SNUM2

ENSSAT
LANNION

VHDL: a Hardware Description Language for Simulation and Logic Synthesis

Annexes

1. Data Types
2. Operators
3. Attributes
4. Subprograms, Packages, and Libraries
5. Inputs and Outputs
6. VHDL examples
7. Synthesis hints

Olivier Sentieys
ENSSAT - Université de Rennes 1
IRISA/INRIA
sentieys@irisa.fr
http://people.rennes.inria.fr/Olivier.Sentieys/?page_id=95

UNIVERSITÉ DE RENNES 1 UMR IRISA informatiques mathématiques Inria

VHDL: a Hardware Description Language for Simulation and Logic Synthesis

Appendix

Data Types

Type Definitions

```
entity xxx is
    Port ( …);
end xxx;
architecture rtl of xxx is
    type myint1 is integer range 1 to 127;
    type myint2 is integer range 1 to 127;
    subtype myint3 is integer range 0 to 127;
    subtype myint4 is integer range 0 to 127;
    signal A: myint1;
    signal B: myint2;
    signal C: myint3;
    signal D: myint4;
begin
    A <= B; -- error: A and B have different types
    C <= D; -- correct
end RTL;
```


Enumerated Data Types

- Enumerated data types are particularly useful for constructing models of computing systems

```
type instr_opcode is ('add', 'sub', 'xor', 'nor', 'beq',
'lw', 'sw');
type state is ('empty', 'half_full', 'half_empty', 'empty');
type day is ('Monday', 'Tuesday',…, 'Sunday');


type boolean is (FALSE, TRUE);
type bit is ('0', '1');
type severity_level is (NOTE, WARNING, ERROR, FAILURE);
type character is ( NUL, SOH, STX, ETX,..., '0','1', ...);
```

Subtypes

- Subset of a predefined or pre-declared type

```
subtype workday is day Monday to Friday;
subtype word is std_logic_vector(15 downto 0);

signal today:day;
signal w1,w2:word;

begin
today <= Monday;
w1 <= w2;
w2 <= X"A5C3" —- X"A5C3"="1010010111000011"
…
```


VHDL Standard Data Types

Type	Range of values	Example declaration
integer	implementation defined	`signal index: integer:= 0;`
real	implementation defined	`variable val: real:= 1.0;`
boolean	(TRUE, FALSE)	`variable test: boolean:=TRUE;`
character	defined in package STANDARD	`variable term: character:= '@';`
bit	0, 1	`signal In1: bit := '0';`
bit_vector	array with each element of type bit	`variable PC: bit_vector(31 downto 0)`
time	implementation defined	`variable delay: time:= 25 ns;`
string	array with each element of type character	`variable name : string(1 to 10) := "model name";`
natural	0 to the maximum integer value in the implementation	`variable index: natural:= 0;`
positive	1 to the maximum integer value in the implementation	`variable index: positive:= 1;`

```
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Arrays

- Arrays are collection of objects with the same type

```
type byte is array (7 downto 0) of std_logic;
type word is array (31 downto 0) of std_logic;
type memory is array (0 to 4095) of word;
```

```
type bus is array (0 to 31) of bit;
type RAM is array (0 to 1024, 0 to 31) of bit;
type PRICE is range 0 to 1000;
type COLORS is (WHITE, BLUE, GREEN, RED, YELLOW, BLACK, PURPLE);
type COLR_PRICE is array (COLOR range WHITE to BLACK) of PRICE;
```

- Arrays may have unknown size
  - Size is then specified during declaration

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
signal m:bit_vector(3 downto 0);
signal c:string(1 to 10); -- string index starts at 1
```


Arrays: examples

```
type index is range 0 to 2;
--array of array
type TAB1 is array (index) of bit_vector(7 downto 0);
--2D array
type TAB2 is array (0 to 3, 1 to 8) of bit;
signal A: TAB1;
signal B: TAB2;
begin
A(0)     <= "01001111";
A(2)(5)  <= '1';
A        <= ("01001111",X"53",X"AD");
B(3,5)   <= '0';
B        <= (('0','1','1','0'),('1','0','1','1'),(…),(…));
…
```

```
type OPTYPE is (ADD,SUB,MUL,DIV,BRA);
type T is array (1 to 10) of OPTYPE;
signal A: T;
begin
A <= (ADD,SUB,MUL,DIV,BRA);
A <= (ADD,SUB,5=>BRA,4=>DIV,3=>MUL);
A <= (ADD,2|4=>SUB,5 to 7=>BRA,others=>DIV);
…
```

Integers

- 32 bits by default
- Can use `range` to restrict size

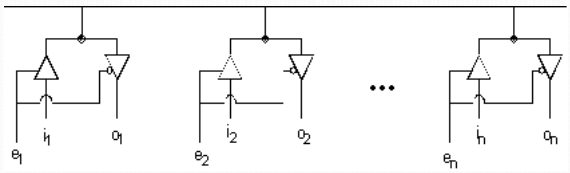
```
-- declaration
signal A,B,C,D: integer range 0 to 15; -- 4 bits
signal E,F,G,H: integer range 0 to 255;
-- examples
A   <= 14;  -- default base = decimal
B   <= 16#A#; -- 10 in hexadecimal
C   <= 10#12#; -- 12 in decimal
D   <= 8#17#; -- 15 in octal
E   <= 16#1B#; -- 27 in hexa
F   <= 2#00101011#; -- 35 in binary
G   <= 2#0010_1011#; -- 35 in binary (more readable)
```


bit and bit_vector

```
-- declaration
signal enable: bit;
signal data1:bit_vector(5 downto 0);
signal data2:bit_vector(7 downto 0);

begin
enable <='1'; -- '
data1  <="010101"; -- "
data1  <=O"35"; -- octal
data2  <=X"5C"; -- hexa
```

Resolved Types



- At any point in time what is the value of the bus signal?
- We need to "resolve" the value
  - Take the value at the head of all drivers
  - Select one of the values according to a resolution function
- Predefined IEEE 1164 resolved types are std_logic and std_logic_vector

STD_LOGIC_1164 Data Types

- Overload and resolve `bit` type
  - 'X': a signal takes this value if assigned '1' and '0' at the same time
  - 'Z': signal is not driven
  - 'L': signal is driven but not equal to Vss
  - 'H': signal is driven but not equal to Vdd
  - 'W': conflict between 'L' and 'H'
  - '-': don't care. Used to specify degree of liberty to synthesis tools
- Library declaration

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.all;
```

```
Type STD_LOGIC is (
'U', -- unitialized
'X', -- Forcing Unknown
'0', -- Forcing 0
'1', -- Forcing 1
'Z', -- High Impedance
'W', -- resistive conflict
     -- or Weak Unknown
'L', -- Weak '0'
'H', -- Weak '1'
'-'  -- don't care
);
```

std_logic Resolution Function

- Pair wise resolution of signal values from multiple drivers
- Resolution operation must be associative

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

resolving values for std_logic types

std_logic_vector: Bus Assignment

MSB LSB

