VHDL MINI-REFERENCE

See the VHDL Language Reference Manual (VLRM) for Additional Details

The following Mini-Reference can be divided into the following parts:

PRIMARY DESIGN UNIT MODEL STRUCTURE - Back To Top

Each VHDL design unit comprises an "entity" declaration and one or more "architectures". Each architecture defines a different implementation or model of a given design unit. The entity definition defines the inputs to, and outputs from the module, and any "generic" parameters used by the different implementations of the module.

Entity Declaration Format - Back To Top

```
entity  name  is
    port( port definition list );-- input/output signal ports
```

```
    generic( generic list);   -- optional generic list
end name;
```

Port declaration format: *port_name: mode data_type;*

The *mode* of a port defines the directions of the singals on that pirt, and is one of: **in**, **out**, **buffer**, or **inout**.

Port Modes:

An **in** port
: can be read but not updated within the module, carrying information into the module. (An in port cannot appear on the left hand side of a signal assignment.)

An **out** port
: can be updated but not read within the module, carrying information out of the module. (An out port cannot appear on the right hand side of a signal assigment.)

A **buffer** port
: likewise carries information out of a module, but can be both updated and read within the module.

An **inout** port
: is bidirectional and can be both read and updated, with multiple update sources possible.

- NOTE: A buffer is strictly an output port, i.e. can only be driven from within the module, while inout is truly bidirectional with drivers both within and external to the module.

Example

```
entity counter is
     port (Incr, Load, Clock: in     bit;
           Carry:             out    bit;
           Data_Out:          buffer bit_vector(7 downto 0);
           Data_In:           in     bit_vector(7 downto 0));
end counter;
```

Generics allow static information to be communicated to a block from its environment for all architectures of a design unit. These include timing information (setup, hold, delay times), part sizes, and other parameters.

Example

```
entity and_gate is
    port(a,b: in  bit;
         c:   out bit);
    generic (gate_delay: time := 5ns);
end and_gate;
```

Architecture - Back To Top

An architecture defines one particular implementation of a design unit, at some desired level of abstraction.

```
architecture arch_name of entity_name is
    ...  declarations ...
begin
    ...  concurrent statements  ...
end
```

Declarations include data types, constants, signals, files, components, attributes, subprograms, and other information to be used in the implementation description. *Concurrent statements* describe a design unit at one or more levels of modeling

abstraction, including dataflow, structure, and/or behavior.

- **Behavioral Model:** No structure or technology implied. Usually written in sequential, procedural style.
- **Dataflow Model:** All datapaths shown, plus all control signals.
- **Structural Model:** Interconnection of components.

VHDL PACKAGES - Back To Top

A VHDL *package* contains subprograms, constant definitions, and/or type definitions to be used throughout one or more design units. Each package comprises a "declaration section", in which the available (i.e. exportable) subprograms, constants, and types are declared, and a "package body", in which the subprogram implementations are defined, along with any internally-used constants and types. The declaration section represents the portion of the package that is "visible" to the user of that package. The actual implementations of subroutines in the package are typically not of interest to the users of those subroutines.

Package declaration format:

```
package package_name is
  ... exported constant declarations
  ... exported type declarations
  ... exported subprogram declarations
end package_name;
```

Example:

```
 package ee530 is
    constant maxint: integer := 16#ffff#;
    type arith_mode_type is (signed, unsigned);
    function minimum(constant a,b: in integer) return integer;
 end ee530;
```

Package body format:

```
package body package_name is
    ... exported subprogram bodies
    ... other internally-used declarations
end package_name;
```

Example:

```
package body ee530 is
   function minimum (constant a,b: integer) return integer is
      variable c: integer; -- local variable
          begin
             if a < b then
                 c := a;  -- a is min
             else
                 c := b;  -- b is min
             end if;
             return c;  -- return min value
          end;
 end ee530;
```

Package Visibility

To make all items of a package "visible" to a design unit, precede the desired design unit with a "use" statement:

Example:

```
use library_name.package_name.all
```

A "use" statement may precede the declaration of any entity or architecture which is to utilize items from the package. If the "use" statement precedes the entity declaration, the package is also visible to the architecture.

User-Developed Packages

Compile user-developed packages in your current working library. To make it visible:

```
use package_name.all;
```

Note: '**std**' and '**work**' (your current working library) are the two default libraries. The VHDL 'library' statement is needed to make the 'ieee' library and/or additional libraries visible.

