VHDL-2006-D3.0 Tutorial

Agenda:

Accellera Standards	Dennis Brophy
IEEE Standards	Edward Rashba
VHDL Tutorial	Jim Lewis
Accellera-VHDL-D3.0	
Fixed and Floating Point Packages	
Whats Next In VHDL	
Questions & Answers	

Accellera VHDL-2006-D3.0

By

Jim Lewis, SynthWorks VHDL Training
jim@synthworks.com

Accellera VHDL-2006-D3.0

- 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 July 2006

*** Accellera VHDL-2006-D3.0 ***
- Approved in July 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.

Formal Generics Types + 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 ;
```

Formal Generics Types + 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
  ) ;
```

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) := "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.

- 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 ;
```

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
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 ;
```

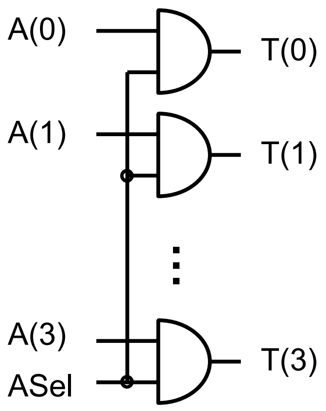
- Calculating Parity without reduction operators:

```
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 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 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

Accellera VHDL-2006-D3.0: Summary

Accellera VHDL-2006-D3.0 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/
    - You do not need to be an Accellera member to participate
  - Ask your colleagues and vendors to participate
  - Join Accellera and help fund the effort
    - Corporate membership

Fixed and Floating Point Packages

By

Jim Lewis, SynthWorks VHDL Training

David Bishop, Kodak

Fixed and Floating Point Packages

SynthWorks

- Fixed Point
  - Package & Types
  - Format
  - Sizing & Overloading
  - Literals in Assignments and Expressions
  - Quirks
- Floating Point
  - Package & Types
  - Format
  - Sizing & Overloading
  - Literals in Assignments and Expressions

These packages are part of Accellera VHDL-2006-D3.0 standard

Fixed Point Package

SynthWorks

- With fixed point, there are parameters that need to be specified.
- In Accellera VHDL-2006-D3.0 they are specified using generics:

```
package ZZZ_fixed_pkg is
  new ieee.fixed_generic_pkg
  generic map (
    fixed_round_style    =>
                  IEEE.math_utility_pkg.fixed_round,
    fixed_overflow_style =>
                  IEEE.math_utility_pkg.fixed_saturate,
    fixed_guard_bits     => 3,      -- # of guard bits
    no_warning           => false   -- show warnings
  );
```

- In the mean time, there is a temporary package fixed_pkg_c.vhd that uses constants and can be edited to be ZZZ_fixed_pkg.

Fixed Point Types

SynthWorks

```
Library YYY_math_lib ;
use YYY_math_lib.ZZZ_fixed_pkg.all ;
```

- ufixed = unsigned fixed point

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

- sfixed = signed fixed point

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

Fixed Point Format

SynthWorks

```
constant A : ufixed(3 downto -3) := "0110100";

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

- Range is required to be downto
- Whole number is on the left and includes 0 index (3 downto 0)
- Fraction is to the right of the 0 index (-1 downto -3)
- Ok to be only a integer or only a fraction

Fixed Point is Full Precision Math

SynthWorks

```
signal A4_3, B4_3 : ufixed ( 3 downto -3 ) ;
signal Y5_3       : ufixed ( 4 downto -3 ) ;
. . .
Y5_3  <= A4_3 + B4_3 ;
```

- Integer portion of the result is one bit bigger than largest argument
- Note that in numeric_std, addition/subtraction is modulo math