```
signal coef, mask: std_logic_vector(5 downto 0);
signal data: std_logic_vector(15 downto 0);
signal result: std_logic_vector(10 downto 0);
signal test: std_logic_vector(0 to 5);

coef <= mask;
result <= data(13 downto 3); -- same width
result <= data(13 downto 5); -- different width
test <= coef(0 to 5); -- same width but different order
test(5) <= coef(0); -- ok
…
test(0) <= coef(5); -- ok
result <= coef & mask(4 downto 0); -- concatenation operator
result(9 downto 5) <= coef and data(7 downto 3);
```

signed / unsigned Types

- Defined in std_logic_arith or numeric_std librairies

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```


Structures: record Type

```
type BUS_OPERATION is (IO_READ,IO_WRITE,MEM_READ,MEM_WRITE);

type operation_bus is record
   address: std_logic_vector(15 downto 0);
   data: std_logic_vector(15 downto 0);
   cmd: bus_operation;
end record;
signal vector: operation_bus;
vector.cmd <= IO_WRITE;   -- assignment of a field
-- assignment of all fields
vector <= (X"1A8F",X"A5A5",MEM_WRITE);
vector <= (cmd=>MEM_WRITE, others=>(others=>'0'));
```

Physical Types

- A physical type allows to define measurement units for some physical quantity, like length, time, pressure, capacity, etc.

```
type Time is range --implementation defined-- ;
  units
        fs; -- femtosecond
        ps = 1000 fs; -- picosecond
        ns = 1000 ps; -- nanosecond
        us = 1000 ns; -- microsecond
        ms = 1000 us; -- millisecond
        sec = 1000 ms; -- second
        min = 60 sec; -- minute
        hr = 60 min; -- hour
  end units;
```

```
type capacitance is range 0 to 1E16
units
ffr; -- femtofarads
pfr = 1000 ffr;
nfr = 1000 pfr;
ufr = 1000 nfr
mfr = 1000 ufr
far = 1000 mfr;
kfr = 1000 far;
end units;

type resistance is range 0 to 1E16
units
l_o; -- milli-ohms
ohms = 1000 l_o;
k_o= 1000 ohms;
m_o = 1000 k_o;
g_o = 1000 m_o;
end units;
```


Physical Types: Example

```
entity inv_rc is
generic (c_load: real:= 0.066E-12); -- farads
port (i1 : in std_logic;
     o1: out: std_logic);
constant rpu: real:= 25000.0; --ohms
constant rpd: real :=15000.0; -- ohms
end inv_rc;

architecture delay of inv_rc is
constant tplh: time := integer (rpu*c_load*1.0E15)*3 fs;
constant tpll: time := integer (rpu*c_load*1.0E15)*3 fs;
begin
   o1 <=      '1' after tplh when i1 = '0' else
              '0' after tpll when i1- = '1' or i1 = 'Z' else
              'X' after tplh;
end delay;
```

explicit type casting and range management

Physical Types: Example (cont.)

```
entity inv_rc is
generic (c_load: capacitance := 66 ffr); -- farads
port (i1 : in std_logic;
      o1: out: std_logic);
constant rpu: resistance:= 25000 ohms;
constant rpd : resistance := 15000 ohms;
end inv_rc;

architecture delay of inv_rc is

constant tplh: time := (rpu/ 1 l_o)* (c_load/1 ffr) *3 fs/1000;
constant tpll: time := (rpu/ 1 l_o)* (c_load/1 ffr) *3 fs/1000;
begin
   o1 <=      '1' after tplh when i1 = '0' else
              '0' after tpll when i1 = '1' or i1 = 'Z' else
              'X' after tplh;
end delay;
```


Other Types

- Real

```
type REAL is range -16#0.7FFFFF8#E+32 to 16#0.7FFFFF8#E+32;
type my_real is range 0.0 to 3.14159;

V1 <= 1.0;
V2 <= 2.14E15;
V3 <= -3.6;
```

- Pointers (`access`)
- File access (`use std.textio.all`)
  - See part on simulation and testbenches

Operators

Operators

- VHDL text or language reference manual for less commonly used operators and types

logical operators	**and**	**or**	**nand**	**nor**	**xor**	**xnor**
relational operators	**=**	**/=**	**<**	**<=**	**>**	**>=**
shift operators	**sll**	**srl**	**sla**	**sra**	**rol**	**ror**
addition operators	**+**	–				
unary operators	**+**	–				
multiplying operators	*****	**/**	**mod**	**rem**		
miscellaneous operators	******	**abs**	**not**	**&**		

Operators: Examples

```
“10010” Sll 1 = “00100”
“10010” Sll 2 = “01000”
“10010” Srl 1 = “01001”
“10010” Sla 1 = “00100”
“10010” Sra 1 = “11001”
“10010” rol 1 = “00101”
“10010” ror 1 = “01001”
“10010” ror -1 = “10010” rol 1
```

```
“10010”&”010”= “10010010”
X“A”&X“2C”=X“A2C”
```

```
A<=‘0’ when B<“1000” else ‘1’;
A<=‘0’ when B(3)=‘0’ else ‘1’;
A<=B(3);
```

```
signal A, B, C: std_logic_vector(5 downto 0);

C <= A + B;
C <= A / 2;
```


Attributes

Attributes

- Data can be obtained about VHDL objects such as types, arrays and signals: `object'attribute`
- Classes of Attributes
  - Value attributes
    - returns a constant value
  - Function attributes
    - invokes a function that returns a value
  - Signal attributes
    - creates a new signal
  - Type Attributes
    - supports queries about the type of VHDL objects
  - Range attributes
    - returns a range

Value Attributes

- Return a constant value

```
type statetype is (state0, state1, state2 state3);
    state_type'left = state0
    state_type'pos(state1) = 1
    state_type'val(2) = state2
    state_type'succ(state2) = state3
    state_type'leftof(state2) = state1
```

- Examples

Value attribute	Value
name'**left** ('**right**)	returns the left (right) most value of name in its defined range
name'**succ(value)** '**pred()** '**leftof()** '**rightof()**	returns the successor of value in name predecessor, value on the left/right
name'**high** ('**low**)	returns the highest (lowest) value of type_name in its range
name'**length**	returns the number of elements in the array name

Function Attributes

- Use of attributes invokes a function call which returns a value

```
if (Clk'event and Clk='1')
```

- Examples: function signal attributes

Function attribute	Function
signal_name'**event**	Return a Boolean value signifying a change in value on this signal
signal_name'**active**	Return a Boolean value signifying an assignment made to this signal. This assignment may not be a new value.
signal_name'**last_event**	Return the time since the last event on this signal
signal_name'**last_active**	Return the time since the signal was last active
signal_name'**last_value**	Return the previous value of this signal

Function Attributes (cont.)

- Function array attributes

Function attribute	Function
array_name'**left**	returns the left bound of the index range
array_name'**right**	returns the right bound of the index range
array_name'**high**	returns the upper bound of the index range
array_name'**low**	returns the lower bound of the index range