Example

```
library lib_name;            -- make library visible
use lib_name.pkg_name.all;   -- make package visible
```

VHDL Standard Packages

STANDARD - basic type declarations (always visible by default)
TEXTIO - ASCII input/output data types and subprograms

To make TEXTIO visible: **use** std.textio.all;

IEEE Standard 1164 Package

This package contained in the 'ieee' library supports multi-valued logic signals with type declarations and functions. To make visible:

```
library ieee;      -- VHDL Library stmt
   use ieee.std_logic_1164.all;
```

Special 12-valued data types/functions to interface with QuickSim II and schematic diagrams.

```
library mgc_portable;            -- Special Mentor Graphics Library
use mgc_portable.qsim_logic.all; -- Quicksim portable data types
```

VHDL IDENTIFIERS, NUMBERS, STRINGS, AND EXPRESSIONS - Back To Top

Identifiers

Identifiers in VHDL must begin with a letter, and may comprise any combination of letters, digits, and underscores. Note that VHDL internally converts all characters to UPPER CASE.

Examples

```
Memory1, Adder_Module, Bus_16_Bit
```

Numeric Constants

Numeric contants can be defined, and can be of any base (default is decimal). Numbers may include embedded underscores to improve readability.

Format: **base#digits#** -- base must be a decimal number

Examples

```
16#9fba#           (hexadecimal)
2#1111_1101_1011#  (binary)
16#f.1f#E+2        (floating-point, exponent is decimal)
```

Bit String Literals

Bit vector constants are are specified as literal strings.

Examples

```
x"ffe"            (12-bit hexadecimal value)
o"777"            (9-bit octal value)
b"1111_1101_1101" (12-bit binary value)
```

Arithmetic and Logical Expressions

Expressions in VHDL are similar to those of most high-level languages. Data elements must be of the type, or subtypes of the same base type. Operators include the following:

- Logical: **and**, **or**, **nand**, **nor**, **xor**, **not** (for boolean or bit ops)
- Relational: =, /=, <, <=, >, >=
- Arithmetic: +, -, *, /, mod, rem, **, abs

  (a mod b takes sign of b, a rem b takes sign of a)
- Concatenate: **&**

  (ex. a & b makes one array)

Examples

```
a <= b nand c;
d := g1 * g2 / 3;
Bus_16 <= Bus1_8 & Bus2_8;
```

VHDL DATA TYPES - Back To Top

Each VHDL objects must be classified as being of a specific data type. VHDL includes a number of predefined data types, and allows users to define custom data types as needed.

Predefined Scalar Data Types (single objects)

VHDL Standard:

- **bit** values: '0', '1'
- **boolean** values: TRUE, FALSE
- **integer** values: -(231) to +(231 - 1) {SUN Limit}
- **natural** values: 0 to integer'high (subtype of integer)
- **positive** values: 1 to integer'high (subtype of integer)
- **character** values: ASCII characters (eg. 'A')
- **time** values include units (eg. 10ns, 20us)

IEEE Standard 1164 (package ieee.std_logic_1164.all)

- **std_ulogic** values: 'U','X','1','0','Z','W','H','L','-'
  'U' = uninitialized
  'X' = unknown
  'W' = weak 'X'
  'Z' = floating
  'H'/'L' = weak '1'/'0'
  '-' = don't care
- **std_logic** resolved "std_ulogic" values
- **X01** subtype {'X','0','1'} of std_ulogic
- **X01Z** subtype {'X','0','1','Z'} of std_ulogic
- **UX01** subtype {'U','X','0','1'} of std_ulogic
- **UX01Z** subtype {'U','X','0','1','Z'} of std_ulogic

Predefined VHDL Aggregate Data Types

- **bit_vector** array (natural range <>) of bit
- **string** array (natural range <>) of char
- **text** file of "string"

IEEE Standard 1164 Aggregate Data Types

(From package: *ieee.std_logic_1164.all*)

- **std_ulogic_vector** array (natural range <>) of std_ulogic
- **std_logic_vector** array (natural range <>) of std_logic

Examples

```
signal dbus: bit_vector(15 downto 0);
dbus (7 downto 4) <= "0000"; (4-bit slice of dbus)
```

```
signal cnt:  std_ulogic_vector(1 to 3);
variable message: string(0 to 20);
```

User-Defined Enumeration Types

An enumerated data type can be created by explicitely listing all possible values.

