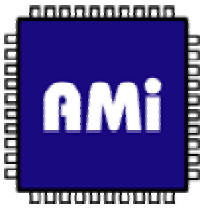


Advanced Electronic Design Automation

VHDL Quick Reference Guide

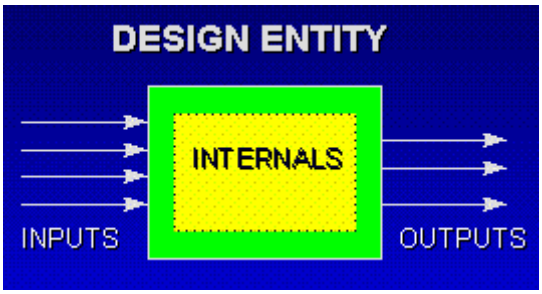
The Quick Reference contains the following sections:

Author: Ian Elliott of Northumbria University

The Design Entity

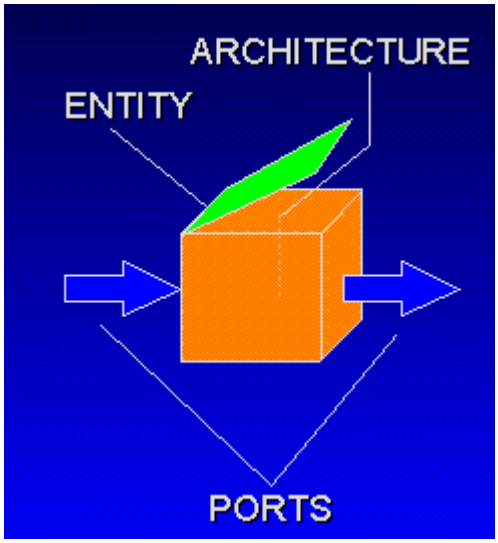
The Design Entity is the basic building block of a VHDL description. It comprises two parts:

- **Entity Declaration**
- **Architecture Body**



The Entity Declaration defines the interface of a hardware element (Ports) along with any parameters of the hardware element (Generics). There can only be one Entity in the library with a given entity_name.

The Architecture Body describes the internals of an Entity in terms of a *Behaviour, Structure* or *Dataflow*, or a combination of these. All statements within an Architecture are *concurrent*, ie. they execute in parallel. An Entity may have many Architectures with different architecture_names.

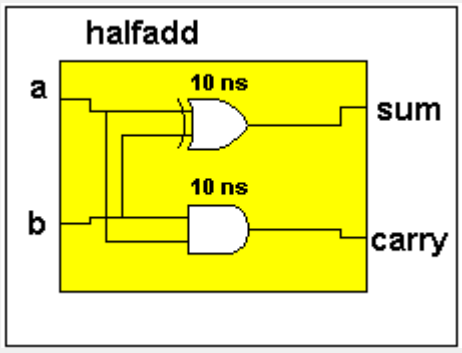


The syntax of the Entity Declaration and the Architecture Body is shown below:

```
--Entity Declaration with library and use clause
library lib_name;
use lib_name.package_name.all;
entity entity_name is
        generic(generic_name : type := default_value);
        port(port_names : direction type);
end entity entity_name; --entity[93]
--Architecture Body
architecture arch_name of entity_name is
        architecture declarations
```

```
begin
        concurrent statements
end architecture arch_name; --architecure[93]
```

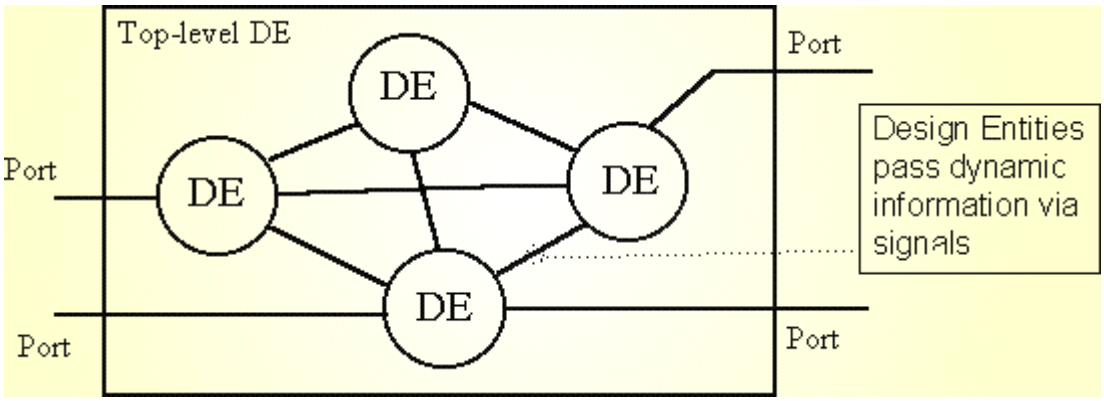
Example




```
entity halfadd is
        generic(delay : time := 10 ns);
        port(a, b : in bit; sum, carry : out bit);
end entity halfadd;
architecture v1 of halfadd is
begin
        sum <= a xor b after delay;
        carry <= a and b after delay;
end architecture v1;
```

The Network Model

A VHDL description of a digital system may be *hierarchical*, ie composed of a number of Design Entities connected together to form a network The Ports of each Design Entity allow dynamic information to pass between them. At the top level, the design may be described in terms of a Test Bench Entity which is a special entity having no ports.



Entity Ports

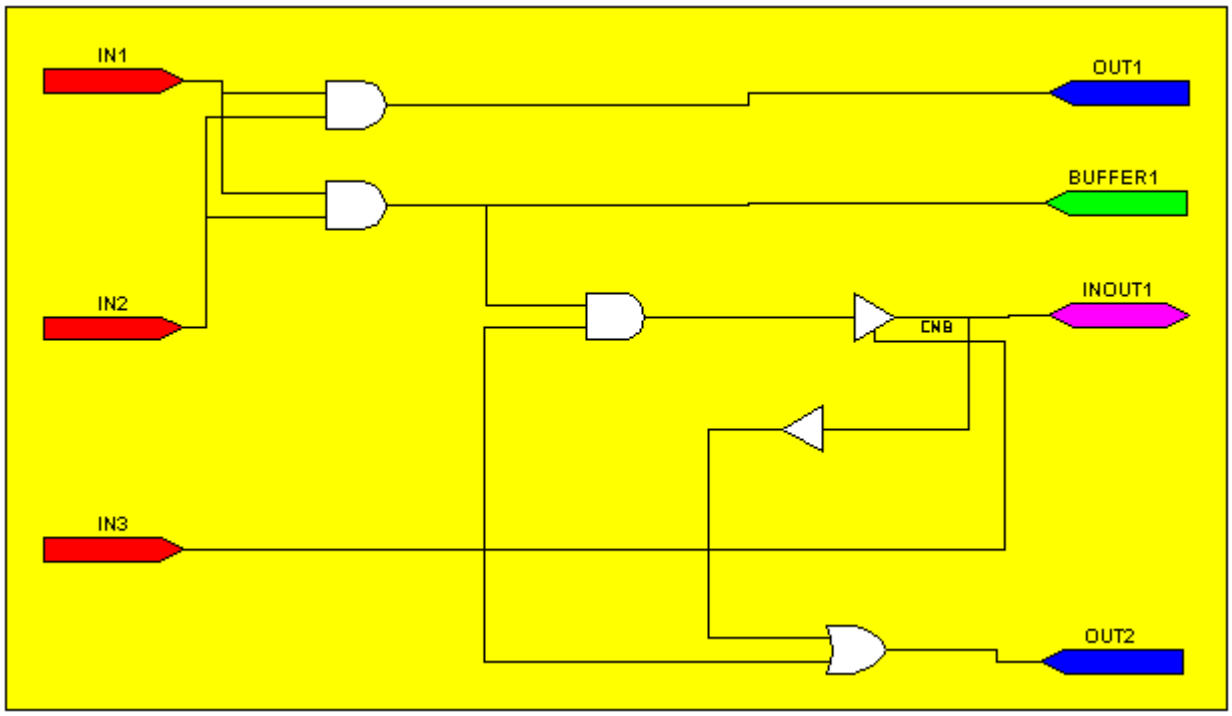
Most Entity Declarations contain a *Port Clause.* (The exceptions are usually Test Bench Entities which do not need to communicate with other Entities). This forms part of the *Entity Header* and defines the interface to the Design Entity. A Port Clause defines the name, type and direction of the ports of the Design Entity. As far as the Architecture Body of the Entity is concerned, the ports are effectively signals which can be accessed by the concurrent statements within the Architecture. The direction or *mode* of a port determines the way in which the statements within the Architecture may access it. The table below summarises the four different port modes.