```
type mem_array is array(0 to 7) of bit_vector(31 downto 0)
    mem_array'left = 0
    mem_array'right = 7
    mem_array'length = 8
```

Range Attributes

- Returns the index range of a constrained array

```
for I in X'range loop
  …
  my_var := X(i);
  …
end loop;
```

- Makes it easy to write loops!

Signal Attributes

- Creates a new "implicit" signal

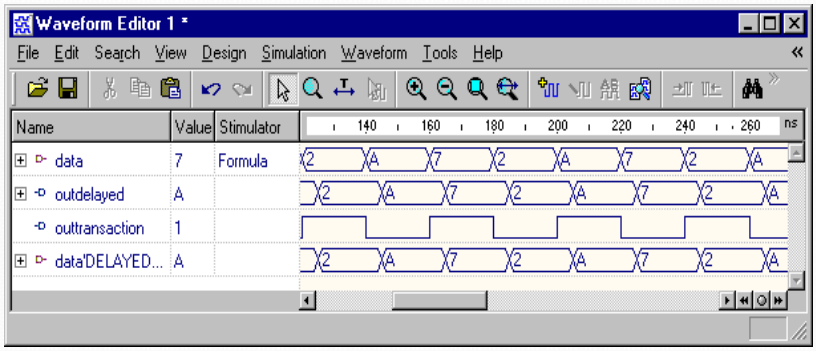
Signal attribute	Implicit Signal
signal_name'**delayed(T)**	Signal delayed by T units of time
signal_name'**transaction**	Signal whose value toggles when signal_name is active
signal_name'**quiet(T)**	True when signal_name has been quiet for T units of time
signal_name'**stable(T)**	True when event has not occurred on signal_name for T units of time

- Internal signals are useful modeling tools

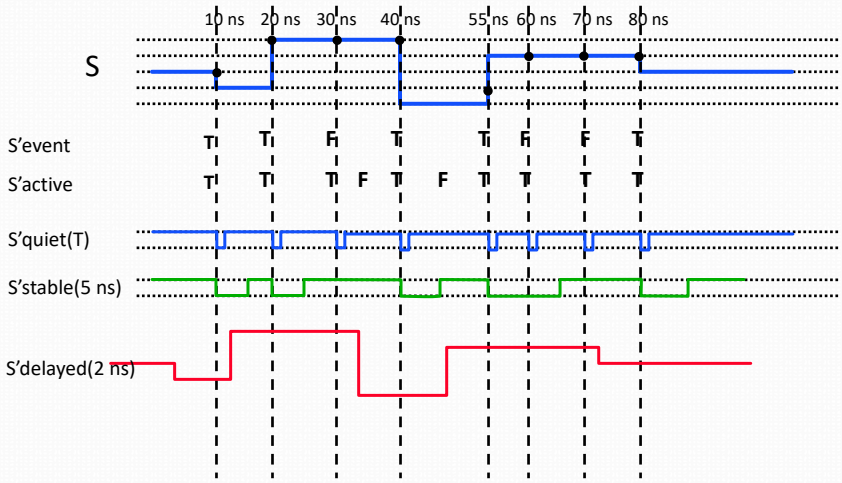
Signal Attributes: Example

```
architecture behavioral of attributes is
begin
  outdelayed <= data'delayed(5 ns);
  outtransaction <= data'transaction;
end attributes;
```

These are real (in simulation) signals and can be used elsewhere in the model



Signal Attributes



Subprograms, Packages, and Libraries

Essentials of Functions

- Place function code in the declarative region of the **architecture** or **process**
  - local to a process or global to the architecture
- Declaration

```
function name [ ( parameter_list ) ]
return type_name ;
```

- Function body

```
function name [ (parameter_list ) ]
return type_name is
 [ variables declarations ]
begin
 sequential instructions
 return (expression)
end [ name ] ;
```

Essentials of Functions

- Formal parameters and mode
  - Default mode is of type in

```
function rising_edge (signal clock: std_logic)
```

- Functions cannot modify parameters
- Function variables initialized on each call
- Wait statements are not permitted in a function!
  - And therefore not in any procedure called by a functions

Function: Example

```
architecture behavioral of dff is
  function rising_edge (signal clock : std_logic) return
boolean is
    variable edge : boolean:= FALSE;
  begin
    edge := (clock = '1' and clock'event);
    return (edge);
  end rising_edge;
begin
output: process
begin
  wait until (rising_edge(Clk));
  Q <= D after 5 ns;
  Qbar <= not D after 5 ns;
end process output;
end architecture behavioral;
```

Function: Example

- A common use of functions: type conversion
- Use of attributes for flexible function definitions
  - Data size is determined at the time of the call
- Browse the vendor supplied packages for many examples

```
function to_bitvector (svalue : std_logic_vector) return bit_vector is
  variable outvalue : bit_vector (svalue'length-1 downto 0);
begin
  for i in svalue'range loop -- scan all elements of the array
    case svalue (i) is
        when '0' => outvalue (i) := '0';
        when '1' => outvalue (i) := '1';
        when others => outvalue (i) := '0';
    end case;
  end loop;
  return outvalue;
end to_bitvector
```


Essentials of Procedures

- Placement of procedures determines visibility in its usage
  - local to a process or global to the architecture
- Declaration

```
procedure name [ ( parameter_list ) ];
```

- Procedure body

```
procedure name [ ( parameter_list ) ] is
[ variable declarations ]
begin
 sequential instructions
end [ name ] ;
```

Essentials of Procedures

- Parameters may be of mode in (read only) and out (write only)
- Default class of input parameters is constant
- Default class of output parameters is variable
- Variables declared within procedure are initialized on each call
- Wait statements are permitted in a procedure

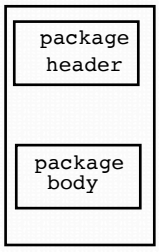
Procedures and Signals

- Procedures can make assignments to signals passed as input parameters
- Procedures may not have a wait statement if the encompassing process has a sensitivity list
- Procedures may modify signals not in the parameter list, e.g., ports
- Signals may not be declared in a procedure
- Procedures may make assignments to signals not declared in the parameter list

```
procedure mread (address : in std_logic_vector (2 downto 0);
  signal R : out std_logic;
  signal S : in std_logic;
  signal ADDR : out std_logic_vector (2 downto 0);
  signal data : out std_logic_vector (31 downto 0)) is
begin
  ADDR <= address;
  R <= '1';
  wait until S = '1';
  data <= DO;
  R <= '0';
end mread;
```

Essentials of Packages

- A package is a collection of reusable objects
- A package may contains procedures, functions, components, constants and types
- A package is composed of
  - Package declaration (header)
  - Package body
- Only package declaration is visible outside the package



Essentials of Packages

- Package declaration
  - Declaration of the functions, procedures, and types that are available in the package
  - Serves as a package interface
  - Only declared contents are visible for external use
- Note the behavior of the use clause
- Package body
  - Implementation of the functions and procedures declared in the package header
  - Instantiation of constants provided in the package header

Example: Package Header