Example

```
type opcodes is (add, sub, jump, call);  -- Type with 4 values
signal instruc: opcodes;                 -- Signal of this type
  ...

if instruc = add then   -- test for value 'add'
  ...
```

Other user-defined types

Custom data types can include arrays, constrained and unconstrained, and record structures.

- *Constrained array:* Upper and lower indexes are specified.

  ### Example

  ```
  type word is array (0 to 15) of bit;
  ```

- *Unconstrained array:* Indexes are specified when a signal or variable of that type is declared.

  ### Examples

  ```
  type memory is array (integer range <>) of bit_vector(0 to 7);
 -- a type which is an arbitrary-sized array of 8-bit vectors
  variable memory256: memory(0 to 255); -- a 256-byte memory array
  variable stack: memory(15 downto 0);  -- a 16-byte memory array
  ```

- *Subtype:* A selected subset of values of a given type. Elements of different subtypes having the same base type may be combined in expressions (elements of different types cannot). Subtypes can be used to detect out-of-range values during simulation.

  ### Examples

  ```
  subtype byte_signed is integer range -128 to 127;
  subtype byte_unsigned is integer range 0 to 255;
  ```

Aliases

An alias" defines an alternate name for a signal or part of a signal. Aliases are often used to refer to selected slices of a bit_vector.

Example

```
signal instruction: bit_vector(31 downto 0);
alias opcode: bit_vector(6 downto 0) is instruction(31 downto 25);
```

```
  ...
opcode <= "1010101";  -- Set the opcode part of an instruction code
```

VHDL OBJECTS: CONSTANTS, VARIABLES, AND SIGNALS - Back To Top

Constants

A *constant* associates a value to a symbol of a given data type. The use of constants may improve the readability of VHDL code and reduce the likelihood of making errors. The declaration syntax is:

constant symbol: type := value;

Examples

```
constant  Vcc:  signal:= '1';   --logic 1 constant
constant  zero4: bit_vector(0 to 3) := ('0','0','0','0');
```

Variables

A *variable* is declared within a blocks, process, procedure, or function, and is updated immediately when an assignment statement is executed. A variable can be of any scalar or aggregate data type, and is utilized primarily in behavioral descriptions. It can optionally be assigned initial values (done only once prior to simulation). The declaration syntax is:

variable symbol: type [:= initial_value];

Examples

```
process
    variable count: integer  := 0;
    variable rega: bit_vector(7 downto 0);
begin
    ...
    count := 7;      -- assign values to variables
    rega  := x"01";
    ...
end;
```

Signals

A *signal* is an object with a history of values (related to "event" times, i.e. times at which the signal value changes).

Signals are declared via signal declaration statements or entity port definitions, and may be of any data type. The declaration syntax is:

signal sig_name: data_type [:=initial_value];

Examples

```
signal clock: bit;
```

```
signal GND:   bit := '0';
signal databus: std_ulogic_vector(15 downto 0);
signal addrbus: std_logic_vector(0 to 31);
```

Each signal has one or more "drivers" which determine the value and timing of changes to the signal. Each driver is a queue of events which indicate when and to what value a signal is to be changed. Each signal assignment results in the corresponding event queue being modified to schedule the new event.

- signal line x

10ns '0' Driver of

20ns '1' signal x

- Event Values
- Times

NOTE: If no delay is specified, the signal event is scheduled for one infinitessimally-small "delta" delay from the current time. The signal change will occur in the next simulation cycle.

Examples

(Assume current time is T)

```
clock   <= not clock after 10ns;      -- change at T + 10ns
databus <= mem1 and mem2 after delay; -- change at T + delay
x       <= '1';                       -- change to '1' at time T + "delta";
```

Element delay models may be specified as either "inertial" or "transport". Inertial delay is the default, and should be used in most cases.

- **Inertial delay:** The addition to an event queue of an event scheduled at time T automatically cancels any events in the queue scheduled to occur prior to time T, i.e. any event shorter than the delay time is suppressed.
- **Transport delay**: Each new event is simply inserted into the event queue, i.e. behavior is that of a delay line. The keyword **transport** is used to indicate transport delays.