Direction	Properties
in	The Design Entity can read the port, ie, the name of the port can only appear on the right hand side (input) of a signal assignment statement.
out	The Design Entity can write to the port, ie. the name of the port can only appear on the left hand side (target) of a signal assignment statement.
buffer	Similar to mode out,but the port may be fed back internally (such as the outputs of a counter). A Buffer port is not bidirectional.
inout	The Design Entity can both read and drive the bidirectional signal, ie. the name of the port can appear on both sides of a signal assignment statement.

The image shown below illustrates the logical equivalent of the port modes. Ports of modes IN and OUT are shown in red and blue respectively. A BUFFER port is shown in green. In this case the signal coming from the AND gate is also feeding the input of another gate (hence the signal will appear on the right hand side of an assignment statement), but the BUFFER port cannot act as an input.

The INOUT port, shown in purple, is truly bi-directional, and therefore will usually involve a three-state buffer. When the input IN3 is active, the three-

state buffer will be enabled and the INOUT1 port is driven by the 2-input AND gate. When the IN3 input is inactive, the three-state buffer is turned off and the INOUT1 port can act as an input to the 2-input OR gate via the buffer.



Generics

Generics are a means of passing instance specific information into a Design Entity. Inside the entity they are effectively a Constant. They are commonly used to pass time delay values into a Design Entity or to set the size of a scalable description.

Syntax:

```
entity entity_name is
        generic(generic_name : type := default_value);
        ......port clause
end entity_name;
```

Examples:

```
entity myreg is
        generic(numbits : positive := 8); --sets width of register
```

```
        port(clock : in bit;
                datain : in bit_vector((numbits - 1) downto 0);
                dataout : out bit_vector((numbits - 1) downto
                0));
end myreg;
entity mygate is
        generic(delay : time := 10 ns); --sets delay of gate
        port(a, b : in bit; f : out bit);
end mygate;
.....in the architecture
        f <= a xor b after delay;
```

Concurrent Statements

Concurrent statements are at the heart of a VHDL description; any of the statements listed below may be used within the statement part of the Architecture Body. The ordering of concurrent statements is not important since they are all effectively active at the same time. Execution of concurrent statements is determined by signal events communicated from one statement to another. For example if the same signal appears on the input side of a number of concurrent statements, then all of the statements would execute at the same time in response to an event on that signal.

Block statement

Process statement

Concurrent Assertion statement

Concurrent Signal Assignment statement

Conditional Signal Assignment

Selected Signal Assignment

Component Instantiation statement

Generate statement

Concurrent Procedure Call

Architecture Local Declarations

The following declarations may appear in the architecture declarative area (between **is** and **begin**). The most commonly used are signal, type and component. These three declare a local signal (to interconnect components or processes), local type (to define the states of a state machine for example) and local components respectively. Any component used inside an architecture must be declared, either in the architecture itself, or in a package which is used by the architecture. The exception to this is the use of *direct instantiation* whereby a design entity can be instantiated without the need for a component declaration (VHDL 1993 only).

Type declaration

Subtype declaration

Signal declaration

Constant declaration

Component declaration

Function declaration

Procedure declaration

Configuration specification

Sequential Statements

These statements are used within the statement part of a process (between **begin** and **end process**) and also within sub-programs (functions and procedures). Sequential statements execute in the order they are written, much the same as in any general purpose high level language. The most important sequential statement is probably the wait statement which is often used in the description of sequential systems. The sequential statements provided in the VHDL language are based on those available in the ADA language.

wait

sequential signal assignment

variable assignment

if then else

case

loop

next

exit

null

Process Declarations

A variable is commonly declared within the declarative region of a sequential process. The variable may be used within the process to hold an intermediate value. Assignments to a variable take immediate effect, whereas signals do not get updated until the end of the entire process (assuming there are no wait statements between the signal assignment and the end of the process).

```
variable variable_name : type;
variable variable_name : type := initial_value;
```

example :

```
--sequential process to model JK flip-flop
process
  --declare a local variable to hold ff state
  variable state : bit := '0';
begin
  --synchronise process to rising edge of clock
  wait until (clock'event and clock = '1');
  if (j = '1' and k = '1') then --toggle
    state := not state;
  elsif (j = '0' and k = '1') then --reset
    state := '0';
  elsif (j = '1' and k = '0') then --set
    state := '1';
  else            --no change
    state := state;
  end if;
  --assign values to output signals
  q <= state after 5 ns;
  qbar <= not state after 5 ns;
```

```
end process;
```

Pre-defined Types and Literals

The predefined types provided by the VHDL language are defined in Package Standard. This package is included implicitly by all Design Entities, ie. there is no need to attach a library or use clause to the Design. Some elements of the Standard Package are implementation dependent, such as the range of predefined type INTEGER for example. The contents of Package Standard are listed below (for Model technology's V-System/PLUS):

```
package standard is
    type boolean is (false,true);
    type bit is ('0', '1');
    type character is (
        nul, soh, stx, etx, eot, enq, ack, bel,
        bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
        dle, dc1, dc2, dc3, dc4, nak, syn, etb,
        can, em,  sub, esc, fsp, gsp, rsp, usp,

        ' ', '!', '"', '#', '$', '%', '&', ''',
        '(', ')', '*', '+', ',', '-', '.', '/',
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', ':', ';', '<', '=', '>', '?',

        '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
        'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '[', '\', ']', '^', '_',

        '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
        'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', '{', '|', '}', '~', del.....);

    type severity_level is (note, warning, error, failure);
    type integer is range -2147483648 to 2147483647;
    type real is range -1.0E308 to 1.0E308;
    type time is range -2147483647 to 2147483647
        units
            fs;
            ps = 1000 fs;
```

```
                ns = 1000 ps;
                us = 1000 ns;
                ms = 1000 us;
                sec = 1000 ms;
                min = 60 sec;
                hr = 60 min;
            end units;
        subtype delay_length is time range 0 fs to time'high;
        impure function now return delay_length;
        subtype natural is integer range 0 to integer'high;
        subtype positive is integer range 1 to integer'high;
        type string is array (positive range <>) of
character;
        type bit_vector is array (natural range <>) of bit;
        type file_open_kind is (
            read_mode,
            write_mode,
            append_mode);
        type file_open_status is (
            open_ok,
            status_error,
            name_error,
            mode_error);
        attribute foreign : string;
    end standard;
```

Literals

Integer types

```
constant freeze : integer := 32; --no decimal point
constant a_pos : positive := 16#ff#;  --hexadecimal
notation
constant b_nat : natural := 2#10101111#; --binary
notation
constant delay_time : time := 10 us;  --physical types must
have units
```

Floating point numbers

```
variable factor : real := 32.0;  --decimal point required
constant x : real := 2.2e-6;   --exponential form
```

Enumeration type literals

```
signal flag : boolean := false; --type boolean can be true or
false
constant myconst : boolean := true
type state_type is (s0, s1, s2, s3); --define a user
enumeration type
signal ps, ns : state_type := s0; --create signals and
initialise
variable temp : bit := '1'; --type bit can be '0' or '1'
signal tied_low : bit := '0';
signal parity : std_logic := 'H';  --could be one of
'U','X','0','1','Z','W','H','L','-'
```