```
package mypack is

        constant SIZE_ROM : natural := 16;

        type mem is array (natural range <>) of std_logic_vector;
        subtype ROM is mem (0 to SIZE_ROM-1);

        component foo
                port(
                        a,b: in std_logic;
                        s: out std_logic);
        end component;

        procedure min (a,b: in integer; c out integer);
        function max (a,b: integer) return integer;

end package mypack;
```


Example: Package Header std_logic_1164

```
package std_logic_1164 is
type std_ulogic is ('U', --Unitialized
'X', -- Forcing Unknown
'0', -- Forcing 0
'1', -- Forcing 1
'Z', -- High Impedance
'W', -- Weak Unknown
'L', -- Weak 0
'H', -- Weak 1
'-' -- Don't care
);
type std_ulogic_vector is array (natural range <>) of std_ulogic;
function resolved (s : std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_ulogic;
type std_logic_vector is array (natural range <>) of std_logic;
function "and" (l, r : std_logic_vector) return std_logic_vector;
--..<rest of the package definition>
end package std_logic_1164;
```

Package Body

- Packages are typically compiled into libraries
- New types must have associated definitions for operations such as logical operations (e.g., and, or) and arithmetic operations (e.g., +, *)
- Examine the package std_logic_1164 stored in library IEEE

```
package body my_package is
--
-- type definitions, functions, and procedures
--
end my_package;
```


Example: Package Body

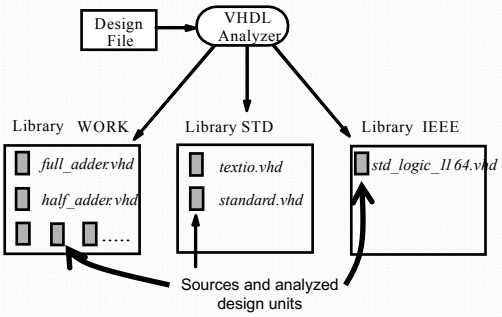
```
package body mypack is

      function max (a,b: integer) return integer; is
      begin
            if a>b then return a;
            else return b;
            end if;
      end max;

End package body mypack ;
```

Essentials of Libraries

- Design units are analyzed (compiled) and placed in libraries
- Logical library names map to physical directories
- Libraries STD and WORK are implicitly declared




```
library ieee;
use ieee.std_logic_1164.all;
use work.mypack.all;

entity ...
```


Visibility Rules

- When multiple design units are in the same file visibility of libraries and packages must be established for each primary design unit (entity)
- The use clause may selectively establish visibility
  - only the function rising_edge() is visible within entity design2

```
library IEEE;
use IEEE.std_logic_1164.all;
entity design1 is
.....

library IEEE;
use IEEE.std_logic_1164.rising_edge;
entity design2 is
.....                                  file.vhd
```

Examples of Packages

numeric_std

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

- Define operations on `std_logic_vector`
- Define types `unsigned` and `signed`
- Define functions on these types
  - Arithmetic: `+,-,*,/,rem,mod`
  - Comparison: `<,<=,>,>=,=,/=`
  - Logical: `not, and, or, nand, nor, xor, xnor`
  - Shift and Rotate: `SHIFT_LEFT, SHIFT_RIGHT, ROTATE_LEFT, ROTATE_RIGHT, sll, srl, rol, ror`
- Conversion functions
  - `TO_INTEGER, TO_UNSIGNED, TO_SIGNED`
- Resize function
  - `RESIZE(v,n)`: Unsigned/ Signed vector extension

std_logic_arith (Synopsys)

- Define types unsigned and signed
- Define arithmetic functions +,-,* on these types
- Define logical functions <,<=,>,>=,=,/=
- Define conversion functions
  - CONV_INTEGER
  - CONV_UNSIGNED
  - CONV_SIGNED
  - CONV_STD_LOGIC_VECTOR
- Unsigned vector extension: EXT
- Signed vector extension: SXT

fixed_pkg

use.ieee.fixed_pkg.all

- Library for fixed-point arithmetic
- Defines types ufixed and sfixed
  - Unsigned/Signed fixed-point numbers

```
type ufixed is array (INTEGER range <>) of STD_LOGIC;
type sfixed is array (INTEGER range <>) of STD_LOGIC;
```

  - Positive indexes for integer part
  - Negative indexes for fractional part

```
signal sa, sb : sfixed (7 downto -6);
signal sc: sfixed (8 downto -6);
signal ua,ub : ufixed (7 downto -6);
signal uc: ufixed (8 downto -6);
begin
sc <= sa + sb;
uc <= ua + ub;
```

fixed_pkg

- Conversion functions from real numbers

```
signal u1,u2 : ufixed(4 downto -4);
signal s1,s2 : sfixed(4 downto -4);
...
u1 <= to_ufixed (9.75,4,-4);                    -- "1001.1100“
u1 <= to_ufixed (9.75,u1);
u1 <= to_ufixed (9.75,u1'high,u1'low);
s1 <= to_sfixed (-9.75,4,-4);                   -- "1.0110.0100"
```

  – ufixed (4 downto -4): 9,75 coded as “1001.1100”
  – sfixed (4 downto -4): -9,75 coded as “1_0110_0100”
  9,75 coded as “0_1001_1100”

fixed_pkg

- Resizing

```
signal u1: ufixed(4 downto -3);
signal u2: ufixed(5 downto -5);
u1 <=  to_ufixed (9.75,u1);   -- u1 = "01001110"
u2 <= resize(u1,u2);          -- u2 ="00100111000“
u2 <= resize(u1,n2'high,u2'low);
u2 <= resize(u1,5,-5);
```

- Overflow

```
u2 <=  to_ufixed (32.75,u2);      -- u2 ="10000011000"
U1 <= resize(u2,u1);              -- u1 = "11111111"
```

- Beware of assignment without resizing

```
signal u1: ufixed(4 downto -3);
signal u2: ufixed(3 downto -4);  -- u1’length = u2’length
u1 <=  to_ufixed (5.75,u1);      -- u1 = "00101110" = 5.75
u2 <= u1;                        -- u2 = "00101110" = 2.875
```

fixed_pkg

- Operations