Examples

```
B <= A after 5ns;            -- inertial delay
C <= transport A after 5 ns; -- transport delay

          5______15 17_________30
A ________|       |_|          |_______________

                 ______________________
B ______________|                      |_________ (Inertial Delay)

                 _______   ___________
C ______________|       |_|           |_________ (Transport Delay)
              10      20 22          35
```

Where there are multiple drivers for one signal, a "resolution function" must be provided to determine the value to be assigned to the signal from the values supplied by the multiple drivers. This allows simulation of buses with multiple sources/drivers.

NOTE: The *std_logic* and *std_logic_vector* types from the ieee library have predefined resolution functions:

Example

```
signal data_line: std_logic;
begin
  block1:
      data_line <= '1';      -- one driver
      ...
  block2:
      data_line <= 'Z';  -- 2nd driver
```

The resolved value is '1' since '1' overrides a 'Z' (floating) value. If the two values had been '1' and '0', the resolved value would have been 'X', indicating an unknown result.

CONCURRENT STATEMENTS - Back To Top

Concurrent statements are included within architecture definitions and within "block" statements, representing concurrent behavior within the modelled design unit. These statements are executed in an asynchronous manner, with no defined order, modeling the behavior of independent hardware elements within a system.

Concurrent Signal Assignment

A signal assignment statement represents a process that assigns values to signals. It has three basic formats.

1. A <= B; A <= B when condition1 elseC **when** condition2 **else** D **when** condition3 **else** E;
2. **with** expression **select** A <= B **when** choice1, C **when** choice2, D **when** choice3, E when others;

For each of the above, waveforms (time-value pairs) can also be specified.

Examples

```
A <= B after 10ns when condition1 else
     C after 12ns when condition2 else
     D after 11ns;

-- 4-input multiplexer (Choice is a 2-bit vector)
with Choice select
     Out <=  In0 after 2ns when "00",
             In1 after 2ns when "01",
             In2 after 2ns when "10",
             In3 after 2ns when "11";

-- 2-to-4 decoder (Y = 4-bit and A = 2-bit vectors)
Y <= "0001" after 2ns when A = "00" else
     "0010" after 2ns when A = "01" else
     "0100" after 2ns when A = "10" else
     "1000" after 2ns ;

-- Tri-state driver: (Y is logic4; X is bit_vector)
Y <= '0' after 1ns when En = '1' and X = '0' else
     '1' after 1ns when En = '1' and X = '1' else
     'Z' after 1ns;

-- A is a 16-bit vector
A <= (others => '0');   -- set all bits of A to '0'
```

The keyword "others" in the last example indicates that all elements of A not explicitly listed are to be set to '0'.

Process Statement - Back To Top

An independent sequential process represents the behavior of some portion of a design. The body of a process is a list of sequential statements.

Syntax:

```
label: process (sensitivity list)
       ... local declarations ...
       begin
       ... sequential statements ...
       end process label;
```

Example

```
DFF: process (clock)
       begin
          if clock = '1' then
             Q  <= D after 5ns;
             QN <= not D after 5ns;
          end if;
     end process DFF;
```

The sequential statements in the process are executed in order, commencing with the beginning of simulation. After the last statement of a process has been executed, the process is repeated from the first statement, and continues to repeat until suspended. If the optional sensitivity list is given, a **wait on** ... statement is inserted after the last sequential statement, causing the process to be suspended at that point until there is an event on one of the signals in the list, at which time processing resumes with the first statement in the process.

Block Statement - Back To Top

A *block* is a grouping of related concurrent statements that can be used in representing designs in a hierarchical manner.

Syntax:

```
label: block (guard expression)
       ... local declarations ...
       begin
       ... concurrent statements ...
       end block label;
```

If a *guard expression* is given, "guarded" a boolean variable GUARD is automatically defined and set to the boolean value of the guard expression. GUARD can then be tested within the block, to perform selected signal assignments or other statements only when the guard condition evaluates to TRUE.

Examples

```
-- D Latch: Transfer D input to Q output when Enable = '1'
block (Enable = '1')
begin
   Q <= guarded D after 5ns;

end block;
```

```
-- D Flip-flop: Transfer D to Q on falling edge of Clock
block (Clock'EVENT and Clock = '0')
begin
   Q <= guarded D after 5ns;
end block;

-- Tristate driver with input B and output A
block (Enable = '1')
begin
    A <= B when GUARD = '1' else 'Z';
end block;
```

In the last example, B is assigned to signal A only when GUARD is true, which implies Enable = '1'.