Array type literals

```
constant flag : bit_vector(0 to 7) := "11110101";  --bit
string literal
variable var1 : bit_vector(7 downto 0) := X"AA";   --
hexadecimal notation
signal bus_9_bit : bit_vector(0 to 8);
.........
bus_9_bit <= O"393"; --octal notation
.........
bus_9_bit <= (others => '0'); --aggregate assignment, set
all bits to '0'
bus_9_bit <= (0 => '1', 2 => '0', others => '1'); --setting
individual bits
signal databus : std_logic_vector(7 downto 0); --ieee
std_logic type
.........
databus <= "ZZZZZZZZ"; --setting the bus to high
impedance
signal bus_n_bits : std_logic_vector((n-1) downto 0); --an
n-bit signal
bus_n_bits <= (others => 'Z'); --sets n bits to 'Z' without a
loop
constant message : string := "hello"; --an array of ASCII
characters
```

Objects and Operators

VHDL Objects

Object	Properties
Signal	Signal objects are used to communicate dynamic events around the model of a hardware system; they have both a time dimension and a value. When a signal object is assigned a new value, it does not take on the value immediately, but after an infinitesimally small time delay known as a delta delay. When a signal changes it is said to have undergone an *event.* Such an event can trigger further assignments to take place. Only a signal can have an event occur on it.
Variable	Can be changed by a *variable assignment statement* (usually within a process). A variable changes immediately and has no time dimension. Variables are most often used within the confines of a process to keep track of a local value such as the state of a memory element or the value of a counter register.
Constant	A constant object is normally set to a particular value when declared; it cannot be changed by assignment in the model. Constants are useful for defining ROM contents or fixed parameters.

VHDL Built-in Types

Type Class	VHDL Name
Numeric	**integer**(-maxint to +maxint) **positive**(1 to +maxint) **natural**(0 to +maxint) **real**(-maxreal to +maxreal)
Enumeration	**boolean**(false, true) **bit**('0', '1') **character**(ASCII set)
Array	**bit_vector**(array of bit) **string**(array of character)

Physical	**time**(units fs, ps , ns, us....)

VHDL Built-in Operators

Operator Class	VHDL Name	Operands
Arithmetic	**+** | **-** | ***** | **/** | **mod** | **rem** | **abs** | ******	Integer, Real and Physical
Logical	**and** | **or** | **not** | **nand** | **nor** | **xor** | **xnor** **sll** | **srl** | **sra** | **sla** | **ror** | **rol**	Boolean, Bit and Bit_vector
Relational	**=** | **/=** | **<** | **>** | **<=** | **>=**	Valid for all types, ordering operators work from left to right
Miscellaneous	**&** (Aggregate operator - concatenates two arrays to form a larger array)	One dimensional arrays

---

Attributes

Attributes supply additional information about an item, ie. a signal, variable, type or component. Certain attributes are predefined for types, array objects and signals. User defined attributes may be declared. These may have no effect on simulation, and are often used to supply information to other design tools such as PCB layout or PLD/FPGA synthesis tools.

Listed below are the most commonly used attributes:

Scalar and Array Attributes

Attribute	Description - X is a Scalar or Array object
X'**high**	The upper bound of X (or upper index if X is an array)

X'**low**	The lower bound of X (or lower index if X is an array)
X'**left**	The leftmost bound of X(or leftmost index if X is an array)
X'**right**	The rightmost bound of X (or rightmost index if X is an array)

Constrained Array Attributes

Attribute	Description - X is a constrained array object
X'**range**	The range of X (often used in **loop** statement)
X'**reverse_range**	The range of X back-to-front
X'**length**	X'**high** - X'**low** + 1 (integer)

Signal Attributes

Attribute	Description - X is a signal object
X'**event**	True when X has an event on it (boolean, often used to detect a clock edge)
X'**active**	True when X is assignment to (boolean)
X'**last_event**	When X last had an event (time)
X'**last_active**	When X was last assigned to (time)
X'**last_value**	Previous value of X (same type as X)

New Signal Creating Attributes

Attribute	Description - X is a signal object
X'**delayed**(t)	A copy of signal X, delayed by time t (same type as X)
X'**stable**(t)	True when X is unchanged for time t (Boolean)
X'**quiet**(t)	True if X is unassigned for time t (Boolean)
X'**transaction**	Toggles when X is assigned to (Bit)

Examples

```
--user defined attribute to define a components PCB package
type ic_package is (dil, plcc, pga);
attribute ptype : ic_package;
attribute ptype of u1 : component is plcc;
attribute ptype of u2 : component is dil;

--loop statement using range of a bit_vector for iteration
variable temp : bit_vector(15 downto 0);
.....
for i in temp'range loop..... --loop 16 times



--suspend a process until clock rises
wait until clock'event and clock'last_value = '0';
```

Structured Types

The VHDL language provides two structured types - **Arrays** and **Records**.

Built-in types bit_vector and string are arrays of bit and characters respectively. These array types are *unconstrained,* which means that the actual number of elements is not defined in the type declaration. The number of elements is defined when an *object* is created.

```
type bit_vector is array (integer range <>) of bit;
type std_logic_vector is array ( natural range <>) of
std_logic;
```

Object declarations:

```
signal my_bus : bit_vector(7 downto 0);
```

General syntax

```
type type_name is type_definition;
```

examples

```
type t_int is range 0 to 9; --user defined numeric type
type t_real is range -9.9 to 9.9;
```

```
type my_state is (reset, idle, acka); --user defined
enumerated type
signal state : my_state := reset;
```

Arrays type declaration

```
type type_name is array(range) of element_type;
```

examples

```
type ram is array (0 to 31) of bit_vector(3 downto 0);
--creating a ram variable with initial contents
variable ram_var : ram :=
("0000","0001","0101",.....,"1111");
```

Records

```
type type_name is record
      element_declarations
end record;
```

examples

```
--declaring subtypes for hours, minutes and seconds
subtype hours is natural;
subtype minutes is integer range 0 to 60;
subtype seconds is integer range 0 to 60;


--record type to define elapsed time
type elapsed_time is
  record
    hh : hours;
    mm : minutes;
    ss : seconds;
  end record;


--function to increment elapsed time by one second
function inctime (intime :elapsed_time) return
elapsed_time is
   variable result : elapsed_time;
begin
   result := intime;
```

```
    --notice use of selected naming to access fields of
    --the record structure (object_name.field_name)
    result.ss := result.ss + 1;
    if result.ss = 60 then
        result.ss := 0;
        result.mm := result.mm + 1;
        if result.mm = 60 then
            result.mm := 0;
            result.hh := result.hh + 1;
        end if;
    end if;
    return result;
end function inctime;
```

Assigning to records using aggregates:

```
--an object which is a record type may be assigned to using
an aggregate
--for example in the case of the object 'result' within the
above function
result := (0,0,0);    --zero time
result := (2,45,6);   --2 hours, 45 minutes and 6 seconds
```

Configurations

The Configuration Declaration is a mechanism for Binding Components to Entity-Architecture pairs within a structural Architecture. Configurations can also be used to assign values to the generics of a component which override the default values.

Syntax:

```
configuration config_name of entity_name is
    for architecture_name
        for instance_label : component_name
            use entity library_name.entity_name(arch_name);
                for arch_name ..--lower level configuration
                specifications
                end for;
        end for;
    end for;
end configuration config_name; --configuration [93]
```

Example:

```
configuration parts of dec2to4_bench is
for structural

  for generator : dec2to4_stim
    use entity work.dec2to4_stim(behavioural);
  end for;

  for circuit : dec2to4
    use entity work.dec2to4(structural);
     for structural
       for all : inv
         use entity work.inv(behaviour)
           generic map(tplh => 10 ns,
                tphl => 7 ns,
                tplhe => 15 ns,
                tphle => 12 ns);
       end for;
       for all : and3
         use entity work.and3(behaviour)
           generic map(tplh => 8 ns,
                tphl => 5 ns,
                tplhe => 20 ns,
                tphle => 15 ns);
       end for;
     end for;
  end for;

end for;
end configuration parts;
```

Packages

A Package is used to group together declarations which may be used by several design units (mainly entities). The declarations include Types, Constants, Components, Attributes, Functions and Procedures. Items contained within a package can be made visible to an entity by attaching a use clause to the entity.

