

Accellera VHDL-2006

By

Jim Lewis, SynthWorks VHDL Training
jim@synthworks.com

MARLUG - Mid-Atlantic Region Local Users Group
ANNUAL CONFERENCE - OCTOBER 12, 2006
Johns Hopkins University Applied Physics Lab – Laurel, MD

SynthWorks

Accellera VHDL-2006

- IEEE VASG - VHDL-200X effort
 - Started in 2003 and made good technical progress
 - However, no \$\$\$ for LRM editing
- Accellera VHDL TSC
 - Took over in 2005,
 - Funded the technical editing,
 - Users reviewed and prioritized proposals,
 - Did super-human work to finalize it for DAC 2006

*** Accellera VHDL-2006 Standard 3.0 ***

- Approved at DAC 2006 by Accellera board
- Ready for industry adoption

Accellera VHDL-2006

- PSL
- IP Protection via Encryption
- VHDL Procedural Interface - VHPI
- Type Generics
- Generics on Packages
- Arrays with unconstrained arrays
- Records with unconstrained arrays
- Fixed Point Packages
- Floating Point Packages
- Hierarchical references of signals
- Process(all)
- Simplified Case Statements
- Don't Care in a Case Statement
- Conditional Expression Updates
- Expressions in port maps
- Read out ports
- Conditional and Selected assignment in sequential code
- hwrite, owrite, ... hread, oread
- to_string, to_hstring, ...
- Sized bit string literals
- Unary Reduction Operators
- Array/Scalar Logic Operators
- Slices in array aggregates
- Stop and Finish
- Context Declarations
- Std_logic_1164 Updates
- Numeric_Std Updates
- Numeric_Std_Unsigned

Many of VHDL's cumbersome syntax issues were fixed

PSL

- PSL will be incorporated directly into VHDL
- Implications
 - PSL Vunit, Vmode, Vprop 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.

VHDL Procedural Interface - VHPI

- Standardized Procedural Programming Interface to VHDL
- Gives access to information about a VHDL model during analysis, elaboration, and execution.
 - For add-in tools such as linters, profilers, code coverage, timing and power analyzers, and
 - For connecting in external models
- Object-oriented C model.
 - Gives direct access as well as callback functions for when an event occurs.

```
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 ;
```

- 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
  ) ;
```

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.float_pkg.all`

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

- Format is Sign Bit, Exponent, Fraction

```
signal A, B, Y : float (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
```

Hierarchical Reference

- Direct hierarchical reference:

```
A <= <<signal .top_ent.u_comp1.my_sig : std_logic_vector >>;
```

- Specifies object class (signal, shared variable, constant)
- path (in this case from top level design)
- type (constraint not required)

- Using an alias to create a local short hand:

```
Alias u1_my_sig is <<signal u1.my_sig : std_logic_vector >>;
```

- Path in this case refers to component instance u1 (subblock of current block).
- Can also go up from current level of heirarchy using "^"

Force and Release

- Forcing a port or signal:

```
A <= force '1' ;
```

- For in ports and signals this forces the effective value
- For out and inout ports this forces the driving value

- Forcing the effective value of an out or inout:

```
A <= force in '1' ; -- driving value, effects output
```

- Can also specify "in" with in ports and "out" with out ports, but this is the default behavior.
- Can force via hierarchical reference.
- Normal driver resolution occurs at levels above force level.

Force and Release

- Releasing a signal:

```
A <= release ;
```

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 ;

```

MAR

2006

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

- Current VHDL syntax:

```

if (Cs1='1' and nCs2='0' and Addr=X"A5") then
  if nWe = '0' then

```

- New: Allow top level of condition to be std_ulogic or bit:

```

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

```

- Create special comparison operators that return std_ulogic
(?=?, ?/=?, ?>, ?>=, ?<, ?<=)

```

if (Cs1 and not nCs2 and Addr?=X"A5") then
  DevSell1 <= Cs1 and not nCs2 and Addr?=X"A5" ;

```

- Does not mask 'X'