Fixed Point Sizing Rules

SynthWorks

Operation	Result Range
A + B, A - B	Max(A'left, B'left)+1 downto Min(A'right, B'right)
A * B	A'left + B'left+1 downto A'right + B'right
A rem B	Min(A'left, B'left) downto Min(A'right, B'right)
Unsigned /, divide	A'left - B'right downto A'right - B'left -1
Unsigned mod	B'left downto Min(A'right, B'right)
Unsigned Reciprocal	-A'right + 1 downto - A'left
Signed /, divide	A'left - B'right+1 downto A'right - B'left
Signed mod	Min(A'left, B'left) downto Min(A'right, B'right)
Signed Reciprocal	-A'right downto -A'left -1
Signed Abs(A)	A'left + 1 downto A'right
Signed -A	A'left + 1 downto A'right

Overloading

SynthWorks

Operation	Ufixed Result	Sfixed Result
Arithmetic + - * / rem mod abs = /= > < >= <=	ufixed op ufixed ufixed op real real op ufixed ufixed op natural natural op ufixed	sfixed op sfixed sfixed op real real op sfixed sfixed op integer integer op sfixed
Shift sll srl sla srl rol ror	ufixed op integer	sfixed op integer
Logic and or xor nand nor xnor	ufixed op ufixed	sfixed op sfixed

Notes:

- Size rules for integer assume that it is fixed'left downto 0
- Size rules for real assume that it is fixed'range

Literals in Assignments

SynthWorks

```
signal A4     : ufixed (3 downto -3) ;
. . .
-- String Literal
A4 <= "0110100" ;  -- 6.5

-- Real and/or Integer Literal
A4 <= to_ufixed( 6.5, A4 ) ;          -- sized by A4

A4 <= to_ufixed( 6.5, 3, -3 ) ;       -- pass indicies
```

- To_ufixed
  - Size of result based on range of an argument (such as A4) or by passing the indicies (3, -3)
  - Overloaded to accept either real or integer numbers
  - Type real and integer limited the precision of the literal

Literals in Expressions

SynthWorks

- Issue: a string literal used in an expression has range based on the direction of the base type and left index (integer'low)

```
signal A4     : ufixed (3 downto -3) ;
signal Y5     : ufixed (4 downto -3) ;
. . .
-- Y5 <= A4 + "0110100" ;   -- illegal,
--              ^^indicies are integer'low to …
```

- Solutions

```
subtype ufixed4_3 is ufixed (3 downto -3) ;
signal A4, B4 : ufixed4_3 ;
signal Y5      : ufixed (4 downto -3) ;
. . .
Y5 <= A4  +  ufixed4_3'("0110100") ;
Y5 <= A4  +  6.5 ;     -- overloading
Y5 <= A4  +  6 ;
```

Quirks: Accumulator

- Size of result needs to match size of one of the inputs

```
signal A4  : ufixed (3 downto -3) ;
signal Y7  : ufixed (6 downto -3) ;
. . .
--  Y7  <= Y7 + A4 ;   -- illegal, result too big

-- Solution, resize the result
Y7 <= resize (
        arg =>                 Y7 + A4,
        size_res =>            Y7,
        overflow_style =>      fixed_saturate,
                            -- fixed_wrap
        round_style =>         fixed_round
                            -- fixed_truncate
      );
```

Fixed Point Conversions

To_ufixed	integer, real, unsigned, sfixed, std_logic_vector to ufixed
To_sfixed	integer, real, signed, ufixed, std_logic_vector to sfixed
Resize	ufixed to ufixed or sfixed to sfixed both with potential rounding
to_real	ufixed or sfixed to real (scalar)
to_integer	ufixed or sfixed to integer (scalar)
to_unsigned	ufixed to unsigned (array)
to_signed	sfixed to signed (array)
to_slv	ufixed or sfixed to slv (array) for top level ports

Floating Point Package

SynthWorks

- With floating point, there are parameters that need to be specified.
- In Accellera VHDL-2006-D3.0 they are specified using generics:

```
package ZZZ_float_pkg is
  new ieee.fixed_generic_pkg
  generic map (
    float_exponent_width => 8,    -- default  'high
    float_fraction_width => 23,   -- default  'low
    float_round_style    =>
                 IEEE.math_utility_pkg.round_nearest,
    float_denormalize    => true,  -- IEEE extended fp
    float_check_error    => true,  -- NAN & overflow
    float_guard_bits     => 3,     -- # of guard bits
    no_warning           => false  -- show warnings
   );
```

- In the mean time, there is a temporary package float_pkg_c.vhd that uses constants and can be edited to be ZZZ_float_pkg.