Syntax:

```
package package_name is
        declarations
end package package_name; --package [93]
```

Example:

```
package demo_pack is
```

```
        constant some_flag : bit_vector := "11111111";
        type state is (reset, idle, acka);
        component halfadd
                port(a, b : in bit; sum, carry : out bit);
        end component;
end package demo_pack;
```

Library and Use Clauses

The VHDL language makes use of Library and Use clauses to organise design libraries and promote design re-use. The operating system dependent aspect of libraries is abstracted out of the language by means of a mapping between a *logical name* and an operating system *path name*; the latter points to the directory containing the library. The library itself is a database containing analysed design units. Typically a library will contain a Package which declares all of the components in the library along with the compiled design units for each component; the latter are required for simulation purposes.

```
library library_name_1, library_name_2, ...;
use library_name_1.package_name_1.all;
use library_name_2.package_name_2.all;
```

Example - Source file listing showing typical format of design units for compilation into a library. This file will be compiled into a library named mylib.

```
entity mycomp1 is....--design entity for component
architecture v1 of mycomp1 is...
.....
entity mycomp2 is....--design entity for component
architecture v1 of mycomp2 is...
.....
entity mycomp3 is....--design entity for component
architecture v1 of mycomp3 is...
.....
--package declaration groups declarations together
package mycomponents is
  component mycomp1 ....end component--component declarations
   component mycomp2 ....end component
   component mycomp3 ....end component
   signal power : std_logic := '1'; --global signal declarations
   signal ground : std_logic := '0';
end package mycomponents;
```

Example - Clauses attached to a design entity using above library of components

```
library mylib; --this clause makes the library name 'mylib' visible
use mylib.mycomponents.all; --makes all mycomponents declarations
visible
```

entity ....--this entities' architecture can use mycomp1, mycomp2 etc..

IEEE Standard Logic

Package std_logic_1164 defines industry standard digital types for modelling real hardware. In addition to defining the standard logic types, the package also includes definitions of the basic logical functions (**and**, **or**, **xor** etc...) for types **std_logic** and **std_logic_vector**, as well as the functions **rising_edge(***signal***)** and **falling_edge(***signal***)**. The latter are used with signals of type std_logic and return a Boolean result which is true if the signal has changed from '0' to '1' (rising_edge) or '1' to '0' (falling_edge) during the current simulation cycle.

Type std_logic supports accurate simulation using 9-values:

Uninitialised	'U'
Forcing Unknown	'X'
Forcing Zero	'0'
Forcing One	'1'
High Impedance	'Z'
Weak Unknown	'W'
Resistive Zero	'L'
Resistive One	'H'
Don't care	'-'

To use IEEE Std_logic, precede Entity Declaration with the following context clause:

```
library ieee;
use ieee.std_logic.all;
```

Example - An Octal D-Type Register with three-state outputs:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY ttl374 IS
```

```
  PORT(clock, oebar : IN std_logic;
       data : IN std_logic_vector(7 DOWNTO 0);
       qout : OUT std_logic_vector(7 DOWNTO 0));
END ENTITY ttl374;
ARCHITECTURE using_1164 OF ttl374 IS
  --internal flip-flop outputs
  SIGNAL qint : std_logic_vector(7 DOWNTO 0);
BEGIN
  qint <= data WHEN rising_edge(clock);
  qout <= qint WHEN oebar = '0' ELSE "ZZZZZZZZ"; --high
impedance
END ARCHITECTURE using_1164;
```

IEEE Standard Logic Support Packages

The following packages are widely used for simulation and synthesis; each package contains overloaded functions for arithmetic operations on digital numbers represented as bit_vectors or std_logic_vectors.

- Std_logic_arith

Defines a set of arithmetic, conversion and comparison functions for types SIGNED*, UNSIGNED*, INTEGER, STD_LOGIC and STD_LOGIC_VECTOR. *Types SIGNED and UNSIGNED are arrays of Std_logic

- Std_logic_signed

Defines a set of signed two's complement arithmetic, conversion and comparison functions for type STD_LOGIC_VECTOR.

- Std_logic_unsigned

Defines a set of unsigned arithmetic, conversion and comparison functions for type STD_LOGIC_VECTOR.

- Numeric_bit

Defines a set of signed and unsigned arithmetic, conversion and comparison functions for types SIGNED and UNSIGNED. The base element of types SIGNED and UNSIGNED is type BIT.

- Numeric_std

Defines a set of signed and unsigned arithmetic, conversion and comparison functions for types SIGNED and UNSIGNED. The base element of types SIGNED and UNSIGNED is type STD_LOGIC.

Example - context clause:

```
LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Numeric_bit | Numeric_std |
Std_logic_arith.ALL;
ENTITY .....
```

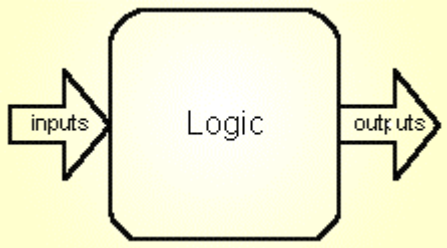
Example:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity edgecntr is
  port(clk, pulsein, clear : in std_logic;
       count : buffer std_logic_vector(3 downto 0));
end entity edgecntr;

architecture v1 of edgecntr is
begin
  cntr : process
  begin
    wait until rising_edge(clk);
    if clear = '1' then
       count <= (others => '0');
    elsif pulsein = '1' then
       count <= count + 1; --std_logic_unsigned function
'+'
    else
       null;
    end if;
  end process;
end architecture v1;
```

Process Templates for Synthesis

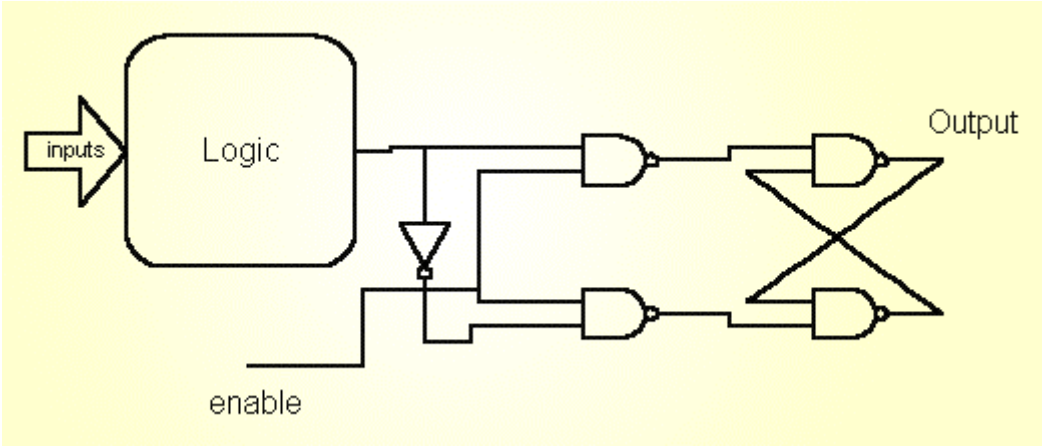
A VHDL Process can be used to describe both combinational and sequential logic. The templates given below illustrate standard formats for describing the basic types of logic for the purposes of synthesis (they can also be used for simulation).

