VHDL Synthesis Reference

Summary

This comprehensive reference provides detailed information with respect to synthesis of VHDL code. It also contains an overview section regarding the syntax of the VHDL Language.

Technical Reference
TR0115 (v1.1) June 10, 2005

VHDL is a hardware description language (HDL). It contains the features of a conventional programming language, a classical PLD programming language, and a netlist, as well as design management features.

VHDL is a large language and it provides many features. This reference does not attempt to describe the full language - rather it introduces enough of the language to enable useful design.

The VHDL Synthesis engine supports most of the VHDL language, however, some sections of the language have meanings that are unclear in the context of logic design - the file operations in the package "textio", for example. The exceptions and constraints on the Synthesizer's VHDL support are listed in the topics 'Unsupported Constructs', 'Ignored Constructs', and 'Constrained Constructs'.

The VHDL Synthesizer uses the VHDL'93 version of VHDL. This version is basically a superset of the previous standard VHDL'87.

Notation Conventions

VHDL is not case-sensitive, so a design description can contain UPPERCASE or lowercase text. In this reference, examples are all lowercase. VHDL reserved words in both the text and the examples are **bold**, for example :

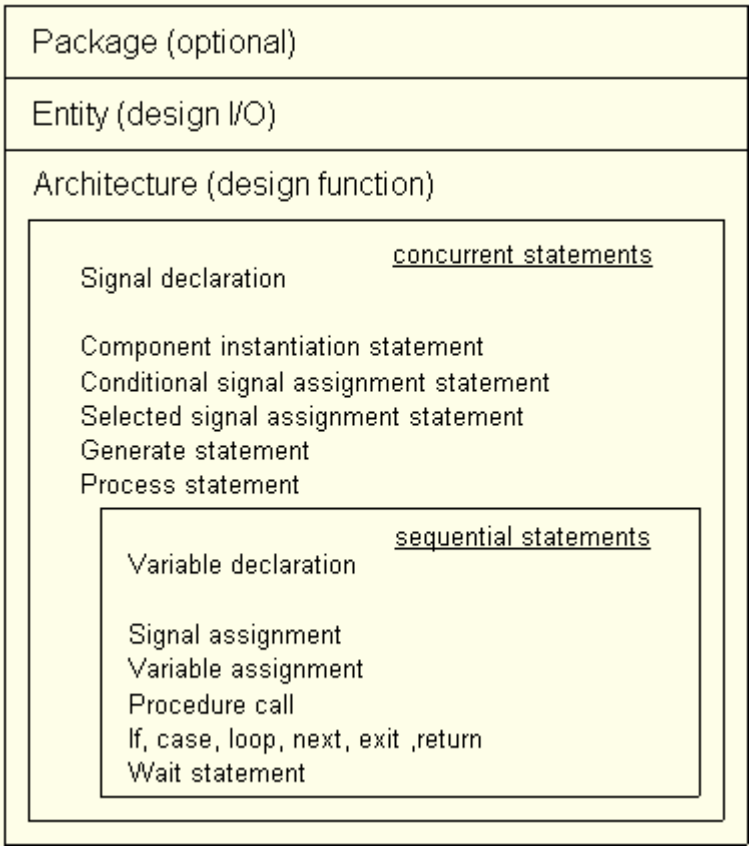
```
entity counter is
   port (clk, reset: in bit; sum: out integer);
end counter ;
```

bold In examples, bold type indicates a reserved word. In the example above, **entity**, **is**, **port**, **in**, **out**, and **end** are all reserved words

plain-text Regular plain type represents a user-definable identifier or another VHDL construct. Reserved words cannot be used as user-defined identifiers. In the example above, the name "sum" is a user-defined identifier.

Structure of a VHDL Design Description

The basic organization of a VHDL design description is shown below:



A **package** is an optional statement for shared declarations. An **entity** contains declarations of the design I/O, and an **architecture** contains the description of the design. A design may contain any number of package, entity and architecture statements. Most of the examples in this guide use a single entity-architecture pair.

An architecture contains concurrent statements. Concurrent statements (like netlists and classic PLD programming languages) are evaluated independently of the order in which they appear. Values are passed between statements by **signals**; an assignment to a **signal** (<=) implies a driver. A **signal** can be thought of as a physical wire (or bundle of wires).

The most powerful VHDL constructs occur within sequential statements. These must be placed inside a particular concurrent statement, the **process** statement, or inside a **function** or **procedure**.

Sequential statements are very similar to programming language statements: they are executed in the order they are written (subject to if statements, return statements, etc.). Values are held in **variables** and **constants**. **Signals** are used to pass values in and out of a **process**, to and from other concurrent statements (or the same statement).

Several concepts are important to the understanding of VHDL. They include: the distinction between concurrent statements and sequential statements, and the understanding that signals pass values between concurrent statements, and variables pass values between sequential statements.

Sequential statements in VHDL refer to statement ordering, not to the type of logic compiled. Sequential statements may be used to compile both combinational and sequential logic.

VHDL can be written at three levels of abstraction: structural, data flow, and behavioral. These three levels can be mixed.

The following topics: Structural VHDL, Data Flow VHDL, and Behavioral VHDL, introduce the structural, data flow, and behavioral design methods and show VHDL code fragments that are written at each level of abstraction.

Variations of the following design are used to illustrate the differences:

```
entity hello is
   port (clock, reset : in  boolean; char : out character);
end hello;

architecture behavioral of hello is
   constant char_sequence : string := "hello world";
   signal step : integer range 1 to char_sequence'high := 1;
begin
   -- Counter
   process (reset,clock)
   begin
      if reset then
         step <= 1;
      elsif clock and clock'event then
         if step = char_sequence'high then
            step <= 1;
         else
            step <= step + 1;
         end if;
      end if;
   end process;

   -- Output Decoder
   char <= char_sequence(step);
end behavioral ;
```

This design compiles to a simple waveform generator with two inputs (clock and reset) and eight outputs. The output sequences through the ASCII codes for each of the eleven characters in the string "hello world". The codes change some logic delay after each rising edge of the clock. When the circuit is reset, the output is the code for 'h' -- reset is asynchronous.

Structural VHDL

A structural VHDL design description consists of component instantiation statements, which are concurrent statements. For example:

```
u0: inv port map (a_2, b_5);
```

This is a netlist-level description. As such, you probably do not want to type many statements at the structural level. Schematic capture has long been known as an easier way to enter netlists.

Structural VHDL simply describes the interconnection of hierarchy. Description of the function requires the data flow or behavioral levels. Component instantiation statements are useful for sections of design that are reused, and for integrating designs.

The design in the following example has been partitioned into two instantiated components. Note that the components are declared but not defined in the example. The components would be defined as entity/architecture pairs.

```
entity hello is
   port (clock, reset : in  boolean; char : out character);
end hello;


architecture structural of hello is
   constant char_sequence : string := "hello world";
   subtype short is integer range 1 to char_sequence'high;
   signal step : short;
   component counter
      port (clock, reset : in  boolean; num : out short);
   end component;
   component decoder
      port (num : in short ; res : out character);
   end component;
begin
   U0 : counter port map (clock,reset,step);
   U1 : decoder port map (step,char);
end  structural;
```

This is useful if counter and decoder had been previously created and compiled into two PALs. The availability of a larger PAL allows you to integrate the design by instantiating these as components and compiling for the larger device.

Data Flow VHDL

Another concurrent statement is the signal assignment. For example:

```
a <= b and c;
m <= in1 when a1 else in2;
```

Assignments at this level are referred to as data flow descriptions. They are sometimes referred to as RTL (register-transfer-level) descriptions.

This example could be rewritten as :

```
entity hello is
   port (clock, reset: in  boolean; char : out character);
end hello;

architecture data_flow of hello is
   constant char_sequence : string := "hello world";
   signal step0, step1 :integer range 1 to char_sequence'high := 0;
begin
-- Output decoder
char <= char_sequence(step1);

-- Counter logic
step1 <= 1 when step0 = char_sequence'high else step0 + 1;

-- Counter flip flops
step0 <= 1 when reset else
   step1 when clock and clock'event;
end data_flow;
```

In data flow descriptions combinational logic is described with the signal assignment (<=). There is no register assignment operator; sequential logic is inferred from incomplete specification (of step0) as in the example above.

Behavioral VHDL

The most powerful concurrent statement is the process statement. The process statement contains sequential statements and allows designs to be described at the behavioral level of abstraction. For example :

```
process (insig)
   variable var1: integer; -- variable declaration
begin
   var1:= insig; -- variable assignment
   var1:= function_name(var1 + 1); -- function call
end process;
```

In hardware design the process statement is used in two ways: one for combinational logic and one for sequential logic. To describe combinational logic the general form of the process statement is :

```
process (signal_name, signal_name, signal_name,......)
begin
   .....
end process;
```

and the general form for sequential logic :

```
process (clock_signal)
begin
   if clock_signal and clock_signal'event then
   ....
   end if;
end process;
```

For combinational logic there is a list of all process input signals after the keyword **process**. For sequential logic there is either: (a) no sensitivity list but there is a **wait** statement; or (b) a sensitivity list containing the clock and the statements are within an if statement.

It is illegal in VHDL for a process to have both a sensitivity list and a wait statement. To have neither implies no logic.

Our example could be viewed as two processes: one for the sequential counter, and one of the combinatorial decoder :

```
entity hello is
   port (clock, reset : in  boolean; char : out character);
end hello;


architecture behavioral of hello is
```

```
   constant char_sequence : string := "hello world";
   signal step : integer range 1 to char_sequence'high := 1;
begin
   counter : process (reset, clock)
   begin
      if reset then
         step <= 1;
      elsif clock and clock'event then
         if step = char_sequence'high then
            step <= 1;
         else
            step <= step + 1;
         end if;
      end if;
   end process ;

   decoder :process (step)
   begin
      char <= char_sequence(step);
   end process;
end behavioral;
```

VHDL Types

VHDL contains the usual programming language data types, such as:

- boolean
- character
- integer
- real
- string

These types have their usual meanings. In addition, VHDL has the types:

- bit
- bit_vector

The type bit can have a value of '0' or '1'. A bit_vector is an array of bits. (Similarly, a string is an array of characters in VHDL just as it is in Pascal).

Most electrical engineers use the IEEE 1164-standard-logic types in place of bit and bit_vector.

- **std_logic**
- **std_logic_vector**

These are declared in the IEEE library in package std_logic_1164

To make these declarations visible, an entity that uses these types is prefixed with a library declaration and a use clause:

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

Definitions for all of the predefined types can be found in the file **std.vhd**, which contains the **package** standard.

The type of a **variable**, **signal**, or **constant** (which are collectively called objects) determines the operators that are predefined for that object. For hardware design, the type also determines the number -- and possibly the encoding scheme used -- for the wires that are implemented for that object.

Type-checking is performed during analysis. Types are used to resolve overloaded subprograms. Users may define their own types, which may be scalars, arrays, or records.

VHDL also allows subtypes. This is simply a mechanism to define a subset of a type. More information on the impact of types and subtypes on synthesis is contained in the topic 'Synthesis of VHDL Types'.

Simulatable, but not necessarily synthesizable

This section is primarily for experienced VHDL simulation users who have VHDL code that has been developed using a VHDL simulator.

VHDL is a standard, how can there be a problem? Many VHDL models are not suitable for synthesis, such as high level performance models, environment models for stimulus/response, or system models including software, hardware, and physical aspects.

Synthesis assumes that the VHDL code describes the logic of a design, and not some model of the design. This assumption puts additional constraints on the programmer. Most of the remainder of this guide describes how to program in VHDL within these constraints.

A design description may be correctly specified in English, but may have no practical hardware implementation (e.g. H.G.Wells' Time Machine). The same is true for a design specified in VHDL, which may have no practical implementation. Just because its written in a Hardware Description Language doesn't mean it describes realizable hardware!

For example, suppose you have a VHDL simulation model of a PAL, lets say the model configures itself from a JEDEC file during simulation initialization. The model actually simulates the programming and logic of the PAL. It describes more than just the hardware, the model also describes the manufacturing step when the PAL was programmed. A synthesizable VHDL model would only describe the component function and not the earlier manufacturing step.

Some other issues

Hardware design adds several additional constraints such as gated clocks. These are not a constraint in a simulation where values may be written to computer memory without concern for electrical glitches. Hardware design requires care be taken in controlling the clocking of memory elements.

A simulation model may also describe the timing characteristics of a design. These are ignored by the synthesis tool, which considers timing a result of the hardware realization of the design. A VHDL model that depends on the timing for correct operation may not synthesize to the expected result.

A simulation model may use enumerated types to:

- represent the encoding of a group of wires (e.g. load store execute), perhaps as part of a state machine description
- represent the electrical characteristic on a single wire (e.g. high impedance, resistive, strong), as well as the state of the simulation (unknown, uninitialized)

Within VHDL, a synthesis system has no way to distinguish the meaning in each case. The synthesizer assumes the encoding representation for enumerated types unless the encoding is explicitly specified using the attribute 'enum_encoding'.

PLD Programming using VHDL

VHDL is a large language. It is an impractical task to learn the whole language before trying to use it. Fortunately, it is not necessary to learn the whole language in order to use VHDL (the same is true of any computer or even human language). This section presents a view of VHDL that should be familiar to users of classic PLD programming languages.

Just as in PLD programming, the design I/O, combinational logic, sequential logic, and state machines can be described. Initially, only signals of type std_logic and std_logic_vector (a 1 dimensional array of std_logic) will be considered. These types allow you to do logical operations (and, or...) and relational operations (equal, greater than,....).

Design I/0

Design I/O is described using a port statement. Ports may have mode IN, OUT , INOUT or BUFFER. The mode describes the direction of data flow. The default mode of a port is IN. Values may be assigned to ports of mode OUT and INOUT or BUFFER, and read from ports mode IN and INOUT or BUFFER. Port statements occur within an entity. For example :

```
entity ent1 is
   port (a0,a1,b0,b1 : in std_logic; c0, c1 : out std_logic);
end ent1;


entity ent2 is
   port (a,b : std_logic_vector(0 to 5);
         sel : std_logic; c : out std_logic_vector(0 to 5)) ;
end ent2;
```

INOUT and BUFFER are used to specify routing on ports. An INOUT port specifies bi-directional dataflow, and a BUFFER port is a unidirectional OUT that you can read from. INOUT describes a signal path that runs through a pin and back into the design: "pin feedback" in PLDs or an IO block in some FPGAs. BUFFER describes a signal path that drives an output buffer to a pin and internal logic: "register feedback" in PLDs or internal routing in FPGAs. INOUT is required to specify pin feedback. Register feedback may be specified using BUFFER or using OUT and an extra signal.

It is also a convention to use another standard, IEEE 1164. To use this standard, the following two lines are written before each entity (or package) to provide visibility to the definition of 'std_logic'. This is not required, it's just a convention.

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

Combinational Logic

Combinational logic may be described using concurrent statements, just like equations in PLD languages. Concurrent statements occur within an architecture. Note that an architecture references an entity.

The equations assign values to signals. Ports are examples of signals; all signals must be declared before they are used. A two bit adder can be described using boolean equations :

```
architecture adder of ent1 is
   signal d, e : std_logic;
begin
   d    <= b0 and a0;
   e    <= b1 xor a1;
   c0   <= (b0 and not a0) or (not b0 and a0);
   c1   <= (e and not d) or (not e and d);
end adder;
```

Conditional assignment can also be performed. Here conditional assignment is used to build a mux:

```
architecture mux1 of ent2 is
begin
   c <= a when sel = '1' else b;
end mux1;
```

Note that omitting the '**else** b' above would specify a latch:

```
c <= a when sel = '1';
```

because this would then have the same meaning as:

```
c <= a when sel = '1' else c;
```

The meaning is different to some PLD languages, which may assume a default else to be 'zero', or perhaps 'don't care'. VHDL'93 is also different from VHDL'87 which required the **else** to be present.

Generate is a concurrent looping structure. This construct allows another possible implementation of the mux. This example also illustrates selecting elements of arrays:

```
architecture mux2 of ent2 is
begin
   for i in 0 to 5 generate
      c(i) <= (a(i) and sel) or (b(i) and not sel);
   end generate;
end mux2;
```

Registers and Tri-state

VHDL does not contain a register assignment operator; registers are inferred from usage. Therefore, a D latch could be described :

```
q <= d when clk = '1';
```

and a D flip flop :

```
q <= d when clk = '1' and clk'event
```

and a D flip flop with asynchronous reset:

```
q <= '0' when rst = '1' else d when clk = '1' and clk'event
```

In practice, the clk'event expression is a little cumbersome. This can be improved upon by using the rising_edge () function from std_logic_1164. In the following example output registers are added to the combinational adder:

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is
   port (a0,a1,b0,b1,clk : in  std_logic; c0, c1 : out std_logic);
end counter;

architecture adder_ff of counter is
   signal d, e, f, g  : std_logic;
begin
  d  <= b0 and a0;
  e  <= b1 xor a1;
  f  <= (b0 and not a0) or (not b0 and a0);
  g  <= (e and not d) or (not e and d);
  c0 <= f when rising_edge(clk);
  c1 <= g when rising_edge(clk);
end adder_ff;
```

Tristates can be added in much the same way as flip flops, by using a conditional assignment of 'Z' (here controlled by an input oe):

```
architecture adder_ff_tri of counter is
   signal d, e, f, g, h, i  : std_logic;
begin
   d  <= b0 and a0;
   e  <= b1 xor a1;
   f  <= (b0 and not a0) or (not b0 and a0);
```

```
   g  <= (e and not d) or (not e and d);
   h  <= f when rising_edge(clk);
   i  <= g when rising_edge(clk);
   c0 <= h when oe = '1' else 'Z';
   c1 <= i when oe = '1' else 'Z';
end adder_ff_tri;
```

Procedures can be used to make the intent of the design a little clearer, such as moving the combinational logic into a procedure. Notice that procedures contain programming language like 'sequential statements' and that intermediate values in the example below are held in variables. Notice also that signals are assigned with "<=", and variables with ":=". Like programming languages, the order of sequential statements is important.

```
architecture using_procedure of counter is
   signal f, g  : std_logic;
   procedure add (signal a0,a1,b0,b1 : std_logic;
      signal c0,c1 : out std_logic) is
   variable x,y : std_logic;
   begin
      x    := b0 and a0;
      y    := b1 xor a1;
      c0   <= (b0 and not a0) or (not b0 and a0);
      c1   <= (y and not x) or (not y and x);
   end;
begin
   add ( a0, a1, b0, b1, f, g);
   c0 <= f when rising_edge(clk);
   c1 <= g when rising_edge(clk);
end using_procedure;
```

State Machines

There is no state transition view in VHDL, however, it does support a behavioral view. This allows design description in a programming-language-like way. Sequential statements may also occur in processes.

```
q <= d when rising_edge(clk);
```

An exactly equivalent statement is :

```
process(clk)
begin
   if rising_edge(clk) then
      q <= d;
   end if;
end process;
```

The process statement may contain many sequential statements. This simple behavioral description is very like a state machine description in a classic PLD language.

```
library ieee;
use ieee.std_logic_1164.all;
entity ent5 is
   port (clk,reset : in std_logic;
         p : buffer std_logic_vector(1 downto 0));
end ent5 ;

architecture counter1 of ent5 is
begin
   process (clk, rst)
   begin
      if reset = '1' then
         p <= "00";
      elsif rising _edge(clk) then
         case p is
         when "00" => p <= "01";
         when "01" => p <= "10";
         when "10" => p <= "11";
         when "11" => p <= "00";
         end case;
      end if;
   end process ;
end counter1 ;
```

Although the process statement can be used as a way to describe state machines, it more generally allows behavioral modeling of both combinational and sequential logic.

It is strongly recommended that you read the topics 'Programming Sequential Logic' and 'Programming Finite State Machines' before attempting to write process statements; it is important to understand the impact of the wait statement on signals and variables in the process statement.

Hierarchy

In VHDL each entity and architecture combination defines an element of hierarchy. Hierarchy may be instantiated using components. Since there is a default binding between a component and an entity with the same name, a hierarchical design instantiating Child in Parent looks like:

```
---Child
library ieee;
use ieee.std_logic_1164.all;

entity Child is
   port (I : std_logic_vector(2 downto 0) ;
         O : out std_logic_vector(0 to 2));
end Child;

architecture behavior of Child is
begin
  o <= i;
end;

---Parent
use ieee.std_logic_1164.all;
entity Parent is
   port(a : std_logic_vector(7 downto 5);
         v : out std_logic_vector( 1 to 3));
end Parent;

architecture behavior of Parent is
 -- component declaration , bound to Entity Child above
   component Child
      port (I : std_logic_vector(2 downto 0) ;
            O : out std_logic_vector(0 to 2));
   end component;
begin
   -- component instantiation
   u0 : Child port map (a,v);
end;
```

Hierarchy also allows VHDL design partitioning, reuse, and incremental testing. VHDL synthesis incorporates some additional semantics of hierarchy; including controlling the logic optimize granularity, hierarchical compile, and instantiating silicon specific components as shown below.

```
---Parent
library ieee;
use ieee.std_logic_1164.all;
entity Parent is
   port (a,en : std_logic_vector(2 downto 0);
         v : out std_logic);
end Parent;

architecture behavior of Parent is
   -- component declaration , unbound
   -- ASSUMES 'TriBuf' and 'Pullup' are silicon specific components
   -- defined by some downstream tool.
   component TriBuf is
      port (I,T : std_logic ; O : out std_logic);
   end component;
   component Pullup is
      port (O : out std_logic);
   end component;
signal tri_net : std_logic;
begin
   -- component instantiations
   u0 : TriBuf port map (a(0), en(0), tri_net);
   u1 : TriBuf port map (a(1), en(1), tri_net);
   u2 : TriBuf port map (a(2), en(2), tri_net);
   u3 : Pullup port map (tri_net);
   v <= tri_net;
end;
```

Types

The use of types other than 'std_logic' and 'std_vector_logic' can make your design much easier to read. It is good programming practice to put all of your type definitions in a package, and make the package contents visible with a use clause. For example:

```
package type_defs is
   subtype very_short is integer range 0 to 3;
end type_defs;

library ieee;
use ieee.std_logic_1164.all;
use work.type_defs.all;

entity counter2 is
   port (clk, reset : std_logic; p : buffer very_short);
end counter2 ;

architecture using_ints of counter2 is
begin
   process(clk,reset)
   begin
      if reset = '1' then
         p <= 0;
      elsif rising_edge(clk) then
         p <= p + 1;
      end if;
   end process;
end;
```

In this example, type integer has been used because the "+" operator is defined for integers, but not for std_logic_vectors.

Sometimes there are other packages written by third parties that you can use, such as the Synopsys packages included with the Synthesizer. One of these packages defines a "+" operation between a std_logic_vector and an integer. Using this package the example can be rewritten:

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

entity counter2 is
   port (clk, reset : std_logic;
         p : buffer std_logic_vector(1 downto 0));
end counter2 ;

architecture using_ints of counter2 is
begin
   process(clk,reset)
   begin
      if reset = '1' then
         p <= "00";
      elsif rising_edge(clk) then
         p <= p + 1;
      end if;
   end process;
end;
```

It is a convention that the Synopsys packages be placed in the IEEE library, however, they are not an IEEE standard. To add these packages to the IEEE library use the lib alias compile option to specify ieee.vhd and synopsys.vhd.

Compiling

You can compile each design unit (an entity and an architecture) one at a time and link the results with some downstream tool, or you can compile your whole design in one pass. The former is probably the best approach during design development, the latter is the easiest if you are importing a pre-existing design.

When compiling the whole design in one pass there can be a problem due to the constraints of downstream tools if the compiled data format is Open Abel 2. There is no problem if the complied data format is EDIF or DSL. The problem may occur with large designs and the solution requires that the design be compiled in partitions of no more than 5000 compiled gates as described below.

The compile and link methodology for EDIF and DSL allows any compile granularity as long as the downstream tool supports linking of multiple files, this linking is usually automatic, when the linker encounters an undefined component it assumes the component is defined in a file of the same name as the component. When compiling lower levels it may be necessary to inhibit IO buffer insertion with a compile option.

It is possible to write VHDL code where the logic of a design unit depends upon another design unit. For example if generics, or ports of an unconstrained array type, or signals declared in packages are used. In this case the parent and child must be compiled at the same time, this case does occur but is not the most common usage.

Considerations when the compiled data format is Open Abel 2

Before compiling a design, you should consider the compile granularity. It is uncommon to compile a large design in one big "Partition". Partitioning a design into several Partitions of 500 to 5000 output gates each is most common.

A Partition may consist of one or more VHDL entities or architecture pairs and may be contained in one or more .vhd files. The method for specifying file names is described in the documentation for the software that calls the Synthesizer's compiler. Each Partition compiles to one output netlist file. Note that specifying the logic compile granularity is distinct from specifying the logic optimize granularity.

To compile a Partition, do the following :

- specify the files(s)
- specify the top level of this Partition
- compile VHDL

Several Partitions may be compiled in any order, then linked with a netlist linker.

For each Partition:

```
{
   /* do one Partition */
      a) specify files(s)
      b) specify the top level of this Partition
      c) compile VHDL
}
```

Link the output files.

Choose a Partition size of 500 to 5000 output gates because :

- Smaller Partitions give faster compiles for design iterations.
- Some downstream tools exhibit performance constraints with large Partitions.

This partitioning is important to your success. If the design is not your design and you don't know how much logic (output gates) it contains, then try some tests on parts of the design before selecting the partitions. Not partitioning a large design is probably a bad choice. Note, however, that partitions of 500 gates are not always required; sometimes a Partition just connects other Partitions and contains no logic.

There is a special case in partitioning for synthesis. If a VHDL component instantiation uses 'generic map', the parent and child must be compiled in the same Partition. The Parent is the Architecture containing the instantiation, and Child is the instantiated Entity. Because generic map implies different logic for each instance, it is possible to have a case where the same Child is compiled in several different Partitions.

Debugging

A very personal issue - here are some suggestions for debugging the specification and implementation of your design.

System level simulation

Simulate your VHDL design before synthesis using a third party VHDL simulator. This allows you to verify the specification of your design. If you don't have a VHDL simulator, run the Synthesizer with the compile option optimize set to zero (to minimize compile time), and simulate the output with your equation or gate level simulator.

Hierarchy

Partition your design into entity/architecture references as components. Compile each entity separately. Simulate using a third party equation or netlist simulator to verify functionality.

Check the size of the logic in this module. Is it about what you expected?

Using hierarchy to represent the modules of your design will result in faster and better optimization, and may allow you to reuse these design units in future designs.

Attribute 'critical'

Critical forces a signal to be retained in the output description. This allows you to trace a VHDL signal using your third party equation or netlist simulator.

The name of the VHDL signal will be maintained - but may be prefixed with instance, block labels, or package names, and suffixed with a "_n", if it represents more than one wire.

Log file

This file contains additional information about inferred logic structures. The information is printed on a per process basis, indicating the inferred structure, type of any control signals, and a name.

flip flop [type] <name> [bit]

latch [type] <name> [bit]

tristate <name> [bit]

critical <name> [bit]

comb fb <name> [bit]

macrocell <name>

This information lists the inferred structure. The name field represents the name of an inferred logic element. The name will be a local signal or variable name from within the VHDL source code. The name of the structure in the output file will be derived within a larger context and may be different. If no user recognizable name exists, the name field will contain "[anonymous]". The bit field is optional. A macrocell name will be a predefined Xblox or LPM name.