```
signal n1,n2 : ufixed(4 downto -3);
signal n3 : ufixed(5 downto -3); -- 5=4+1
begin
n1 <=  to_ufixed (5.75,n1);      -- n1 = "00101110" = 5.75
n2 <=  to_ufixed (6.5,n2);       -- n2 = "00110100" = 6.5
n3 <= n1+n2;                    -- n3 = "001100010" = 12.25
!!Warning!! if n3 is ufixed(4 downto 0) then n3="00110001" = 6.125
```

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

fixed_pkg

- ufixed_high, ufixed_low, sfixed_high, sfixed_low
  - Enable for signals to be declared without knowledge of operation result sizes

operation	Operator assumed
'+'	"+"
'-'	"-"
'*'	"*"
'/'	"/", divide
'1'	reciprocal
'M' or 'm'	"mod", modulo
'R' or 'r'	"rem", remainder
'A' or 'a'	abs
'N' or 'n'	"-" - unary
others	index

```
signal a : sfixed (5 downto –3);
signal b : sfixed (7 downto –9);

signal adivb : sfixed (sfixed_high (5,-3, '/', 7,-9)
                       downto sfixed_low (5,-3, '/', 7,-9));
signal adivb:sfixed (sfixed_high (a'high, a'low, '/', b'high, b'low)
                 downto sfixed_low (a'high, a'low, '/', b'high, b'low));
signal adivb:sfixed(sfixed_high (a,'/',b) downto sfixed_low (a,'/',b));
```

fixed_generic_pkg.vhd

- Generic package used to define fixed_pkg

```
use IEEE.fixed_float_types.all;
package fixed_generic_pkg is
generic (
   --Round style, fixed_round, fixed_truncate
fixed_round_style : fixed_round_style_type := fixed_round;
   --Overflow style, fixed_saturate (largest possible number),
   --fixed_wrap (discard high bits)
fixed_overflow_style : fixed_overflow_style_type := fixed_saturate;
   --Guard bits are added to the bottom of some operation for rounding.
   --any natural number (including 0) are valid.
fixed_guard_bits : NATURAL := 3;
   --when "false" issue warnings, when "true" be silent.
no_warning : BOOLEAN := false
);
```


fixed_generic_pkg.vhd

- Generic package used to define fixed_pkg

```
package fixed_pkg is
new work.fixed_generic_pkg
generic map (
 fixed_round_style => IEEE.fixed_float_types.fixed_round,
 fixed_overflow_style => IEEE.fixed_float_types.fixed_saturate,
 fixed_guard_bits => 3,  --number of guard bits
 no_warning => false ); --show warnings
```

- Or to define your own package

```
package my_fixed_pkg is new ieee.fixed_generic_pkg
generic map (
 fixed_round_style => IEEE.fixed_float_types.fixed_truncate,
 fixed_overflow_style => IEEE.fixed_float_types.fixed_wrap,
 fixed_guard_bits => 0, --Don't need the extra guard bits
 no_warning => true );  --turn warnings off
```

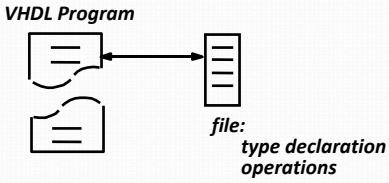
std.env

```
procedure stop: pauses simulation
procedure finish: stops simulation
procedure resolution_limit: returns simulator
  resolution
…
```


Inputs and Outputs

File Objects and Declarations

- The file type permits us to declare and use file objects



- Files can be distinguished by the type of information stored

```
type text is file of string;
type IntegerFileType is file of integer;
```

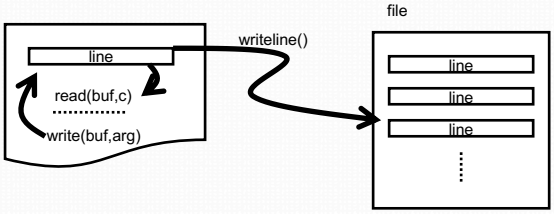

Binary File I/O (VHDL 1993)

- VHDL provides read(f,value), write(f, value) and endfile(f)
- VHDL 93 also provides File_Open() and File_Close()
- Explicit vs. implicit file open operations

```
entity io93 is
end entity io93;
architecture behavioral of io93 is
begin
process
  type IntFileType is file of integer;
  -- file declarations
  file dataout :IntFileType;
  variable count : integer:= 0;
  variable fstatus: FILE_OPEN_STATUS;
begin
  file_open(fstatus, dataout,
       "myfile.txt", write_mode);
  for j in 1 to 8 loop
    write(dataout,count);
    count := count+2;
  end loop;
  wait; -- an artificial way to stop the process
end process;
end architecture behavioral;
```

The TEXTIO Package

- A file is organized by *lines*
- read() and write() procedures operate on line data structures
- readline() and writeline() procedures transfer data from-to files
- All procedures encapsulated in the TEXTIO package in the STD lib.
  - Procedures for reading and writing the pre-defined types from lines
  - Pre-defined access to std_input and std_output
  - Overloaded procedure names



Example: Use of the TEXTIO Package

- This is an example of formatted IO
- The First Parameter is = 5 The Second Parameter is = 0110
- ...and so on

```
use STD.Textio.all;
entity formatted_io is
end formatted_io;
architecture behavioral of formatted_io is
begin
process is
  file outfile :text;
  -- declare the file to be a text file
  variable fstatus :File_open_status;
  variable count: integer := 5;
  variable val: bit_vector(3 downto 0):= X"6";
  variable buf: line; -- buffer to file
begin
  file_open(fstatus, outfile, "myfile.txt",
  write_mode); -- open the file for writing
```

```
L1: write(buf, "This is an example of
formatted I/O");
L2: writeline(outfile, buf);
-- write buffer to file
L3: write(buf, "The First Parameter is = ");
L4: write(buf, count);
L5: write(buf, ' ');
L6: write(buf, "The Second Param. is = ");
L7: write(buf, val);
L8: writeline(outfile, buf);
L9: write(buf, "...and so on");
L10: writeline(outfile, buf);
L11: file_close(outfile);
-- flush the buffer to the file
wait;
end process;
end architecture behavioral;
```

Extending TEXTIO for Other Datatypes

- Hide the ASCII format of TEXTIO from the user
- Create type conversion procedures for reading and writing desired datatypes, e.g., std_logic_vector
- Encapsulate procedures in a package
- Install package in a library and make its contents visible via the use clause

Example: Type Conversion

- Text based type conversion for user defined types
- Note: writing values vs. ASCII codes

```
procedure write_v1d (variable f: out
text; v: in std_logic_vector) is
  variable buf: line;
  variable c : character;
begin
  for i in v'range loop
    case v(i) is
        when 'X' => write(buf, 'X');
        when 'U' => write(buf, 'U');
        when 'Z' => write(buf, 'Z');
        when '0' => write(buf,
                character'('0'));
        when '1' => write(buf,
                character'('1'));
        when '-' => write(buf, '-');
        when 'W' => write(buf, 'W');
        when 'L' => write(buf, 'L');
        when 'H' => write(buf, 'H');
        when others => write(buf,
                character'('0'));
    end case;
  end loop;
  writeline (f, buf);
end procedure write_v1d;
```

Example: Type Conversion

- read() is a symmetric process