Combinational Logic



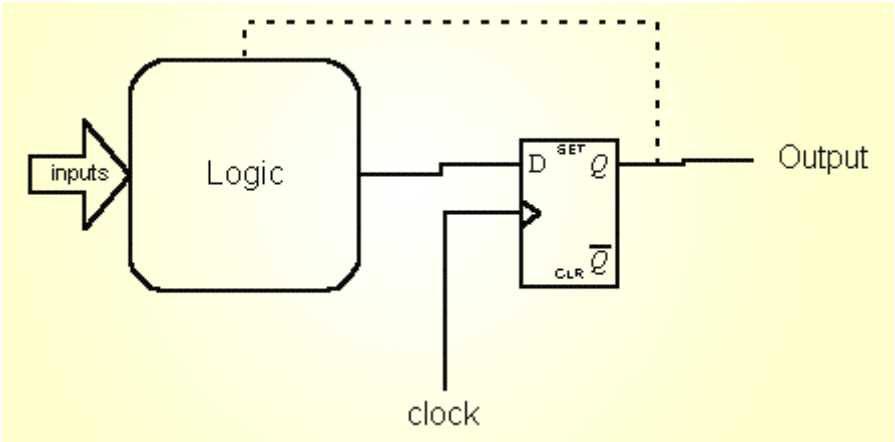

```
--combinational logic
process(all_inputs)
begin
   ...To avoid unwanted latches....
   ...assign all outputs a default value at the top of the
process
   ...OR specify output values for ALL possible input
combinations.
   ...Describe combinational logic using if..then..else,
   ...case..when, loop statements etc.
end process;
```

Latches plus Logic



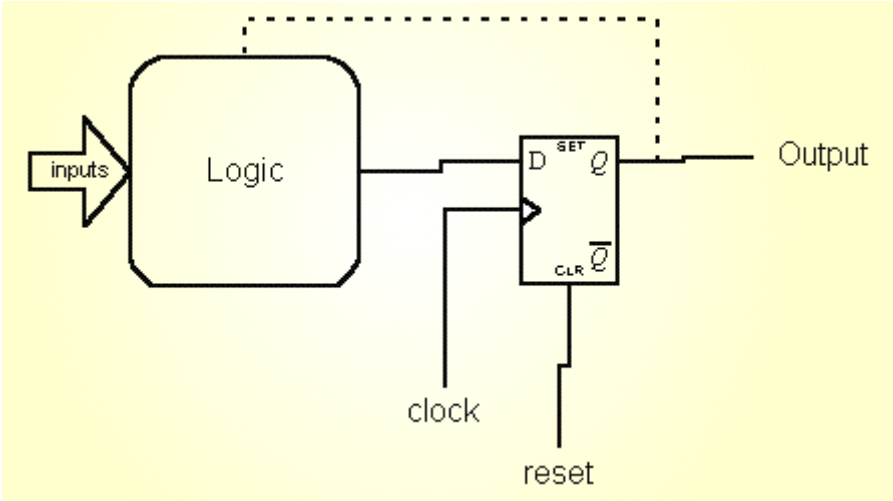

```
--latches plus logic
process(all_inputs)
begin
   if enable = '1' then
      ...latches plus logic
      ...when en -> '0' outputs retain value
   end if;
end process;
```

Flip-flops plus logic



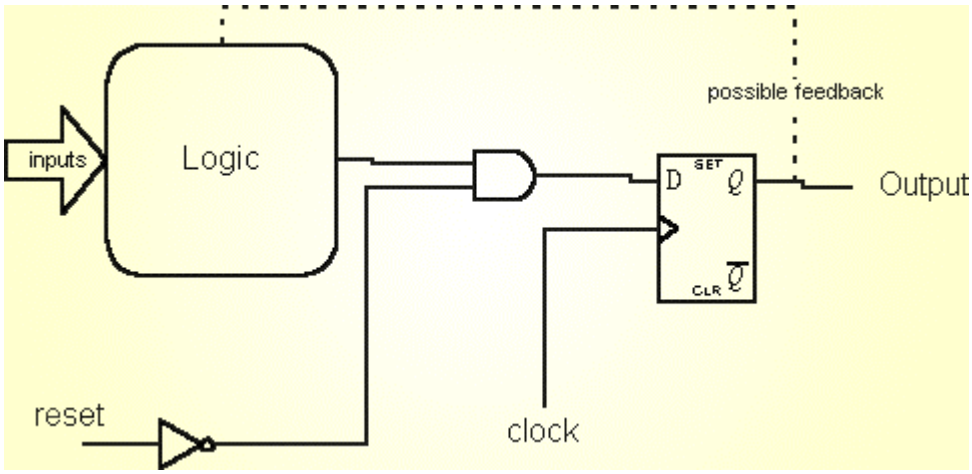

```
--flip-flops plus logic
process
begin
   wait until rising_edge(clock);
      ...flip-flops plus logic
      ...any assigned output is a flip-flop output
end process;
```

Asynchronous Reset




```
--asynchronous reset
process(clock, reset)
begin
  if reset = '1' then
    ..asynchronous reset action
  elsif rising_edge(clock) then
    ...flip-flops plus logic
  end if;
end process;
```

Synchronous Reset




```
--synchronous reset
process
begin
wait until rising_edge(clock);
```

```
  if reset = '1' then
    ...synchronous reset action
  else
    ...flip-flops plus logic
  end if;
end process;
```

Block Statement

A Block defines a region of visibility within an Architecture. (An Architecture is itself a Block).

Block statements are used to partition a design into sub-units without having to use components. The declarative part of a Block may declare local signals, types and functions etc. which are only visible within the enclosing block. Blocks may have Ports and Generics (with Port and Generic mapping to signals within the parent Architecture).

Signals declared outside a Block are visible within the Block, but the reverse is not true. Blocks may be nested.

Syntax:

```
label : block is    --is [93]
        local_declarations
begin
        concurrent_statements
end block label;
```

Example:

```
architecture ..
  signal address, offset : natural range 0 to 1023; --10-bit
+ve numbers
  signal suboff, clear, clock : bit;
begin
 cntr10 : block
   signal count : natural := 0;  --declare a local signal
 begin --dataflow model of address counter
   count <= 0 when clear = '1' else
      ((count + 1) mod 1024) when (clock'event and
clock = '1')
      else count; --10-bit counter with async clear
   address <= count when suboff = '0' else
```

```
        (count - offset) when ((count - offset) >= 0)
        else (1024 - abs(count - offset)); --arithmetic logic
end block cntr10;
```

Process Statement

A process statement is a concurrent statement which contains a sequential algorithm. The statements within a process are sequential statements and they execute in the order that they appear. A process can contain wait statements, which suspend the execution of the process, or it may contain a *sensitivity list*, **but not both**.

A process never terminates. It is either active or suspended.

Syntax:

```
optional_label : process (optional sensitivity list) is  --is [93]
        process declarations
begin
        sequential statements
end process optional_label;
```

Concurrent Assertion Statement

The concurrent assertion statement is often used in the statement part of an entity declaration. This is an optional part of the entity declaration which follows the entity header containing the generic and port clauses. The statement is typically used to check for timing violations occurring on inputs to the entity such as a set-up time, hold time or minimum pulse width requirement. The check is defined by the boolean condition which will often refer to the signal ports of the entity.

Being a concurrent statement, it may also be placed in an architecture statement part.

```
assert Boolean_condition [ report String_literal ] [ severity
Level ] ;
```

Concurrent Signal Assignment Statement

The Concurrent Signal Assignment Statement assigns a value to a target signal whenever any of the signals on the right hand side (input) of the statement have an event on them. Every concurrent signal assignment statement creates a driver for the target signal. The *expression* may be a logical expression involving other signals, in which case the statement represents simple combinational logic. Alternatively the *expressions* may be values which are scheduled to occur on the target signal during the simulation. The default delay mode for all signal assignments in VHDL is inertial, which means that input signal pulses which are shorter than the specified delay clause will be ignored by the statement, ie. the input signal fails to overcome the inertia of the logic circuit. If pure time delays are being modelled, such as in the case of a delay line, the keyword 'transport' precedes the value expression.

signal_name <= [**transport**] *expression* [**after** *delay*]
{,*expression2* [**after** *delay2*]};

The following examples illustrate the use of the concurrent signal assignment statement:

```
--a half adder with delays
sum <= a xor b after 5 ns;
carry <= a and b after 10 ns;
--creating a waveform -__--_ , at 50 ns the signals stays at
'0'
pulse <= '1', '0' after 10 ns,
         '1' after 30 ns,
         '0' after 50 ns;

--generating a clock, this statement triggers itself
clock <= not clock after period/2;
```