Concurrent Procedure Call - Back To Top

An externally defined procedure/subroutine can be invoked, with parameters passed to it as necessary. This serves the same function and behaves in the same manner as a "process" statement, with any signals in the passed parameters forming a sensitivity list.

Example

```
ReadMemory (DataIn, DataOut, RW, Clk);
(where the ReadMemory procedure is defined elsewhere)
```

Component instantiation - Back To Top

Instantiates (i.e. create instances of) predefined components within a design architecture. Each such component is first declared in the declaration section of that architecture, and then "instantiated" one or more times in the body of the architecture.

- In the declaration section: list the "component declaration" and one or more "configuration specifications".

  The "component declaration" defines the component interface, which corresponds to the component's entity declaration. This allows the VHDL compiler to check signal compatibilities.

  Example

```
component adder
   port(a,b:  in  bit_vector(7 downto 0);
        s:    out bit_vector(7 downto 0);
        cin:  in  bit;
        cout: out bit);
end component;
```

- The "configuration specification" identifies specific architecture(s) to be used for each instance of the component. (There may be multiple architectures for a given component.)

  Examples

```
for ALL:     comp1 use entity work.comp1 (equations);
for ADDER1:  adder use entity work.adder (equations);
```

```
for ADDER2:  adder use entity work.adder (dataflow);
```

In all three examples, the prefix **work.** indicates that the current working library contains the indicated component models. In the first example, architecture *equations* of entity *comp1* is used for all instances of *comp1*. In the other examples, architecture *equations* is to be used for instance *ADDER1* of component *adder*, and architecture *dataflow* is to be used for instance ADDER2 of component *adder*.

Component Instantiation Each instance of a declared component is listed, an instance name assigned, and actual signals connected to its ports as follows:

instance_name: component_name **port map** (port list);

The port list may be in either of two formats:

- (1) "Positional association": signals are connected to ports in the order listed in the component declaration.
- Ex. A1: adder port map (v,w,x,y,z)
  - v,w, and y must be bit_vectors, y and z bits
- (2) "Named association": each signal-to-port connection is listed explicitly as "signal=>port".

Example

A1: adder port map(a=>v, b=>w, s=>y, cin->x, cout->z);

(The signal ordering is not important in this format)

Example:

```
architecture r1 of register is
   component jkff
      port(J,K,CLK: in bit;
           Q,QN:    out bit);
   end component;
   for ALL: jkff use entity work.jkff (equations);
   -- Use architecture equations of entity jkff
      for all instances

   component dff
      port(D,CLK: in bit;
           Q,QN:  out bit);
   end component;
   for DFF1: dff  use entity work.dff  (equations);
   for DFF2: dff  use entity work.dff  (circuit);
   --Use different architectures of dff for instances
     DFF1 and DFF2
begin
   JKFF1: jkff port map (j1,k1,clk,q1,qn1);
   JKFF2: jkff port map (j2,k1,clk,q2,qn2);
   DFF1:  dff  port map (d1,clk,q4,qn4);
   DFF2:  dff  port map (d2,clk,q5,qn5);
end.
```

Concurrent assertion - Back To Top

A *concurrent assertion statement* checks a condition (occurrence of an event) and issues a report if the condition is not true. This can be used to check for timing violations, illegal conditions, etc. An optional severity level can be reported to indicate the nature of the detected condition.

Syntax:

```
assert  (clear /= '1') or (preset /= '1')
report "Both preset and clear are set!"
severity warning;
```

Generate statement - Back To Top

A *generate statement* is an iterative or conditional elaboration of a portion of a description. This provides a compact way to represent what would ordinarily be a group of statements.

Example

Generate a 4-bit full adder from 1-bit full_adder stages:

```
  add_label:        -- Note that a label is required here
    for i in 4 downto 1 generate
        FA: full_adder port map(C(i-1), A(i), B(i), C(i), Sum(i));
    end generate;

The resulting code would look like:

    FA4: full_adder port map(C(3), A(4), B(4), C(4), Sum(4));
    FA3: full_adder port map(C(2), A(3), B(3), C(3), Sum(3));
    FA2: full_adder port map(C(1), A(2), B(2), C(2), Sum(2));
    FA1: full_adder port map(C(0), A(1), B(1), C(1), Sum(1));
```

SEQUENTIAL STATEMENTS - Back To Top

Sequential statements are used to define algorithms to express the behavior of a design entity. These statements appear in process statements and in subprograms (procedures and functions).