```
procedure read_v1d (variable f: in
   text; v : out std_logic_vector) is
  variable buf: line;
  variable c : character;
begin
  readline(f, buf);
  for i in v'range loop
  read(buf, c);
  case c is
    when 'X' => v (i) := 'X';
    when 'U' => v (i) := 'U';
    when 'Z' => v (i) := 'Z';
    when '0' => v (i) := '0';
    when '1' => v (i) := '1';
    when '-' => v (i) := '-';
    when 'W' => v (i) := 'W';
    when 'L' => v (i) := 'L';
    when 'H' => v (i) := 'H';
    when others => v (i) := '0';
  end case;
 end loop;
end procedure read_v1d
```


Useful Code Blocks

- Formatting the output

```
write (buf, "This is the header");
writeline (outfile,buf);
write (buf, "Clk =");
write (buf, clk);
write (buf, ", N1 =");
write (buf, N1);
```

- Text output will appear as follows

```
This is the header
Clk = 0, N1 = 01001011
```

Useful Code Blocks

- Reading formatted input lines

```
# this file is parsed to separate comments
0001 65 00Z111Z0
0101 43 0110X001
```

bit vector integer std_logic_vector

- The code block to read such files may be

```
while not (endfile(vectors) loop
readline(vectors, buf);
if buf(1) = '#' then
    continue;
end if;
read(buf, N1);
read (buf, N2);
read (buf, std_str);
```

convert to std_logic_vector

Useful Code Blocks: Filenames

```
process is
variable buf : line;
variable fname : string(1 to 10);
begin
--
-- prompt and read filename from standard input
--
write(output, "Enter Filename: ");
readline(input,buf);
read(buf, fname);
--
-- process code
--
end process;
```

- Assuming "input" is mapped to simulator console
  - Generally "input" and "output" are mapped to standard input and standard output respectively

Useful Code Blocks: Testing Models

```
library IEEE;

use IEEE.std_logic_1164.all;
use STD.textio.all;
use work.classio.all;
-- the package classio has been compiled into the working directory

entity checking is
end checking; -- the entity is an empty entity

architecture behavioral of checking is
begin

-- use file I/O to read test
-- vectors and write test results

end architecture behavioral;
```

my I/O library

Testing process

Useful Code Blocks: Testing Models

```
process is
  -- use implicit file open
  file infile : TEXT open read_mode is "infile.txt";
  file outfile : TEXT open write_mode is "outfile.txt";
  variable check : std_logic_vector (15 downto 0) := x"0008";
begin
  -- copy the input file contents to the output file
  while not (endfile (infile)) loop
    read_v1d (infile, check);
    --
    -- Model to test here
    --
    write_v1d (outfile, check);
  end loop;
  file_close(outfile); -- flush buffers to output file
  wait; -- artificial wait for this example
end process;
```

Useful Code Blocks: Assert Example

- Report statements may be used in isolation

```
architecture check_times of DFF is
  constant hold_time: time:=5 ns;
  constant setup_time : time:= 2 ns;
begin
process
  variable lastevent: time;
begin
  if d'event then
    assert NOW = 0 ns or (NOW - lastevent) >= hold_time
    report "Hold time too short"
    severity FAILURE;
    lastevent := NOW;
  end if;
  -- check setup time
  -- D flip flop behavioral model
end process;
end architecture check_times
```


Useful Code Blocks: Stimuli Generation

- Tester module to generate periodic signals and apply test vectors

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use WORK.classio.all; -- declare the I/O package
entity srtester is -- this is the module generating the tests
  port (R, S, D, Clk : out std_logic;
        Q, Qbar : in std_logic);
end entity srtester;
architecture behavioral of srtester is
begin
  clk_process: process -- generates the clock waveform with
  begin -- period of 20 ns
    Clk<= '1', '0' after 10 ns, '1' after 20 ns, '0' after 30 ns;
    wait for 40 ns;
  end process clk_process;
```

Useful Code Blocks: Stimuli Generation (cont.)

```
io_process: process          -- this process performs the test
  file infile : TEXT is in "infile.txt"; -- functions
  file outfile : TEXT is out "outfile.txt";
  variable buf : line;
  variable msg : string(1 to 19) := "This vector failed!";
  variable check : std_logic_vector (4 downto 0);
begin
  while not (endfile (infile)) loop -- loop through all test vectors in
    read_v1d (infile, check);        -- the file
    -- make assignments here
    wait for 20 ns;   -- wait for outputs to be available after applying
    if (Q /= check (1) or (Qbar /= check(0))) then -- error check
        write (buf, msg);
        writeline (outfile, buf);
        write_v1d (outfile, check);
    end if;
  end loop;
  wait; -- important to allow the simulation to halt!
end process io_process;
end architectural behavioral;
```


Useful Code Blocks: Stimuli Generation

- Test generation vs. File I/O: how many vectors would be need?

```
process
begin
  databus <= (others => '0');
  for N in 0 to 65536 loop
     databus <= to_unsigned(N,16) xor shift_right(to_unsigned(N,16),1);
     for M in 1 to 7 loop
       wait until rising_edge(clock);
     end loop;
     wait until falling_edge(Clock);
  end loop;
--
-- rest of the the test program
--
end process;
```

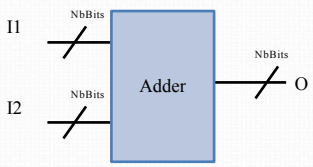
Useful Code Blocks: Stimuli Generation (cont.)

```
while not endfile(vectors) loop
 readline(vectors, vectorline); -- file format is 1011011
 if (vectorline(1) = '#' then
   next;
 end if;
 read(vectorline, datavar);
 read((vectorline, A); -- A, B, and C are two bit vectors
 read((vectorline, B); -- of type std_logic
 read((vectorline, C);

 -- signal assignments
 Indata <= to_stdlogic(datavar);
 A_in <= unsigned(to_stdlogicvector(A)); -- A_in, B_in and C_in are of
 B_in <= unsigned(to_stdlogicvector(B)); -- unsigned vectors
 C_in <= unsigned(to_stdlogicvector(C));
 wait for ClockPeriod;
end loop;
```


Examples: Combinational Logic

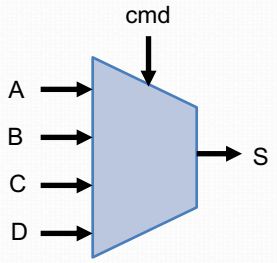
- Generic Adder




```
ENTITY Adder IS
  GENERIC (Tadd: TIME ; NbBits: INTEGER);
  PORT    (I1: IN  std_logic_vector(NbBits-1 DOWNTO 0);
           I2: IN std_logic_vector(NbBits-1 DOWNTO 0);
           O: OUT std_logic_vector(NbBits-1 DOWNTO 0));
END Adder;
ARCHITECTURE RTL_sim OF Adder IS
BEGIN
PROCESS
  VARIABLE e1, e2, s: INTEGER;
BEGIN
  wait on entree1, entree2;
  e1 := Conv_Integer (I1);
  e2 := Conv_Integer (I2);
  s := e1 + e2;
  O <= Conv_Std_Logic_Vector (s, NbBits) AFTER Tadd;
END PROCESS;
END RTL_sim;
```