Conditional Signal Assignment

The Conditional Signal Assignment statement is a concurrent statement which is somewhat similar to the if..then..else statement in the sequential part of the VHDL language. The statement executes whenever an event occurs on any signal appearing on the right hand side (input) of the statement, ie. in any of the *expressions* or *boolean_conditions*. Unlike the Selected Signal Assignment statement, each condition is tested in sequence (*condition1* then *condition2* etc.), the first to return a true result determining the value assigned to the target signal; the remaining conditions are ignored. In the 1987 version of VHDL the else part is

mandatory. However, the 1993 standard allows the else part to be missed out. This implies that the conditional signal assignment statement could be used to describe simple flip-flops and latches. A new keyword, unaffected, was added in the 1993 release for use in concurrent signal assignments to indicate that under certain conditions the target signal is unaffected, thereby implying memory.

```
signal_name <= expression1 when boolean_condition1 else
               expression2 when boolean_condition2 else
expression3;
```

The following examples illustrate the use of the conditional signal assignment statement:

```
--a d-type flip-flop
q <= d when (clock'event and clock = '1');
--a 4 input multiplexer
q <= i0 when (a = '0' and b = '0') else
     i1 when (a = '1'  and  b = '0') else
     i2 when (a = '0'  and  b = '1') else
     i3 ;
--an 8-input priority encoder, in7 is highest priority input
level <= "111" when in7 = '1' else
         "110" when in6 = '1' else
         "101" when in5 = '1' else
         "100" when in4 = '1' else
         "011" when in3 = '1' else
         "010" when in2 = '1' else
         "001" when in1 = '1' else
         "000";
```

Selected Signal Assignment

The Selected Signal Assignment statement is a concurrent statement which is somewhat similar to a Case statement in the sequential part of the VHDL language. All signals appearing on the right hand (input) side of the statement can cause the statement to execute. This includes any signals in the *select_expression* as well as in any of the *expressions* and *choices* listed in each 'when' limb of the statement.

There is no priority associated with any particular 'when' alternative which means that choices must not overlap and all possible choices for the

select_expression must be included. Otherwise a final 'when others' limb must be included to cover those choices not elaborated.

```
with select_expression select
signal_name <= expression1 when choice1,
      expression2 when choice2,
      expression3 when choice3| choice4,  --multiple
alternative choices
      expression4 when choice5 to choice7, --discrete range
of choices
      expression5 when others;  --covers all possible
choices not mentioned
```

The selected signal assignment statement is useful for describing multiplexers and arbitrary combinational logic based on a truth table description, as the following example illustrates:

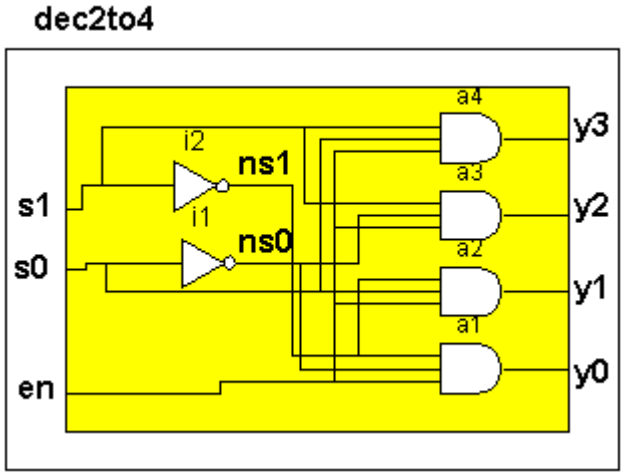
```
entity fulladd is
   port(a,b,cin : in bit; s,cout : out bit);
end fulladd;

architecture zero_delay_behave of fulladd is
begin
   with a&b&cin select --select expression is an aggregate
     s <= '1' when "010"|"100"|"001"|"111",'0' when
others;
   with a&b&cin select
     cout <= '1' when "011"|"101"|"110"|"111",'0' when
others;
end zero_delay_behave;
```

Component Instantiation Statement

A Component Instantiation Statement creates an *occurence* of a component. The label is compulsory, in order to differentiate between instantiations of the same component. The *port_association_list* defines which local signals connect to which ports of the component. The association list can be *positional* or *named* (see examples). Ports may be left unconnected by associating the key word open to a component port. A component having generics has a *generic_association_list* which maps values to the generics for a given instance; these will override values defined by the component's entity declaration.

instance_label : *component_name*
generic map(*generic_association_list*)
port map(*port_association_list*);

The following example shows a 2-to-4 decoder described in terms of gates:




```
entity dec2to4 is
  port(s0,s1,en : in bit; y0,y1,y2,y3 : out bit);
end dec2to4;

architecture structural of dec2to4 is

  --components must be declared before being used
  component inv
    port(a : in bit; b : out bit);
  end component;
  component and3
    port(a1,a2,a3 : in bit; o1 : out bit);
  end component;
  signal ns0,ns1 : bit;

begin

i1 : inv port map(s0,ns0); --positional association
i2 : inv port map(s1,ns1);

--positional association, actuals are connected by position
a1 : and3 port map(en,ns0,ns1,y0);
a2 : and3 port map(en,s0,ns1,y1);
```

```
a3 : and3 port map(en,ns0,s1,y2);
--named association formal => actual
a4 : and3 port map(a1 => en, a2 => s0, a3 => s1, o1 => y3);

end structural;
```

Direct Instantiation (VHDL 1993)

The 1993 release of VHDL allows design entities to be instantiated in other design entities directly, ie. without the need to declare a component or set up a binding. The syntax for direct instantiation is shown below:

```
label : entity lib_name.entity_name(architecture_name) port map (.....);
G1 : entity WORK.BLOCK1(RTL) port map (A, B, F);
```

Generate Statement

The Generate Statement is used to *replicate* concurrent statements over a specified range. If the concurrent statements are component instantiations, then this creates an array of components. This is very useful for creating regular structures like shift registers, memory circuits and ripple carry adders.

The label is compulsory with a generate statement.

The statement may be nested to create two dimesional arrays of components.

The if.. generate statement is often used within a generate structure to account for irregularities (see example).

```
label : for parameter in range generate
   concurrent_statements
        label : if condition generate
              concurrent_statements
        end generate label;
end generate label;
```

Example 1 - n-bit binary adder:

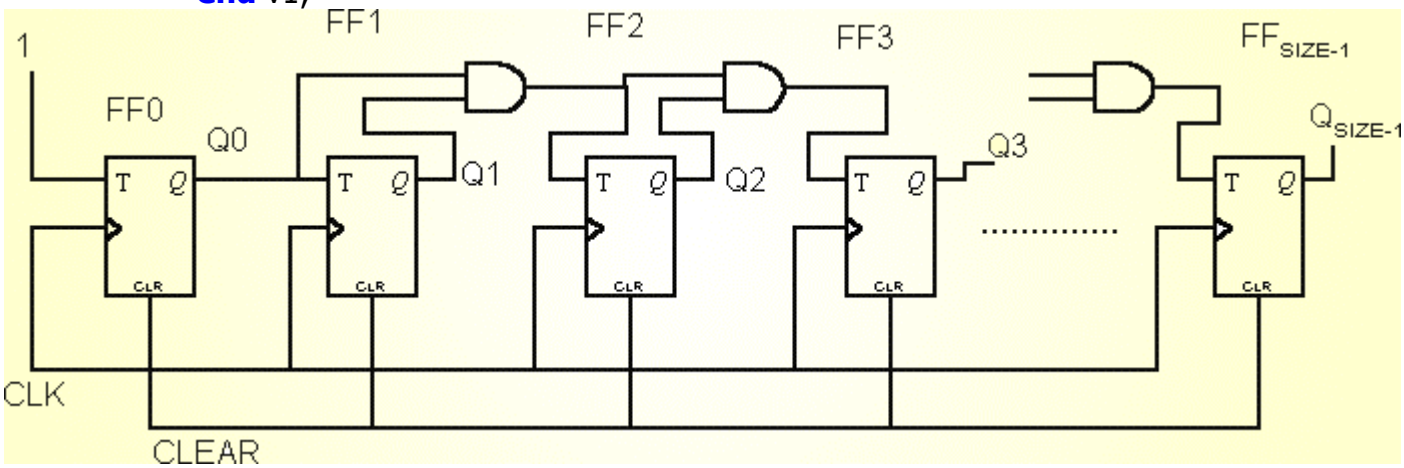
```
--n-bit binary adder using generated signal assignments
entity addn is
  generic(n : positive := 3);   --no. of bits less one
  port(addend, augend : in bit_vector(0 to n);
       carry_in : in bit; carry_out, overflow : out bit;
       sum : out bit_vector(0 to n));
end addn;
```

```
architecture generated of addn is
  signal carries : bit_vector(0 to n);
begin
addgen : for i in addend'range --range is 0 to n
  generate
     lsadder : if i = 0 generate --lsb is a special case
          sum(i) <= addend(i) xor augend(i) xor carry_in;
          carries(i) <= (addend(i) and augend(i)) or
                        (addend(i) and carry_in) or
                         carry_in and augend(i));
     end generate;
     otheradder : if i /= 0 generate --all other stages cascade
          sum(i) <= addend(i) xor augend(i) xor carries(i-1);
          carries(i) <= (addend(i) and augend(i)) or
                        (addend(i) and carries(i-1)) or
                        (carries(i-1) and augend(i));
     end generate;
  end generate;
  carry_out <= carries(n);
  overflow <= carries(n-1) xor carries(n);
end architecture generated;
```