Note that the design statistics printed at the end of the compile may not be the same as the sum of the per process inference information. There are three possible reasons for this:

- Optimization may remove or change inferred structures.
- Additional combinational feedback paths explicitly specified (i.e. not inferred) between processes.

- Additional instantiated macrocells.

The compiler will also report in the log file signal assignment and usage that is legal VHDL but that indicates a possible programming error. These reports can indicate subtle problems such as comparing arrays of different lengths, or mismatch between a component port declaration and its entity port declaration.

700:NOTE: Signal 'name' is used but not assigned and is driven by its default value.

701:NOTE: Port 'name' is not assigned and is driven by its default value.

702: NOTE: Signal 'name' is assigned but not used.

703: NOTE: Port 'name' is not used.

Report and assert statements

The VHDL report statement specifies no logic and is useful for following the execution of the compiler -- perhaps to see when functions are called or to see iterations of a loop. For example:

```
entity loop_stmt is
   port (a: std_logic_vector (0 to 3);
         m: out std_logic_vector (0 to 3));
end loop_stmt;

architecture example of loop_stmt is
begin
   process (a)
      variable b: integer;
   begin
      b := 1;
      while b < 7 loop
         report "Loop number = " & integer'image(b);
         b := b + 1;
      end loop;
   end process;
end example;
```

If an assert statement is used in place of a report statement, the value of the assert condition must be false in order for a message to be written. In synthesis, if the value of condition depends upon a signal, it is not possible for the compiler to evaluate to either true or false. In this case no message is written (i.e. as if true). This can lead to confusion during debugging. The best plan is not to use signals or variables in the assert condition. Also note: the execution of the report or assert statement should not depend on an if or case statement - that in turn depends on signals or variables.

Downstream Tools

Third-party tools that take the output from the Synthesizer's compiler are referred to as downstream tools. Downstream tools sometimes make use of attributes (also called properties or parameters) within the netlist to direct their operation. Attribute examples include placement information such as pin number or logic cell location name. These are added using VHDL attributes or VHDL generic maps.

The supported features depend on the output format and the way it supports properties. The output format depends on the OEM environment that calls the compiler.

Understanding Synthesis Tools

It is important to understand that synthesis tools do not design for you. Synthesis tools do handle design details to enable you to be more productive.

The single most productive thing you can do is to be aware of what, and how much hardware you are describing using an HDL.

Conversely, writing HDL without considering the hardware, and expecting the synthesis tool to 'do the design' is a recipe for disaster. A common mistake is to create a design description, validate with a simulator, and assume that a correct specification must also be a good specification.

Programming Combinational Logic

This section shows the relationship between basic VHDL statements and combinational logic. The resulting logic is represented by schematics (one possible representation of the design), provided to illustrate this relationship. The actual implementation created by the Synthesizer depends upon other VHDL statements in the design that affect the logic minimization, and on the target technology, which affects the available gate types.

Most of the operators and statements used to describe combinational logic are the same as those found in any programming language. Some VHDL operators are more expensive to compile because they require more gates to implement (like programming languages where some operators take more cycles to execute). You need to be aware of these factors. This section describes the relative costs associated with various operators.

If an operand is a constant, less logic will be generated. If both operands are constants, the logic can be collapsed during compilation, and the cost of the operator is zero gates. Using constants (or more generally metalogic expressions) wherever possible means that the design description will not contain extra functionality. The result will compile faster and produce a smaller implementation.

Certain operators are generally restricted to use with specific types. See 'Logical Operators' and 'Arithmetic Operators' for more information.

In VHDL, operators can also be redefined for any type. This is known as operator overloading, but it is outside the scope of this reference.

Logical Operators

VHDL provides the following logical operators:

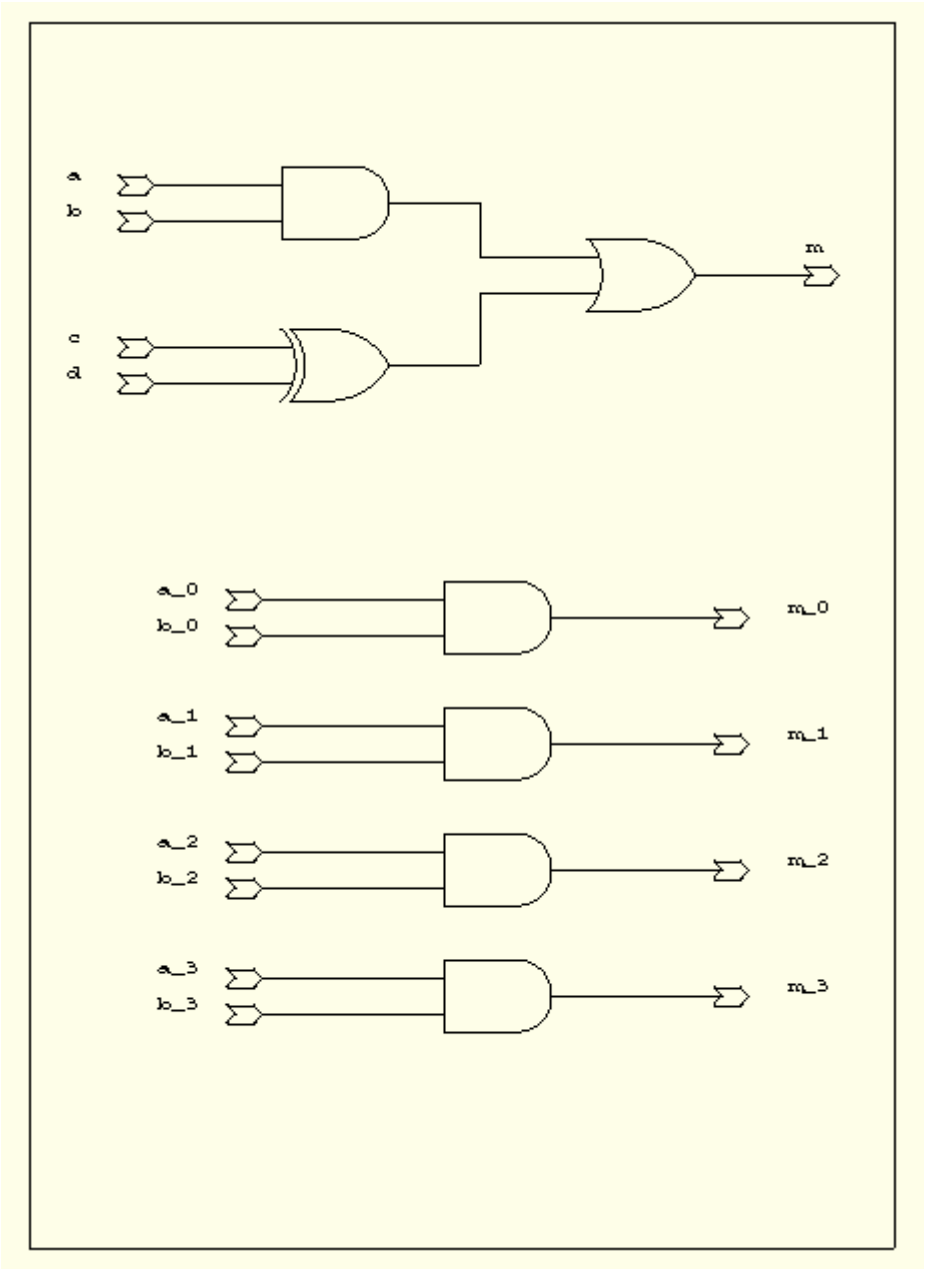
- and
- or
- nand
- nor
- xor
- xnor
- not

These operators are defined for the types bit, boolean and arrays of bit or boolean (for example, bit_vector). The compilation of logic is fairly direct from the language construct, to its implementation in gates, as shown in the following examples:

```
entity logical_ops_1 is
   port (a, b, c, d: in bit;  m: out bit);
end logical_ops_1;


architecture example of logical_ops_1 is
   signal e: bit;
```

```
begin
   m <= (a and b) or e; --concurrent signal assignments
   e <= c xor d;
end example;

entity logical_ops_2 is
   port (a, b: in bit_vector (0 to 3);  m: out bit_vector (0 to 3));
end logical_ops_2

architecture example of logical_ops_2 is
begin
   m <= a and b;
end example;
```

Corresponding Schematic Representation



Relational Operators

VHDL provides the following relational operators:

= Equal to

/= Not equal to

> Greater than

< Less than

>= Greater than or equal to

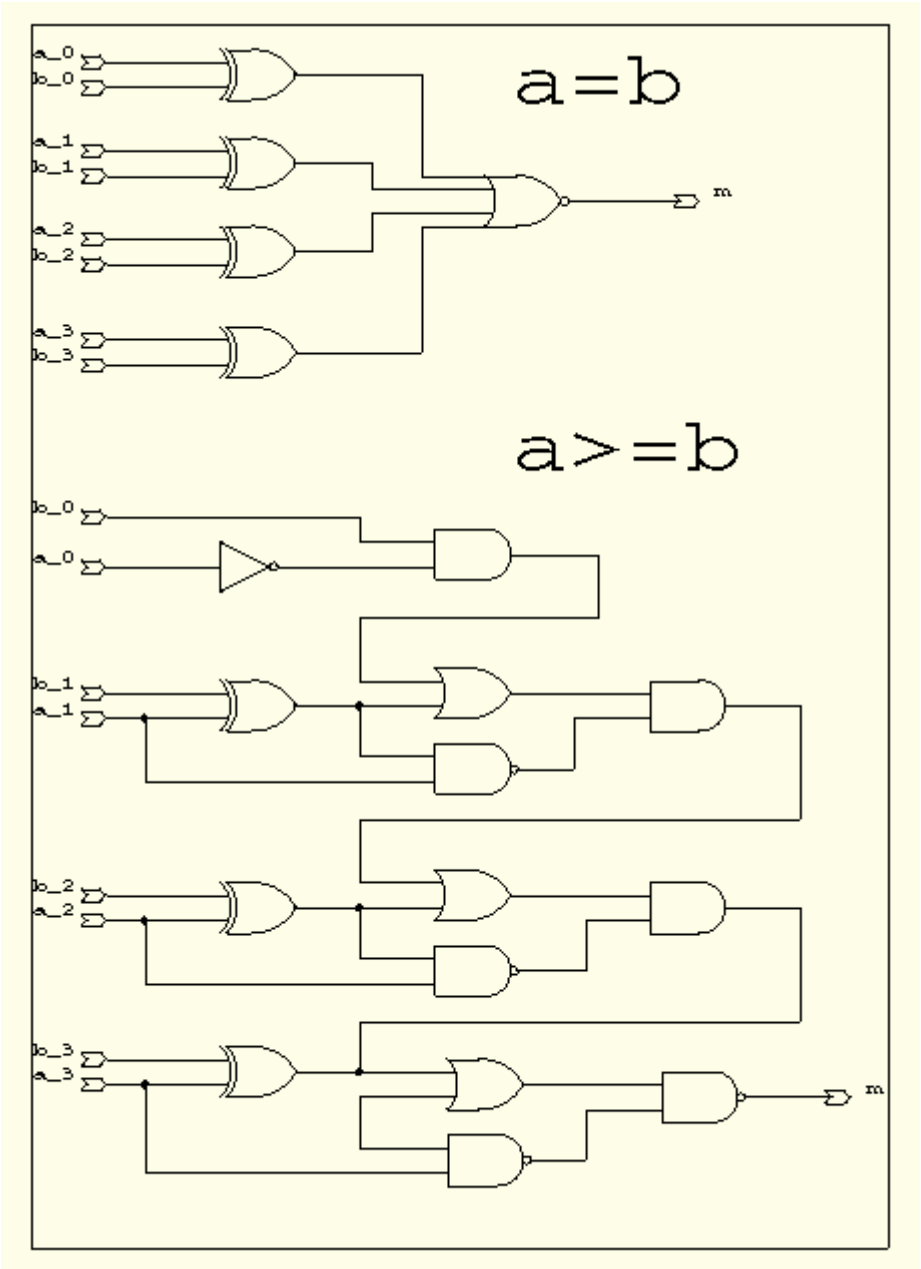
<= Less than or equal to

The equality operators (= and /=) are defined for all types. The ordering operators (>=, <=, >, <) are defined for numeric types, enumerated types, and some arrays. The resulting type for all these operators is boolean.

The simple comparisons, equal and not equal, are cheaper to implement (in terms of gates) than the ordering operators. To illustrate, the first example below uses an equal operator and the second uses a greater-than-or-equal-to operator. As you can see from the schematic, the second example uses more than twice as many gates as the first.

```
entity relational_ops_1 is
   port (a, b: in bit_vector (0 to 3);  m: out boolean);
end relational_ops_1;

architecture example of relational_ops_1 is
begin
   m <= a = b;
end example;

entity relational_ops_2 is
   port (a, b: in integer range 0 to 3;  m: out boolean);
end relational_ops_2;

architecture example of relational_ops_2 is
begin
   m <= a >= b
end example;
```

Corresponding Schematic Representation



Arithmetic Operators

The arithmetic operators in VHDL are defined for numeric types. These are:

+ Addition

- Subtraction

* Multiplication

/ Division

mod Modulus

rem Remainder

abs Absolute Value

** Exponentiation

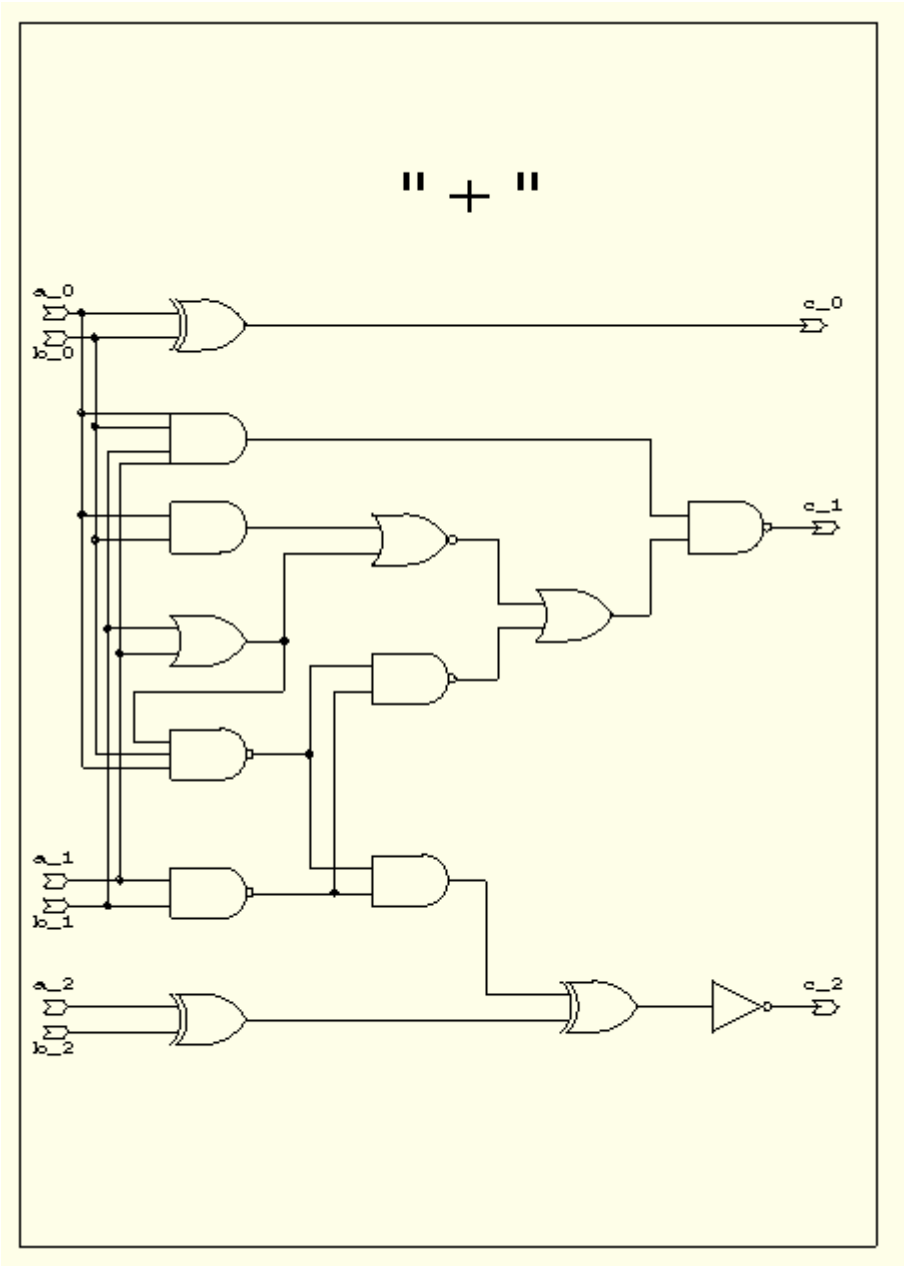
While the adding operators (+, -) are fairly expensive in terms of gates, the multiplying operators (*, /, mod, rem) are very expensive. The Synthesizer does make special optimizations, however, when the right hand operator is a constant and an even power of 2.

The absolute (abs) operator is inexpensive to implement. The ** operator is only supported when its arguments are constants.

The following example illustrates the logic due to an addition operator (and the use of package and type declaration):

```
package example_arithmetic is
   type small_int is range 0 to 7;
end example_arithmetic;

use work.example_arithmetic.all;

entity arithmetic is
   port (a, b: in small_int;  m: out small_int);
end arithmetic;

architecture example of arithmetic is
begin
   m <= a + b;
end example;
```

Corresponding Schematic Representation



Control Statements

VHDL provides the following concurrent statements for creating conditional logic:

- conditional signal assignment
- selected signal assignment

VHDL provides the following sequential statements for creating conditional logic:

- if
- case

Examples of concurrent control statements are conditional signal assignments:

```
entity control_stmts is
   port (a, b, c: boolean;  m: out boolean);
end control_stmts;

architecture example of control_stmts is
begin
   m <= b when a else  c;
end example;
```

All possible cases must be used for selected signal assignments. You can be certain of this by using an **others** case:

```
entity control_stmts is
   port (sel: bit_vector (0 to 1); a,b,c,d : bit;  m: out bit);
end control_stmts;

architecture example of control_stmts is
begin
   with sel select
      m <= c when b"00",
      m <= d when b"01",
      m <= a when b"10",
      m <= b when others;
end example;
```

The same functions can be implemented using sequential statements and occur inside a process statement. The condition in an **if** statement must evaluate to true or false (that is, it must be a boolean type).

The following example illustrates the **if** statement:

```
entity control_stmts is
```

```
   port (a, b, c: boolean;  m: out boolean);
end control_stmts;

architecture example of control_stmts is
begin
   process (a, b, c)
      variable n: boolean;
   begin
      if a then
         n := b;
      else
         n := c;
      end if;
      m <= n;
   end process;
end example;
```

Using a **case** statement (or selected signal assignment) will generally compile faster and produce logic with less propagation delay than using nested **if** statements (or a large selected signal assignment). The same is true in any programming language, but may be more significant in the context of logic synthesis.

If statements and selected signal assignments are also used to infer registers.

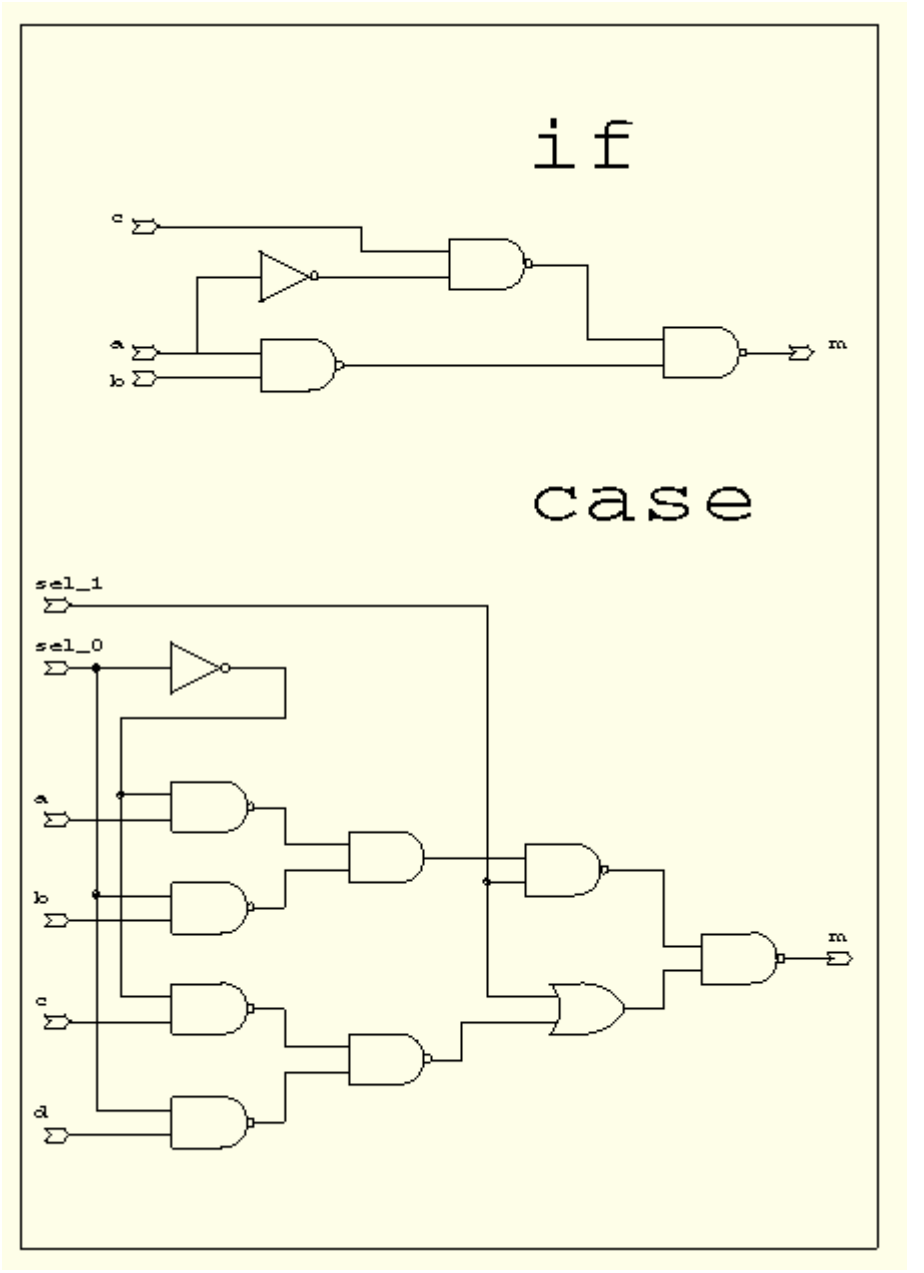
VHDL requires that all the possible conditions be represented in the condition of a **case** statement. To ensure this, use the **others** clause at the end of a **case** statement to cover any unspecified conditions.

The following example illustrates the **case** statement:

```
entity control_stmts is
   port (sel: bit_vector (0 to 1);  a,b,c,d : bit;  m: out bit);
end control_stmts;

architecture example of control_stmts is
begin
   process (sel,a,b,c,d)
   begin
      case sel is
         when b"00"  => m <= c;
         when b"01"  => m <= d;
         when b"10"  => m <= a;
         when others => m <= b;
```

```
      end case;
   end process;
end example;
```

Corresponding Schematic Representation



Subprograms and Loops

VHDL provides the following constructs for creating replicated logic:

- generate
- loop
- for loop
- while loop
- function
- procedure

Functions and procedures are collectively referred to as subprograms. Generate is a concurrent loop statement. These constructs are synthesized to produce logic that is replicated once for each subprogram call, and once for each iteration of a loop.

If possible, **for loop** and **generate** ranges should be expressed as constants. Otherwise, the logic inside the loop may be replicated for all the possible values of loop ranges. This can be very expensive in terms of gates.

```
entity loop_stmt is
   port (a: bit_vector (0 to 3);  m: out bit_vector (0 to 3));
end loop_stmt;


architecture example of loop_stmt is
begin
   process (a)
      variable b:bit;
   begin
      b := '1';
      for i in 0 to 3 loop --don't need to declare i
         b := a(3-i) and b;
         m(i) <= b;
      end loop;
   end process;
end example;
```

A loop statement replicates logic, therefore, it must be possible to evaluate the number of iterations of a loop at compile time. This requirement adds a constraint for the synthesis of a **while loop** and an unconstrained **loop** (but not a **for loop**). These loops must be completed by a statement whose execution depends only upon metalogic expressions. If, for example, a loop completion depends on a signal (i.e. not a metalogic expression), an infinite loop will result.

Placing a **report** statement within the loop is a useful technique for debugging. A message will be reported to the screen at each iteration of the loop.

```
entity loop_stmt is
   port (a: bit_vector (0 to 3);  m: out bit_vector (0 to 3));
end loop_stmt;

architecture example of loop_stmt is
begin
   process (a)
      variable b: integer;
   begin
      b := 1;
      while b < 7 loop
         report "Loop number = " &  integer'image(b);
         b := b + 1;
      end loop;
   end process;
end example;
```

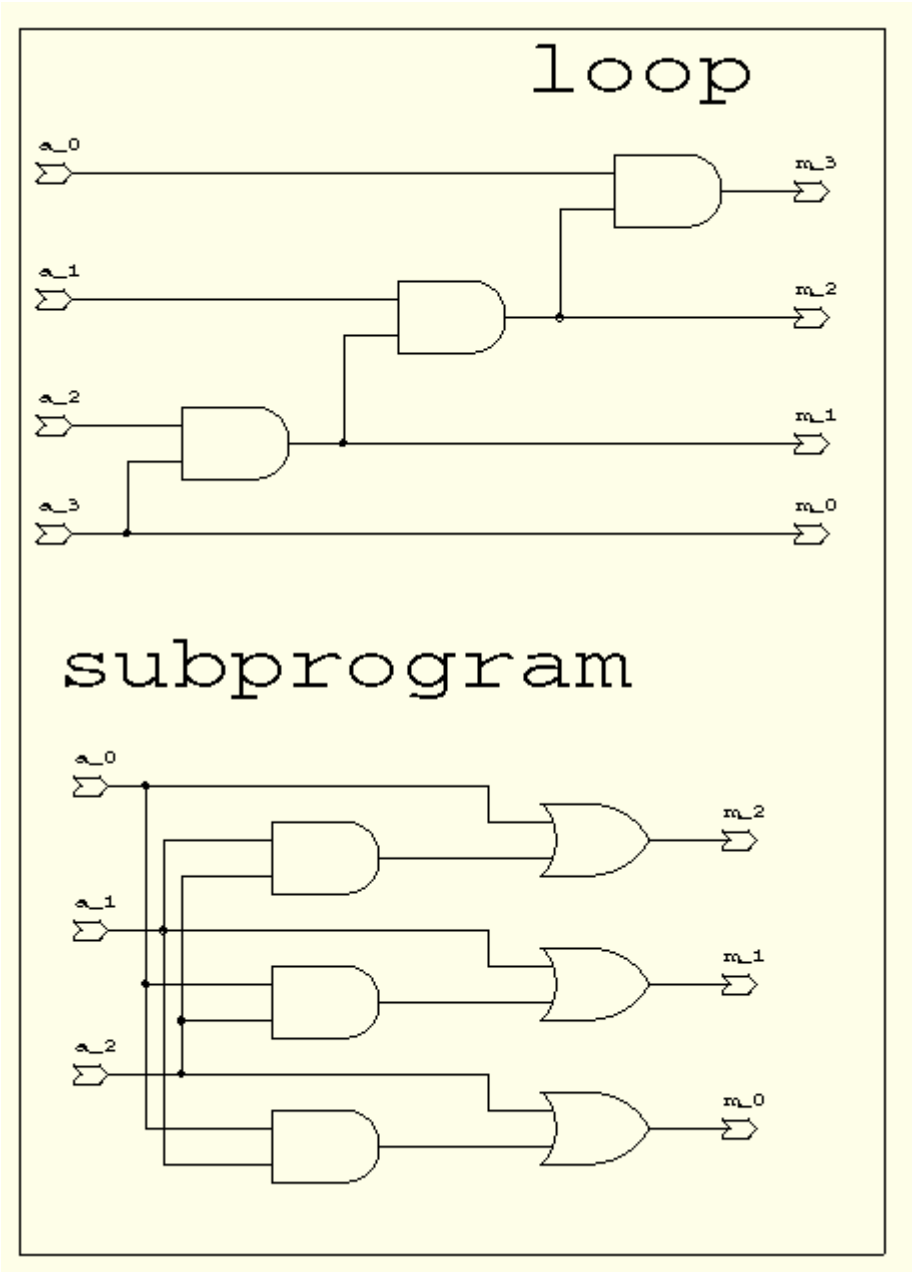
Loop statements may be terminated with an **exit** statement, and specific iterations of the loop statement terminated with a **next** statement. When simulating, an **exit** or **next** may be used to speed up simulation time. For synthesis, where each loop iteration replicates logic, there is probably no speedup. In addition, the **exit** or **next** may synthesize logic that gates the following loop logic. This may result in a carry-chain-like structure with a long propagation delay in the resulting hardware.