Wait statement - Back To Top

- suspends process/subprogram execution until a signal changes, a condition becomes true, or a defined time period has elapsed. Combinations of these can also be used.

Syntax:

```
wait [on signal_name {,signal_name}]
     [until condition]
     [for time expression]
```

Example

Suspend execution until one of the two conditions becomes true, or for 25ns, whichever occurs first.

```
wait until clock = '1' or enable /='1' for 25ns;
```

Signal assignment statement - Back To Top

Assign a waveform to one signal driver (edit the event queue).

Example

```
A <= B after 10ns;
C <= A after 10ns;  -- value of C is current A value
```

Variable assignment statement - Back To Top

Update a process/procedure/function variable with an expression. The update takes affect immediately.

Example

```
A := B and C;
D := A;         -- value of D is new A value
```

Procedure call - Back To Top

Invoke an externally-defined subprogram in the same manner as a concurrent procedure call.

Conditional Statements - Back To Top

Standard *if..then* and *case* constructs can be used for selective operations.

```
if condition then
   ... sequence of statements...
elsif condition then
   ... sequence of statements...
else
   ... sequence of statements...
end if;
```

NOTE: *elsif* and *else* clauses are optional.

```
case expression is
   when choices => sequence of statements
   when choices => sequence of statements
   ...
   when others => sequence of statements
end case;
```

NOTE: *case* choices can be expressions or ranges.

Loop statements - Back To Top

Sequences of statements can be repeated some number of times under the control of **while** or **for** constructs.

```
label: while condition loop
   ... sequence of statements ...
   end loop label;

label:  for loop_variable in range loop
 ... sequence of statements...
```

```
end loop label;
```

NOTE: the label is optional.

Loop termination statements - allow termination of one iteration, loop, or procedure.

next [when condition**];** -- end current loop iteration

exit [when condition**];** -- exit innermost loop entirely

return expression; -- exit from subprogram

NOTES: 1. The next/exit condition clause is optional.

2. The return expression is used for functions.

- 8. **Sequential assertion** - same format as a concurrent assertion.

PROCEDURES - Back To Top

A *procedure* is a subprogram that is passed parameters and may return values via a parameter list.

Example

```
procedure  proc_name (signal clk: in vlbit;
                      constant d: in vlbit;
                      signal data: out vlbit) is
    ... local variable declarations ...
begin
    ... sequence of statements ...
end proc_name;
```

Procedure call: proc_name(clk1, d1, dout);

FUNCTIONS - Back To Top

A *function* is a subprogram that is passed parameters and returns a single value. Unlike procedures, functions are primarily used in expressions.

Example

```
-- Convert bit_vector to IEEE std_logic_vector format
-- (attributes LENGTH and RANGE are described below)
function bv2slv (b:bit_vector) return std_logic_vector is
    variable result: std_logic_vector(b'LENGTH-1 downto 0);
begin
    for i in result'RANGE loop
        case b(i) is
            when '0' => result(i) := '0';
            when '1' => result(i) := '1';
        end case;
    end loop;
    return result;
end;

-- Convert bit_vector to unsigned (natural) value
```

```
function b2n (B: bit_vector) return Natural is
    variable S: bit_vector(B'Length - 1 downto 0) := B;
    variable N: Natural := 0;
begin
    for i in S'Right to S'Left loop
        if S(i) = '1' then
            N := N + (2**i);
        end if;
    end loop;
    return N;
end;
```

Function Calls:

```
    signal databus:  vector4(15 downto 0);
    signal internal: bit_vector (15 downto 0);
    variable x: integer;
     ....
    databus <= bv2slv (internal);
    x := b2n(internal);
```

Data conversion between ieee types and bit/bit_vector (functions in "ieee.std_logic_1164")

To_bit(sul) - from std_ulogic to bit
To_bitvector(sulv) - from std_ulogic_vector/std_logic_vector
To_StdULogic(b) - from bit to std_ulogic
To_StdLogicVector(bv) - from bit_vector or std_ulogic_vector
To_StdULogicVector(bv)- from bit_vector or std_logic_vector
To_X01(v) - from bit, std_ulogic, or std_logic to X01
To_X01Z(v) - from bit, std_ulogic, or std_logic to X01Z
To_UX01(v) - from bit, std_ulogic, or std_logic to UX01