Hwrite, Hread, Owrite, Oread

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

```

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

procedure hread (
    Buf           : inout Line ;
    VALUE         : out bit_vector ;
    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;
) 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." &
    LF & " Actual value = " & to_hstring (Data) &
    LF & " Expected value = " & to_hstring (ExpData) &
    LF & " 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" = "ZZZZZZZZ"
```

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

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 ;

```

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) ;

```

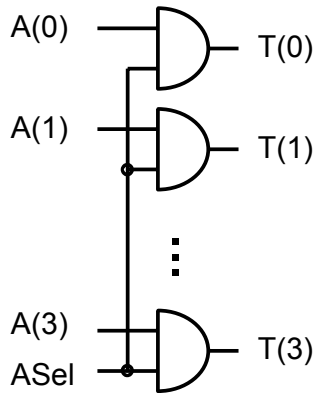
Array / Scalar Logic Operators

- 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 be 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) ;

```

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) ;

```

Stop and Finish

- STOP - Stop like breakpoint
- FINISH - Stop and not able to continue
- Defined in package ENV in library STD

```

package ENV is
  procedure STOP ( STATUS: INTEGER );
  procedure FINISH ( STATUS: INTEGER );
  . . .
end package ENV;

```

- Usage:

```

use std.env.all ;
. . .
TestProc : process begin
  . . .
  Stop(0) ;
end process TestProc ;

```


Context Declaration = Primary Design Unit

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

```
Context project1_Ctx is
  library ieee, YYY_math_lib ;
  use std.textio.all ;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all ;
  use YYY_math_lib.ZZZ_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, . . .

Std Logic 1164 Updates

- Goals: Enhance current std_logic_1164 package
- A few items on the list are:
 - std_logic_vector is now subtype of std_ulogic_vector
 - Uncomment xnor operators
 - Add shift operators for vector types
 - Add logical reduction operators
 - Add array/scalar logical operators
 - Added text I/O read, oread, hread, write, owrite, hwrite

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:
 - Array / scalar addition operators
 - TO_X01, IS_X for unsigned and signed
 - Logic reduction operators
 - Array / scalar logic operators
 - TextIO for numeric_std

Numeric Std Unsigned

- Overloads for std_ulogic_vector to have all of the operators defined for ieee.numeric_std.unsigned
- Replacement for std_logic_unsigned that is consistent with 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
Notes:			
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 Standards Next Steps

- Constrained Random Stimulus Generation
 - Random value generation with dynamic weighting
 - Randomly generate sequences of stimulus
- Functional Coverage
- Interfaces
- Verification Data Structures:
 - associative arrays, queues, FIFOs, and memories
- Direct C and Verilog/SystemVerilog Calls
- Object Orientation

- Goal = HDVL: Hardware Description and Verification Language
 - Full verification capabilities in one consistent language

Accellera VHDL-2006 3.0: Summary

Accellera VHDL-2006 3.0 is done and ready for adoption

- Tell your vendors about features you want supported.
 - Be specific and prioritize your requests
- Help us with the next revision ...
 - Participate! Don't sit on the bench and wait and watch.
 - See <http://www.accellera.org/activities/vhdl/>
 - Ask your colleagues and vendors to participate
 - Join Accellera and help fund the effort
 - Corporate membership

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

A design and verification engineer's introduction to VHDL syntax, RTL coding, and testbenches. Students get VHDL hardware experience with our FPGA based lab board.

VHDL Testbenches and Verification 3 days

http://www.synthworks.com/vhdl_testbench_verification.htm

Learn to simplify writing tests by creating transaction-based testbenches.

Intermediate VHDL Coding for Synthesis 2 Days

http://www.synthworks.com/intermediate_vhdl_synthesis.htm

Learn RTL (hardware) coding styles that produce better, faster, and smaller logic.

Advanced VHDL Coding for Synthesis 2 Days

http://www.synthworks.com/advanced_vhdl_synthesis.htm

Learn to avoid RTL coding issues, problem solving techniques, and advanced VHDL constructs.

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