Examples: Combinational Logic

- Multiplexer




```
ENTITY MUX IS
 GENERIC (Tmux: TIME; NbBits : INTEGER);
 PORT     (A,B,C,D: IN  std_logic_vector(NbBits-1 DOWNTO 0);
          cmd: IN std_logic_vector(1 DOWNTO 0);
          S: OUT std_logic_vector(NbBits-1 DOWNTO 0));
END MUX;
ARCHITECTURE dataflow OF MUX IS
BEGIN
  S <= A WHEN (cmd = "00")
     ELSE B WHEN (cmd = "01")
     ELSE C WHEN (cmd = "10")
     ELSE D;
END dataflow;
```

```
ARCHITECTURE dataflow2 OF MUX IS
BEGIN
  WITH cmd SELECT
        S <=     A  WHEN (cmd = "00")
                 B  WHEN (cmd = "01")
                 C  WHEN (cmd = "10")
                 D  WHEN OTHERS;
END dataflow2;
```

```
ARCHITECTURE beh OF MUX IS
BEGIN
PROCESS (cmd, A, B, C, D)
BEGIN
  IF cmd = "00" THEN
         S <= A  ;
  ELSIF cmd = "01" THEN
         S <= B;
  ELSIF cmd = "10" THEN
         S <= C;
  ELSIF cmd = "11" THEN
         S <= D;
  END IF;
END PROCESS;
END beh;
```


Examples: Combinational Logic

- Decoder

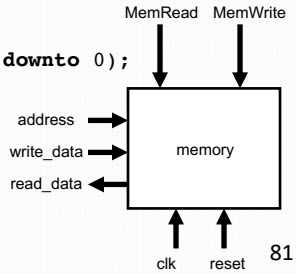
```
library ieee;
use ieee.numeric_std.all;
entity decoder is
        generic (NbOut : integer;
                 Log2NbOut : integer) ;
        port (O: out std_logic_vector (NbOut-1 downto 0) ;
              cmd: in std_logic_vector (log2NbOut-1 downto 0));
end decoder;
architecture dataflow of decodeur is
begin
 assign: for i in O'range generate
        O(i) <= '1' when to_integer(unsigned(cmd)) = i
                    else '0';
 end generate assign;
end dataflow;
```

Processes + Conditional Signal Assignments

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity memory is
port (address, write_data: in std_logic_vector (7 downto 0);
      MemWrite, MemRead, clk, reset: in std_logic;
      read_data: out std_logic_vector (7 downto 0));
end entity memory;

architecture behavioral of memory is
  signal dmem0,dmem1,dmem2,dmem3: std_logic_vector (7 downto 0);
begin
mem_proc: process (clk) is
-- process body for memory write operation
end process mem_proc;
-- read operation CSA
end architecture behavioral;
```



Process + CSAs: The Write Process

```
mem_proc: process (clk) is
begin
  if (rising_edge(clk)) then -- wait until next clock edge
    if reset = '1' then -- initialize values on reset
        dmem0 <= x"00";  -- memory locations are initialized to
        dmem1 <= x"11";  -- some random values
        dmem2 <= x"22";
        dmem3 <= x"33";
    elsif MemWrite = '1' then -- if not reset then check for memory write
        case address (1 downto 0) is
          when "00" => dmem0 <= write_data;
          when "01" => dmem1 <= write_data;
          when "10" => dmem2 <= write_data;
          when "11" => dmem3 <= write_data;
          when others => dmem0 <= x"ff";
        end case;
    end if;
  end if;
end process mem_proc;
```

Process + CSAs: The Read Statement

```
-- memory read is implemented with a conditional signal assignment
read_data <=   dmem0 when address (1 downto 0) = "00" and MemRead = '1' else
               dmem1 when address (1 downto 0) = "01" and MemRead = '1' else
               dmem2 when address (1 downto 0) = "10" and MemRead = '1' else
               dmem3 when address (1 downto 0) = "11" and MemRead = '1' else
               x"00";
```


Example: ln2 function

```
package util_pkg is
   function ln2(n: integer) return natural;
end package;

package body util_pkg is
-- n=17 result=5; n=16 result=4; n=15 result=4
function ln2(n: integer) return natural is
   variable b: integer :=n;
   variable result: integer;
begin
   if n<1 then
      result:= 0;
   else
      result:=1;
      b:=n-1;
      while b>1 loop
         result:=result+1;
         b:=b/2;
      end loop;
    end if;
    return result;
end;
end util_pkg;
```

Example: ln2 function

```
use work.util_pkg.all;
entity decoder is
  generic (NbOut: integer) ;
  port (O: out std_logic_vector (NbOut-1 downto 0);
        cmd: in std_logic_vector (ln2(NbOut)-1 downto 0));
end decoder;

architecture df of decodeur is
begin
       assign: for i in O'range generate
          O(i) <= '1' when conv_positif(cmd) = i
               else '0';
       end generate assign;
end df;
```


Relational Operators

- "?=", "?/=","?<", "?<=", "?>","?>="
- Return bit or std_logic

```
Reg1Sel <= Cs1 and not (nCs2) and (Addr ?="101000") ;

-- equivalent to
if Cs1 and not (nCs2) and (Addr ?= "1010--") then
-- "?=" understands '-' as don't care
```

Examples: Case

```
process (Request)
begin
case? Request is -- each case choice must be non-overlapping
  when "1---" => Grant <= "1000" ;
  when "01--" => Grant <= "0100" ;
  when "001-" => Grant <= "0010" ;
  when "0001" => Grant <= "0001" ;
  when others => Grant <= "0000" ;
end case? ;
end process ;
```

Case with don't care

```
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 -- 2008
  when "0000" => Y <= "00" ;
  when CHOICE2 => Y <= "01" ; -- 2008
  when "0110" => Y <= "10" ;
  when ONE1 & "00" => Y <= "11" ; -- 2008
  when others => Y <= "XX" ;
end case ;
end process ;
```


Examples: Case Generate

```
g1: case mode generate
when 0 =>
   c1 : use entity work.comp(rtl1) port map (a => a, b=>b);
when 1 =>
   c1 : use entity work.comp(rtl2) port map (a => a, b=>b);
when 2 =>
   c1 : use entity work.comp(rtl3) port map (a => a, b=>b);
end generate;
```

Examples: others, downto

```
variable V : std_logic_vector(7 downto 0);
begin
V := (others => '0'); -- "00000000"
V := ("10", others => '0'); -- "10000000" (2008)
V := (4 => '1', others => '0'); -- "00010000"
V := (3 downto 0 => '0', others => '1'); -- "11110000"
V := ("0000", others => '1'); -- "00001111“ (2008)