A **function** is always terminated by a **return** statement, which returns a value. A **return** statement may also be used in a **procedure**, but it never returns a value.

```
entity subprograms is
   port (a: bit_vector (0 to 2);  m: out bit_vector (0 to 2));
end subprograms;

architecture example of subprograms is
   function simple (w, x, y: bit) return bit is
   begin
      return (w and x) or y;
   end;
begin
   process (a)
   begin
      m(0) <= simple(a(0), a(1), a(2));
      m(1) <= simple(a(2), a(0), a(1));
```

```
      m(2) <= simple(a(1), a(2), a(0));
   end process;
end example;
```

Corresponding Schematic Representation



Shift and Rotate Operators

VHDL provides the following shift and rotate operators:

- sll
- srl
- sla
- sra
- rol
- ror

The left operand may be a one dimensional array whose element type is either BIT or BOOLEAN, and the right operand is of type INTEGER. If the right operand is a constant (or a metalogic expression), these operations imply no logic.

```
entity sr_1 is
   port (a,b,c: in bit_vector (5 downto 0 );
         ctl : integer range 0 to 2**5 -1;
         w,x,y: out bit_vector (5 downto 0);
end sr_1;

architecture example of sr_1 is
begin
   w <= a sll ctl;  -- shift left , fill with '0'
   x <= a sra ctl;  -- shift right, fill with a'left [ a(5) ]
   y <= a rol ctl;  -- rotate left
end example;
```

Note that a negative right argument means a shift or rotate in the opposite direction. If the right argument is non constant (not metalogic expression), and has a subtype which has a range that includes a negative number, a bidirectional shift or rotate structure will be constructed. This can be very expensive. For example :

```
function to_natural ( arg : bit_vector) return natural;
function to_integer ( arg : bit_vector) return integer;

a sll to_natural (bv);
a sll to_integer (bv);  ------ EXPENSIVE
```

Tri-states

There are two possible methods to describe tristates: either using the 'Z' in the type std_logic defined in ieee.std_logic_1164, or using an assignment of NULL to turn off a driver. The first method applies to the type std_logic only, the second method applies to any type. The first method is the one commonly used.

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

entity tbuf2 is
   port (enable : boolean;
      a : std_logic_vector(0 to 4);
      m : out std_logic_vector(0 to 4));
end tbuf2;

architecture example of tbuf2 is
begin
   process (enable, a)
      if enable then
         m <= a;
      else
         m <= 'Z';
      end if;
   end process;
end;
```

or the equivalent concurrent statement :

```
architecture example2 of tbuf2 is
begin
   m <= a when enable else 'Z';
end;
```

An internal tristate bus may be described as in the following example. Pullups may be connected to an internal tristate bus by instantiating a pullup component.

```
architecture example3 of tbuf2 is
begin
   m <= a0 when enable0 else 'Z';
   m <= a1 when enable1 else 'Z';
```

```
   m <= a2 when enable2 else 'Z';
end;
```

The assignment of **null** to a **signal** of kind **bus** turns off its driver. When embedded in an **if** statement, a **null** assignment is synthesized to a tristate buffer.

```
package example_bus is
   subtype bundle is bit_vector (0 to 4);
end example_bus;

use work.example_bus.all;
entity tbuf is
   port (enable: boolean; a: bundle; m: out bundle bus);
end tbuf;

architecture example of tbuf is
begin
   process (enable, a)
   begin
      if enable then
         m <= a;
      else
         m <= null;
      end if;
   end process;
end example;
```

Programming Sequential Logic

Programming sequential logic in VHDL is like programming in a conventional programming language, and unlike programming using a traditional PLD programming language. There is no register assignment operator, and no special attributes for specifying clock, reset, etc. In VHDL you program the behavior of a sequential logic element, such as a latch or flip-flop, as well as the behavior of more complex sequential machines.

This section shows how to program simple sequential elements such as latches and flip-flops in VHDL. This is extended to add the behavior of Set and Reset (synchronous and asynchronous).

Sequential Logic Behavior

The behavior of a sequential logic element can be described using a **process** statement (or the equivalent **procedure** call, or concurrent signal assignment statement). The behavior of a sequential logic element (latch or flip-flop) is to save a value of a signal over time. This section shows how such behavior may be programmed.

The techniques shown here may be extended to specify Set and Reset, both synchronous and asynchronous, as shown in later sections. There are often several ways to describe a particular behavior, the following examples typically show two styles each, however, there is no particular 'right' style. The choice of style is simply that which helps the programmer specify the clearest description of the particular design.

For example, the designer may choose to copy the procedures for latches and flip-flops from the following examples, and describe a design in terms of equations and procedure calls. Alternatively the designer may choose to describe a design in a more behavioral form.

There are three major methods to program this register behavior: using conditional specification , using a wait statement, or using guarded blocks. Conditional specification is the most common method.

Conditional Specification

This relies on the behavior of an **IF** statement, and assigning in only one condition:

```
if clk then
   y <= a;
else
         -- do nothing
end if;
```

This describes the behavior of a latch, if the clock is high the output (y) gets a new value, if not the output retains its previous value. Note that if assignment had been made in both conditions, the behavior would be that of a mux:

```
if clk then
   y <= a;
else
   y <= b;
end if;
```

The key to specification of a latch or flip-flop is incomplete assignment using the **IF** statement; there is no particular significance to any signal names used in the code fragments. Note, however, that incomplete assignment is within the context of the whole **process** statement.

The latch could be described as transparent low:

```
if clk then
        -- do nothing
else
   y <= a;
end if;
```

Or more concisely:

```
ifnot clk then
   y <= a;
end if;
```

A rising edge Flip-flop is created by making the latch edge sensitive:

```
if clk and clk'event then
   y <= a;
end if;
```

In all these cases the number of registers or the width of the mux are determined by the type of the signal "y".

Wait Statement

The second method uses a **wait** statement:

```
wait until expression;
```

This suspends evaluation (over time) until the expression evaluates to "true". A flip-flip may be programmed:

```
wait until clk
y <= a;
```

It is not possible to describe latches using a **wait** statement.

Guarded Blocks

The guard expression on a **block** statement can be used to specify a latch.:

```
lab : block (clk)
begin
   q <= guarded d;
end block;
```

It is not possible to describe flip-flops using guarded blocks.

Latches

The following examples describe a level sensitive latch with an **and** function connected to its input. In all these cases the **signal** "y" retains it's current value unless the clock is true:

```
-- A Process statement :
process (clk, a, b) -- a list of all signals used in the process
begin
   if clk then
      y <= a and b;
   end if;
end process;

-- A Procedure declaration, creates a latch
-- when used as a concurrent procedure call
procedure my_latch (signal clk, a, b : boolean; signal y : out boolean)
begin
   if clk then
      y <= a and b;
   end if;
end;

-- an example of two such calls:
label_1: my_latch ( clock, input1, input2, outputA );
label_2: my_latch ( clock, input1, input2, outputB  );

-- A concurrent conditional signal assignment,
-- note that "y" is both used and driven
y <= a and b when clk else y;
y <= a and b when clk;
```

Flip-Flops

The following examples describe an edge sensitive flip-flop with an **and** function connected to its input. In all these cases the **signal** "y" retains it's current value unless the clock is changing :

```
-- A Process statement :
process (clk)   -- a list of all signals that result in propagation delay
begin
   if clk and clk'event then   -- clock rising
      y <= a and b;
   end if;
end process;

-- A Process statement containing a wait statement:
process     -- No list of all signals used in the process
begin
   wait until not clk;   -- clock falling
   y <= a and b;
end process;

-- A Procedure declaration, this creates a flip-flop
-- when used as a concurrent procedure call
procedure my_ff (signal clk, a, b :  boolean; signal y : out boolean)
begin
   if not  clk  and clk'event then   -- clock falling
      y <= a and b;
   end if;
end;
-- A concurrent conditional signal assignment,
-- note that "y" is both used and driven
y <= a and b when clk and clk 'event else y;
y <= a and b when clk and clk 'event ;   -- the last else is not required
```

It is sometimes clearer to write a function to detect a rising edge :

```
function rising_edge (signal s : bit ) return boolean is
begin
   return s = '1' and s'event;
end;
```

Using this function, a D flip flop can be written as :

```
q <= d when rising_edge(clk);
```

Gated Clocks and Clock Enable

The examples in this section assume the clock is a simple signal. In principle, any complex boolean expression could be used to specify clocking. However, the use of a complex expression implies a gated clock.

As with any kind of hardware design, there is a risk that gated clocks may cause glitches in the register clock, and hence produce unreliable hardware. You need to be aware of the constraints of the target hardware and, as a general rule, use only simple logic in the **if** expression.

It is possible to specify a gated clock with a statement such as:

```
if clk1 and ena then
      -- register assignments here
end if;
```

which implies a logical AND in the clock line.

To specify a clock enable use nested **if** statements:

```
if clk1 then
   if ena then
      -- register assignments here
   end if;
end if;
```

This will connect 'ena' to the register clock enable if the clock enable compile option is used. If the clock enable option is not used then the data path will be gated with 'ena'. In neither case will 'ena' gate the 'clk1' line.

Synchronous Set or Reset

To add the behavior of synchronous set or reset, simply add a conditional assignment of a constant immediately inside the clock specification.

```
-- Set:
process (clk)
begin
   if clk and clk'event then -- clock rising
      if set then
         y <= true; -- y is type boolean
      else
         y <= a and b;
      end if;
```

```
   end if;
end process;

-- 29 bits reset , 3 bits set by init
process
begin
   wait until clk -- clock rising
   if init then
      y <= 7; -- y is type integer
   else
      y <= a + b;
   end if;
end process;
```

Asynchronous Set or Reset

To describe the behavior of asynchronous set or reset the initialization is no longer within the control of the clock. Simply add a conditional assignment of a constant immediately outside the clock specification.

```
-- Reset using a concurrent statement statement:
y <= false when reset else a when clk and clk'event else y;

-- and using the function rising_edge described earlier :
y <= false when reset else a when rising_edge(clk);

-- Reset using sequential statements:
process (clk, reset)
begin
   if reset then
      q <= false;  -- y is type boolean
   else
      if clk and clk'event then -- clock rising
         q <= d;
      end if;
   end if;
end process;
```

```
procedure ff_async_set (signal clk, a, set:  boolean;
            signal q : out boolean)
begin
   if set then
      q <= true;
   elsif clk and clk'event then -- clock rising
      q <= a; -- D input
   end if;
end;
```

Asynchronous Set and Reset

To describe the behavior of both asynchronous set and reset, simply add a second conditional assignment of a constant immediately outside the clock specification.

```
-- Reset and Set using a concurrent statement
q <= false when reset else
   true when preset else
      d when clk and clk'event;

-- Reset and Set using a sequential statements
process (clk, reset, preset)
begin
   if reset then
      q <= false; -- q is type boolean
   elsif preset then
      q <= true
   else
      if clk and clk'event then -- clock rising
         q <= d;
      end if;
   end if;
end process;
```

Asynchronous Load

To describe the behavior of asynchronous load, replace the constant used for set or reset with a signal or an expression. Asynchronous load is actually implemented using both flip flop asynchronous preset and flip flop asynchronous reset.

```
-- Load using a concurrent statement
q <= load_data when load_ctl = '1' else d when rising_edge(clk);


-- Load using a sequential statement
process (clk, load_ctl,load_data)
begin
   if load_ctl = '1' then
      q <= load_data;
   elsif rising_edge(clk) then
      q <= d;
   end if;
end process;
```

Register Inference Rules

Storage elements are inferred by the use of the **if** statement. Register control signals are specified with the expression in an **if** statement, the control signal function is specified by the assignments (or lack of assignments) in the branches of the **if** statement.

```
if if expression then
   then branch
else
   else branch
end if;
```

Multiple nested **if** statements are combined to specify multiple register control signals. The execution of the first **if** statement may not be conditional on any other statements, unless the condition is a metalogic expression.

The scope of register inference is a single concurrent statement.

Reset/Preset

One branch of the **if** statement assigns a constant (metalogic expression) to the register. The other branch assigns input to the register.

Clock

One branch of the **if** statement assigns to the register, the other branch does not assign to the register (or assigns the register output). A register is inferred because its value is incompletely specified.

Clock Enable

One branch of the **if** statement assigns to the register input in the clock expression, the other branch does not assign to the register. Must occur immediately within the clock **if** statement.

Inference priority

Control signals are inferred with the following priority, listed with the highest priority first (not all combinations are supported) :

- Asynchronous reset / preset
- Clock
- Clock Enable
- Synchronous reset

Programming Finite State Machines

Finite state machines (FSMs) can be classified as Moore or Mealy machines. In a Moore machine, the output is a function of the current state only; thus can change only on a clock edge. Whereas a Mealy machine output is a function of the current state and current inputs, and may change when any input changes.

This section shows the relationship between these machines and VHDL code. Each example illustrates a single machine. This is not a constraint, just a simplification. If there were multiple machines, they could have different clocks. In this case, synchronization would be the responsibility of the designer.

Feedback Mechanisms

There are two ways to create feedback - using signals and using variables. With the addition of feedback you can build state machines.

It is possible to describe both combinational and sequential (registered) feedback systems. When using combinational feedback to create asynchronous state machines it is often helpful, but not required, to mark the feedback signal with the Synthesizer user attribute **critical**.

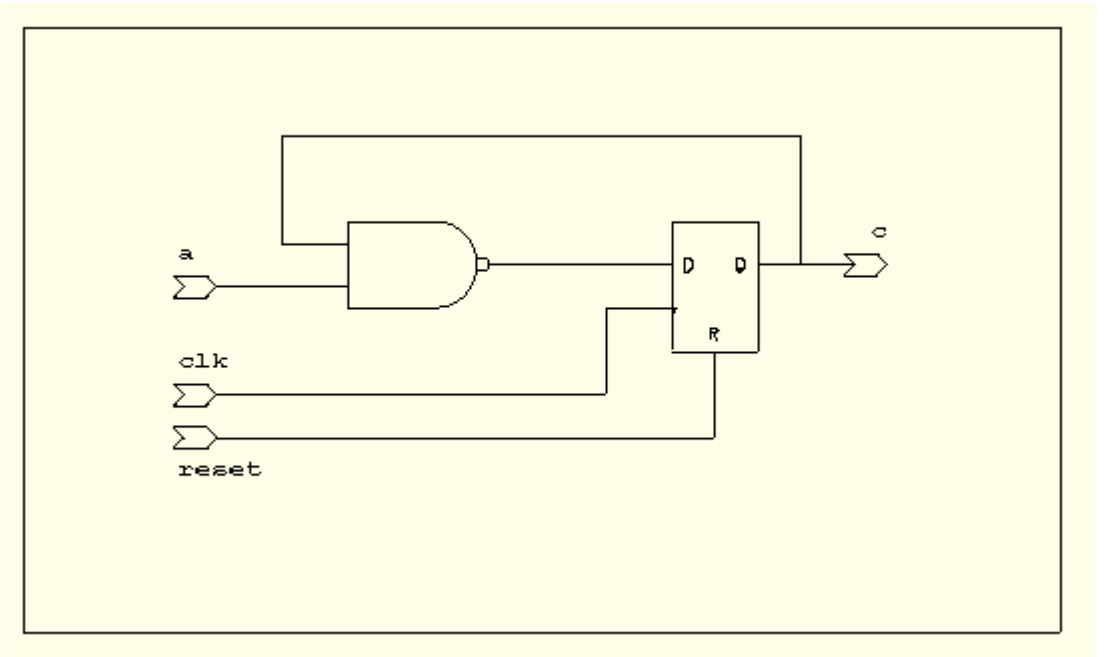
Feedback on signals

```
architecture example of some_entity is
   signal b: bit;
   function rising_edge (signal s : bit ) return boolean is
   begin
      return s = '1' and s'event;
   end;
begin
   process (clk, reset)
   begin
      if reset = '1' then
         c <= '0';
      elsif  rising_edge(clk)
         c <= b;
      end if;
   end process;

   process (a, c) -- a combinational process
   begin
      b <= a and c;
   end process;
end example;
```

A more concise version of the same feedback is shown in the following example:

```
use work.my_functions.all; -- package containing
                           -- user function rising_edge
architecture example of some_entity is
begin
   process(clk,reset)
   begin
      if reset = '1' then
         c <= '0';
      elsif rising_edge(clk)
         c <= a and c;
      end if;
   end process;
end example;
```

Corresponding Schematic Representation



Feedback on variables

Variables exist within a process, and processes suspend and reactivate. If a variable passes a value from the end of a process back to the beginning, feedback is implied. In other words, feedback is created when variables are used (placed on the right hand side of an expression, in an **if** statement, etc.) before they are assigned (placed on the left hand side of an expression).

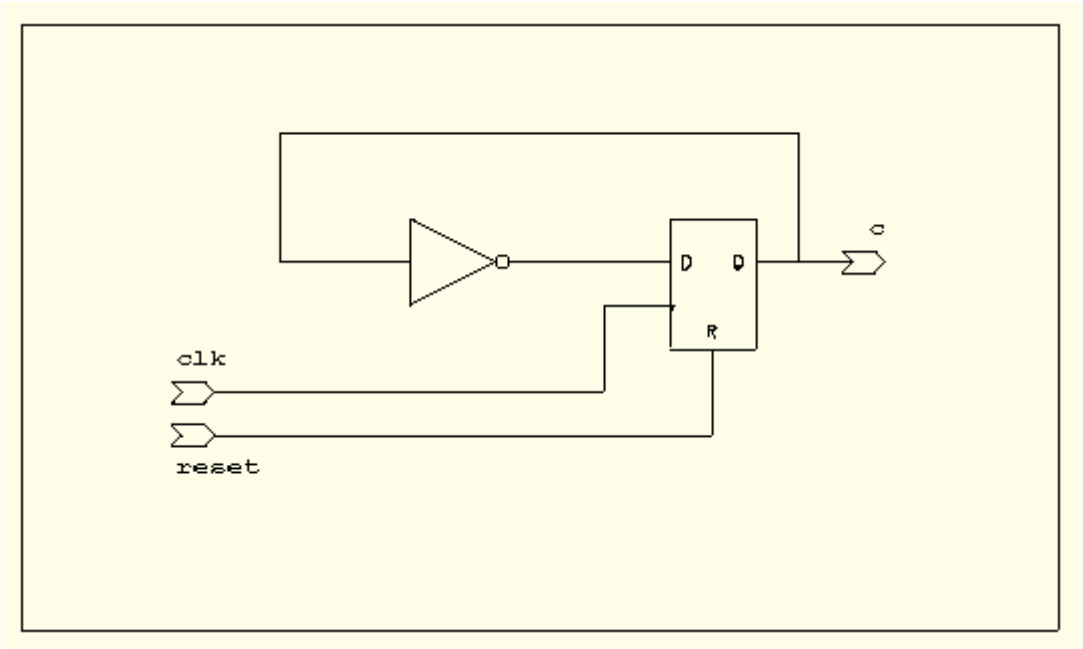
Feedback paths must contain registers, so you need to insert a **wait** statement.

```
process
   variable v: bit;
begin
   wait until clk = '1';
   if reset = '1' then
      v <= '0';
   else
      v := not v; --v is used before it is assigned
      c <= v;
   end if;
end process;
```

A flip-flop is inserted in the feedback path because of the **wait** statement. This also specifies registered output on signal **a**.

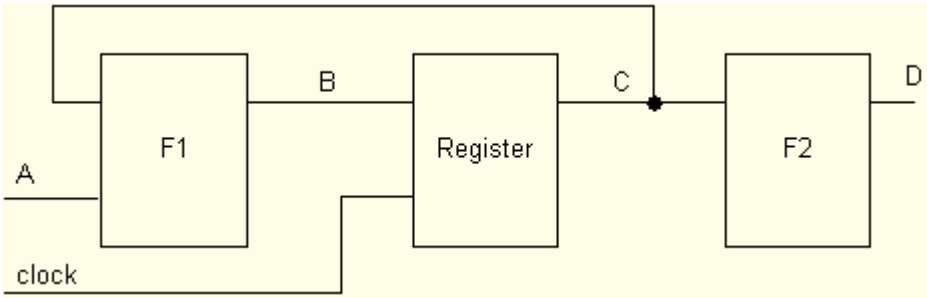
If a variable is declared inside a function or procedure, the variable exists only within the scope of the subprogram. Since a **wait** statement can only be placed within a **process** statement (a Synthesizer constraint), variables inside subprograms never persist over time and never specify registers.

Corresponding Schematic Representation



Moore Machine

In the following architecture, F1 and F2 are combinational logic functions. A simple implementation maps each block to a VHDL process.




```
entity system is
   port (clock: boolean; a: some_type;.d: out some_type);
end system;

architecture moore1 of system is
   signal b, c: some_type;
begin
```

```
   process (a, c)
   begin
      b <= F1(a, c);
   end process;
   process (c)
   begin
      d <= F2(c);
   end process;
   process
   begin
      wait until clock;
      c <= b;
   end process;
end moore1;
```

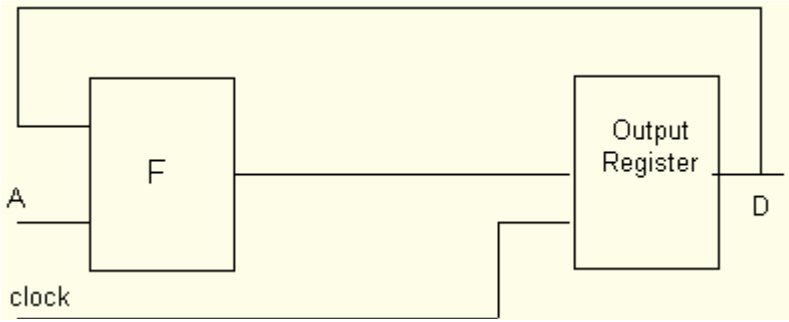
A more compact description of this architecture could be written as follows:

```
architecture moore2 of system is
   signal c: some_type;
begin
   process (c) -- combinational logic
   begin
      d <= F2(c);
   end process;
   process     -- sequential logic
   begin
      wait until clock;
      c <= F1(a, c);
   end process;
end moore2;
```

In fact, a Moore Machine can often be specified in one process.

Output registers

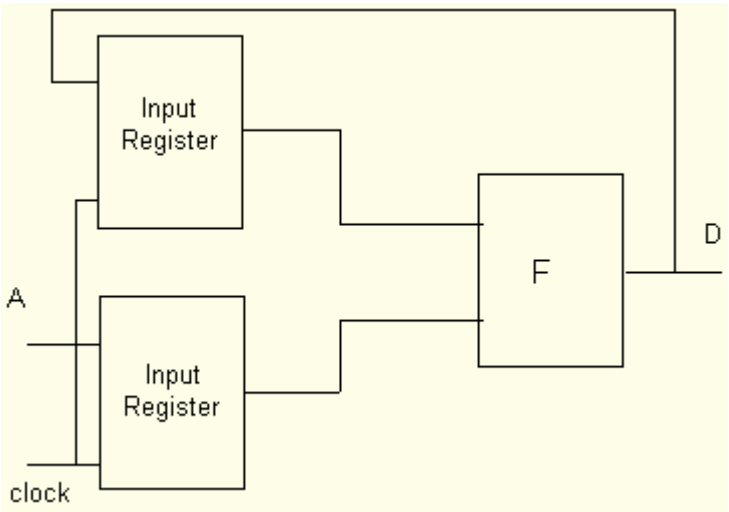
If the system timing requires no logic between the registers and the output (the shortest output propagation delay is desired), the following architecture could be used:




```
architecture moore3 of system is
begin
   process
   begin
      wait until clock;
      d <= F(a,d)
   end process;
end moore3;
```

Input Registers

If the system timing requires no logic between the registers and the input (if a short setup time is desired), the following architecture could be used:

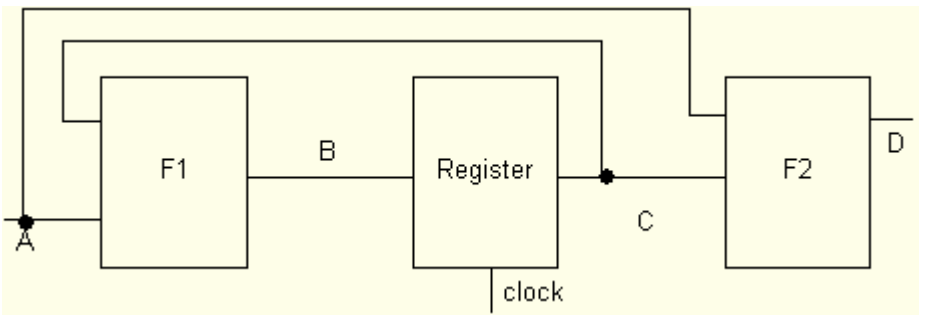



```
architecture moore4 of system is
   signal a1, d1 : some_type;
begin
   process
```

```
   begin
      wait until clock;
      a1 <= a;
      d1 <= d;
   end process;
   process (a1, d1)
   begin
      d <= F(a1,d1)
   end process;
end moore4;
```

Mealy Machine

A Mealy Machine always requires two processes, since its timing is a function of both the clock and data inputs.




