouraging your EDA vendor(s) to support VHDL standards,
  - Participating in VHDL standards working groups, and / or
  - Purchasing your VHDL training from SynthWorks

## SynthWorks VHDL Training

Comprehensive VHDL Introduction 4 Days

[http://www.synthworks.com/comprehensive\\_vhdl\\_introduction.htm](http://www.synthworks.com/comprehensive_vhdl_introduction.htm)

A design and verification engineers introduction to VHDL syntax, RTL coding, and 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](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](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>