Other "ieee.std_logic_1164" functions - Back To Top

rising_edge(s) - true if rising edge on signal s (std_ulogic)
falling_edge(s) - true if falling edge on signal s (std_ulogic)

Additional Mentor Graphics-supplied functions for elements of types Bit_vector (implemented as overloaded operator definitions):

library mgc_portable;
use mgc_portable.qsim_logic.ALL;

Arithmetic between bit_vectors: use normal binary operator tokens

a + b, a - b, a * b, a / b, a mod b, a rem b

Logical operations between all signal types and vectors of signal types in the "ieee" library.

and, or, nand, nor, xor, xnor, not

Shift/rotate left/right logical/arithmetic operators:

sll, srl, sra, rll, rrl

Ex. a := x sll 2; -- "shift left logical" bit_vector x by 2 bits

Relational operations: =,/=,<,>,<=,>=

Type conversion:

to_bit (from integer)
to_integer (from bit_vector)

OBJECT ATTRIBUTES - Back To Top

An *object attribute* returns information about a signal or data type.

Signal Condition Attributes (for a signal S)

S'DELAYED(T) - value of S delayed by T time units
S'STABLE(T) - true if no event on S over last T time units
S'QUIET(T) - true if S quiet for T time units
S'LAST_VALUE - value of S prior to latest change
S'LAST_EVENT - time at which S last changed
S'LAST_ACTIVE - time at which S last active
S'EVENT - true if an event has occurred on S in current cycle
S'ACTIVE - true if signal S is active in the current cycle
S'TRANSACTION - bit value which toggles each time signal S changes

Examples

```
if (clock'STABLE(0ns)) then  -- change in clock?
    ...      -- action if no clock edge
else
    ...      -- action on edge of clock
end if;

if clock'EVENT and clock = '1' then
    Q <= D after 5ns;       -- set Q to D on rising edge of clock
end if;
```

Data Type Bounds (Attributes of data type T)

T'BASE - base type of T
T'LEFT - left bound of data type T
T'RIGHT - right bound
T'HIGH - upper bound (may differ from left bound)
T'LOW - lower bound

Enumeration Data Types (Variable/signal x of data type T)

T'POS(x) - position number of value of x of type T
T'VAL(x) - value of type T whose position number is x
T'SUCC(x) - value of type T whose position is x+1
T'PRED(x) - value of type T whose position is x-1

T'LEFTOF(x) - value of type T whose position is left of x
T'RIGHTOF(x) - value of type T whose position is right of x

Array Indexes for an Array A (Nth index of array A)

A'LEFT(N) - left bound of index
A'RIGHT(N) - right bound of index
A'HIGH(N) - upper bound of index
A'LOW(N) - lower bound of index
A'LENGTH(N) - number of values in range of index
A'RANGE(N) - range: A'LEFT to A'RIGHT
A'REVERSE_RANGE(N) - range A'LEFT downto A'RIGHT

NOTE: For multi-dimensional array, Nth index must be indicated in the attribute specifier. N may be omitted for a one-dimensional array.

Examples

```
for i in (data_bus'RANGE) loop
    ...
for i in (d'LEFT(1) to d'RIGHT(1)) loop
    ...
```

Block Attributes (of a block B)

B'BEHAVIOR - true if block B contains no component instantiations
B'STRUCTURE - true if no signal assignment statements in block B

THE TEXTIO PACKAGE - Back To Top

TEXTIO is a package of VHDL functions that read and write text files. To make the package visible:

use std.textio.all;

Data Types:

text - a file of character strings
line - one string from a text file

Example Declarations

```
file Prog: text is in "file_name"; --text file "file_name"
variable L: line;              -- read lines from file to L
```

Reading Values From a File:

readline(F, L)
Read one line from "text" file F to "line" L

read(L, VALUE, GOOD);

Read one value from "line" L into variable VALUE

- GOOD is TRUE if successful
- Data_type of VALUE can be bit, bit_vector, integer, real, character, string, or time.

Writing values to a file:

writeline(F, L);

Write one line to "text" file F from "line" L

write(L, VALUE, JUSTIFY, FIELD);

Write one value to "line" L from variable VALUE

- Data_type of VALUE can be bit, bit_vector, integer, real, character, string, or time.
- JUSTIFY is "left" or "right" to justify within the field
- FIELD is the desired field width of the written value

Back To Top