Floating Point Types

SynthWorks

```
Library YYY_math_lib ;
use  YYY_math_lib.ZZZ_float_pkg.all ;
```

- Main type is unconstrained:

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

- Package also defines subtypes:
  - IEEE 754 Single Precision

```
subtype float32  is float( 8 downto -23);
```

  - IEEE 754 Double Precision

```
subtype float64  is float(11 downto -52);
```

  - IEEE 854 Extended Precision

```
subtype float128 is float(15 downto -112);
```

Floating Point Format

SynthWorks

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

  8  76543210  12345678901234567890123
  S  EEEEEEEE  FFFFFFFFFFFFFFFFFFFFFFF

E = Exponent is biased by 127
F = Fraction with implied 1 left of the binary point

value = 2**(E-127) * (1 + F)

0  10000001  10100000000000000000000
= +1 *  2**(129 - 127)  *  (1.0 + 0.5 + 0.125)
= +1 *  2**2            *  (1.625)              =  6.5
```

- Range is required to be downto
- Sign Bit = A'left = bit 8 (0 = positive, 1 = negative)
- Exponent = range A'left - 1 downto 0 = 7 downto 0
- Mantissa = range -1 downto A'right = -1 downto -23
- Sign, Exponent and Mantissa are always present

Special Numbers

SynthWorks

- Zero (Positive 0 = Negative 0)

```
0 00000000 00000000000000000000000  -- Positive
1 00000000 00000000000000000000000  -- Negative
```

- Infinity

```
0 11111111 00000000000000000000000  -- Positive
1 11111111 00000000000000000000000  -- Negative
```

- NAN - Not A Number

```
1 11111111 00000000000000000000001
```

- Exponent with all 0 is reserved for zero and denormal numbers
- Exponent with all 1 is reserved for infinity and NAN

Range of Values

SynthWorks

- Large positive number (Exponent of all 1 is reserved)

```
0 11111110 00000000000000000000000
= +1 * 2**(254 - 127) * (1.0 + 0)
= 2**(127)
```

- Smallest positive number without denormals

```
0 00000001 00000000000000000000000
= +1 * 2**(1 - 127) * (1.0 + 0)
= 2**(-126)
```

- Extended small numbers = Denormals, but only when enabled

```
0 00000000 10000000000000000000000
= +1 * 2**(1 - 127) * (0 + 0.5)
= +1 * 2**(-126)    * 2**(-1)
= 2 **(-127)
```

Floating Point Types

SynthWorks

```
signal A32, B32, Y32 : float (8 downto -23) ;

. . .

Y32 <= A32 + B32 ;
```

  - Floating point result will have the maximum exponent and maximum mantissa of its input arguments.

- Also need to specify:

Rounding	Default = round_nearest
round_nearest, round_zero, round_inf, round_neginf	
Denormals: On / Off	Default = on = true
Check NAN and Overflow	Default = on = true
Guard Bits: Extra bits for rounding.	Default = 3

Overloading

Operation	Float Result
Arithmetic + - * / rem mod abs = /= > < >= <=	float op float float op real real op float float op integer integer op float
Logic and or xor nand nor xnor	float op float

Notes:
- Integers and reals are converted to a float that is the same size as the float argument

Literals in Assignments

```
signal A_fp32  : float32 ;
. . .
-- String Literal
A_fp32 <= "01000000110100000000000000000000" ; -- 6.5

-- Real and/or Integer Literal
A_fp32 <= to_float(6.5, A_fp32);  -- size using A_fp32

A_fp32 <= to_float(6.5, 8, -32);  -- pass indicies
```

- To_float
  - Needs to size the result based on range of an argument (such as A_fp32) or by passing the indicies (8, -32)
  - Overloaded to accept either integers or real numbers
  - Note the required precision of type real and integer is limited by the language

Literals in Expressions

SynthWorks

- Issue: a string literal used in an expression has range based on the direction of the base type and left index (integer'low)