```
architecture mealy of system is
   signal c: some_type;
begin
   process (a, c) -- combinational logic
   begin
      d <= F2(a, c);
   end process;
   process        -- sequential logic
   begin
      wait until clock;
      c <= F1(a, c);
   end process;
end mealy;
```

Synthesis of VHDL Types

In VHDL, types are used for type-checking and overload resolution. For logic design, each type declaration also defines the encoding and number of wires to be produced. For subtypes, checking and overloading use the base type of the subtype.

Each subtype declaration defines a subset of its type and can specify the number or wires, and possibly the encoding scheme.

During compilation by the Synthesizer, ports with types that compile to multiple wires are renamed by appending "_n", where n is an incremented integer starting from zero.

Enumerated Types

As a default, enumerated types use binary encoding. Elements are assigned numeric values from left to right, and the value of the leftmost element is zero.

The number of wires will be the smallest possible n, where number of elements <= 2**n.

The type **bit** is synthesized to one wire.

The type **character** is synthesized to eight wires.

Don't Cares

Unused encodings are implicitly compiled as "don't care" conditions; these allow the Synthesizer to perform additional logic optimizations. Subtypes use the element encodings of their base, and types define additional "don't care" conditions. Don’t care may be explicitly specified using 'enum_encoding' as described in the next section.

For example:

The declaration	Is synthesized as
type direction **is** (left, right, up, down);	Two wires
type cpu_op **is** (execute, load, store);	Two wires; the encoding of 11 is a "don't care"
subtype mem_op **is** cpu_op **range** load **to** store;	Two wires; the encodings of 00 and 11 are "don't cares"

In the example below, logic will be generated with inputs 11 and 00 as "don’t care" conditions for evaluating **output_var**.

```
variable operation: mem_op;
...
case operation is
   load  => output_var :=...;
   store => output_var :=...;
end case;
```

User Defined Encoding

Users may redefine the encoding of an enumerated type using the attribute 'enum_encoding'. For example, cpu_op might be redefined with one hot encoding:

```
attribute enum_encoding of enum_t : type is "001 010 100";
                    -- or  ... : type is "one hot";
                    -- or  ... : type is "1-hot";
```

or kept as two bits with a different encoding:

```
attribute enum_encoding of enum_t : type is "01 10 11";
```

The definition of the encoding may contain a string consisting of '0' '1' 'Z' 'M' or '-', delimited by ' ' (white space). The encoding of each enumerated element must have the same number of characters. Each encoding should be unique. The encoding 'Z' represents a high impedance, the encoding '-' represents a don't care, and the encoding 'M' represents a metalogic value. In addition there are two predefined encodings "one hot" and "gray".

Users must be aware that the enum_encoding attribute allows the user to redefine the semantics of an enumerated type. In certain cases this may results in synthesis creating logic that does not have the same behavior as the original VHDL source! In general, this is not a big problem; it is, however, a pitfall to be aware of, as explained below.

Enumerated types in programming languages are defined as having unique and ascending values. In order to maintain behavior the enum_encoding specified by the user should be unique and ascending. Non-unique encoding should be avoided. For non-ascending encoding, the user must overload the ordering operators < <= > >= for the re-encoded type of each ordering operator used.

Std_logic_1164

The library 'ieee' contains the package 'std_logic_1164'; this in turn declares an enumerated type 'std_ulogic':

```
type std_ulogic is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

This type and its derivatives 'std_logic' and 'std_logic_vector' are often used in VHDL simulation. This allows the user to maintain information about the simulation model itself as well as describe the design. The values 'U' 'X' 'W' and '-' are referred to as metalogical values because they represent the state of a model rather than the logic of a design.

An object of type std_logic is encoded as one wire because the library ieee (supplied with the Synthesizer) contains the encoding definition:

```
attribute enum_encoding of std_ulogic : type is  "M M 0 1 Z M 0 1 -";
```

The attribute defines the semantics for each element:

'0' 'L'	Logic value 0
'1' 'H'	Logic value 1
'Z'	Logic value tristate
'U' 'X' 'W'	Metalogic value
'-'	Don't care value

The 'U' 'X' 'W' and '-' values have the same synthesis semantics -- except as arguments to the IEEE Standard 1076.3 function STD_MATCH. The semantics are defined in 1076.3 and allow don't care logic optimization if evaluation results in assigning a metalogic value or don't care value.

These semantics are designed for compatibility with simulation; if an 'X' propagates in simulation, there may be don't care optimization. Note that some operations don't propagate unknowns:

- "=" with one metalogic argument is always false.
- "/=" with one metalogic argument is always true.
- an ordering operator with a metalogic argument is illegal.
- a case choice containing metalogic is always ignored.

The function ieee.numeric_std.std_match provides wildcard matching for the don't care value.

One Hot Encoding

User defined encoding may be used to specify one hot encoding. For instance, the enumerated type 'state_type' (below) could be redefined as one hot simply by changing the enum_encoding attribute.

```
type state_type is (st0,st1,st2,st3,st4,st5,st6,st7,st8,
                    st9,st10,st11,st12,st13,st14,st15);
```

There are two possible forms:

```
-- either
attribute enum_encoding of state_type : type is "one hot";
-- or
attribute enum_encoding of state_type : type is
      "0000000000000001 " &   --st0
      "0000000000000010 " &   --st1
      "0000000000000100 " &
      "0000000000001000 " &
      "0000000000010000 " &
```

```
      "0000000000100000 " &
      "0000000001000000 " &
      "0000000010000000 " &
      "0000000100000000 " &
      "0000001000000000 " &
      "0000010000000000 " &
      "0000100000000000 " &
      "0001000000000000 " &
      "0010000000000000 " &
      "0100000000000000 " &
      "1000000000000000";     --st15
```

The encoding is specified in ascending order so the ordering operators ("<" "<=" ">" ">=") function as expected, and so writing additional functions to define these operations is not needed. Don't care conditions are handled automatically and transparently to the user.

An alternative method to describe one hot encoding is to use arrays of 'std_logic' (or even 'bit'). This method may be slower to compile and require additional explicit don't care specification. The recommended style is to use enumerated types and enum_encoding.

Gray Encoding

User defined encoding may be used to specify a predefined gray encoding. For example:

```
type state_type is (st0,st1,st2,st3,st4,st5,st6,st7,st8,
                st9,st10,st11,st12,st13,st14,st15);
attribute enum_encoding of state_type : type is "gray";


-- this is the same as :
attribute enum_encoding of state_type : type is
      "0000 " &    --st0
      "0001 " &    --st1
      "0011 " &    --st2
      "0010 " &    --st3
      "0110 " &
      "0111 " &
      "0101 " &
      "0100 " &
      "1100 " &
      "1101 " &
      "1111 " &
```

```
"1110 " &
"1010 " &
"1011 " &
"1001 " &
"1000";        --st15
```

Because of the nature of a gray code, the encoding is NOT specified in ascending order so the ordering operators ("<" "<=" ">" ">=") do NOT function as expected. For example the expression st2 < st3 should be true, but the encoded expression “0011” < “0010” is false. So writing additional functions to define ordering operators for state_type is required if these operations are used.

Numeric Types

Numeric types consist of integer, floating point, and physical types. Two encoding schemes are used by the Synthesizer for numeric types:

- Numeric types and subtypes that contain a negative number in their range definition are encoded as 2's complement numbers.
- Numeric types and subtypes that contain only positive numbers are encoded as binary numbers.

The number of wires that are synthesized depends on the value in its subtype declaration that has the largest magnitude. The smallest magnitude is assumed to be zero for numeric types.

Floating point numbers are constrained to have the same set of possible values as integers - even though they can be represented using floating point format with a positive exponent.

Numeric types and subtypes are synthesized as follows:

The declaration	**Is synthesized as**
type int0 **is range** 0 **to** 100	A binary encoding having 7 bits
type int1 **is range** 10 **to** 100	A binary encoding having 7 bits
type int2 **is range** -1 **to** 100	A 2's complement encoding having 8 bits (including sign)
subtype int3 **is** int2 **range** 0 **to** 7	A binary encoding having 3 bits

`Warning` - take great care when using signed scalar numbers. These are encoded as twos-complement, which is a fixed width encoding.

This can be a problem when mixing objects that have different signed subtypes - each will have different widths and result in unexpected behavior. This is not a problem during simulation since these objects are always encoded as a fixed , 32 bit, width.

It is probably safest to use unsigned scalar types. Another option is to use an array of bits to explicitly specify the width; this is the approach taken by the Synopsys and IEEE 1076.3 synthesis package.

If the type of the object to which the result is assigned has more bits than either of the operands, the result of the numeric operations is automatically sign extended or zero extended. Sequential encoded types are zero extended , and two's compliment numbers are sign extended.

If the type of the object to which the result is assigned has fewer bits than either of the operands, the result of the numeric operations is truncated.

If a numeric operation has a result that is larger than either of the operands, the new size is evaluated before the above rules are applied.

For example, a "+" generates a carry that will be truncated , used , or sign (or zero) extended, according to the type of the object to which the result is assigned.

```
type short is integer 0 to 255;
subtype shorter is short range 0 to 31;
subtype shortest is short range 0 to 15;

signal op1,op2,res1 : shortest;
signal res2 : shorter;
signal res3 : short
begin
   res1 <= op1 + op2;   -- truncate carry
   res2 <= op1 + op2;   -- use carry
   res3 <= op1 + op2;   -- use carry and zero extend
```

Note that if shorter had been declared as:

```
subtype shorter is short range 0 to 16;
```

the encoding of integers rounded up to the nearest power of two would have the same result.

Arrays and Records

Composite types (arrays and records) are treated as collections of their elements. Subtypes of composite types are treated as collections of the elements of the subtype only.

Managing Large Designs

Many of the VHDL design descriptions in this guide consist of a single entity (the design I/O) and its architecture (the design functionality). This view is sufficient for many users, but as your designs get larger you will also want to consider the issues of partitioning and design management.

This section introduces some additional VHDL constructs for partitioning and sharing code modules. These are **block**, **component**, **package**, and **library** statements. Of these, only **component** has special meaning in the context of synthesis, so you can refer to any of the standard VHDL texts for detailed descriptions.

Using Hierarchy

A VHDL entity can have multiple architectures. A particular entity/architecture pair (referred to as a design entity) can also be referenced from another architecture as a VHDL **component**. Instantiating components within another design provides a mechanism for integrating partitioned designs or for using other designs in the current design. You can manage the relationship between a component declaration and various design entities by using **configuration** specifications. Because of default configurations, such specifications are not required.

During synthesis, a **component** is also used to tell the Logic Optimizer about the hierarchy of your design. Using components in a large design will result in a design that optimizes faster and produces more efficient results. This is because using components adds the designer's knowledge of the hierarchy of a design to the description, this in turn is used by the compiler to specify the domain of the Logic Optimizer. Hierarchy is also useful in the debugging of large designs, in reusing design units. For VHDL synthesis there are some additional semantics of hierarchy. It is used to specify logic optimize granularity, hierarchical compiles, and silicon specific components.

Controlling the logic optimize granularity

The domain of the Synthesizer's Logic Optimizer is an architecture, which is the amount of logic the Optimizer will optimize at one time. Using hierarchy to reflect the structure of your design will allow efficient use of the Optimizer.

Specifying an architecture containing a large amount of logic may take a long time to optimize. Optimizing many small architectures can be quick but may not give satisfactory results if the Optimizer doesn't see enough of the design at one time.

There is no right answer, but keep in mind:

- An architecture should typically contain logic that synthesizes between 500 and 5000 gates.
- There is no lower bound, an architecture could simply specify connectivity and imply no gates.
- You can cause a Child to be optimized as part of each of its Parents by applying the synthesis attribute "ungroup" to the component declaration.

Hierarchical compile

It is not necessary (and possibly not even a good idea)to compile a whole design in one pass. Large designs are commonly compiled in multiple partitions and the resulting netlists linked together. Since you can compile more than one entity/architecture in one pass, compile granularity is distinct from optimize granularity.

From the point of view of the VHDL code, you don't have to do anything special for hierarchical compile (although there are some constraints imposed by netlist semantics). Simply compile the parent without the Child :

```
---Parent
library ieee;
use ieee.std_logic_1164.all;
entity Parent is
   port (a : std_logic_vector(7 downto 5);
         v : out std_logic_vector( 1 to 3));
end Parent;
architecture behavior of Parent is
   -- component declaration , unbound
   component Child
      port (I : std_logic_vector(2 downto 0) ;
            O : out std_logic_vector(0 to 2));
   end component;
begin
   -- component instantiation
   u0 : Child port map (a,v);
end;
```

This results in a netlist containing an instance of Child but no definition of Child. The Child entity is then compiled and the resulting netlist linked by a downstream tool. In practice many cases are possible, a Parent may have a Child_1 defined and compiled at this time, and Child_2 compiled at a different time.

Silicon specific components

It is also possible that the Child is never defined to the synthesis tool, but defined by a downstream tool. You can use this to specify primitives in the target hardware. The primitives could be as simple as I/O buffers, or clock buffers. They might also be a pre-defined component such as a special counter, or an XBLOX or LPM macrocell. For simulation using third party tools prior to simulation you may need simulation models of such components. These models should not be visible to the synthesis compiler.

```
---Parent
library ieee;
use ieee.std_logic_1164.all;
entity Parent is
   port (a : std_logic_vector(7 downto 5);
         v : out std_logic_vector( 1 to 3));
end Parent;
architecture behavior of Parent is
```

```
   -- component declaration , unbound
   component IN_BUF_3
      port (I : std_logic_vector(2 downto 0) ;
            O : out std_logic_vector(0 to 2));
   end component;
   component OUT_BUF_3
      port (I : std_logic_vector(2 downto 0) ;
            O : out std_logic_vector(0 to 2));
   end component;
   signal x : std_logic_vector(2 downto 0);
   --you may need to add this attribute , see the text below.
   attribute macrocell : Boolean;
   attribute macrocell of IN_BUF_3, OUT_BUF_3 : component is true;
begin
   -- component instantiations
   u0 : IN_BUF_3 port map (a,x);
   u2 : OUT_BUF_3 port map (x,v);
end;
```

If the ports of your component have a type corresponding to multiple bits, you should add the macrocell **attribute** as shown. Adding the attribute is not required if the component has only single bit ports (such as those with type std_logic). The macrocell **attribute** changes the naming conventions for expanding component bus names.

Blocks

Designs can be partitioned using **block** statements or **component** statements. These constructs have the same meaning as blocks and components in schematic capture.

Block statements can be used to partition concurrent statements, as in the following example:

```
architecture partitioned of some_design is
begin
   a_block: block
      begin
      -- concurrent statements here
      end block;
   another: block
      begin
      -- concurrent statements here
      end block;
end partitioned;
```

Direct Instantiation

Each element of the design hierarchy (each entity architecture combination) may be directly instantiated within another. For example :

```
-- The design leaf
entity child is
   port (a, b: bit; c : out bit);
end child;

architecture behavior of child is
begin
   c <= a and b;
end behavior;

-- The design root
entity parent is
   port (a, b: bit; c: out bit);
end parent;

architecture family of parent is
   signal w, r: bit;
   use work.all;
begin
   huey: entity child port map (a, b, w); --direct instantiations
   luey: entity child port map (a, w, r);
   duey: entity child port map (a, r, c);
end family;
```

Components and Configurations

VHDL allows any number of **entity**-**architecture** pairs, which are referred to as design entities. These design entities can be referenced from another architecture as **components**. The mapping of design entities is managed using a **configuration** specification, which associates particular component instances with a specified design entity.

The first example contains three **component** instantiations:

```
-- The component definition
entity goose is
   port (a, b: bit; c : out bit);
end goose;
```

```
architecture snow_goose of goose is
begin
   c <= a and b;
end snow_goose;

-- The design definition
entity flock is
   port (a, b: bit; c: out bit);
end flock;

architecture three_geese of flock is
   signal w, r: bit;
   component goose --component declaration
      port (a, b: bit; c: out bit);
   end component;
begin
   one: goose port map (a, b, w); --component instantiations
   two: goose port map (a, w, r);
   three: goose port map (a, r, c);
end three_geese;
```

In this example, the **architecture** three_geese contains a declaration of a **component** goose and three instantiations of that component, but no definition of the component's configuration. By default, VHDL uses an **entity** of the same name as the component (in this case goose), which is defined at the beginning of the design.

You can override the default component definition by using a **configuration** specification. For example, a configuration specification could have been used to describe another architecture of entity flock, as follows:

```
architecture three_birds of flock is
   signal w, r: bit;
   component bird --component declaration
      port (a, b: bit;  c: out bit);
   end component;
   for all: bird use entity work.goose; --configuration specification
begin
   one: bird port map (a, b, w); --component instantiations
   two: bird port map (a, w, r);
   three: bird port map (a, r, c);
end three_birds;
```

In a configuration specification, instantiation labels (in this example, "one," "two," and "three") can be used instead of the reserved word **all** to indicate that the configuration applies to particular instances of the specified component. Configurations have many other capabilities that are described in the standard VHDL texts.

If a design contains multiple design entities, you need to specify which one is used as the root (top level) of the design. The default is the last entity analyzed. You can override this default by using the elaborate compile option.

Package Declarations and Use Clauses

The **package** declaration can be used to declare common types and subprograms. For example:

```
package example_package is
   type shared_enum is (first, second, third, last);
end example_package;
```

In order for the contents of a package to be visible from inside an entity or an architecture, you need to place a **use** clause before the entity declaration. For example:

```
use work.example_package.all;
entity design_io is
   ...
end design_io;
```

Placing a **use** clause before an entity causes the contents of the specified package to be visible to that entity and its architecture(s), but nowhere else.

The **work** library is the default name of the current library. For now, just treat it as template and always include it in the **use** clause.

Since the VHDL visibility rules ignore file boundaries, the package might be in one file, the use clause and entity declaration in another, and the architecture in a third file. VHDL requires that these units have already been analyzed when they are referenced in the code, therefore the order in which the files are specified to the compiler is important. It is not required that design units be placed in different files.

To define common subprograms, a **package body** is used. For information on this construct, and other applications of the use clause, refer to the standard VHDL texts.

VHDL Design Libraries

In VHDL, a design library is defined as "an implementation-dependent storage facility for previously analyzed design units". The library "work" is a special case, as it is an alias for the current library.

A library is simply an external VHDL file or files, so files specified directly to the compiler are in the library "work". Files specified through a vhdl library statement (by direct association or alias association) are contained in the "work" library Some files stored in the installation directory are also saved as pre-analyzed binary ".mm0" files,

Libraries are made visible within the source code by the **library** statement. To make the library units within the library visible outside the library, it is necessary to add **use** statements:

```
library stuff;
use stuff.all;    -- Makes visible all design units in stuff.
use useless.all; -- Makes all declarations in the design unit
                 named useless visible.
```

or enter the following statement for each design unit:

```
use stuff.useless.all;
```

Direct association

A library is defined as a file of the same name. The library statement above will cause the Synthesizer to read a file named "stuff.vhd". The compiler searches for the file in the current directory, then in the installation directory. An eight-character limit is imposed on library names by some versions of the DOS operating system.

Synthesizer VHDL Libraries

The library files supplied with the Synthesizer contain the following packages :

STD.VHD	IEEE 1076 package 'standard'
IEEE.VHD	IEEE 1164 package 'std_logic_1164'
NUM_BIT.MM0	IEEE 1076.3 package 'numeric_bit'
NUM_STD.MM0	IEEE 1076.3 package 'numeric_std'
METAMOR.VHD	Synthesis specific package 'attributes' Synthesis specific package 'array_arith'
VLBIT.VHD	Viewlogic package 'pack1076'
SYNOPSYS.VHD	Synopsys package 'std_logic_arith' Synopsys package 'std_logic_unsigned' Synopsys package 'std_logic_signed' Synopsys package 'std_logic_misc'
XBLOX.VHD	package 'macros'
LPM.VHD	package 'macros200' package 'macros201'

Documentation for these packages is included within the VHDL source files, short descriptions follow. The XBLOX and LPM libraries may only be used in association with XBLOX or LPM compilers, and may not be included in your version of the product.

std.standard

The VHDL 1076 package, declares bit, bit_vector, boolean, etc.

ieee.std_logic_1164

The IEEE standard 1164 package, declares std_logic, std_logic_vector, rising_edge(), etc.

ieee.numeric_bit

This package is part of the IEEE 1076.3 Draft Standard VHDL Synthesis Package. The package is supplied in binary compiled form. The source code is available from the IEEE as part of the Standard.

This package defines numeric types and arithmetic functions for use with synthesis tools. Two numeric types are defined:

- UNSIGNED: represents an UNSIGNED number in vector form
- SIGNED: represents a SIGNED number in vector form

The base element type is type BIT. The leftmost bit is treated as the most significant bit. Signed vectors are represented in two's complement form. This package contains overloaded arithmetic operators on the SIGNED and UNSIGNED types. The package also contains useful type conversions functions, clock detection functions, and other utility functions.

This package is in the binary file num_bit.mm0. To use this package the library alias for IEEE should be set to num_bit.vhd. (IEEE <path>\num_bit.vhd)

See the topic 'VHDL Design Libraries' (in the Links section below) for information on alias association.

ieee.numeric_std

This package is part of IEEE 1076.3 Draft Standard VHDL Synthesis Package. The package is supplied in binary compiled form. The source code is available from the IEEE as part of the Standard.

This package defines numeric types and arithmetic functions for use with synthesis tools. Two numeric types are defined:

- UNSIGNED: represents an UNSIGNED number in vector form
- SIGNED: represents a SIGNED number in vector form

The base element type is type STD_LOGIC. The leftmost bit is treated as the most significant bit. Signed vectors are represented in two's complement form. This package contains overloaded arithmetic operators on the SIGNED and UNSIGNED types. The package also contains useful type conversions functions.

This package is in the binary file num_std.mm0, the package depends upon IEEE.STD_LOGIC_1164. To use this package the library alias for IEEE should be set to include ieee.vhd and num_std.vhd. (IEEE <path>\ieee.vhd <path>\num_std.vhd)

metamor.attributes

Declarations of the synthesis specific attributes.

metamor.array_arith

This package contains subprograms that allow arithmetic operations on arrays for optimizing third party synthesis packages. These functions are intended to be hidden from the end user within other functions contained in a third party package. There would be two implementations of the package body, one optimized for synthesis (uses these functions), and the other optimized for simulation.

The documentation with the file describes the list of assumptions and example usage. More examples of the use of these functions can be found in vlbit.vhd and synopsys.vhd.

vlbit.pack1076

This package contains type and subprogram declarations for Viewlogic's built-in type conversion and bus resolution functions. The package has been optimized for use with the Synthesizer's compiler. Vlbit based designs may (or may not) require some modification; this is described below.

Vlbit designs may make use of register inference conventions that are different from those used by the Synthesizer. The case to look for is preset/reset, which is specified in a wait statement along with the clock. Using the Synthesizer, this will result in a gated clock, which is probably not what you want. You should replace the wait statement with the if-then style of register inference.

You should validate using simulation and also check to see that the number of registers used and their type (flip-flop/latch, preset/reset, sync/async) are what you expected. The compiler reports register types, and number of instances in the log file.

ieee.std_logic_arith, ieee.std_logic_unsigned, ieee.std_logic_signed, ieee.std_logic_misc

These packages are versions of the Synopsys packages that have been optimized for use with the VHDL Synthesizer's compiler. When importing designs you should validate using simulation and also check the number of registers used and their type (flip-flop/latch, preset/reset, sync/async) to ensure they are what you expected. The compiler reports register types, and number of instances in the log file.

These packages are in the file synopsys.vhd and are usually placed in a library named IEEE (although they are not an IEEE standard). To use these packages the library alias for IEEE should be set to include ieee.vhd and synopsys.vhd.

xblox.macros

This package contains component declarations for Xblox macrocells, for use with the Xblox compiler. These components may be instantiated in your design in the usual way. For example:

```
u1 : compare port map (d1,d2, a_ne_b => x);
```

The package is based on ieee.std_logic_1164.std_logic. If you wish to use datatypes other than std_logic, then create your own package by copying from this one. There are no hidden magic words, except that the port and generic names must match the Xblox specification. All components that are Xblox macrocells must have the synthesis attribute 'macrocell' set to 'true'.

lpm.macros200, lpm.macros201

This package contains component declarations for Lpm macrocells, for use with an LPM compiler. These components may be instantiated in your design in the usual way. For example:

```
u1 : lpm_compare generic map (4,"unsigned")
              port map (d1,d2, aeb => x);
```

The package is based on ieee.std_logic_1164.std_logic. If you wish to use datatypes other than std_logic, then create your own package by copying from this one. There are no hidden magic words, except that the port and generic names must match the LPM specification. All components that are LPM macrocells must have the synthesis attribute 'macrocell' set to 'true'.

LPM requires instance specific Properties. These are specified by using VHDL generics. The component declarations include these generic declarations. Instance specific values are specified with a generic map. Some examples are :

```
signal d1 : std_logic_vector(3 downto 0)
signal d2 : std_logic_vector(0 to 3)
signal d3,d4 : std_logic_vector(7 downto 6)
....
u1 : lpm_compare generic map (4)      --default is "signed"
               port map (d1,d2, aeb => x);
u2 : lpm_compare generic map (2,"unsigned")
               port map (d3,d4, y1, y2);  --agb not used
u3 : lpm_compare generic map
               (representation =>"unsigned", width => 2 )
               port map (d3,d4, z);      -- alb is used
```

Hierarchical Compilation

The whole design need not be recompiled when only a single **architecture** changes. The VHDL Synthesizer supports this feature through hierarchical compilation. The granularity of hierarchical compilation is the **component**.

This feature requires that the user maintain and link the resulting elements of the hierarchy (components) external to the Synthesizer. The user is also responsible for checking the root and leaf interfaces for consistentcy. This feature is only available with output formats that support hierarchy.

If a **component** has no **entity** visible when the design root is compiled, no entity is bound to that component. This results in a hierarchy instantiation in the output file with no definition for that leaf of the hierarchy. The leaf entity that was not visible during the first compilation is generated by a second compilation using the Synthesizer.

Because the binding between root and leaf is external to the VHDL compiler (the user links these together) certain VHDL features are not available at the hierarchical compilation boundary. The user is responsible to ensure that component and entity port definitions match exactly. Some things to watch out for include:

- Leaf entity and component names must be the same.
- Leaf entity and component port names and subtypes must be the same.
- Leaf instance may not have a 'generic map'.
- Leaf may not have a port that has a type that is unconstrained.
- Ports that have an array type must have matching directions in the entity and component declaration.
- Leaf component declaration may not contain a port map (the component instantiation may still contain a port map)
- Root and leaf must not reference a signal declared outside of their scope (e.g. a signal declared in a package).
- Configurations are not supported at (or across !!) the hierarchical compilation boundary.

Logic and Metalogic

A HDL design description consists of code to serve three distinct functions.

Logic expressions - logic in the hardware implementation. The value of a logic expression changes over time. In VHDL terms its value depends upon a signal.

Metalogic expressions - logic about (not in) the hardware implementation. The value of a metalogic expression does not change over time. In VHDL terms its value must not depend upon a signal.

Metalogic values - logic value extensions for tools such as simulators or synthesis tools. Metalogic values describe the state of the design model.

Metalogic expressions are important in synthesis as they imply no hardware. This allows them to compile faster, and generally produces more efficient synthesis results. In addition, some constraints on VHDL for synthesis depend upon certain expressions being metalogic expressions (i.e., they must not vary over time).

Metalogic values are tool specific values (specific to simulators or synthesis tools) added to the design description. An understanding of the required values may be important when porting VHDL code from say a simulator to a synthesis tool(in addition to the additional constraints of EE design !).

In a classic PLD programming language, design description consists of logic expressions, constant metalogic expressions, and perhaps 'X' (mapped to 0 or don't care) as a metalogic value.

Logic expressions

Logic expressions are familiar to hardware engineers, any classic PLD programming language consists of logic expressions. In VHDL examples of logic expressions might be :

```
(a and b) or c
d + e
```

If a,b,c,d, and e are **signals**.

