IEEE 1076-2008
VHDL-200X

By

Jim Lewis, SynthWorks VHDL Training
jim@synthworks.com

IEEE

accellera

IEEE 1076-2008

- IEEE VASG - VHDL-200X effort
  - Started in 2003 and made good technical progress
  - However, no $$$ for LRM editing
- Accellera VHDL TSC
  - Took over in Fall 2005,
  - Prioritized IEEE proposals,
  - Finalized LRM text,
  - Completed Accellera standard in July 2006
  - Vendors implemented some features and provided feedback
  - In Spring 2008, Accellera forwarded standard to IEEE VASG for IEEE standardization.

*** VHDL-2008 *** Approved in September by IEEE REVCOM

IEEE 1076-2008

- Biggest Language change since 1076-1993

- 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

VHDL-2008 Big Ticket Items

PSL

- PSL has been incorporated directly into VHDL
  - Vunit, Vmode, Vprop are VHDL Design Units
  - Properties are VHDL block declarations
  - Directives (assert, cover) are VHDL concurrent statements

IP Protection and Encryption

- A pragma-based approach
  - Keywords and constructs specify algorithms and keys
  - Constructs demarcated protected envelopes of VHDL code

VHDL Procedural Interface - VHPI

- Standardized Procedural Programming Interface to VHDL
  - Gives tools access to information about a VHDL model during analysis, elaboration, and execution

Formal Type and Subprogram Generics + Packages with Generic Clause

```
package ScoreBoardPkg is
  generic (
    type BaseType ;
    function check(A, E : BaseType) return boolean
  ) ;
  . . .
end ScoreBoardPkg ;
```

- Specify generics in a package instance to create a new package

```
library IEEE ;
  use ieee.std_logic_1164.all ;
package ScoreBoardPkg_slv8 is new work.ScoreBoardPkg
  generic map (
    BaseType => std_logic_vector(7 downto 0),
    check => std_match ) ;
```

Composites with Unconstrained Elements

Arrays of Unconstrained Arrays

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

```
signal A : std_logic_matrix(5 downto 0)(7 downto 0) ;
```

Records with Unconstrained Array Elements

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

```
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) := "0110100" ;

  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  FFFFFFFFFFFFFFFFFFFFFFF

E = Exponent has a bias of 127
F = Fraction with implied 1 left of the binary point

0  10000000  00000000000000000000000  =  2.0
0  10000001  10100000000000000000000  =  6.5
0  01111100  00000000000000000000000  =  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 hierarchy 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.

- 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 ;
```

Case? = Case With Don't Care

- '-' represents don't care in case? choice
- Allow '-' in case? choice provided all choices 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? choice 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
DevSel1 <= 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"  =  "1111111"
7D"127" =  "1111111"
```

- 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"  =  "XXXXXXX"
7X"ZZ"  =  "ZZZZZZZ"
```

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 ;
```

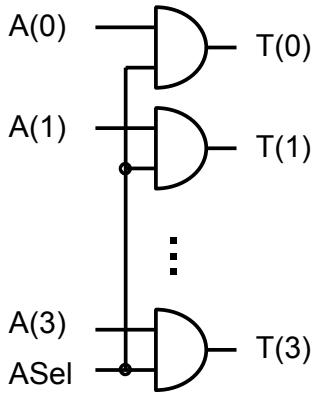
- Calculating Parity without reduction operators:

```
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 replicated to form an array.

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

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

Array / Scalar Logic Operators

- Common application: 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 STOP ;
  procedure FINISH ( STATUS: INTEGER );
  procedure FINISH ;
  function RESOLUTION_LIMIT return DELAY_LENGTH;
end package ENV;
```

- Usage:

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

Context Declaration = Primary Design Unit

SynthWorks

- 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

SynthWorks

- 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 logical 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

Operator	Left	Right	Result
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

IEEE 1076-2008 VHDL: Summary

- Approved by REVCOM in September 2008
- Users are interested in the new features
- Vendors have started implementing it.
- Next Steps
  - Working group has proposals for OO and Constrained Random.
  - There are requests for implementing Functional Coverage, Interfaces, Verification Data Structures (associative arrays, queues, FIFOs, Scoreboards, and memories), and Direct C and Verilog/SystemVerilog Calls
- Seeking new funding model as the Accellera has indicated they are not interested in proceeding with the above work.

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 4 days
http://www.synthworks.com/vhdl_testbench_verification.htm
Learn essential verification techniques including self-checking, transaction-based testing, data structures (linked-lists, scoreboards, memories), and randomization

VHDL Coding for Synthesis 4 Days
http://www.synthworks.com/vhdl_rtl_synthesis.htm
Learn VHDL RTL (FPGA and ASIC) coding styles, methodologies, design techniques, problem solving techniques, and advanced language constructs to produce better, faster, and smaller logic.

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