```
signal A, Y : float32 ;
. . .
-- Y <= A + "01000000110100000000000000000000"; -- ill
--             ^^ range integer'low to ...
```

- Solutions

```
signal A, Y : float32 ;
. . .
Y <= A + float32'("01000000110100000000000000000000");
Y <= A + 6.5 ;    -- overloading
Y <= A + 6 ;      -- overloading
```

Floating Point Conversions

SynthWorks

To_float	**integer, real, ufixed, sfixed, signed, unsigned, and std_logic_vector to float**
Resize	**float to float with potential rounding, …**
to_real	**float to real (scalar)**
to_integer	**float to integer (scalar)**
to_sfixed	**float to sfixed (array)**
to_ufixed	**float to ufixed (array)**
to_unsigned	**float to unsigned (array)**
to_signed	**float to signed (array)**
to_slv	**float to slv (array) for top level ports**

Going Further

SynthWorks

- Until vendors implement Accellera VHDL-2006-D3.0, download math_utility_pkg.vhd, fixed_pkg_c.vhd, and float_pkg_c.vhd from:

  http://vhdl.org/vhdl-200x/vhdl-200x-ft/packages/files.html

- Current methodology
  - Create a library named, YYY_math_lib, where YYY = project or company
  - Copy fixed_pkg_c to ZZZ_fixed_pkg and float_pkg_c to ZZZ_float_pkg
  - Set the constants to appropriate values
  - Compile into the library
  - For different settings, make additional copies of the packages

- With package generics (see Accellera VHDL-2006-D3.0 standard),
  - package instantiations replace copies of a package with constants

What's Next in VHDL

By

Jim Lewis, SynthWorks VHDL Training

DATE 07 accellera IEEE

SynthWorks

What's Next in VHDL

- **VHDL = Verification & Hardware Description Language**
- Verification focused
  - OO/Classes
  - Verification Data Structures
  - Randomization
  - Functional Coverage
  - Incorporate the good things from SystemVerilog, SystemC, IEEE 1850/E, Vera
- RTL: items to do, but verification is higher priority

Caution: Changes presented here are a work in progress

OO / Classes

SynthWorks

- Classes are the foundation for both data structures and randomization of transactions
- Status: Have proposal from Peter Ashenden
- Proposal extends protected types into classes
  - Similar to other programming languages (particularly Java).
  - Adds shared variable ports
- Since classes extend protected types, can do some prototyping with the current language.

OO / Classes

SynthWorks

```
type BoundedFIFO is protected class
  procedure put ( e : in element_type );    -- methods
  procedure get ( e : out element_type );
end protected class BoundedFIFO;
```

```
type BoundedFIFO is protected class body
  constant size : positive := 20;
  type element_array is array (0 to size-1) of element_type;

  variable elements : element_array;
  variable head, tail : natural range 0 to size-1 := 0;
  variable count : natural range 0 to size := 0;

  procedure put ( e : in element_type ) is begin
    if count = size then wait until count < size; end if;
    elements(head) := e;
    head := (head + 1) mod size;  count := count + 1;
  end procedure put;

  procedure get ( e : out element_type ) is begin   . . .

end protected class body BoundedFIFO;
```

Verification Data Structures

SynthWorks

- Basic Requirements
  - Linked-Lists
  - FIFOs
  - Mailboxes (Put, Get)
  - Transaction Interfaces (Put, Get)
  - Scoreboards (PutValue, CheckValue, ErrCount)
  - Memories (MemInit, MemRead, MemWrite)

- The current plan is to make these class based.

A 1 Item MailBox

SynthWorks

- An interface specifies contracts on a class (like in Java):

```
type putable is interface
   procedure put ( e : in element_type );
   procedure try_put (e : in element_type;  ok : out boolean);
end protected interface putable;

type getable is interface    . . .   -- see proposal
```

- Mailbox class implements putable and getable:

```
type mailboxPCType is protected class implements putable,
getable
   function flag_up return boolean;
   procedure put  ( e : in element_type );
   procedure try_put (e : in element_type;  ok : out boolean);
   procedure get ( e : out element_type );
   procedure try_get (e : out element_type; ok : out boolean);
end protected class mailbox;
```

A 1 Item MailBox

- Producer port is "putable".
- Consumer port is "getable"
- Any type that implements these can be used

```
entity tlm is
end tlm ;

architecture structural of tlm is
  component producer is
    port ( shared variable data_source : inout putable );
  end component producer;

  component consumer is
    port ( shared variable data_sink : inout getable );
  end component consumer;

  shared variable MailBox : mailboxPCType;

begin

  u_producer : producer port map ( data_source => MailBox );
  u_consumer : consumer port map ( data_sink   => MailBox );

end architecture tlm;
```

Randomization

- Useful when testing numerous configurable features.
  - Testing in an isolated features is straightforward
  - Testing interactions is a large verification space
    - difficult to simulate completely
    - difficult to predict corner cases
    - Randomization can reasonably coverage this space

- Use of coverage is important to identify which design features have been tested.

- Status: Have proposal from Jim Lewis

- Current proposal is based on SystemVerilog

Randomization

SynthWorks

- Supporting two forms of randomization:
  - Class based
  - Procedural
- Class based randomization
  - Group values of a transaction together
  - Specify relationships/constraints between them
  - Randomize as a single item
- Procedural Randomization
  - Randomizing single values
  - CaseRand
  - Sequence

Class Based Randomization

SynthWorks

- A class with constraints:

```
Type TxPacketCType is class

  Rand Variable BurstLen   : integer ;   -- Public Variables
  Rand Variable BurstDelay : integer ;

  Constraint BurstPkt is (
    BurstLen in (1 to 10) ;
    BurstDelay in (1 to 6) when BurstLen <= 3 else
    BurstDelay in (3 to 10) ;
  ) ;

End class TxPacketCType ;
```

Class Based Randomization

```
TxProc : process
  variable TxPacket : TxPacketCTType ;
  variable RV : RandomClass ;
begin
  . . .
  TxOuterLoop: loop

    TxPacket.randomize ;

    for i in 1 to TxPacket.BurstLen loop
      DataSent := RV.RandSlv(0, 255, DataSent'length);
      Scoreboard.PutExpectedData(DataSent) ;
      WriteToFifo(DataSent) ;
    end loop ;

    wait for TxPacket.BurstDelay * tperiod_Clk - tpd ;
    wait until Clk = '1' ;

  end loop TxOuterLoop ;
. . .
end process TxProc ;
```

Procedural Randomization

- Randomization within code using individual randomization calls, RandCase, or Sequence construct

```
I0 := 1; I1 := 1; I2 := 1;
for i in 1 to 3 loop

  RandCase is
    with I0 =>
      CpuWrite(CpuRec, DMA_WORD_COUNT, DmaWcIn);
      I0 := 0 ; -- modify weight

    with I1 =>
      CpuWrite(CpuRec, DMA_ADDR_HI, DmaAddrHiIn);
      I1 := 0 ; -- modify weight

    with I2 =>
      CpuWrite(CpuRec, DMA_ADDR_LO, DmaAddrLoIn);
      I2 := 0 ; -- modify weight

  end case ;

end loop ;
CpuWrite(CpuRec, DMA_CTRL, START_DMA or DmaCycle);
```

Functional Coverage

- Functional coverage supplements other forms of coverage
- Tool based or structural coverage can tell you:
  - there was a FIFO read
  - the FIFO was empty
  - however, it has no way to tell you both happened.
- Assertions via PSL can tell you this
- Functional coverage constructs provide capability to:
  - bin values of an object into separate categories
  - correlate cross coverage between items
- Status: proposal is a work in progress
  - As always, all input welcome.

Summary

- Both Standards and Vendors are heavily influenced by users
  - Make sure to voice your opinion
- 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

VHDL = <u>V</u>erification & <u>H</u>ardware <u>D</u>escription <u>L</u>anguage