Metalogic expression

An example of a simple metalogic expression is one using constants. In VHDL examples might be:

```
('0' and '1') or '1'
 e + f
```

If e and f are constants, generics, generates, loop iterators or, in VHDL speak, are static, then the expression is a static expression and also metalogical. Metalogic expressions may also contain variables. More on this later in this section.

A more useful example of a metalogic expression might be the loop expression :

```
for i in 4 to 9 loop
   left(i) <= right(i+2);
end loop;
```

The expression i+2 implies no logic. It is a metalogic expression, used (and the loop statement) to specify information about the design, which does not appear in the implementation. The result is more

concise, and the relationship between the arrays left and right is more clear. Of course, five distinct assignments would produce the same result.

An expression containing a variable will be metalogical if the variable's value depends only on a metalogic expression. Metalogical Variables are very powerful, but it is only possible to tell if they are metalogical from the context. An expression is said to be a metalogic expression if it is a static expression, a metalogic expression may in addition contain variables whose values depend only upon metalogic expressions.

A larger example of metalogic might be the following function, which converts a bit vector to an integer. The logic generated may be different at each function call, depending upon the argument passed at each call.

```
constant too_long_msg : STRING :
              =  "Array too long to be integer.";
constant too_short_msg : STRING :
              = "Null array passed to subprogram.";
function to_integer ( arg : BIT_VECTOR ) return INTEGER is
   variable result : INTEGER := 0;
   variable w : INTEGER := 1;
begin
   -- Report null range
   assert arg'length > 0 report too_short_msg severity NOTE;
   -- Assert array size limit.
   assert arg'length < 32 report too_long__msg;

   -- Calculate bit_vector value.
   for i in arg'reverse_range loop
      if arg (i) = '1' then
         result := result + w;
      end if;
      -- test before multiplying w by 2, to avoid overflow
         if i /= arg'left then
            w := w + w;
         end if;
   end loop;
   return result;
end to_integer;
```

Reviewing this function you can see that the variable 'w' depends only on the initial value (w: integer := 1;) and the current value of 'w' (w := w + w). It can be said that 'w' is always a metalogical variable and the assignments to 'w' imply no logic.

The variable 'result' depends on the initial value of 'result' (metalogic), the value of 'w' (metalogic), and 'arg', which depends on the argument the function is called with. If the function is called with a metalogic parameter, say :

```
to_integer("010101");
```

then arg is a constant, and hence metalogic. It also follows that 'result' is metalogic. The function implies no logic, just pull up and pull down. However, if the function were called with a logical parameter, arg would not be metalogic, so hardware is implied. For example:

```
to_integer(some_signal);
```

In this case the algorithm implemented is such that the hardware is simply wires. (hint: a binary representation of 'w' is always a single 1 and many 0s).

Variables declared in subprograms allow metalogic expressions. The same is true of variables declared in a process. However, variables in a process usually depend on the sensitivity list of a wait statement (statement and list may be explicit or implied). Therefore, they are usually not metalogical. In simulation terms, variables in a process persist over time. Variables in a subprogram are created when the subprogram is called and destroyed when it returns (like the difference between static variables and automatic variables in C).

Metalogic values

Metalogic values are extensions added to the design description. They provide additional information for tools to allow the tools to produce better results. Two examples are unknowns (X) for simulation and don't care (-) for logic optimization. These metalogic values are added as alternatives to logic values (0,1) within the tools. These metalogic values may have different meanings to different tools.

Unknowns allows you to detect design description errors during simulation. Errors such as unconnected inputs or connected outputs (try writing boolean equations for these !) clearly do not describe logic. Unknowns due to uninitialized registers (but not unknowns injected due to timing errors) also highlight boolean logic errors. As long as a simulation propagates such metalogic you know that the design description does not represent logic.

Don't care works around one of the limitations of a boolean representation, allowing logic minimizers and technology mappers to produce more compact description. A high level language provides a more elegant solution, in which the user never has to consider don't cares. This alternative is to describe the design using multi-valued enumerated types in place of arrays of booleans.

An understanding of metalogic values is significant because the output of a synthesis tool is boolean logic (0,1); therefore, the metalogic values are removed (and possibly used) during synthesis. This is significant if the operation of a design depends upon metalogic values. A design that depends on some signal having a value X has two possible implementations: the signal is either 0 or 1 (but never X).

Within VHDL, the only common use of metalogic values is some of the elements of the enumerated type std_ulogic :

```
std_ulogic : type is ('U','X','0','1','Z','W','L','H','-');
```

The IEEE standard 1076.3 specifies that four of these values ('U' 'X' 'W' '-') are metalogic values, with specific semantics. However, to a simulator they are just elements of an enumerated types. For

synthesis, the attribute 'enum_encoding' is used to describe which elements describe logic values and which describe metalogic values. The Synthesizer follows the standard and considers '0' , '1', 'Z'. 'L' and 'H' as logic values and the remainder as metalogic values. The metalogic values may be used within the Synthesizer's logic minimization.

When using std_logic, the metalogic values 'U' 'X' 'W' and '-' have one meaning to a simulation tool and another (don't care) to a synthesis tool. Within the Synthesizer, metalogic values are not simply thrown away, but are treated in expressions as don't cares as specified by enum_encoding. Signals do not propagate metalogic values, only '0' '1' and possibly 'Z'.

The use of metalogical values is one possible difference between a simulation model and a hardware design. For example, with one metalogic argument an equality operation will always return false in synthesis, but in simulation the result will depend upon the current value of the other argument; therefore, unknown handling may be used for simulation and ignored for synthesis:

```
assert not isome_signal = 'X' report "unknown, bad news" severity error;
```

The function 'is_x' from 'ieee.std_logic_1164' may be used as a run time synthesis or simulation flag. This function will always return false within synthesis, and its result depends upon the current value during simulation.

```
if is_x('W') then
   assert false report "simulation code" severity note;
else
   assert false report "synthesis code" severity note;
end if;
```

WARNING: Such tricks may impair your validation methodology.

Macrocells

Complex logic cells such as add or compare may be inferred from VHDL expressions or instantiated as VHDL statements. Instantiated silicon specific logic cells of a fixed size using components is discussed in the topic 'Using Hierarchy' under the section 'Silicon specific components' (see the Related Topics section below).

Instantiating parameterized components for macrocell compilers is described in the related topics. Examples of macrocell compilers are LPM compilers and the Logiblox compiler.

If the formal port declarations are unconstrained, or generics are used, the macrocell becomes a parameterized macrocell. Parameterized macrocells are only supported for the EDIF and Open Abel 2 output formats.

The compiler reports instantiated parameterized macrocells :

```
component : u1 : Parameterized Macrocell "compare"
```

In addition macrocells may be automatically inferred by the compiler. Whether inferred or instantiated, macrocells usually give better synthesis results in terms of both area and delay; compilation is usually faster too. The log file will contain the names of macrocells inferred for each process.

Parameterized Macrocell Instantiation

If no entity is bound to a component and the formal port declarations are unconstrained, or generics are used, the macrocell becomes a parameterized macrocell. Parameterized macrocells are only supported for the EDIF and Open Abel 2 output formats. Parameterized macrocells are implemented by silicon specific macrocell compilers in downstream tools, invocation of these compilers is usually automatic. Macrocell compilers may require specific properties added to the component or instance (see documentation for the compiler), these may be added with a generic map or with a VHDL attribute.

The compiler reports instantiated parameterized macrocells as:

```
component : u1 : Parameterized Macrocell "compare"
```

For example, the Compare macrocell from the Xblox library is declared with unconstrained ports and a style parameter:

```
component compare
   generic (style : string := "");
   port (a, b: std_logic_vector;
         a_eq_b, a_ne_b, a_lt_b, a_gt_b, a_le_b, a_ge_b :
                                 out std_logic);
end component;
attribute macrocell of compare : component is true;
```

The macrocell may be instantiated with input ports whose size varies with each instantiation. The parameter style may be specified or left as the LPM or Logiblox default. And, in the usual VHDL manner, named association may be used to pick from the out ports. For example:

```
U1: compare port map (a_byte, b_byte, a_eq_b => eql );
U2: compare port map (a_byte, b_byte, a_eq_b => eql ,
             a_ge_b => bigger);
U3: compare generic map ("RIPPLE")
        port map (a_word, b_word, a_le_b =>lss);
```

Combinatorial Macrocell Inference

Inference occurs transparently to the user when the output format supports parameterized macrocells. Inference maps VHDL relational and arithmetic operators to format specific macrocells. For example, the multiply operation below will results in a multiply macrocell in the LPM format, and a set of adder macrocells in the Logiblox format.

```
p <= a * b;
```

The relational operations map to the Compare macrocell. The following two concurrent statements are equivalent :

```
neq <= a_nibble /= b_nibble;
U1: compare port map (a_nibble, b_nibble, a_ne_b => neq);
```

Macrocell inference only occurs if both operands are VHDL signals (or more formally are not metalogic expressions). So for example, adding two VHDL constants will not produce an adder macrocell.

Sequential Macrocell Inference

If a process contains both inferred flip flops and an inferred combinational macrocell, the compiler can infer a sequential macrocell. An example is a counter with reset described using a concurrent statement.

```
count  <= 0 when reset = '1' else count +1 when
                rising_edge(clock);
```

Sequential macrocells often have a synchronous load control, which may be specified using an if statement. Load inference has the lowest priority of all register control inference. For example, an accumulator with load:

```
process(RST,CLK)
begin
   if RST then                  -- Reset
      Q <= 0;
   else
      if (CLK  and CLK'event) then
         if  load  then
            Q <= P;
         else
            Q <= P + Q;
```

```
            end if;
         end if;
      end if;
   end process;
end behavior;
```

The characteristic of load having a lower priority than clock enable for instance, is a characteristic of the target macrocell and is simply reflected in the VHDL macrocell inference engine. Sometimes your design may specify different behavior - but you still want to take advantage of macrocell inference. Suppose your design specified a counter with an enable and a load that has a higher priority than clock enable. You could do the following :

```
process (RST,CLK)
begin
   if RST then                -- Async Reset
      Q <= 0;
   else
      if CLK and CLK'event then
         if LD or CE then  -- load dominates clock enable,
                     -- so OR clkena pin
            if LD then     -- sync load
               Q <= D;
            else
               Q <= Q + 1;
            end if;
         end if;
      end if;
   end if;
end process;
```

Resource Sharing

Resource sharing is a compiler technique for sharing inferred macrocells in order to reduce the design area. The domain of resource sharing is a process, macrocells inferred within a single process may be shared, macrocells inferred in different processes will not be shared. The actual resource sharing performed will depend upon the target silicon architecture and on the compiler. If you write code that depends upon resource sharing you should check the log with debugging turned on to see that the implementation was as you expected.

Synthesis Attributes

One feature of VHDL that may not be familiar to programmers is attributes. VHDL has many predefined attributes which allow access to information about types, arrays, and signals. Some examples are :

```
integer'high    -- has a value of 2147483647
integer'low     -- has a value of -2147483647
```

If a subtype of type integer is declared

```
subtype shorter is integer range 0 to 100;
shorter'high   -- has a value of 100
shorter'low    -- has a value of 0
```

and

```
shorter'base'high    -- has a value of 2147483647
```

when used with an array the 'high attribute has a value of the array index:

```
type my_array is array (0 to 99) of boolean;
variable info : my_array;
info'high      -- has a has a value of 99
```

There is a set of attributes which give access to information about **signal** waveforms. Most signal attributes are for simulation, and have no meaning in the context of synthesis. However one, 'event, is useful. It may be used on signals to specify edge sensitivity. It is usually used in combination with a value test to specify a rising or falling edge.

```
signal clock : boolean;
not clock and clock'event    -- specifies a falling edge.
```

User-defined attributes

VHDL allows the user to define their own attributes. The Altium Designer-based VHDL tools use this capability to define attributes for synthesis, and to define attributes (also called properties or parameters) for downstream tools. The declaration of synthesis attributes are described in the following sections. Synthesis attributes described here are not passed to downstream tools. To use these attributes, either make them visible (use metamor.attributes.all), or copy to your VHDL source description. The value of these attributes must be locally static.

```
package attributes is

   ------------------------------------------------------
   -- User defined place and route information passed to
   -- output file
   ------------------------------------------------------
   attribute pinnum : string;
```

```
   attribute part_name : string;
   attribute property : string;

   ------------------------------------------------------
   -- User defined encoding of enumerated types
   ------------------------------------------------------
   attribute enum_encoding : string;

   ------------------------------------------------------
   -- User specified critical nodes
   ------------------------------------------------------
   attribute critical : boolean;

   ------------------------------------------------------
   -- User specified macrocells
   ------------------------------------------------------
   attribute macrocell : boolean;

end attributes;
```

Array_to_numeric (Synthesis attribute)

Some type conversion functions can be very slow to compile during VHDL synthesis. This attribute accelerates compilation in one specific and common case: converting arrays to numbers. An example is converting a bit_vector to an integer. This particular conversion specifies no logic but is slow to compile.

The Synthesizer provides an attribute, **array_to_numeric** (and also a **numeric_to_array** attribute), to short circuit the compilation of such functions as follows:

```
function to_integer ( arg : BIT_VECTOR ) return INTEGER is
   variable result : natural := 0;
   variable w : natural := 1;
   attribute array_to_numeric of to_integer : function is true;
begin
   -- Calculate bit_vector value.
   for i in arg'reverse_range loop
      if arg (i) = '1' then
         result := result + w;
      end if;
```

```
      -- test before multiplying w by 2, to avoid overflow
      if i /= arg'left then
         w := w + w;
      end if;
   end loop;
   return result;
end to_integer;
```

The attribute may only be applied to functions with an array formal parameter returning a numeric type when the parameter and the return value have the same synthesis encoding. For the array argument, 'left is assumed to be the most significant bit.

The array argument is treated as signed or unsigned depending on the subtype of the function return value. If the subtype of the return value ('natural' , the subtype of the variable 'result' in the example above) is signed (integer is signed), the array argument is sign extended. If the subtype is unsigned (natural is unsigned), the argument is zero extended.

When this attribute is true, the formal parameter is returned by the function with the subtype of the returned object. Since this function short circuits the semantics of VHDL it should be used with caution.

Clock_buffer (Synthesis attribute)

This attribute is ignored if the compiler output format is not EDIF. If the output format is EDIF and input and output buffers are being inserted, this attribute causes a clock buffer to be added in place of an input buffer. If buffers are not being inserted, the user may simply instantiate a technology specific clock buffer.

The attribute must be declared as :

```
attribute clock_buffer : boolean;
```

For example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity prep7 is
   generic (width : natural := 15);
   port (CLK, RST,LD,CE : in std_logic;
         D : in std_logic_vector (width downto 0);
         Q : buffer std_logic_vector (width downto 0));
   -- declare clock_buffer attribute
   attribute clock_buffer : boolean;
   -- mark port CLK as using a clock buffer
   attribute clock_buffer of CLK : signal is true;
end prep7;
```

Critical (Synthesis attribute)

This introduces nodes into the design, but does so from the VHDL source. The attribute **critical** allows the user to specify signals in the VHDL description whose timing is critical. An assignment to such a specified signal may imply a node in the output logic description. **Critical** is also used to put factoring under control of the user.

```
attribute critical of a,b,c : signal is true;  --a,b,c are nodes
```

In general, the Synthesizer will create a logic minimized design description in which there may be no one to one mapping between objects in the VHDL source description and combinational nodes in the output logic description.

Sometimes this 'minimum logic' description (where logic nodes are collapsed as controlled by the Optimizer) is not optimal for the propagation delay or layout of the resulting logic. In this event, the user may control the logic minimization by means of the attribute **critical**, which is applied to a **signal** in the VHDL source description.

This may be of use when the delay of the resulting logic can benefit from the designers knowledge of the structure or circuit (electrical/timing) characteristics of the implementation and not simply depend on being logically minimal. **Critical** constrains both the logic Optimizer and the synthesis function as specified by the user. It is also used to specify signals that will have net attributes for downstream tools.

Example

```
library ieee;
use ieee.std_logic_1164.all;
package encode2 is
   subtype byte is std_logic_vector (7 downto 0);
   subtype state_type is std_logic_vector (4 downto 0);
   constant st0 : state_type := "00101";
   constant st1 : state_type := "00000";
   constant st2 : state_type := "10000";
   constant st3 : state_type := "00100";
   constant st4 : state_type := "10100";
   constant st5 : state_type := "01100";
   constant st6 : state_type := "01000";
   constant st7 : state_type := "10101";
   constant st8 : state_type := "10001";
   constant st9 : state_type := "11000";
   constant st10 : state_type := "10011";
   constant st11 : state_type := "00011";
   constant st12 : state_type := "00001";
```

```
   constant st13 : state_type := "01101";
   constant st14 : state_type := "01001";
   constant st15 : state_type := "11001";
   constant dont_care : state_type := "-----";
end;

library ieee;
use ieee.std_logic_1164.all;
use work.encode2.all;

entity state_machineis
   port (clk,rst : boolean;
      i : byte;
      o : out byte);
end state_machine;

architecture instance of state_machine is
   signal machine : state_type;
begin
   process (clk,rst)
   begin
      if rst then
         machine <= st0;
      elsif clk and clk'event then
         case machine is
         when st0 =>
            case I is
            when "00000000"      => machine <= st0;
            when "00000001" to "00000011" => machine <= st1;
            when "00000100" to "00011111" => machine <= st2;
            when "01000000" to "00111111" => machine <= st3;
            when others          => machine <= st4;
            end case;
         when st1 =>
            if I(1 downto 0) = "11" then
               machine <= st0;
```

```
   else
      machine <= st3;
   end if;
when st2 =>
   machine <= st3;
when st3 =>
   machine <= st5;
when st4 =>
   if (I(0) or I(2) or I(4)) = '1' then
      machine <= st5;
   else
      machine <= st6;
   end if;
when st5 =>
   if (I(0) = '0') then
      machine <= st5;
   else
      machine <= st7;
   end if;
when st6 =>
   case I(7 downto 6) is
   when "00" => machine <= st6;
   when "01" => machine <= st8;
   when "10" => machine <= st9;
   when "11" => machine <= st1;
   end case;
when st7 =>
   case I(7 downto 6) is
   when "00"  => machine <= st3;
   when "11"  => machine <= st4;
   when others => machine <= st7;
   end case;
when st8 =>
   if (I(4) xor I(5)) = '1' then
      machine <= st11;
   elsif I(7) = '1'  then
```

```
      machine <= st1;
   end if;
when st9 =>
   if I(0) = '1' then
      machine <= st11;
   end if;
when st10 =>
   machine <= st1;
when st11 =>
   if i = "01000000" then
      machine <= st15;
   else
      machine <= st8;
   end if;
when st12 =>
   if i = "11111111" then
      machine <= st0;
   else
      machine <= st12;
   end if;
when st13 =>
   if (I(5) xor I(3) xor I(1)) = '1' then
      machine <= st12;
   else
      machine <= st14;
   end if;
when st14 =>
   case I is
   when "00000000"      => machine <= st14;
   when "00000001" to "00111111" => machine <= st12;
   when others          => machine <= st10;
   end case;
when st15 =>
   if (I(7) = '1') then
      case I(1 downto 0) is
      when "00" => machine <= st14;
```

```
            when "01" => machine <= st10;
            when "10" => machine <= st13;
            when "11" => machine <= st0;
            end case;
         end if;
        when others => machine <= dont_care;
        end case;
      end if;
   end process;

with machine select
   O <= "00000000" when st0,
       "00000110" when st1,
       "00011000" when st2,
       "01100000" when st3,
       "1------0" when st4,
       "-1----0-" when st5,
       "00011111" when st6,
       "00111111" when st7,
       "01111111" when st8,
       "11111111" when st9,
       "-1-1-1-1" when st10,
       "1-1-1-1-" when st11,
       "11111101" when st12,
       "11110111" when st13,
       "11011111" when st14,
       "01111111" when st15,
       "--------" when others;
end;

library metamor;
use metamor.attributes.all;
use work.encode2.all;

entity prep4 is
   port (clk,rst : boolean;
```

```
      i : byte;
      o : out byte);
end prep4;

architecture top_level of prep4 is
   component state_machine
   port (clk,rst : boolean;
      i : byte;
      o : out byte);
   end component;
   signal q1,q2,q3 : byte;
   attribute critical of q1,q2,q3 : signal is true;  --q1,q2,q3 are nodes
begin
   u1 : statemachine port map (clk,rst,i,q1);
   u2 : statemachine port map (clk,rst,q1,q2);
   u3 : statemachine port map (clk,rst,q2,q3);
   u4 : statemachine port map (clk,rst,q3,o);
end;
```

Critical is used in this example to separate the output encoder of one instance from the input decoder of the next; the result is a faster design. Critical is used in this case because neither the inputs or outputs of the components are registered. The state machine inputs are also encoded in such a way that they (just) fit within 16 product terms. In the multiple instance case, manual specification of the critical nodes in the combined output/input logic using the critical attribute produces better results than automatic synthesis.

The relationship between the name of a VHDL signal specified as critical, and its equivalent netlist node may be complex. For example, a one bit signal may result in no node if its use is redundant, or many nodes if hierarchy is used. The name of the VHDL signal may be maintained unless this would lead to a conflict. It may be prefixed with instance or block labels, or package names, and suffixed with a number if it represents more than one wire, or have a machine generated name.

Enum_encoding (Synthesis attribute)

You may need to specify different machine encoding for different hardware technologies. For example, one hot encoding may be preferred for an FPGA but not for a CPLD. The enum_encoding attribute allows you to specify how symbolic values declared as enumerated types are actually encoded in hardware.

The most common application of the enum_encoding attribute is in the specification of a state machine. Other applications include the resolution of multi-valued logic (such as std_ulogic) and for introducing don't cares into a decoder function.

Foreign (Synthesis attribute)

VHDL has an external language interface to allow users to specify modules in some non-VHDL form; the implementation is VHDL tool specific. The **foreign** attribute supports external HDLs . This mechanism is only supported using those output formats that support hierarchy and linking.

This attribute may be applied to an architecture. Its value specifies the name of the external module. For example :

```
entity abel_code is
   port (a,b : bit_vector(0 to 7) ; sum : out bit_vector(0 to 8));
end abel_code;

architecture simple of abel_code is
   attribute foreign of simple : architecture is "adder";
begin
end simple;
```

These statements in the architecture are ignored, and a call to the foreign language module 'adder' is generated when the entity abel_code is instantiated in a VHDL design. The inputs and outputs of adder must match the port declarations in VHDL. There are two constraints: the VHDL ports must have locally static types, and VHDL generics are not passed to the external module.

For example, the adder might be described in Abel :

```
MODULE adder
a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7 pin;
b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7 pin;
sum_0, sum_1, sum_2, sum_3, sum_4, sum_5, sum_6, sum_7, sum_8 pin;

a = [a_7..a_0];
b = [b_7..b_0];
sum = [sum_8..sum_0];

EQUATIONS
sum = a + b;

END;
```

A side effect of the **foreign** attribute is that the foreign module might be defined in VHDL. An easier way to do this is provided by the hierarchical compilation feature.

Inhibit_buf (Synthesis attribute)

For the XNF and EDIF output formats you may set a compile option that automatically inserts input and output buffers for the target device. Sometimes you may wish to override this buffer insertion on a per-pin basis. This may be done by attaching the **inhibit_buf** attribute to the top level port (as in the example insertion of a clock buffer shown below, or with any silicon specific IO structure).

```
library ieee;
use ieee.std_logic_1164.all;
entity Parent is
   port (clk : std_logic;
         a : std_logic_vector (7 downto 5);
         v : out std_logic_vector (3 downto 1));
   -- inhibit automatic input buffer on signal clk
   -- because of clock buffer instantiation
   attribute inhibit_buf : Boolean;
   attribute inhibit_buf of clk : signal is true;
end Parent;

architecture behavior of Parent is
   -- vendor specific clock buffer
   component CLKBUF
   port (I : std_logic;  O : out std_logic);
   end component;
   signal clk_buf : std_logic;
begin
   -- 3 flip flops
   v <= a when rising_edge(clk_buf);
   -- instantiate clock buffer
   u0 : CLKBUF port map (clk,clk_buf);
end;
```

Macrocell (Synthesis attribute)

The **macrocell** attribute is used to specify an alternate multi-bit port name expansion convention in two special cases (unbound components and root entity) and also to mark unbound components for use by a macrocell compiler. The **macrocell** attribute is required for parameterized macrocells such as XBLOX and LPM.

The alternate naming convention is "NameNumber" with no index delimiter character so that in the port names of IN_BUF_3 and OUT_BUF_3 will be I0,I1,I2,O0,O1,I2. This is for linking with logic from other

tools such as certain schematic capture packages. Examples of multi-bit types are integer and std_logic_vector. Examples of single bit types are bit and std_logic.

When a component instance has no entity bound to it the **macrocell** attribute is used to specify the alternate naming convention for the component's multi-bit formal ports. This is used to allow linking child components from other some tools.

When a top level entity has a **macrocell** attribute set true the top level ports will use the alternate convention. This is used to allow linking the output netlist with a parent netlist from some other tools.