Example 2 - n-bit Synchronous Binary Counter:

```
--a T-type flip flop
library ieee;
use ieee.std_logic_1164.all;
entity tff is
  port(clk, t, clear : in std_logic; q : buffer std_logic);
end tff;

architecture v1 of tff is
begin
  process(clear, clk) --signals process is sensitive to
  begin
    if clear = '1' then --asynchronous reset
      q <= '0';
   elsif rising_edge(clk) then
      if t = '1' then
        q <= not q;
      else
        null; --no change , q retains current value
      end if;
```

```
    end if;
  end process;
end v1;
```




```
--An scalable synchronous binary counter
library ieee;
use ieee.std_logic_1164.all;
entity bigcntr is
  generic(size : positive := 32);
  port(clk, clear : in std_logic;
       q : buffer std_logic_vector((size-1) downto 0));
end bigcntr;

architecture v1 of bigcntr is

  component tff is
    port(clk, t, clear : in std_logic; q : buffer std_logic);
  end component;

  signal tin : std_logic_vector((size-1) downto 0);

begin

  --generate counter T-type flip-flops
  genttf : for i in (size-1) downto 0 generate
    ttype : tff port map (clk, tin(i), clear, q(i));
  end generate;
  --generate counter carry AND gates
  genand : for i in 0 to (size-1) generate
    t0 : if i = 0 generate --T(0) tied to logic-1
      tin(i) <= '1';
    end generate;
    t1_size : if i > 0 generate --cascaded 2-input AND gates
      tin(i) <= q(i-1) and tin(i-1);
```

```
    end generate;
  end generate;

end v1;
```

Concurrent Procedure Call

A Procedure is one of the two types of sub-program provided by VHDL, the other is a Function. Both Procedures and Functions embody a group of sequential statements into a well defined routine which carries out some specific task. Functions are used to compute and return a value, given a set of input parameters. A common function provided within the Std_logic_1164 package is rising_edge(..). This function is passed a signal parameter and returns a Boolean value indicating whether or not the signal has undergone a rising edge transition. The function rising_edge(..) makes use of the attributes of the signal being passed to it, such as event and last_value, to deduce the return value.

Procedures can have multiple parameters of mode IN, OUT and INOUT. Signals which are either IN or INOUT parameters of a procedure are effectively in the sensitivity list for that procedure. This results in the procedure being called (invoked) whenever an event occurs on any of the actual signal parameters being passed into the procedure. In this way concurrent procedures can be used instead of component instantiation statements.

Syntax:

procedure_name [(*actual_parameter1, actual_parameter2,...*)];

Example - creating a D-Type Flip-Flop using a procedure:

```
dff1 : dff_proc(clk, clear, q);  --dff_proc is a procedure
```

Type Declaration

A type declaration creates a new type which can be assigned to any of the VHDL objects (signal, variable and constant). The most common use of the type declaration is to declare an enumeration type to represent the state of a Finite State Machine. Commonly used types are often declared inside of a package and the package can then be used by any design unit by including a use clause.

type *identifier* **is** *type_definition* ;

Subtype Declaration

Subtypes are often used to create a type which can make use of the functions supported by the base type, while having a constrained set of values, or being subject to a resolution function.

The most common example of this is the sub-type std_logic which is a resolved subtype of std_ulogic. This means that objects of type std_logic can make use of all of the functions provided for objects of type std_ulogic in addition to being *resolved*, ie. such objects can have multiple drivers.

subtype *subtype_name* **is**
[*resolution_function*] *base_type_name* [**range** *constraint*];

Signal Declaration

Creates a signal object. Signals declared within Architectures are *local* signals. They are only accessible within the confines of the Architecture. A signal may be given an initial value on declaration; if this is omitted then the signal takes on the value corresponding to the left hand element of the signal type declaration. For example, signals of type **bit** are initialised to '0' by default, whereas signals of type **std_logic** are initialised to 'U' (Unitialised).

signal *signal_name* : *type*;
signal *signal_name* : *type* := *initial_value*;

Constant Declaration

constant *constant_name* : *type* := *value*;

Component Declaration

An architecture which instantiates other design entities in the form of components must declare them explicitly in the architecture declarative part.

component *component_name* **is** --is [93]

```
        generic(generic_list);
        port(port_list);
end component;
```

Function Declaration

Sub-programs (Functions and Procedures) can be declared in the declarative part of an architecture (or an entity). However it is more usual for them to be declared in a package body. This allows any design unit to make use of them by means of a context clause.

```
function function_name(parameter_list) return type is
        declarations
begin
        sequential_statements
end function function_name;
```

Procedure Declaration

Sub-programs (Functions and Procedures) can be declared in the declarative part of an architecture (or an entity). However it is more usual for them to be declared in a package body. This allows any design unit to make use of them by means of a context clause.

```
procedure procedure_name (parameter_list) is
        declarations
begin
        sequential statements
end procedure procedure_name;
```

Configuration Specification

The Configuration Specification is used in the declarations part of the architecture body. It is used to bind together a component instance with a library unit, ie. an entity-architecture pair. It is not as powerful as the primary design unit known as a configuration declaration.

```
for instance_name : component_name
        use entity library_name.entity_name(architecture_name);
```

Examples:

```
FOR rom : rom256x8 USE ENTITY
work.rom256x8(version2);
FOR ALL : andg3 USE ENTITY work.andg3(behaviour); --
ALL selects every instance of andg3
```

Wait

The wait statement provides a mechanism for suspending execution of a process until one or more conditions are met.

There are three types of wait statement:

wait on *signal_event*;

wait until *boolean_condition*;

wait for *time_expression*;

A single wait statement can combine all three conditions or have none at all.

The following statement suspends a process indefinitely:

wait;

Execution of a wait statement causes the simulator to complete the current simulation cycle and increment time by one delta. This has the effect of updating all signals which have previously been assigned with their corresponding driver values. This would otherwise occur after execution of the process statement itself.

If a process contains wait statements, it cannot have a sensitivity list:

```
process(a,b,c)
begin
.....
wait on a,b,c ; X incorrect !
```

```
.....
end process;
```

The **wait on** *signal_event* statement provides an alternative to the sensitivity list which can appear after the word **process**. The following processes p1 and p2 are exactly the same:

```
p1 : process (a,b,c)
begin
.....
.....
.....
end process p1;
p2 : process
begin
.....
.....
wait on a,b,c;
end process p2;
```

Both processes are sensitive to events on signals a, b and c. In the lower example, the wait statement is placed at the end of the process, since all processes are executed once at the start of a simulation.