(S(3 downto 0), S(7 downto 4)) <= S;      -- swap nibbles
(3 downto 0 => S, 7 downto  4 => S) <= S; -- using named association
```


Examples

- Hierarchical references (^: up, @: root)

```
alias int1 << signal .tb.uut.o_n : std_logic >>; -- hierarchical signal name
alias int2 << signal ^.^.a : std_logic >>; -- signal a two levels above
alias int3 << variable @lib.pack.v : bit >>; -- variable in a package (shared)
alias int4 << signal .tb_top.u_ioc.int1 : std_logic >>;
```

- Force, release

```
int4 <= force ‘1’;
...
int4 <= force ‘0’;
...
int4 <= release;
```

- Sensitivity list
  - All: specify all read signals

```
process (all)
begin
case state is
   when idle =>
      if in1 then nextState <= Go1;
      end if;
   when Go1 =>
      nextState <= Go2;
   when Go2 =>
      nextState <= Idle;
end case;
end process;
```

Example: State Machine

```
library IEEE;
use IEEE.std_logic_1164.all;
entity state_machine is
  port(reset, clk, x : in std_logic; z : out std_logic);
end entity state_machine;
architecture behavioral of state_machine is
  type statetype is (state0, state1);
  signal state, next_state: statetype := state0;
begin
  comb_process: process (state, x) is
  begin
    -- process description here
  end process comb_process;
  clk_process: process is
  begin
    -- process description here
  end process clk_process;
end architectural behavioral;
```


Example: Output and Next State Functions

```
comb_process: process (state, x) is
begin
case state is -- depending upon the current state
  when state0 => -- set output signals and next state
    if x = '0' then
        next_state <= state1;
        z <= '1';
    else next_state <= state0;
        z <= '0';
    end if;
  when state1 =>
    if x = '1' then
        next_state <= state0;
        z <= '0';
    else next_state <= state1;
        z <= '1';
    end if;
  end case;
end process comb_process;
```

Combination of the next state and output functions

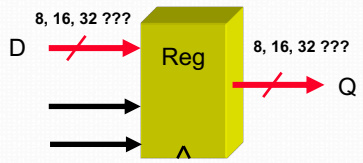
Example: Clock Process

- Use of synchronous reset to initialize into a known state

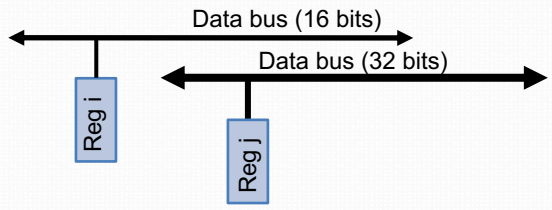
```
clk_process: process is
begin
  wait until (clk'event and clk = '1');
  -- wait until the rising edge
  if reset = '1' then
  -- check for reset and initialize state
    state <= statetype'left;
  else state <= next_state;
  end if;
end process clk_process;
end behavioral;
```


Example: N-bit Register

```
entity generic_reg is
  generic (n: positive:=2);
  port (clk, reset, enable : in std_logic;
        d : in std_logic_vector (n-1 downto 0);
        q : out std_logic_vector (n-1 downto 0));
end entity generic_reg;
architecture behavioral of generic_reg is
begin
  reg_process: process (clk, reset)
  begin
    if reset = '1' then
          q <= (others => '0');
    elsif (rising_edge(clk)) then
          if enable = '1' then q <= d;
          end if;
    end if;
  end process reg_process;
end behavioral;
```



Example: Using the Generic Register




```
architecture structure of system is
        signal BusData : std_logic_vector(15 downto 0);
        signal BusAddress : std_logic_vector(31 downto 0);
        signal SLoadi, SLoadj, SClock, SEnablei, SEnablej : std_logic;
begin
        Regi : generic_reg
                generic map (16)
                port map (BusData, SLoadi, SEnablei, SClock, BusData);
        Regj : generic_reg
                generic map (32)
                port map (BusAddress, SLoadj, SEnablej, SClock, BusAddress);
end structure;
```


Synthesis Hints

- Initialization
  - Do not specify initial values in your declaration of signals. Most synthesis compilers will ignore them
  - Specify the number of bits necessary for a signal explicitly in the declaration
- Inferring Storage
  - Avoid latch inferred for a signal in a process by having every execution path through the process assigned a value for that signal
  - Using variables in a process before they are defined infers a latch for that variable
  - To avoid the inference of latches, make sure that default values are assigned to signals before a conditional block or a loop
  - To ensure that combinational logic is generated from a process or concurrent signal assignment statements (conditional or selected) every possible execution path through the code must determine all output values

Synthesis Hints

- Inferring Storage
  - Use if-then statements to infer flip flops rather than wait statements
- Optimizations
  - Using don't care values to cover the when others case in a case statement or selected signal assignment statement can enable the synthesis compiler to optimize the logic
  - Move common complex operations out of the branches of if-then-else statements and place them after the conditional code
  - Using a case statement rather than an if-then-elsif construct will produce less logic (no priority logic)
  - Use parentheses in simple concurrent signal assignments statements to control concurrency
  - Use of the selected signal assignment statement will generally produce less logic than conditional signal assignment statements
  - The for-loop is synthesized by first unrolling the loop. Loop dependencies, where computations in one iteration are dependent on computations in another iteration, can lead to long signal paths.
  - Minimize signal assignment statements within a process and use variables

Synthesis Hints

- Optimizations
  - Do not use don't care symbols in comparisons (no hardware equivalent)
  - Check vendor specific constraints on the permitted types and range of the for-loop index
  - Keep in mind that the code should "imply" hardware structures
  - All loop indices must have statically determinable loop ranges
  - The while-loop statement is generally not supported for synthesis
  - The choice of level sensitive conditional expressions vs. edge detection expressions in your VHDL code should be guided by the parts available in your target library. For example, if latches are not available then the synthesis tools may try to create a latch by synthesizing the gate level equivalents (e.g. reset, clock edge)
- Consistency with Pre-Synthesis Functional Simulation
  - Simulation models delays using the after clause. During synthesis the delay values of the operations are derived from the synthesized implementation
  - Comparisons maybe modeled in the simulation differently from that actually synthesized into hardware (simulation can have U,X,W,… hardware only have 0,1)
  - Include all signals in a process in the sensitivity list of the process to avoid pre-synthesis and post-synthesis simulation mismatches

Synthesis Hints

- Consistency with Pre-Synthesis Functional Simulation
  - Conditional and selected signal assignment statements are equivalent to process construct. CSA are always active and more simulation overhead but are better for synthesis
  - Use of variables will lead to faster simulation but may reduce the effectiveness of the inference mechanisms. The lesson is that writing code for optimal simulation speed is not the same as optimal synthesis