```
library ieee;
use ieee.std_logic_1164.all;
entity Parent is
   port (a : std_logic_vector (7 downto 5);
         v : out std_logic_vector ( 1 to 3));
end Parent;
architecture behavior of Parent is
   -- component declaration , unbound
   component IN_BUF_3
      port (I : std_logic_vector (2 downto 0) ;
            O : out std_logic_vector (0 to 2));
   end component;
   component OUT_BUF_3
      port (I : std_logic_vector (2 downto 0) ;
            O : out std_logic_vector (0 to 2));
   end component;
   signal x : std_logic_vector (2 downto 0);
   attribute macrocell : Boolean;
   attribute macrocell of IN_BUF_3, OUT_BUF_3 : component is true;
begin
   -- component instantiation
   u0 : IN_BUF_3 port map (a,x);
   u2 : OUT_BUF_3 port map (x,v);
end;
```

Note that if the component formal ports have an unconstrained type (such as XBLOX or LPM instances) the **macrocell** attribute must be used. Components that have unconstrained formal ports or generics are parameterized, attaching the **macrocell** attribute also indicates that a downstream macrocell compiler (such as an XBLOX or LPM compiler) will be used.

Numeric_to_array (Synthesis attribute)

Some type conversion functions can be very slow to compile during VHDL synthesis. This attribute accelerates compilation in one specific and common case: converting numbers to arrays. An example is converting integer to std_logic_vector.

The Synthesizer provides an attribute, **numeric_to_array** (and also a **array_to_numeric** attribute), to short circuit the compilation of such functions as follows:

```
function int_2_v( int, bits : integer ) return std_logic_vector is
   variable out_vec : std_logic_vector((bits-1) downto 0) := (others =>
   '0');
   variable tmp : integer := int;


   --attribute numeric_to_array speeds compilation
   attribute numeric_to_array : boolean;
   attribute numeric_to_array of int_2_v : function is true;


begin
   for i in 0 to (bits-1) loop
      if ((tmp mod 2) = 1) then
         out_vec(i) := '1';
      end if;
      tmp := tmp/2;
   end loop;
   return out_vec;
end int_2_v;
```

The attribute may only be applied to functions with a numeric formal parameter returning an array type when the parameter and the return value have the same synthesis encoding. For the array result 'left is assumed to be the most significant bit.

When this attribute is true, the formal parameter is returned by the function with the subtype of the returned object. Note that you may have more than one formal argument, its the first that is returned, other arguments may be used to set the returned subtype as shown in the example. Since this function short circuits the semantics of VHDL it should be used with caution.

Part_name (Synthesis attribute)

The Synthesizer allows designers to pass place and route information to fitters, or netlists. This information has no meaning to the Synthesizer, it is simply passed from VHDL to the output file.

The **part_name** attribute is used to specify the target device, it may be applied to the top level entity. The attribute is declared as :

```
attribute part_name : string;
```

The value may be specified as follows (for backward compatibility, the library 'metamor' is still used in the coding example):

```
library metamor;
use metamor.attributes.all
entity special_attributes is
   port(c : bit_vector (3 to 5);
      d : bit_vector (27 downto 25);
      e : out boolean) ;

   --usage of part_name
   attribute part_name of special_attributes : entity is  "22v10";
end special_attributes;
```

The device compile option will override the value of the **part_name** attribute.

Pinnum (Synthesis attribute)

The Synthesizer allows designers to pass place and route information to fitters, or netlists. This information has no meaning to the Synthesizer, it is simply passed from VHDL to the output file.

The **pinnum** attribute is used to specify the pinout in the target device, and may be applied to ports in the top level entity. The attribute is declared as :

```
attribute pinnum : string;
```

Its value is a string containing a comma (',') delimited list of pad names or pin numbers. These values are assigned to the elements of the port in a left to right order. For example :

```
library metamor;
use metamor.attributes.all
entity special_attributes is
   port (a, b : in  integer range 0 to 7;
       c : bit_vector (3 to 5);
       d : bit_vector (27 downto 25);
       e : out boolean) ;

   -- usage of pinnum
   attribute pinnum of a : signal is  "4,5,6,7";   -- extra pin ignored
                         -- a(0) gets "6"
   attribute pinnum of b : signal is  "8,9";       -- missing pin number
                         -- b(0) not assigned
   attribute pinnum of c : signal is  "a3,b4,a1"; -- ascending order
```

```
                        -- c(3) gets "a3"
   attribute pinnum of d : signal is  "w1,W2,w99"; -- descending order
                        -- d(27) gets "w1"
   attribute pinnum of e : signal is  "2";      -- single bit
end special_attributes;
```

Notes

For backward compatibility, the library 'metamor' is still used in the coding example.

Property (Synthesis attribute)

The Synthesizer allows designers to pass place and route information to fitters, or netlists. This information has no meaning to the Synthesizer, it is simply passed from VHDL to the output file.

If the output is not EDIF then the **property** attribute is used to pass an arbitrary string to the output file. If applied to an entity the value is included at the head of the output file, if applied to a port the value is included as a property of the port in the output file. The attribute is declared as :

```
attribute property : string;
```

The value is passed directly to the output file; therefore, you will need to know the legal syntax for that file. The second example shows how using VHDL functions can make this task less error prone.

```
library metamor;
use metamor.attributes.all
entity special_attributes is
   port (c : bit_vector (3 to 5);
       d : bit_vector (27 downto 25);
       e : out boolean) ;

   -- usage of property on an entity
   attribute property of special_attributes : entity is
      "lca some text" & CR &
      "lca more text" & CR &
      "lca yet more text" & CR &
      "amdmach Mach Specific STuff";

   -- usage of property on a port
   attribute property of e : signal is "Fast";
end special_attributes;
```

Strings are passed to the output file exactly as specified in the VHDL source, and case is maintained. A characteristic of VHDL is that a new line character is not legal within a string; therefore, to create several lines, strings and a new line are concatenated using "xxx" & CR & "yyy" as shown in the

example above. This can get a little cluttered unless you declare functions for commonly used string values. For example:

```
package xilinx is
   function timespec (name, from, too, delay : string) return string;
end;

package body xilinx is
-- returns an XNF timespec symbol
   function timespec(name,from,too,delay : string) return string is
   begin
      return "SYM, XXX" & name &
         ", TIMESPEC, LIBVER=2.0.0, " & name &
         "=from:" & from & ":to:" & too & "=" & delay & CR &
         "END" & CR;
   end;
end;

library ieee,metamor;
use ieee.std_logic_1164.all;
use metamor.attributes.all;
use work.xilinx.all;
entity MORE_ATTRIBUTES is
   port (d,c,ce,r,tri : in std_logic;
         q,p : out std_logic;
         w : out std_logic_vector(2 downto 0));

   attribute property of MORE_ATTRIBUTES : entity is
      timespec("TS1","FFS","FFS","30ns") &
      timespec("TS2","PADS","LATCHES","35ns") &
      timespec("TS3","FFS","RAMS","25ns");

   attribute property of q,w : signal is "FAST";
   -- 4 pins are "FAST"
end;
```

Note

For backward compatibility, the library 'metamor' is still used in the coding examples.

Ungroup (Synthesis attribute)

The **ungroup** attribute removes hierarchy from the design and also overrides the default logic optimize behavior. By default, the logic Optimizer works on the logic within a single architecture. The logic is separately optimized for any component instantiated within the architecture maintaining the hierarchy.

By removing the Child from the hierarchy, **ungroup** causes a Child component to be optimized as part of its Parent. If a Child component has an **ungroup** attribute with a value true, its architecture is optimized as part of its Parent architecture. Multiple instances of a component with an **ungroup** attribute cause the logic for each instance to be added to the parent prior to optimization.

In the following trivial example, if **ungroup** is true the result is a wire, if **ungroup** is not present (or false) the implementation is an AND with its inputs connected.

```
---Child
library ieee;
use ieee.std_logic_1164.all;
entity Child is
   port (A, B : std_logic;
         C : out std_logic);
end Child;
architecture behavior of Child is
begin
   C <= A and B;
end;

---Parent
use ieee.std_logic_1164.all;
entity Parent is
   port (X : std_logic;
         Y : out std_logic);
end Parent;
architecture behavior of Parent is
   -- component declaration , bound to Entity Child above
   component Child
      port (A, B : std_logic;
            C : out std_logic);
   end component;
   -- UNGROUP TRUE, child is optimized as part of Parent
   attribute ungroup : Boolean;
   attribute ungroup of Child : component is true;
```

```
begin
   -- component instantiation
   u0 : Child port map (X, X, Y);
end;
```

The **ungroup** attribute might be used when compiling a design made up of components specifying a small amount of logic (such as TTL components). Unbridled use of the **ungroup** attribute can result in attempts to optimize large Parent blocks of logic, which may take a significant time.

Xilinx_BUFG (Synthesis attribute)

This attribute is ignored if the compiler output format is not Xilinx EDIF. If the output format is Xilinx EDIF and input and output buffers are being inserted, this attribute causes IBUFs to be replaced by BUFGs. If buffers are not being inserted, the user may simply instantiate a BUFG.

The attribute must be declared as :

```
attribute Xilinx_BUFG : boolean;
```

For example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity prep7 is
   generic (width : natural := 15);
   port (CLK, RST,LD,CE : in std_logic;
      D : in std_logic_vector (width downto 0);
      Q : buffer std_logic_vector (width downto 0));

   -- declare Xilinx layout attribute
   attribute Xilinx_BUFG : boolean;

   -- mark ports CE and LD as using BUFG
   -- (CLK will get BUFG by default)
   attribute Xilinx_BUFG of CE, LD : signal is true;
end prep7;
```

Xilinx_GSR, FPGA_GSR (Synthesis attribute)

The attributes **Xilinx_GSR** and **FPGA_GSR** have identical behavior. These attributes are ignored if the compiler output format is not EDIF for certain FPGAs. If the output format is EDIF for certain FPGAs then this attribute is used to mark a signal that uses the global set or reset resource. The marked signal must be a top level port. The marked signal is disconnected from any flip-flop preset or clear, and for Xilinx the flip-flop has an INIT property added. Note, some FPGAs have an active low GSR and require the appropriate signal in the VHDL be coded as active low.

A STARTUP (or equivalent) symbol must be instantiated in the top level and connecting its GSR input to a port. If the whole design is compiled in one pass (regardless of the number of entities) then no **FPGA_GSR** attribute is required. For example:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity design is
   port (d,set,clk  : std_logic;
         q   : out std_logic);
end design;

architecture design of design is
   component startup is
      port (gsr : std_logic);
   end component;
begin
   u0 : startup port map (set);
   q <= '1' when set = '1' else d when rising_edge(clk);
end design;
```

Alternatively if the VHDL is compiled in several parts (and the EDIF or XNF linked later) then the top level must have a STARTUP (or equivalent) symbol, and the other levels have a **FPGA_GSR** attribute attached to their reset port. For Example:

```
-------Parent.vhd
library IEEE;
use IEEE.std_logic_1164.all;
entity parent is
   port (d,set,clk  : std_logic;
         q   : out std_logic);
end parent;

architecture parent of parent is
```

```
   component startup is
      port (gsr : std_logic);
   end component;

   component child is
      port (d,set,clk   : std_logic;
            q   : out std_logic);
   end component;
begin
   u0 : startup port map (set);
   u1 : child port map (d,set,clk,q);
end parent;

-------------Child.vhd
library IEEE;
use IEEE.std_logic_1164.all;
entity child is
   port (d,set,clk   : std_logic;
         q   : out std_logic);
end child;

architecture child of child is
   attribute FPGA_gsr : boolean;
   attribute FPGA_gsr of set : signal is true; -- routed using GSR
begin
   q <= '1' when set = '1' else d when rising_edge(clk);
end child;
```

Attributes for Downstream Tools

The Synthesizer's compiler makes use of user defined attributes to pass information to downstream tools. The compiler also recognizes some attributes such as **critical** or **ungroup** to control its own operation. These are not passed to downstream tools. This section discusses the rules for passing attributes to downstream tools and shows some specific examples.

The use of attributes is complicated by the diverse usage by downstream tools; also by the distinction between port, instance, and net which does not exist in RTL or behavioral VHDL code but usually exist in the output netlist; and also by any hierarchy flattening that may occur.

To pass attributes to downstream tools you will need to know the netlist format, and the attributes plus their netlist location recognized by the downstream tool. Please refer to the downstream tool documentation for its legal attribute names, values and locations.

In the VHDL source, attributes may be passed as:

- the attribute as a name/value pair, this passes the pair to the netlist
- the value of the attribute "property", this passes only the value to the netlist
- the value of the attribute "pinnum", this passes pin numbers to the netlist
- the value of the attribute "part_name", this passes the specific device to the netlist.

Attributes passed as name/value pairs should be a type of string, integer, boolean, or a vector. These are representations usually expected by downstream tools. The following table shows how these VHDL attributes (attached to specific VHDL objects) are propagated to EDIF, OpenAbel 2, and DSL, as well as the objects to which they are attached. Note, your version of the compiler may only support some of these netlist formats.

EDIF	part_name	pinnum	property	name/value
Entity, top level	cell interface (1)			cell interface
Port, top level		port (1)		port
Entity, lower level				cell interface
Port, lower level				port
Signal inferring flip-flop, latch, or tristate				instance
Signal with attribute 'critical' true				net
Component				instance
Component instance label				instance
Open Abel 2	**part_name**	**pinnum**	**property**	**name/value**
Entity, top level	DEVICE		PROPERTY	
Port, top level		PINS	PINS	
DSL	**part_name**	**pinnum**	**property**	**name/value**
Entity, top level			file (5)	
Port, top level				

Note (1) Some downstream tools do not support this.

Note (2) EXT if inserting IO buffers else SIG.

Note (3) Hierarchy removed by flattening during compile.

Note (4) If then attribute type is boolean then only attribute name is written (if value is true).

Note (5) Unassociated with any netlist object.

Attributes attached to a signal may be attached to an instance, a net or a port in the netlist according to the priority, highest to lowest, listed below. (The attributes may also be ignored.)

- If the signal is a port then attribute is attached to a netlist port.
- If the signal infers a flip-flop, latch, or tristate then attribute is attached to a netlist instance
- If the signal has a synthesis attribute "critical" then attribute is attached to a netlist net.
- Other cases are ignored.

In order to place an attribute on a net the synthesis attribute, **critical**, must be attached to a VHDL signal. The same attribute is ignored when attached to ports or registers. You may need to create temporary signals in your VHDL source to distinguish attribute placement if there is a conflict between net, port, and instance. For example, an out port that infers a flip-flop may require a slew attribute on the netlist port and a location attribute on the netlist instance of the flip-flop. These are distinct in the netlist but may be represented by the same signal in the VHDL source. In this case an extra signal must be added to the VHDL to support attribute passing.

Some examples follow.

For explicit instances, attach the VHDL attribute to the instance label as shown below:

```
architecture x of y is
   -- Placement hints
   attribute CHIP_PIN_LC of u0 : label is "LAB2";
   attribute CHIP_PIN_LC of u2 : label is "LAB7";
begin
   u0 : buffer port map (a,b);
   u1 : buffer port map (x,y);
   ....
```

You may also use generics to pass instance parameters. This approach is useful if the child component won't be synthesized, but will have a simulation model that needs to see the attribute value.

```
architecture x of y is
   component ROM
      generic (filename : string)
      port (A : std_logic_vector (7 downto 0) ;
         D : out std_logic_vector (4 downto 0));
   end component;
begin
   uo : ROM generic map ("init.prg") port map (address,data);
   ....
```

Adding attributes to netlist instances of inferred flip-flops, latches, or tristates is done using VHDL attributes attached to the signal.

```
architecture x of y is
```

```
   signal q : std_logic_vector (3 downto 0);
   -- Register placement hint,
   -- all 4 flip-flops get REGTYPE=IOC
   attribute REGTYPE of q : signal is "IOC" ;
begin
   q <= d when rising_edge(clk);
   ....
```

You may attribute nets , but only if the compiler is allowed to retain the signals in the output netlist with the Synthesizer's Critical attribute.

```
architecture x of y is
   signal c : std_logic;
   --allow Synthesizer to keep the logic
   attribute critical of c : signal is true;
   -- using attribute property
   attribute property of c : signal is "X";
   -- adds the value to the instance driving the net
   -- using  name/value , assume W is attribute type integer
   attribute W  of c : signal is 100;
   -- assume  SC is declared as boolean.
   attribute SC of c : signal is true;
begin
   c <= a or b;
   d <= c or d;
   ....
```

Netlist ports may be attributed in the same way:

```
entity x is
   port (a,b : std_logic;
         d : out std_logic);
end x;
architecture x of y is
   signal c : std_logic;
   -- using attribute property
   attribute property of a : signal is "NODELAY"; -- adds the value to the
   port
   -- using  name/value , assume TNM is attribute type string
   attribute TNM of b : signal is "name_list";
```

```
   -- assume FAST is declared as boolean.
   attribute FAST of d : signal is true;
begin
   c <= a or b;
   d <= c or d;
   ....
```

Synthesis Coding Issues

Synthesis Coding Issues

A common misconception is that a synthesis compiler 'synthesizes VHDL' - this is incorrect. The tool synthesizes your design expressed in VHDL.

Understanding the hardware that you are specifying is the simplest rule for success. This is particularly important for critical timing. Conversely the easiest way to fail is write a model of your design and then wonder why the synthesis tool didn't 'do the design' for you.

What does synthesize mean in this context? It means to 'transform a logic design specification into an implementation' - nothing you couldn't do yourself. A synthesis tool simply handles the details of this transformation for you.

This section contains examples of user coding problems. They are all real user issues, some may be obvious, others are not.

Test for High Impedance

The following example means 'if sig is floating' - quite a reasonable test to perform in a simulation model. However, a synthesis tool has to transform this into a hardware element that matches this behavior.

```
if sig = 'Z' then     -- sig is std_logic
   -- do something
end if;
```

The code specifies a logic cell that looks at the drive of its fan-in then outputs true if not driven, and false if driven true or false. Such a cell does not exist in most programmable silicon. IEEE 1076.3 specifies that this comparison should always be false, so the statements inside the **if** are not executed, and no logic is generated.

Long Signal Paths - Nested ifs

Multiple nested **if** or **elsif** clauses can specify long signal paths.

```
if sig = "000" then
   -- first branch
elsif sig = "001" then
   -- second branch
elsif sig = "010" then
   -- third branch
elsif sig = "011" then
   -- fourth branch
elsif sig = "100" then
   -- fifth branch
else
   -- last branch
end if;
```

This code is an inefficient way to describe logic - a **case** statement would be much better. A good example is the test for the fourth branch, which depends on three previous tests and describes a long signal path, with the resulting logic delay.

```
case sig is
   when "000" =>     -- first branch
   when "001" =>     -- second branch
   when "010" =>     -- third branch
   when "011" =>     -- fourth branch
   when "100" =>     -- fifth branch
   when others =>    -- last branch
end case;
```

In practice, if the branches contain very little logic, or there are few branches, then there may be little difference. However, the **case** statement generally results in a better implementation.

Long Signal Paths - Loops

Loops are very powerful, but each iteration of a loop replicates logic. A variable that is assigned in one iteration of a loop and used in the next iteration results in a long signal path. This signal path may not be obvious. An example where a long signal path is the expected behavior might be a carry chain (the variable c below):

```
function "+" (a,b:bit_vector)return bit_vector is   -- assumes a,b
descending
   variable sum : bit_vector (a'length downto 0);
   variable c :bit := '0';
begin
   for i in a'reverse_range loop
     sum(i) := a(i) xor b(i) xor c;
     c := (a(i) and c) or (b(i) and c) or (a(i) and b(i));
   end loop;
   sum(a'length) := c;
   return sum;
end;
```

Simulation-optimized code

It is likely that code written for optimal simulation speed will not be an optimal description of the logic.

In the following example it is assumed that only one control input will be active at a time. The description is efficient for simulation, but a poor logic description because the independence of the control signals is not described within the VHDL code.

```
out1 <= '0';
out2 <= '0';
out3 <= '0';

if in1 = '1' then
   out1 <= '1';
elsif in2 = '1' then
   out2 <= '1';
elsif in3 = '1' then
   out3 <= '1';
end if;
```

The independence of the control signals needs to be contained within the design description. The result may be slightly slower simulation, but a smaller logic implementation after synthesis.

```
out1 <= '0';
out2 <= '0';
out3 <= '0';

if in1 = '1' then
   out1 <= '1';
end if;
if in2 = '1' then
   out2 <= '1';
end if;
if in3 = '1' then
   out3 <= '1';
end if;
```

Note that the issue is not a long signal path, but an unclear specification of the design. The best Optimizer in the world can't turn an inefficient algorithm into an efficient one. An algorithm that is efficient from one viewpoint may not be efficient from another.

Port Mode inout or buffer

Simply an issue of overspecification - **Inout** specifies bi-directional dataflow, **buffer**, like **out**, specifies unidirectional dataflow. There are very few occasions in hardware design when bi-directional data flow on a single wire is actually what you want. Use **inout** when you want to specify a signal path that is actually routed through a pin, such as a Xilinx IOB or a PLD pin feedback resource.

Users often use **inout** when they have a logical output they wish to read from, in this case use mode **buffer**. This results in a signal path internal to the target device It is not a good idea to use **inout** on lower levels of hierarchy when separately compiling each design unit. Doing so may be a problem for third party linkers. If the design units are compiled at the same time, the implementation will be two wires, one for data flow in each direction.

Using Simulation Libraries

Compiling simulation models with a synthesis tool is generally understood to be an impractical way to do hardware design. Such models, even if the Synthesizer will accept them, may be correct designs, but are rarely good designs.

The same applies to libraries of functions written for simulation. They may be acceptable to the synthesis tool, but are unlikely to produce good synthesis results. It is critically important that libraries be tuned for synthesis. This is typically done by keeping the same package interface and modifying the package body.

Type Conversion Functions

Usually type conversion functions specify no logic, although this is not always the case. Most logic free functions compile fairly quickly. There is, however, one common exception: a function that performs an array to integer conversion. For example :

```
function to_integer ( constant arg : bit_vector ) return natural is
   alias xarg : bit_vector (arg'length -1 downto 0) is arg;
                          -- normalize direction
   variable result : natural := 0;
   variable w : natural := 1;
begin
   for i in xarg'reverse_range loop
      if xarg (i) = '1' then
         result := result + w;
      end if;
      if (i /= xarg'left) then
         w := w + w;
      end if;
   end loop;
   return result;
end to_integer;
```

This function will be slow to compile if arg'length is greater than 16 to 24 bits (depending on your computer speed/memory). This is the case because one of the "+" operators results in an adder being

built for each iteration of the loop (even though the function describes no logic). These adders are removed on data flow analysis.

One solution to this problem is the use of the **array_to_numeric** attribute.

Depending on Initial Value

The initial value of a signal or variable is the value specified in the object's declaration (if not specified there is a default initial value). The initial value of such an object is its value when created. Signals and variables declared in processes are created at 'time zero'. Variables declared in subprograms are created when the subprogram is called.

The value at time zero has no clear meaning in the context of synthesis, therefore, the initial value of signals and process variables must be used with care. This issue does not arise with the initial value of variables declared in subprograms.

You should not depend on the initial value of signals or process variables if they are not completely specified in the process in which they are used. In this case, the compiler will ignore the time zero condition and use the driven value - effectively ignoring the single transition from the time zero state. If such signals or variables are not assigned, you may reliably use their initial value. Obviously, signals assigned in another process will never depend upon the initial value. For example :

```
   signal res1 : bit := '0';
begin
   process (tmpval,INIT)
   begin
      if (tmpval = 2**6 -1) then
         res1 <= '1';
      elsif (INIT ='1') then
         res1 <= '1';
      end if;
   end process;
```

In this case 'res1' is never assigned low - the code will be synthesized as a pull-up. However during simulation at time zero, 'res1' starts at '0', makes one transition to '1' and stays there. If this is really the intent, the solution is to use a flip-flip.

This design probably depends upon a wire floating low at power up, and probably has no realizable implementation. A solution might be :

```
process (tmpval,INIT)
   begin
      if (tmpval = 2**6 -1) then
         res1 <= '1';
      elsif (INIT ='1') then
         res1 <= '1';
      else
         res1 <= '0';      -- drive it low *****
      end if;
end process;
```

Assign to Array Index

For an assignment such as:

```
a(b) <= c;
```

If b is not a constant, then some care should be taken with this expression. This is because the statement means element 'b' of 'a' gets the value of 'c'; AND all the other elements of 'a' get their previous value (i.e. are unchanged). In hardware this implies storage of data. If this assignment is not clocked, combinational feedback paths will be created.

A typical usage might be :

```
a(b) <= c when rising_edge(clk);
```

If the assignment is clocked as in the example above (and the clock enable compile option is on), the element select logic will drive the flip-flop clock enable control for an efficient implementation. However, an explicit clock enable will override the implicit clock enable. In the following example 'clk_ena' will be connected to the clock enable control and the select logic will be included in the data path.

```
if rising_edge(clk) then
   if clk_ena = '1' then
      a(b) <= c;
   end if;
end if;
```

Don't Care

The semantics of the '-' element of std_logic_1164 are not the same as the semantics of Don't Care in some PLD programming languages. The '-' in 1164 is a unique element of the nine value type std_logic, and not a wildcard.

For example, if

```
a <= "00010"
b <= a = "00---"
```

then b is never true !

If you wish to ignore comparison on some bits, then be explicit:

```
b <= a(4 downto 3) = "00";
```

will produce the desired result.

Unintended Latches

Latches are inferred using incomplete specification in an **if** statement. The following example specifies a latch gated by 'address_strobe', which may not be the intent.

```
process (address, address_strobe)
begin
   if address_strobe = '1' then
      decode_signal <= address = "101010";
   end if;
end process;
```

This says, when address_strobe is '0', then decode_signal holds its previous value, resulting in the latch implementation. In this case the intent is probably to ignore decode_signal when address_strobe is '0'. However, you need to be explicit.

```
if address_strobe = '1' then
   decode_signal <= address = "101010";
else
   decode_signal <= false;
end if;
```

The log file will contain the name and line number of all inferred elements including (unintended) latches.

Unintended Combinational Feedback

It is possible to specify unintended combinational feedback paths by using variables (declared in a process) before they are assigned, or by incomplete specification.

In the following example, if the ReadPtr(i) is never equal to '1', Qint keeps its previous value. It may be a characteristic of the design that one bit of ReadPtr is always '1', but nothing says this is so. Qint is incompletely specified and a feedback path exists, which includes Qint when ReadPtr is all zeros.

```
process (ReadPtr, Fifo)
begin
   for i in ReadPtr'range loop
      if ReadPtr(i) = '1' then
         Qint <= Fifo(i);
      end if;
   end loop;
end process;
```

This case is coded for by making certain Qint is always assigned, in which case its value is defaulted to all zeros, and the unintended feedback path is removed.