The **wait until** *boolean_condition* statement will suspend a process until the specified boolean condition becomes true. It is usual for the boolean condition to involve signals. These signals will be in an effective sensitivity list created by the statement. Whenever an event occurs on one or more of the signals, the boolean condition is tested, and if true, the process is resumed. For example, the two statements below suspend a process until a rising edge occurs on the signal named clock.

```
wait until (clock'event and clock = '1'); --signal clock is of
type bit
wait until rising_edge(clock); --signal clock is of type
std_logic
```

examples:

```
entity cntr3 is
      port(clock : in bit; count : out natural);
```

```
end cntr3;

architecture using_wait of cntr3 is
begin
      process
      begin
        wait until (clock'event and clock = '1');
        count <= 0;
        wait until (clock'event and clock = '1');
        count <= 1;
        wait until (clock'event and clock = '1');
        count <= 2;
      end process;
end using_wait;
wait for 10 ns;
--updates signals and suspends a
--process for 10 ns.

wait for 0 ns;
--updates signals and advances
--delta time.
```

Sequential signal assignment

Signal assignment statements appearing inside a process statement are *sequential* signal assignments. Within a process, only one driver is allowed per signal. Therefore multiple assignments to the same signal behave in a totally different way to multiple concurrent assignments. Each signal assignment within a process to a given signal contributes to the overall driver for that signal.

Syntax:

```
signal_name <= expression;
signal_name <= expression1 after delay1,
                           expr2 after del2,
                           expr3 after del3,....;
```

Example:

```
process begin
rx_data <= transport 11 after  10 ns; --(1)
rx_data <= transport 20 after  22 ns; --(2)
```

```
rx_data <= transport 35 after  18 ns; --(3)
end process;
```

In the above example, the driver for signal rx_data is updated following the execution of each statement as follows:

after (1) : rx_data <------ curr @ now ,11 @ 10 ns

after (2) : rx_data <------ curr @ now ,11 @ 10 ns , 20 @ 22 ns

after (3) : rx_data <------ curr @ now ,11 @ 10 ns , 35 @ 18 ns (previous transaction is overwritten)

In the above case, after statement (3) is executed the transaction '20 @ 22 ns' is deleted from the driver and replaced with transaction '35 @ 18 ns', since the delay for the latter is shorter. The rules governing how signal drivers are affected by multiple sequential signal assignments are complex and therefore beyond the scope of this guide. A basic rule which must be observed when modelling digital hardware is:

- **Within the confines of a process there is only one driver allowed per signal**

Multiple signal assignments to the same signal within a process are often used to avoid unwanted latches being created when a combinational process is synthesised. All output signals can be assigned default output values at the top of the process. These values can be conditionally overwritten by statements in the body of the process which ensures that all outputs have a defined output value for all possible input combinations. See the example below:

```
process(I)
 begin
        GS <= '1'; --set default outputs
        A <= "000";
        if I(7) = '1' then
                A <= "111";  --override default A
        elsif I(6) = '1' then
                A <= "110";
        elsif I(5) = '1' then
                A <= "101";
        elsif I(4) = '1' then
                A <= "100";
        elsif I(3) = '1' then
                A <= "011";
```

```
    elsif I(2) = '1' then
        A <= "010";
    elsif I(1) = '1' then
        A <= "001";
    elsif I(0) = '1' then
        null; --assign defaul outputs
    else
        GS <= '0'; --override default GS
    end if;
end process;
```

The following process creates a repetitive clock

```
clock_process : process
begin
    clock <= '0', '1' after 50 ns;
    wait for 100 ns;
end process;
```

Variable assignment

A variable assignment takes effect immediately; there is no time aspect. The values of signals may be assigned to variables provided the types match. Variables are used inside processes to hold local values.

variable_name := *expression*;

examples:

```
x := y;
q := d;
```

If then else

Note that both the elsif and else parts of the statement are optional and the elsif part is repeatable. The *boolean_expression* is usually a relational expression which returns either true or false. If none of the conditions are true, the statements following the else keyword are executed, if an else part is included. Each condition is tested sequentially, the first to return a true result causes the statement immediately following to execute. The next statement to execute is that which follows the 'end if' keyword.

if *boolean_expression* **then**

```
        {sequential_statements}
 {elsif boolean_expression then
        {sequential_statements} }
[ else
        {sequential_statements} ]
end if;
```

Examples:

```
if (x < 10) then
   a := b;
end if;
if (day = sunday) then
   weekend := true;
elsif (day = saturday) then
   weekend := true;
else
   weekday := true;
end if;
```

Case

When *expression* evaluates to a value which matches one of the *choices* in a **when** statement part, the statements immediately following will be executed up to the next **when** part. If none of the specified choices match, the **case** statement should include an **others** clause in the final when part. After the selected statements have executed control is transferred to the statement following the **end case** key words.

```
case expression is
    when choice { | choice } =>
        {sequential_statements}
  { when choice { | choice } =>
        {sequential_statements} }
end case;
```

Examples:

```
case instruction is
  when load_accum =>
        accum <= data;
  when store_accum =>
        data_out <= accum;
  when load|store =>
```

```
            process_io(addr);
    when others =>
            process_error(instruction);
end case;
.....
variable word3 : bit_vector(0 to 2);
variable number : natural;

case word4 is
    when "000" =>
        number := 0;
    when "100" =>
        number := 1;
    when "010" =>
        number := 2;
    when "110" =>
        number := 3;
    when "001" =>
        number := 4;
    when "101" =>
        number := 5;
    when "011" =>
        number := 6;
    when "111" =>
        number := 7;
end case;
```

Loop

Used for repetitive execution of a statement or statements. The sequential statements enclosed within a 'while' loop will execute repeatedly as long as the *boolean_expression* returns a true value. A 'for' loop will execute as many times as specified in the *discrete_range* which can take the form:

simple_expression **to**|**downto** *simple_expression*

In the above, the two *simple_expressions* are usually integers specifying the number of times the loop is to execute.

A simple loop statement without an iteration scheme or *boolean_condition* is an infinite loop.

The index variable **i** is declared locally by the **for loop** statement. There is no need to declare variable **i** explicitly in the process, function or procedure. In a **for loop** statement the index variable **i** must not be assigned a value within the body of the loop, ie. **i** must not appear on the left hand side of a variable assignment statement. However, **i** can be used on the right hand side of a statement.

The syntax of the three variations on the loop statement are shown below:

```
--looping through a set range
[loop_label : ] for identifier in discrete_range loop
                { sequential_statements }
end loop [loop_label];

--entry test loop
[loop_label : ] while boolean_expression loop
                { sequential_statements }
end loop [loop_label] ;


--unconditional loop
[loop_label : ] loop
                { sequential_statements }
end loop [loop_label] ;
```

Examples:

```
while (day = weekday) loop
  day := get_next_day(day);
end loop;
-----------------------------------
for i in 1 to 10 loop
  i_squared(i) := i * i;
end loop;
-------------------------------------------------
variable binword8 : bit_vector(0 to 7) := "10101111";
variable outputnum : integer := 0;

for i in 0 to 7 loop
  if  binword8(i) = '1' then
    outputnum := outputnum + 2**i ;
  end if;
end loop;
-------------------------------------------------
type day_of_week is (sun,mon,tue,wed,thu,fri,sat);
```

```
for i in day_of_week loop
  if i = sat then son <= mow_lawn;
  elsif i = sun then church <= family;
  else dad <= go_to_work;
  end if;
end loop;
```

Next

The **next** statement is used to prematurely terminate the current iteration of a while, for or infinite loop.

```
next;
next loop_label;
next loop_label when condition;
```

Example:

```
for i in 0 to 7 loop
  if skip = '1' then
    next;
  else
    n_bus <= table(i);
    wait for 5 ns;
  end if;
end loop;
```

Exit

The **exit** statement is used to prematurely terminate a while, for or infinite loop.

```
exit ;
exit loop_label;
exit loop_label when condition;
```

Example:

```
for i in 0 to 7 loop
  if finish_loop_early = '1' then
    exit;
```

```
    else
        n_bus <= table(i);
        wait for 5 ns;
    end if;
end loop;
```

Null

The **null** statement performs no action. It is usually used with the case or if..then statement, to indicate that under certain conditions no action is required.

```
null;
```

A jointly validated MSc course taught over the internet; a programme supported by EPSRC under the Integrated Graduate Development Scheme (IGDS).

website www.ami.ac.uk

Site developed by CRAL. Bolton Institute.
Last updated 01.12.01 RA