```
process (ReadPtr, Fifo)
begin
   Qint <= (others => '0');  -- because of possible comb feedback
   for i in ReadPtr'range loop
      if ReadPtr(i) = '1' then
         Qint <= Fifo(i);
      end if;
   end loop;
end process;
```

The log file will contain the name and line number of all inferred elements, including (unintended) combinational feedback.

Observe the Register Inference Conventions

Synthesis tools infer storage devices (such as latches and flip flops) from incomplete assignment of variables or signals. To the other extreme it is possible to specify storage elements that the synthesis tool won't recognize.

For example:

```
process (clk1,clk2)
begin
   if rising_edge(clk1) then
      if rising_edge(clk2) then
         q <= d;
      end if;
   end if;
end process;
```

This probably describes a flip-flop that loads when its two clocks change at the same instant. It will function during simulation (because of the discrete nature of simulation time) but no hardware element has this behavior, and the compiler will report an error.

It is also possible to specify code that has implementable behavior, which one synthesis tool recognizes and another doesn't. For portable code, keep to the register inference conventions.

VHDL Quick Reference

This section contains quick reference information for VHDL syntax presented in an example-based style. It consists of a partial listing of VHDL constructs, focusing on those that are frequently used for hardware design. For complete information, refer to the IEEE Standard VHDL Language Reference Manual (LRM).

Lexical Elements

- comments from -- to end of line
- characters 'a' 'Z' ':'
- strings "hi there"
- bit strings b"0101" o"05" x"5"
- integers 123_456 2E2 2#0101#
- identifiers , a letter followed by letters, numbers, or underbar :
  hello hello7 h_e_l_l_o
- extended identifiers , any characters delimited by backslash (extended identifiers are case sensitive):
  \hello\ \hell____o\ \And\ \7*\

Reserved Words

The following is a list of words that are reserved in standard VHDL (regardless of case) and cannot serve as user-defined identifiers.

A-H	I-P	record
abs	**if**	**register**
access	**impure**	**reject**
after	**in**	**rem**
alias	**inertial**	**report**
all	**inout**	**return**
and	**is**	**rol**
architecture		**ror**
array	**label**	
assert	**library**	**select**
attribute	**linkage**	**severity**
	literal	**shared**
begin	**loop**	**signal**
block		**sla**

body
buffer
bus

case
component
configuration
constant

disconnect
downto

else
elsif
end
entity
exit

file
for
function

generate
generic
group
guarded

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
port
postponed
procedure
process
pure

Q-Z

range

sll
sra
srl
subtype

then
to
transport
type

unaffected
units
until
use

variable

wait
when
while
with

xnor
xor

Declarations and Names

The following code fragments illustrate the syntax of VHDL statements :

Declarations

```
-- OBJECTS
constant alpha : character := 'a';
variable total : integer ;
variable sum : integer := 0;
signal data_bus : bit_vector (0 to 7);

-- TYPES
type opcodes is (load,store,execute,crash);
type small_int is range 0 to 100;
type big_bus is array ( 0 to 31 ) of bit;
type glob is record
   first : integer;
   second : big_bus;
   other_one : character;
end record;

-- SUBTYPES
subtype  shorter is integer range 0 to 7;
subtype  smaller_int is small_int range 0 to 7;
```

Names

```
-- Array element
big_bus(0)

-- Record element
record_name.element
```

Sequential Statements

The following code fragments illustrate the syntax of VHDL sequential statements :

```
--IF STATEMENT
if increment and not decrement then
   count := count +1;
elsif not increment and decrement then
   count := count -1;
```

```
elsif increment and decrement then
   count := 0;
else
   count := count;
end if;

--CASE STATEMENT
case day is
when Saturday to Sunday =>
   work := false;
   work_out := false;
when Monday | Wednesday | Friday =>
   work := true;
   work_out := true;
when others =>
   work := true;
   work_out := false;
end case;

-- LOOP,NEXT,EXIT STATEMENTS
L1 : for i in 0 to 9 loop
   L2 : for j in opcodes loop
      for k in 4 downto 2 loop -- loop label is optional
         if k = i next L2;      -- go to next L2 loop
      end loop;
      exit L1 when j = crash; -- exit loop L1
   end loop;
end loop;

-- WAIT STATEMENT
wait until clk;

-- VARIABLE ASSIGNMENT STATEMENT
var1 := a or b or c;
-- SIGNAL ASSIGNMENT STATEMENT
sig1 <= a or b or c;
```

Subprograms

The following code fragments illustrate the syntax of VHDL subprograms :

```
-- FUNCTION DECLARATION
-- parameters are mode in
-- return statements must return a value
function is_zero (n : integer) return boolean is
   -- type, variable,constant,subprogram declarations
begin
   -- sequential statements
   if n = 0 then
      return true;
   else
      return false;
   end if;
end;

-- PROCEDURE DECLARATION
-- parameters may have mode in , out or inout
procedure count (incr : boolean; big : out bit;
                 num : inout integer) is
   -- type, variable,constant,subprogram declarations
begin
   -- sequential statements
   if incr then
      num := num +1;
   end if;
   if num > 101 then
      big := '1';
   else
      big := '0';
   end if;
end;
```

Concurrent Statements

The following code fragments illustrate the syntax of VHDL concurrent statements :

```
-- BLOCK STATEMENT
label5 :   -- label is required
block
   -- type, signal,constant,subprogram declarations
begin
   -- concurrent statements
end block;

-- PROCESS STATEMENT , sequential first form
label3 :      -- label is optional
process
   -- type, variable,constant,subprogram declarations
begin
   wait until clock1;
   -- sequential statements
end process;

-- PROCESS STATEMENT , sequential second form
process ( clk) -- ALL signals that cause the
           -- output to change
   -- type, variable,constant,subprogram declarations
begin
   if clk then
      -- sequential statements
      local <= en1 and en2;
      -- sequential statements
   end if;
end process;

-- PROCESS STATEMENT , combinational
process ( en1, en2, reset ) -- ALL signals used in
                    -- process
   -- type, variable,constant,subprogram declarations
```

```
begin
   -- sequential statements
   local <= en1 and en2 and not reset;
   -- sequential statements
end process;

-- GENERATE STATEMENT
label4 : -- label required
for i in 0 to 9 generate
   --  declarations
begin      -- begin is optional if no declarations
   -- concurrent statements
   label : if i /= 0 generate
      -- concurrent statements
      sig(i) <= sig(i-1);
   end generate;
end generate;

-- COMPONENT INSTANTIATION
-- label is required
-- positional association
U1 : decode port map (instr, rd, wr);
-- named association
U2 : decode port map (r=> rd, op => instr, w=> wr);

-- DIRECT INSTANTIATION
-- label is required
-- positional association
U1 : entity decode port map (instr, rd, wr);
-- named association
U2 : entity decode port map (r=> rd, op => instr, w=> wr);

-- CONDITIONAL SIGNAL ASSIGNMENT
total <= x + y;
sum <= total + 1 when increment else total -1;
```

```
-- SELECTED SIGNAL ASSIGNMENT
   with reg_select select
      enable <= "0001" when "00",
                "0010" when "01",
                "0100" when "10",
                "1000" when "11";
```

Library Units

The following code fragments illustrate the syntax of VHDL statements :

```
-- PACKAGE DECLARATION
package globals is
   -- type,constant, signal ,subprogram declarations
end globals;

-- PACKAGE BODY DECLARATION
package body globals is
   -- subprogram definitions
end globals;

-- ENTITY DECLARATION
entity decoder is
   port (op : opcodes; r,w : out bit);
end decoder;

-- ARCHITECTURE DECLARATION
architecture first_cut of decoder is
   -- type, signal,constant,subprogram declarations
begin
   -- concurrent statements
end first_cut;

-- CONFIGURATION DECLARATION
configuration example of decoder is
   -- configuration
end example;
```

```
-- LIBRARY CLAUSE
-- makes library , but not its contents visible
library utils;

-- USE CLAUSE
use utils.all;
use utils.utils_pkg.all;
```

Attributes

ATTRIBUTES DEFINED FOR TYPES

T'base - the base type of T

T'left - left bound of T

T'right - right bound of T

T'high - high bound of T

T'low - low bound of T

T'pos(N) - position number of N in T

T'val(N) - value in T of position N

T'succ(N) - T'val(T'pos(N) +1)

T'pred(N) - T'val(T'pos(n) -1)

T'leftof(N) - T'pred(N) if T is ascending

- T'succ(N) if T is descending

T'rightof(N) - T'succ(N) if T is ascending

- T'pred(N) if T id descending

T'image(N) - string representing value of N

T'value(N) - value of string N

ATTRIBUTES DEFINED FOR ARRAYS

A'left(N) - left bound of Nth index of A

A'right(N) - right bound of Nth index of A

A'high(N) - high bound of Nth index of A

A'low(N) - low bound of Nth index of A

A'range(N) - range of Nth index of A

A'reverse_range(N) - reverse range of Nth index of A

A'length(N) - number of values in Nth index of A

A'ascending - true if array range ascending

ATTRIBUTES DEFINED FOR SIGNALS

S'event - true if an event has just occurred on S

S'stable - true if an event has not just occurred on S

S'last_value - last value of S

STRING ATTRIBUTES

E'simple_name - string "E"

E'path_name - hierarchy path string

E'instance_name - hierarchy and binding string

VHDL Constructs

The following sections provide a partial list of VHDL constructs.

Design Entities and Configurations

Entity Declarations
- Generics
- Ports

Architectures

Configuration Declarations

Subprograms and Packages

Subprogram declarations

Subprogram bodies

Subprogram overloading

Signatures

Operator overloading

Package declarations

Package bodies

Types

Scalar types
- Enumerated types
- Integer types

Composite types
- Array types
- Record types

Declarations

Type declarations
Subtype declarations
Objects
  Constant declarations
  Signal declarations
  Variable declarations
   Interface declarations
  Alias declarations
Attribute declarations
Component declarations
Group declarations

Specifications

Attribute specifications
Configuration specifications

Names

Simple names
Selected names
Indexed names
Slice names
Attribute names

Expressions

Operators
  Logical operators
  Relational Operators
  Adding operators
  Multiplying operators
  Miscellaneous operators
Operands
  Literals
  Aggregates
  Function calls
  Qualified expressions
  Type conversions

Sequential Statements

Wait statement
Assertion statement
Signal assignment statement
Variable assignment statement
Procedure call statement
If statement
Case statement
Loop statement
Next statement
Exit statement
Return statement
Null statement

Concurrent Statements

Block statement
Process statement
Concurrent Procedure call statement
Concurrent Assertion statement
Concurrent Signal assignment statement
Conditional signal assignment
Selected signal assignment
Component instantiation statement
Generate statement

Visibility

Use clauses

All Lexical Elements

Predefined Language Environment
Predefined attributes (but not signal attributes except 'event)
Package STANDARD

Unsupported Constructs

The following constructs are not supported, their use will result in a Constraint message.

- Access types
- File types
- Signal attributes (except 'event , 'stable,and'last_value)
- Textio package
- Impure functions
- Shared variables

Ignored Constructs

The following constructs are ignored. They may be used in VHDL simulation, but the Synthesizer will not generate any logic.

- Disconnect specifications
- Resolution functions
- Signal kind register
- Waveforms, except the first element value

Constrained Constructs

The following constructs are constrained in their usage. Constrained constructs fall into two classes - statements constrained in where they may be used, and constrained expressions. The use of a constrained construct will result in a Constraint message.

Constrained statements

- A wait statement may only be first statement in a process.
- Signal attributes 'event , 'stable, and 'last_value are valid only in where they specify a clock edge.
- Subprograms calls and hierarchy instantiations cannot be recursive.
- Formal part of a named association may not be a function call.
- All tristate buffers driving an internal tristate bus must be in the same architecture.
- A process sensitivity list must contain all signals that the process is sensitive to.

Constrained expressions

Certain expressions must be metalogic expressions, which simply means the value of the expression must not depend upon a signal (the value of the expression will not vary over time).

- Operands of ** must be metalogic expressions.
- Assertion statement condition, severity, and message must be metalogic expressions, if the message is to be reported.
- Type and subtype constraint declarations must be metalogic expressions.
- Floating point and physical types are constrained to the same values as the equivalent integer type.
- While loop and unconstrained loop execution completion must depend only on metalogic expressions.

Error Messages

VHDL Synthesis Error Messages

The following error message topics are intended to help you determine the cause of a problem in your VHDL source file. Each error message produced by the compiler is listed, along with more detailed explanations and suggested workarounds and tips (recommendations). Words in single quotes will be substituted in the actual error message. Note that the workarounds listed are only suggestions; it is not possible for the language compiler to know what you are actually trying to describe with a particular set of VHDL language statements.

0-series Error Messages

Error Message: $Y0001

Summary

Unexpected end of file.

Description

The Compiler has encountered the end of the source file before reaching the end of the current library unit (entity, architecture, configuration, package or package body).

Recommendation

Check to make sure that you have not omitted one or more **end** statements from the source file. Also check to ensure that the disk file is not corrupt or truncated.

Error Message: $Y0002

Summary

Syntax error near 'operator'.

Description

The Compiler has encountered an unexpected sequence of characters or language tokens. The error is associated with the indicated VHDL operator.

Recommendation

Check to make sure you are using the operator properly. Also check to make sure there is no other syntax error on the same line, or on previous lines, that might cause the error.

Check carefully to make sure that you have placed semicolons in their proper locations on previous lines.

Error Message: $Y0003

Summary

Syntax error near 'name'.

Description

The Compiler has encountered an unexpected sequence of language elements. The error is associated with the indicated identifier name.

Recommendation

Check to make sure you are using the identifier properly. Also check to make sure there is no other syntax error on the same line, or on previous lines, that might cause the error.

Check carefully to make sure that you have placed semicolons in their proper locations on previous lines.

Error Message: $Y0004

Summary

Based literal format is incorrect.

Description

The Compiler has encountered a based literal (a literal that has been specified as having a base between 2 and 16) that does not have a valid format.

Recommendation

Check the syntax of the literal to make sure it conforms to the requirements of the specified number base.

Error Message: $Y0005

Summary

Unexpected non-graphic character found.

Description

The Compiler has encountered a character that is not a part of the defined VHDL character set.

Recommendation

Check to make sure that the text editor used to create the source file has not placed illegal characters (such as word processor control codes) into your source file.

Error Message: $Y0006

Summary

An identifier may not begin with the special character 'character'.

Description

The Compiler has encountered an identifier or other VHDL name that begins with a non-alphabetic character. Identifiers in VHDL must begin with an upper or lower case letter. Identifiers may not begin with numbers, underscores, or other special characters.

Recommendation

Check to make sure the identifier conforms to the VHDL requirements for identifier names.

Also check to make sure you have not misplaced an operator or other special character.

Error Message: $Y0007

Summary

Unable to open file 'name'.

Description

The Compiler has encountered an error when attempting to open the indicated file.

Recommendation

Check to ensure that the indicated filename is correctly spelled and exists in the current directory or the directory indicated in the file path.

Error Message: $Y0008

Summary

A 'name' must not contain a new line character.

Description

The Compiler has encountered a new line character in a quoted string or an extended name. Strings and extended identifiers in VHDL must not contain new line characters.

Recommendation

Check to make sure that you have placed a terminating quote character on the end of the string, or terminating backslash on an extended name. For readability in your editor you may prefer shorter strings, in this case use the concatenation operator (&) to break the string into multiple parts on multiple lines.

Error Message: $Y0009

Summary

A 'name' must not contain a CR character.

Description

The Compiler has encountered a carriage return character in a quoted string or extended identifier. Strings and extended identifiers in VHDL must not contain CR characters.

Recommendation

Check to make sure that you have placed a terminating quote character on the end of the string. If the string is too long to enter on one line, use the concatenation operator (&) to break the string into multiple parts on multiple lines. If you require that a carriage return character be embedded in the string, use the syntax:

```
'string1' & CR & 'string 2'
```

to concatenate two sub-strings with a carriage return character.

Error Message: $Y0010

Summary

A 'name' must not contain a non-graphic character.

Description

The Compiler has encountered an illegal character in a quoted string or extended identifier.

Recommendation

Check to make sure that the string or extended identifier indicated contains only valid VHDL characters. If the string or extended identifier appears to include only valid characters, check to make sure your text editor or word processor has not inserted illegal non-graphic characters.

Error Message: $Y0011

Summary

A bit string must not contain a new line character.

Description

The Compiler has encountered a new line character in a bit string. Binary bit strings must consist only of the characters '0', '1' and '_'. Octal bit strings must consist only of the characters '0' to '7' and '_'. Hexadecimal bit strings must consist only of the characters '0' to 'f' and '_'.

Recommendation

Check to make sure that you have placed a terminating quote character on the end of the bit string.

Error Message: $Y0012

Summary

Bit string delimiters do not match.

Description

The Compiler has encountered an unexpected character at the end of a bit string.

Recommendation

Check to make sure that you have used the same character delimiter at the beginning and end of the bit string. If you have used the replacement character '%' in the bit string, make sure that the same replacement character is used as both the first and second delimiter.

Error Message: $Y0013

Summary

Illegal binary value 'character' in bit string.

Description

The Compiler has encountered an unexpected character while reading a bit string. The Compiler has encountered an invalid binary format bit string. Binary bit strings must include only the characters '0' through '1', and the special character '_'.

Recommendation

Check to make sure that the bit string is in a valid binary number format, or change the base specification to reflect the format used.

Error Message: $Y0014

Summary

A Bit string must not have '_' as its first element.

Description

The Compiler has encountered an illegal use of the special character '_' in a bit string. The '_' character may not be used as the first or last character in a bit string.

Recommendation

Check to make sure that the bit string does not begin with a '_' character.

Error Message: $Y0015

Summary

A bit string must not contain consecutive under bars '__'.

Description

The Compiler has encountered an illegal use of the special character '_' in a bit string. The '_' character can only be used to provide separation between numeric characters in a bit string and must be entered as a single character.

Recommendation

Check to make sure that the bit string does not include extraneous '_' characters.

Error Message: $Y0016

Summary

A bit string must not have '_' as its last element.

Description

The Compiler has encountered an illegal use of the special character '_' in a bit string. The '_' character can only be used to provide separation between numeric characters in a bit string. The '_' character may not be used as the first or last character in a bit string.

Recommendation

Check to make sure that the bit string does not end with an extraneous '_' character.

Error Message: $Y0017

Summary

Illegal octal value 'character' in bit string.

Description

The Compiler has encountered an invalid octal format bit string. Octal bit strings must include only the characters '0' through '7' and the special character '_'.

Recommendation

Check to make sure that the bit string is in a valid octal number format, or change the base specification to reflect the format used.

Error Message: $Y0018

Summary

Illegal hex value 'character' in bit string.

Description

The Compiler has encountered an invalid hexadecimal format bit string. Hexadecimal bit strings must include only the characters '0' through '9', 'A' through 'F', 'a' through 'f,' and the special character '_'.

Recommendation

Check to make sure that the bit string is in a valid hexadecimal number format, or change the base specification to reflect the format used.

Error Message: $Y0019

Summary

Based literal contains illegal character 'character'.

Description

The Compiler has encountered an invalid based numeric literal. Numeric literals entered in non-decimal format must include only the characters appropriate for the base specification. (e.g. '0' through '7' and the special character '_' if the base specifier is 8).

Recommendation

Check to make sure that the based numeric literal is in a valid numeric format that matches the base specification.

Error Message: $Y0020

Summary

Literal Base must not be greater than 16.

Description

The Compiler has encountered an invalid based numeric literal. The literal base specification must be in the range of 2 to 16.

Recommendation

Check to make sure that the literal has a valid base specification.

Error Message: $Y0021

Summary

Literal Base must not be less than 2.

Description

The Compiler has encountered an invalid based numeric literal. The literal base specification must be in the range of 2 to 16.

Recommendation

Check to make sure that the literal has a valid base specification.

Error Message: $Y0022

Summary

Illegal literal format, missing 'E'.

Description

The Compiler has encountered a floating point literal that is incorrectly specified.

Recommendation

Check to make sure that the floating point literal is specified correctly. (Note, however, that floating point numbers are only supported as integers during synthesis. The fractional part of a floating point number will be truncated).

Error Message: $Y0023

Summary

A number must not contain '_character'.

Description

The Compiler has encountered an invalid sequence of characters in a numeric literal. Numeric literals may include '_' (underscore) characters to improve readability, but must not include other, non-numeric characters. (Values entered in hexadecimal format may also include the characters 'A' through 'F' or 'a' through 'f').

Recommendation

Check to make sure that there are no invalid characters used in the numeric literal and that the '_' character is used properly.

Error Message: $Y0024

Summary

A number must not have 'character' as its last character.

Description

The Compiler has encountered an invalid character at the end of a numeric literal.

Recommendation

Check to make sure that there are no missing or additional delimiters (such as white space or new line) at the end of the number.

Error Message: $Y0025

Summary

An identifier may not contain consecutive under bars '__'.

Description

The Compiler has encountered an invalid sequence of characters in an identifier. Identifiers may include '_' (underscore) characters to improve readability, but they must not be used consecutively.

Recommendation

Check to make sure there are no extraneous consecutive '_' characters in the identifier.

Error Message: $Y0026

Summary

An identifier may not contain a 'character'.

Description

The Compiler has encountered an invalid character in an identifier. Identifiers may consist of letters, digits, and '_' (underscore) characters but must not include other special or non-graphic characters.

Recommendation

Check to make sure that there are no invalid characters used in the identifier. Also consider using the extended identifier syntax; an extended identifier may contain any graphic character. Extended identifiers have a backslash (\) as their first and last character. Also note that extended identifiers are case sensitive.

Error Message: $Y0027

Summary

An identifier may not have '_' as its last character.

Description

The Compiler has encountered an invalid sequence of characters in an identifier. Identifiers may include '_' (underscore) characters to improve readability, but the '_' character must not be used as the first or last character in the identifier.

Recommendation

Check to make sure that the '_' character is not used as the last character in the identifier.

Error Message: $Y0028

Summary

A character literal must not contain a non-graphic character.

Description

The Compiler has encountered a quoted character that is not a graphic character.

Recommendation

Check to make sure that the text editor you have used to create the source file has not placed illegal characters (such as word processor control codes) into your source file.

Error Message: $Y0029

Summary

'mm': unknown command option 'name'.

Description

The Compiler software has been invoked with an unknown command option.

Recommendation

Check the Compiler documentation for information about compiler options and option formats.

Error Message: $Y0030

Summary

Unable to create a temporary file.

Description

The Compiler has encountered a system error while attempting to write a file to the disk.

Recommendation

Check to make sure that you have sufficient space on the disk. Also check to make sure the disk drive is not write protected or a read-only device. If you have a networked system, check to ensure that you have adequate network privileges.

Error Message: $Y0031

Summary

Unable to open a temporary file.

Description

The Compiler has encountered a system error while attempting to open an existing temporary file.

Recommendation

Check to make sure the disk drive or network directory is available. If you have a networked system, check to make sure that you have adequate network privileges.

Error Message: $Y0032

Summary

Unable to write to a temporary file.

Description

The Compiler has encountered a system error while attempting to write a file to the disk.

Recommendation

Check to make sure that you have sufficient space on the disk. Also check to make sure the disk drive is not write protected or a read-only device. If you have a networked system, check to ensure that you have adequate network privileges.

Error Message: $Y0033

Summary

Out of memory.

Description

The Compiler has encountered a system error while attempting to allocate memory.

Recommendation

Synthesis software can require large amounts of memory, depending on the size of the design. Check your design to ensure that you have not described a circuit that is impractical to synthesize (such as one that includes very large array or integer ranges, or describes complex mathematical functions).

Also check to ensure that your system has adequate physical memory and that there is enough free memory to run the synthesis software.

If your design is very large, you should consider partitioning it into multiple, smaller design modules and synthesize those modules independently.

Error Message: $Y0034

Summary

Disk is full.

Description

The Compiler has encountered a system error when attempting to write a file to the disk.

Recommendation

Check to make sure that you have adequate disk space.

Error Message: $Y0035

Summary

Software security protection check failed.

Description

The Compiler was unable to find the software security device.

Recommendation

Check to make sure that the software security device is connected properly before running the software.

Error Message: $Y0036

Summary

Design too large for Demonstration version.

Description

The Compiler is operating in demonstration mode. In this mode, you are restricted in the size of design that can be processed.

Recommendation

Check to make sure that the number of semicolons in your design is within the restriction imposed by the demonstration version. If you are not intending to run the software in demonstration mode, check to ensure that the software security device is properly attached.

Error Message: $Y0082

Summary

Unable find package 'standard' in the file `std.vhd`.

Description

The Compiler has encountered a problem in the standard library file, `std.vhd`.

Recommendation

Check to make sure the `std.vhd` file has not become corrupted. If necessary, re-install the `std.vhd` file from the installation disk.

Error Message: $Y0083

Summary

Entity 'name' does not exist in the design.

Description

The Compiler was unable to find the indicated entity in the specified input source files.

Recommendation

Check to make sure that you have specified the top-level entity correctly. Check also to make sure you have specified all necessary source files and that the desired top-level entity exists.

Error Message: $Y0084

Summary

Architecture 'name' does not exist in the design.

Description

The Compiler was unable to find the indicated architecture in the specified input source files.

Recommendation

Check to make sure that you have specified the top-level architecture correctly.

Also check to make sure you have specified all necessary source files and that the desired top-level architecture exists.

Error Message: $Y0085

Summary

Input file and output file have the same name.

Description

The Compiler has determined that the input and output file names you have specified are the same.

Recommendation

Check to make sure that you have specified the correct file names for input and output files and check to make sure you have specified the correct file name extensions.

Error Message: $Y0086

Summary

Incorrect version of library STD.

Description

The Compiler has encountered a problem in the standard library file, `std.vhd`. The version of the file is not correct.

Recommendation

Check to make sure the `std.vhd` file has not become corrupted. If necessary, re-install the `std.vhd` file from the installation disk.

Also check to make sure you do not have an old version of `std.vhd` somewhere on your path, or in your project directory.

Error Message: $Y0087

Summary

Incorrect version of library METAMOR.

Description

The Compiler has encountered a problem in the standard library file, `metamor.vhd`. The version of the file is not correct.

Recommendation

Check to make sure the `metamor.vhd` file has not become corrupted. If necessary, re-install the `metamor.vhd` file from the installation disk.

Also check to make sure you do not have an old version of `metamor.vhd` somewhere on your path, or in your project directory.

Error Message: $Y0088

Summary

Install error, file 'name' is missing.

Description

The Compiler has encountered a missing file, this file is part of the software and must be present.

Recommendation

If the directory containing the file is on a networked drive, check that the drive is shared. Once you verify that the file is missing, re-install the software.

Error Message: $Y0089

Summary

Install error, file 'name' is incorrect version.

Description

The Compiler has encountered a file that is part of the software but is from another version of the software. This file is incompatible and must be replaced.

Recommendation

Re-install the software.

100-series Error Messages

Error Message: $Y0100

Summary

A description 'name' is used in an expression as a primary, expected a signal, a variable, or a constant.

Description

The Compiler has encountered a primary expression element that is not a legal as part of an expression. An object of class signal, variable or constant was required. These objects include signals, variables, constants, generics, enumerated type elements, functions and attribute values.

Recommendation

Check to make sure that you have correctly specified the expression.

Also check to make sure the indicated name has been declared and is not hidden by another declaration.

Error Message: $Y0101

Summary

'name' has not been declared as a 'description'.

Description

The Compiler has encountered a component instantiation that does not reference a known component, entity or configuration.

Recommendation

Check to make sure that you have provided a component declaration for the indicated component.

If the component declaration exists in a package, make sure you have provided the necessary **use** statement to make the contents of that package visible. If this is a direct instantiation, check that the keywords **entity** or **configuration** are not missing.

Error Message: $Y0102

Summary

Mode conflict associating actual 'name' with formal 'name'.

Description

The Compiler has determined that the mode (direction) of the actual parameter indicated is not compatible with the mode of the formal parameter.

For example, you cannot connect an actual that is itself an **out** port, to a formal that is an **inout** port.

Recommendation

Check to make sure that the mode specified in the component declaration is compatible with the mode of the actual parameter.

Check to make sure the mode on the component declaration is the same as the mode on its entity port declaration.

Also check to make sure you have associated the actual parameters to formal parameters as expected. A mode conflict is actually an electrical rules check and usually indicates a design error. It is often possible to work around this error using a temporary signal as the actual.

Error Message: $Y0103

Summary

No actual is specified for generic 'name'.

Description

The Compiler has encountered an incomplete generic mapping. The actual generic value is missing in the generic map.

Recommendation

Check to make sure that all required generic parameters have been specified, or add a default value to the declaration of this generic.

Error Message: $Y0104

Summary

Port 'name' has mode **IN**, is unconnected and has no explicit default value.

Description

The Compiler has determined that the indicated port of an entity has been left unspecified or specified as open. Input ports that do not have default values must be connected.

Recommendation

Check to make sure that all necessary input ports have been specified with actual parameters, or add default values to those ports that will be left unconnected.

Error Message: $Y0105

Summary

Port 'name' has mode description has a type that is unconstrained and may not be unconnected.

Description

The Compiler has encountered a port mapping that is invalid, due to the use of a formal port that has an unconstrained array type. All ports that have unconstrained types must be connected.

Recommendation

Check to make sure that you have described the intended port mapping and have not inadvertently omitted one or more ports from the port map or specified this port as open.

Error Message: $Y0106

Summary

Block specification must be an Architecture, Block label, or Generate label.

Description

The Compiler has encountered a configuration that references an invalid design unit type or other unknown label.

Recommendation

Check to make sure that the block specification in the configuration specifies a valid architecture, block label or generate label.

Error Message: $Y0107

Summary

'name' is not an Entity.

Description

The Compiler expected an entity name in a direct instantiation of an entity or in a configuration statement, but has instead encountered an identifier that is not a known entity name, or that has been declared as some other type of design unit or object.

Recommendation

Check to make sure that you have entered the name of the entity correctly.

Also check to make sure you have not used the same name to identify a local signal or other object and that the entity is made visible with a **use** statement or with a selected name such as work.entity.

Error Message: $Y0108

Summary

'name' is not a Type or Subtype.

Description

The Compiler expected a type or subtype name, but has instead encountered an identifier that is not a type or subtype.

Recommendation

Check to make sure the type or subtype has been entered properly.

Also check to make sure the type or subtype has been declared correctly and is visible in the current region of the design. If the type or subtype declaration was made within a package, make sure you have provided the appropriate **use** statement to make that declaration visible.

Error Message: $Y0109

Summary

A **Return** statement in a procedure must not return an expression.

Description

The Compiler has encountered a **return** statement within a procedure that includes a return value. Return values are not allowed in procedures.

Recommendation

Check to make sure that a procedure is what you really intended to create. If you need to return values from a procedure, you will need to use procedure parameters of mode **out** or **inout**, or replace the procedure with a function.

Error Message: $Y0110

Summary

A **Return** statement in a function must return an expression.

Description

The Compiler has encountered a **return** statement within a function that does not specify a return value. Functions must be provided with return values at all possible exit points.

Recommendation

Check to make sure that all **return** statements within the function have valid return values.

Error Message: $Y0111

Summary

Operator function has too few parameters.

Description

The Compiler has encountered an operator function (overloaded operator) that does not have the required number of parameters for the specified operator.

Recommendation

Check to make sure that the number of function parameters matches the requirements of the specified operator.

Error Message: $Y0112

Summary

Operator function has too many parameters.

Description

The Compiler has encountered an operator function (overloaded operator) that does not have the required number of parameters for the specified operator.

Recommendation

Check to make sure that the number of function parameters matches the requirements of the specified operator.

Error Message: $Y0113

Summary

Cannot type convert a NULL, an aggregate, or a string literal.

Description

The Compiler has encountered a type conversion that is invalid. Explicit type conversions are only allowed between closely related types, such as between arrays with the same dimensions. Explicit type conversions are not allowed for a null, aggregate or string literals.

Recommendation

Check to make sure that the explicit type conversion is being used for closely related types, or use a type conversion function.

Error Message: $Y0114

Summary

'name' has no Architecture named 'name'.

Description

The compiler was unable to find the specified architecture name.

Recommendation

Check to make sure that the correct architecture name has been used.

Also check to ensure that the specified architecture exists in the design source files.

Error Message: $Y0115

Summary

'name' is already declared as a description.

Description

The Compiler has encountered a duplicate declaration for the indicated identifier name.

Recommendation

Check to make sure that you are specifying the correct identifier name and that the name is unique in this declarative region.

Remove one of the duplicate declarations.

Error Message: $Y0116

Summary

Name at end of 'description' does not match 'description' name.

Description

The Compiler has encountered a mismatched name at the end of a design unit, subprogram or other end-terminated section of the design.

Recommendation

Check to make sure that you have used the correct name at the end of the section.

Also check to make sure that you have not omitted one or more **end** statements.

Error Message: $Y0117

Summary

Block configuration must be an Architecture.

Description

The Compiler has encountered an invalid binding of a block with an architecture in a configuration statement or declaration.

Recommendation

Check to make sure that the name specified in the block configuration is an architecture and that the architecture specified exists in the design.

Error Message: $Y0119

Summary

Unable to determine the range of a non-scalar type.

Description

The Compiler has encountered a problem when attempting to determine the range of a non-scalar (composite) data type such as a record that has no range.

Recommendation

Check to make sure that a range is actually needed, or rewrite the design so that a scalar data type is used.

Error Message: $Y0120

Summary

Illegal subtype constraint.

Description

The Compiler has encountered a subtype declaration or usage that is illegal, due to an incorrect constraint specification. The constraint (such as a range specifier) must match the requirements of the base type.

Recommendation

Check to make sure that the subtype and base type are compatible with the constraint specified.

Error Message: $Y0121

Summary

'name' is not an array.

Description

The Compiler expected an array object or literal.

Recommendation

Check to make sure that the object or literal you are specifying is an array type.

Error Message: $Y0123

Summary

Attempt to select element of an object whose type is not a record.

Description

The Compiler has encountered an invalid use of a record field specifier. The object referenced in the statement is not a record type.

Recommendation

Check to make sure that the object you are specifying is a record type of object. If you did not intend to specify a record field, check to make sure you have not inadvertently used a '**.**' operator or other record-related syntax.

Error Message: $Y0124

Summary

'name' does not conform to declaration in package.

Description

The Compiler has encountered an invalid declaration in a package body. The declaration for the indicated name must match the declaration in the corresponding package.

Recommendation

Check to make sure that you have specified the declaration properly in the package body. If the declaration is for a subprogram, check to make sure the parameters are correctly specified and have matching class, mode, type and names.

Also check to make sure the specified identifier has been properly declared (as a prototype) in the package.

Error Message: $Y0125

Summary

'name' is not a Physical Unit.

Description

The Compiler has encountered an apparent use of a physical type literal that does not specify a valid physical type unit.

Recommendation

Check to make sure that the physical type definition includes the physical unit you have specified. If you did not intend to specify a physical type literal, check to make sure you have not inadvertently omitted an operator or other language element from the statement.

Error Message: $Y0126

Summary

description 'name' may not be a prefix for .ALL.

Description

The Compiler has encountered a .**all** specification (such as in a **use** statement) that is not valid. The .**all** keyword may only be prefixed with a package, library, entity or architecture.

Recommendation

Check to make sure that you have specified a valid package, library, entity or architecture name in the statement.

Error Message: $Y0127

Summary

Physical unit prefix must be a number.

Description

The Compiler has encountered an apparent use of a physical type literal that does not specify a valid physical type prefix value. Physical type prefix values must be numbers.

Recommendation

Check to make sure that the physical type definition includes a valid numeric prefix. If you did not intend to specify a physical type literal, check to make sure you have not inadvertently omitted an operator or other language element from the statement.

Error Message: $Y0128

Summary

Range is not within the range of the base type.

Description

The Compiler has encountered an invalid specification of a range. Range specifications must specify ranges of values that are within the range of the specified base type.

Recommendation

Check to make sure that the correct base type has been referenced and check to make sure that the range specified falls within the range of the base type.

Error Message: $Y0129

Summary

Illegal NULL in expression, NULL must be in a simple assignment.

Description

The Compiler has encountered an illegal use of NULL. When used as a value, NULL may only be used in the right hand side of a simple assignment and may not appear within an expression.

Recommendation

Check to make sure that NULL was really intended in the expression. You may be able to simplify the expression to a simple assignment by using a selected assignment or similar statement.

Error Message: $Y0130

Summary

Others must be the last choice in a selected signal assignment.

Description

The Compiler has encountered an illegal use of the choice **others**. **Others** is only allowed as the last choice in a series of choices.

Recommendation

Check to make sure that the **others** choice is at the end of the series of choices.

Error Message: $Y0131

Summary

Others must be the last choice in a **case** statement.

Description

The Compiler has encountered an illegal use of the choice **others**. **Others** is only allowed as the last choice in a **case** statement.

Recommendation

Check to make sure that the **others** choice is at the end of the **case** statement.

Error Message: $Y0132

Summary

Others must be the only choice in a selected alternative.

Description

The Compiler has encountered an illegal use of the choice **others**, it may not be or'd with another choice. For example, a case of the following form is illegal:

```
when '000' | others => .....
```

Recommendation

Check to make sure that the **others** choice is the only choice in the selected alternative. You can probably remove the or'd choice.

Error Message: $Y0133

Summary

Others must be the only choice in a case alternative.

Description

The Compiler has encountered an illegal use of the choice **others**, it may not be or'd with another choice. For example, a case of the following form is illegal:

```
when '000' | others => .....
```

Recommendation

Check to make sure that the **others** choice is the only choice in the case alternative.

Error Message: $Y0134

Summary

The label at the end of the 'description' does not match 'description' label.

Description

The Compiler has encountered an **end** statement that references a concurrent statement label other than expected.

Recommendation

Check to make sure that you have terminated the concurrent statement with the correct label.

Error Message: $Y0135

Summary

An **Exit** statement must be within a **loop** statement.

Description

The Compiler has encountered an incorrect use of the **exit** statement. **Exit** is used to terminate execution of a **loop** and must be used within a loop.

Recommendation

Check to make sure that the **exit** statement is being used within a loop.

Also check to make sure you have not inadvertently terminated the loop prior to the **exit** statement with a misplaced **end** statement.

Error Message: $Y0136

Summary

An **Exit** statement specifies a label that is not a Loop label.

Description

The Compiler has encountered an **exit** statement that specifies an invalid loop label.

Recommendation

Check to make sure that the optional loop label has been correctly specified.

Check to make sure the loop (or loops) in which the **exit** statement is being used are correctly labeled.

Error Message: $Y0137

Summary

A **Next** statement specifies a label that is not a Loop label.

Description

The Compiler has encountered a **next** statement that specifies an invalid loop label.

Recommendation

Check to make sure that the optional loop label has been correctly specified.

Check to make sure the loop (or loops) in which the **next** statement is being used are correctly labeled.

Error Message: $Y0138

Summary

A **Return** statement must be within a Function or Procedure.

Description

The Compiler has encountered a **return** statement that is not within a function or procedure (subprogram). **Return** statements are used to exit from a subprogram and must not be used outside of a subprogram.

Recommendation

Check to make sure that the **return** statement is being properly used within a function or procedure.

Error Message: $Y0139

Summary

A passive process may not contain a signal assignment.

Description

The Compiler has encountered a process that is passive (such as one entered in the entity declaration) and has one or more signal assignments.

Recommendation

Check to make sure the process has been entered in the desired location of the source file. If the process is not intended to be passive, it must be located within an architecture declaration.

Error Message: $Y0140

Summary

Process has a sensitivity list and a **wait** statement.

Description

The Compiler has encountered a **wait** statement being used in a process that includes a sensitivity list. A process may not include both a **wait** statement and a sensitivity list.

Recommendation

Check the design requirements to determine if a sensitivity list is required. If you are creating a design intended for synthesis, you should consider using the sensitivity list in conjunction with appropriate conditional logic to define the behavior of the circuit. Remove either the sensitivity list or the **wait** statement to correct the problem.

Error Message: $Y0141

Summary

Illegal NULL in concurrent signal assignment.

Description

The Compiler has encountered an illegal assignment to NULL in a concurrent signal assignment. VHDL does not allow assignments of NULL in concurrent signal assignments.

Recommendation

Check to make sure that you really need to assign the signal to NULL. If you are attempting to describe an output enable, you should use the std_logic data type and assign the signal a value of 'Z', rather than NULL. If you require an assignment of NULL, modify the design so that the assignment is performed within a process or subprogram.

Error Message: $Y0142

Summary

Missing block guard expression or signal 'guard'.

Description

The Compiler has encountered an invalid or incomplete specification of a guarded assignment. A guarded assignment requires either a guarded block or implicit or explicit signal guard.

Recommendation

Check to make sure that a guard expression or the implicit signal guard has been specified for the guarded block. If guard is not an implicit signal, check to make sure it has been properly declared as a Boolean type.

Error Message: $Y0143

Summary

'guard' is not a signal.

Description

The Compiler has encountered an invalid use of the signal guard. Guard is not an implicit signal in this context and is not declared as a Boolean-type signal.

Recommendation

Check to make sure that you have not specified the wrong signal name.

Also check to make sure you have correctly declared the explicit guard signal.

Error Message: $Y0144

Summary

Signal 'guard' is not type 'boolean'.

Description

The Compiler has encountered an invalid use of the special signal guard. The condition expression of the guarded block does not evaluate to a Boolean result, or guard has been declared as a non-Boolean type.

Recommendation

Check to make sure that the condition expression evaluates to a Boolean result. If guard is explicitly declared and used in a guarded signal assignment, it must be declared as a Boolean.

Error Message: $Y0145

Summary

Target of un-guarded assignment is guarded.

Description

The Compiler has encountered an inconsistent use of a guarded assignment. The target of an assignment is guarded, but the **guarded** keyword has not been specified.

Recommendation

Check to make sure that you have specified the **guarded** keyword for all assignments to guarded signals.

Error Message: $Y0146

Summary

'name' is not a Procedure.

Description

The Compiler expected to encounter a procedure name, but the name specified is not a procedure.

Recommendation

Check to make sure that you have correctly entered the procedure name with the correct number of arguments, each of the correct type.

Also check to make sure there is no other local declaration that hides the procedure declaration and that the procedure declaration is visible in the current region of the design.

Error Message: $Y0147

Summary

Positional association must not follow named association.

Description

The Compiler has encountered an incorrect use of a port mapping or subprogram arguments. When positional association is used in combination with named association, the positional associations must be specified prior to any named associations.

Recommendation

Check to make sure that the ports have been specified in the correct order.

Also check to make sure you have not inadvertently omitted one or more named associations.

Error Message: $Y0148

Summary

Attribute 'name has not been declared.

Description

The compiler has encountered an attribute name that has not been declared.

Recommendation

Check to make sure that the attribute has been properly declared. If the attribute declaration is in a package, make sure the package has been properly loaded from the library and make sure the package contents have been made visible with a **use** statement.

Error Message: $Y0149

Summary

Attribute not defined for this object.

Description

The Compiler has encountered an attribute use that is not defined for the object the attribute is being applied to.

Recommendation

Check to make sure that the attribute has been defined for the type of the object. Use a type conversion, if necessary, to convert the object to the correct data type, or use an attribute that has been declared for the data type.

Error Message: $Y0150

Summary

Prefix for attribute 'base must be a type or subtype.

Description

The Compiler has encountered an invalid use of the predefined attribute 'base. The 'base attribute is used to find the base type for a subtype and so must only be applied to a type or subtype.

Recommendation

Check to make sure that the type specified is a subtype.

Error Message: $Y0151

Summary

Attribute 'attribute must not have a parameter.

Description

The Compiler has encountered an invalid use of a predefined attribute. The indicated attribute must not have a parameter.

Recommendation

Check to make sure that you are using the correct attribute.

Error Message: $Y0152

Summary

Attribute 'base must be the prefix of another attribute.

Description

The Compiler has encountered an incorrect use of the predefined 'base attribute.

Recommendation

'Base must be used in conjunction with another attribute, such as 'left, 'right, 'high, or 'low.

Check to make sure that you are using the attribute correctly.

Error Message: $Y0153

Summary

Attribute 'attribute may not have a parameter if the prefix is a scalar type.

Description

The Compiler has encountered a predefined attribute being used incorrectly with a parameter. The indicated attribute may not include an attribute parameter when used with scalar types.

Recommendation

Check to make sure that the attribute is being used correctly.

Error Message: $Y0154

Summary

Prefix of attribute 'attribute must be a discrete type.

Description

The Compiler has encountered a predefined attribute being used incorrectly. The indicated attribute requires a prefix that is a discrete type (an enumeration type or integer).

Recommendation

Check to make sure that the attribute is being used correctly and that the prefix is an enumeration type or integer.

Error Message: $Y0155

Summary

Attribute 'attribute must have a parameter.

Description

The Compiler has encountered an invalid use of a predefined attribute. The indicated attribute requires a parameter.

Recommendation

Check to make sure that you are using the correct attribute. Add an attribute parameter if necessary.

Error Message: $Y0157

Summary

Signal attribute prefix is not a signal.

Description

The Compiler has encountered a predefined attribute being used incorrectly. The indicated attribute requires a prefix that is a signal identifier.

Recommendation

Check to make sure that the attribute is being used correctly and that the prefix is a signal identifier.

Note that most signal attributes are not supported for synthesis.

Error Message: $Y0158

Summary

In an aggregate, positional associations must occur before named associations.

Description

The Compiler has encountered an incorrect use of an aggregate. When positional association is used in combination with named association, the positional associations must be specified prior to any named associations.

Recommendation

Check to make sure that the elements of the aggregate have been specified in the correct order. Also check to make sure you have not inadvertently omitted one or more named associations.

Error Message: $Y0159

Summary

Choice **Others** must only occur once in an aggregate.

Description

The Compiler has encountered more than one use of the **others** choice in an aggregate. **Others** may only be used once to define the default assignment in a aggregate.

Recommendation

Check to make sure that you have only provided one **others** choice in the aggregate.

Error Message: $Y0160

Summary

Choice **Others** must be last element of an aggregate.

Description

The Compiler has encountered an invalid use of the **others** choice in an aggregate. **Others** may only be used once to define the default assignment in a aggregate and must be the last element in the aggregate.

Recommendation

Check to make sure that you have only provided one **others** choice in the aggregate and that it is the last choice.

Error Message: $Y0161

Summary

Choice **Others** must be the only choice in an aggregate element association.

Description

The Compiler has encountered an illegal use of the choice **others** - it may not be or'd with another choice. For example, a case of the following form is illegal:

```
when 7 | others => .....
```

Recommendation

Check to make sure that the **others** choice is the only choice in the selected alternative. You can probably remove the or'd choice.

Error Message: $Y0162

Summary

Unable to 'action' library file 'name'.

Description

The Compiler was unable to perform an action (read or write) on the specified library file.

Recommendation

For read, check to make sure the library file exists and is located in the current working directory, in the library directory, or is correctly specified in the list of files in a library alias.

For write of a compiled library file, check to make sure you have write privileges in the specified directory.

Error Message: $Y0163

Summary

'name' is a 'description' and not a 'description'.

Description

The Compiler has encountered an unexpected use of the indicated identifier.

Recommendation

Check to make sure that the identifier has been entered correctly and is the expected type of object, design unit, loop, block, or subprogram.

Error Message: $Y0164

Summary

'name' is not a user defined attribute.

Description

The Compiler has encountered an invalid use of the indicated identifier. The name used is not declared as a user defined attribute.

Recommendation

Check to make sure that the attribute name has been correctly entered. If an attribute was not intended, check to make sure the ' (single quote) character has not been incorrectly used.

Error Message: $Y0165

Summary

Subtype has more dimensions than base type.

Description

The Compiler has encountered an invalid specification of a subtype. Array subtype declarations must specify ranges of values that are within the range of the specified base type and must have the same number of array dimensions.

Recommendation

Check to make sure that the correct base type has been referenced and that the dimensions of the subtype are compatible with the range of the base type.

Error Message: $Y0166

Summary

Subtype index is incompatible with base type index.

Description

The Compiler has encountered an invalid specification of a subtype. An array subtype index must specify a value that is within the range of the specified base type.

Recommendation

Check to make sure that the correct base type has been referenced and that the index of the subtype is compatible with the range of the base type.

Error Message: $Y0167

Summary

Base type of subtype must not be a record.

Description

The Compiler has encountered an invalid specification of a subtype. The base type of a subtype may not be a record.

Recommendation

Check to make sure that the correct base type has been referenced, and that the base type is not a record type.

Error Message: $Y0168

Summary

Base type for index constraint must be an array.

Description

The Compiler has encountered an invalid specification of an index constraint. An array index constraint may only be used when the base type is an array.

Recommendation

Check to make sure that the correct base type has been referenced.

Error Message: $Y0169

Summary

Base type of subtype must be an unconstrained array.

Description

The Compiler has encountered an invalid specification of an unconstrained array subtype. The base type of a subtype must be an unconstrained array.

Recommendation

Check to make sure that the correct base type has been referenced, or constrain the array subtype with a valid range.

Error Message: $Y0170

Summary

'name' was declared outside of the function in which it is used.

Description

The Compiler has encountered an object being referenced within a function, but that object was not declared within the function.

Recommendation

Check to make sure that you are referencing an object that is local to the function, or has been passed into the function via the parameter list.

Error Message: $Y0171

Summary

A function may not contain a **wait** statement.

Description

The Compiler has encountered a **wait** statement within a function. Functions may not include **wait** statements.

Recommendation

Note that **wait** statements are legal within procedures, but are not supported for synthesis.

Error Message: $Y0172

Summary

A function must be completed by a **return** statement.

Description

The Compiler has encountered a function that does not contain a **return** statement. Functions must have at least one **return** statement with a return value.

Recommendation

Check to make sure that a **return** statement is the last statement of the function and that the **return** statement is not dependent on an **if** statement or other conditional expression.

Error Message: $Y0173

Summary

The subtype indication given in the full declaration of 'name' must conform to that given in the deferred constant declaration.

Description

The Compiler has encountered a full constant declaration that is incompatible with the associated deferred constant declaration. The subtype indications specified for the deferred and full constant declarations are incompatible.

Recommendation

Check to make sure that the same subtype indications of the deferred and full constant declaration are compatible.

Error Message: $Y0174

Summary

Range of a physical type must be an integer.

Description

The Compiler has encountered a physical type declaration that includes non-integer units.

Recommendation

Check to make sure that the physical type declaration includes a valid base unit and that subsequent units are defined using an integer multiplier.

Error Message: $Y0175

Summary

Subprogram declaration 'name', does not have a corresponding body.

Description

The Compiler was unable to find a function or procedure body corresponding to the subprogram declaration indicated.

Recommendation

Check to make sure that you have provided a function or procedure body for the subprogram. If the subprogram has been declared within a package, make sure that a corresponding package body has been provided that includes the function or procedure body.

Also check to make sure the declarations in the package and package body match.

Error Message: $Y0176

Summary

Deferred constant declaration 'name', is not declared in a package body.

Description

The Compiler was unable to find a full constant declaration corresponding to the deferred constant declaration indicated.

Recommendation

Check to make sure that you have provided a full constant declaration for the indicated deferred constant.

Check to make sure that a corresponding package body has been provided for the package containing the deferred constant declaration.

Error Message: $Y0177

Summary

Return statement must be within a Function or Procedure.

Description

The Compiler has encountered a **return** statement that is not within a function or procedure body.

Recommendation

Check to make sure that you are using the **return** statement correctly to exit from a subprogram.

Also check to ensure that you have not incorrectly placed one or more **end** statements that would cause the subprogram to be prematurely terminated.

Error Message: $Y0178

Summary

Next statement must be within a **Loop** statement.

Description

The Compiler has encountered a **next** statement that is not within a loop.

Recommendation

Check to make sure that you are using the **next** statement correctly in the loop.

Also check to ensure that you have not incorrectly placed the **end loop** statement before the **next** statement.

Error Message: $Y0179

Summary

Reserved word **UNAFFECTED** may only appear as a waveform in a concurrent signal assignment.

Description

The Compiler has encountered an invalid use of the indicated keyword. The **unaffected** keyword is only allowed in waveforms that are part of a concurrent signal assignment. (Note also that only the first item in a waveform is supported in synthesis).

Recommendation

Check the proper use of the **unaffected** keyword and modify the design.

Error Message: $Y0180

Summary

Attempt to assign to a port with mode **IN**.

Description

The Compiler has encountered an invalid use of a port with mode **in**. It is not legal to assign values to ports that have been declared as mode **in**.

Recommendation

Check to make sure that you are assigning to the correct port in your design. If you need to assign a value to the port, use mode **inout** or **out**.

Error Message: $Y0181

Summary

Attempt to assign to a port with mode **LINKAGE**.

Description

The Compiler has encountered an invalid use of a port with mode **linkage**. It is not legal to assign values to ports that have been declared as mode **linkage**.

Recommendation

Check to make sure that you are assigning to the correct port in your design. If you need to assign a value to the port, use mode **inout** or **out**.

Error Message: $Y0182

Summary

Attempt to assign to implicit signal 'guard'.

Description

The Compiler has encountered an invalid use of the implicit signal guard. This signal is created as a result of a guard expression, and may not have a value assigned to it.

Recommendation

Check to make sure that you have not specified the wrong identifier name in the assignment. If you are attempting to modify the guard expression dynamically, you will need to rewrite the design so there are multiple guarded blocks specified with the required guard expressions.

Error Message: $Y0183

Summary

Attempt to assign to an alias of a port with mode **IN**.

Description

The Compiler has encountered an invalid use of a port with mode **in**. It is not legal to assign values to ports, or to aliases of ports, that have been declared as mode **in**.

Recommendation

Check to make sure that you are assigning to the correct port in your design. If you need to assign a value to the port, use mode **inout** or **out**.

Error Message: $Y0184

Summary

Attempt to assign to an alias of a port with mode **LINKAGE**.

Description

The Compiler has encountered an invalid use of a port with mode **linkage**. It is not legal to assign values to ports, or to aliases of ports, that have been declared as mode **linkage**.

Recommendation

Check to make sure that you are assigning to the correct port in your design. If you need to assign a value to the port, use mode **inout** or **out**.

Error Message: $Y0185

Summary

A description cannot be the target of a Signal assignment statement.

Description

The Compiler has encountered an invalid use of a signal assignment (<=). The target of a signal assignment must be a signal or port.

Recommendation

Check to make sure that the left side of the assignment is a signal or port. If you are assigning to a variable, use the variable assignment operator := .

Note, however, that variable assignments occur immediately within the process and signal assignments are executed at the end of a process. Such a substitution may change the behavior of your design.

Error Message: $Y0186

Summary

A description cannot be the target of a Variable assignment statement.

Description

The Compiler has encountered an invalid use of a variable assignment (:=). The target of a variable assignment must be a variable.

Recommendation

Check to make sure that the left side of the assignment is a variable. If you are assigning to a signal or port, use the signal assignment operator <= .

Note, however, that variable assignments occur immediately within the process and signal assignments are executed at the end of a process. Such a substitution may change the behavior of your design.

Error Message: $Y0187

Summary

Expected a static expression, description 'name' is illegal here.

Description

The Compiler has encountered an expression that is invalid in the current context. A static expression (one that can be evaluated at compile time and does not depend on a signal or variable) is expected.

Recommendation

Check to make sure the expression is static.

Error Message: $Y0188

Summary

Signal 'name' is not readable as it has mode **OUT**.

Description

The Compiler has encountered an invalid use of a port with mode **out**. It is not legal to read values of ports, or aliases of ports, that have been declared as mode **out**.

Recommendation

Check to make sure that you are specifying the correct port in your design. If you need to read the value of a port, use mode **buffer**. You could also consider mode **inout**. This, however, specifies bi-directional data flow and is often over-specification.

Error Message: $Y0189

Summary

'name' is not a static signal name.

Description

The Compiler has encountered an object or expression that is invalid in the current context. A static expression (one that can be evaluated at compile time and does not depend on a signal or variable) is expected.

Recommendation

Check to make sure that the object or expression is static.

Error Message: $Y0190

Summary

Enumerated type contains duplicate element 'name'.

Description

The Compiler has encountered two or more identical enumerated values in an enumerated type declaration.

Recommendation

Check to make sure that you have not incorrectly typed in the enumerated values for the type. Remove or rename the duplicate entries.

Error Message: $Y0191

Summary

Array type is not constrained.

Description

The Compiler has encountered an unsupported use of an unconstrained array type.

Recommendation

Check to make sure that all array types in your design are provided with valid array bounds (ranges).

Error Message: $Y0192

Summary

Function parameter must be mode **IN**.

Description

The Compiler has encountered an incorrect use of a function parameter.

Recommendation

All formal parameters of a function must be of mode **in** (which is the default mode) and may not be assigned values within the function. If you require that one or more parameters of your subprogram be of mode **out** or **inout**, then you will need to use a procedure, rather than a function.

Error Message: $Y0193

Summary

Package Body with no Package of same name.

Description

The Compiler has encountered a package body that does not correspond to any package in the design.

Recommendation

Check to make sure that a package has been provided corresponding to the package body and that the package and package body names are consistent and in the same library.

Error Message: $Y0194

Summary

Unable to determine type of array index.

Description

The Compiler has encountered an array index that is of an unknown type.

Recommendation

Check to make sure that the expression used for the array index results in a integer or other valid index value. Introducing an intermediate signal or variable can help to resolve data type ambiguities.

Error Message: $Y0195

Summary

Array must have an index constraint.

Description

The Compiler has encountered an array without an index constraint, in a context where unconstrained arrays are not allowed.

Recommendation

Check to make sure that the array is provided with an index constraint.

Error Message: $Y0196

Summary

Prefix of a selected name cannot be a slice name.

Description

The Compiler has encountered an invalid selected name. The prefix of a selected name must be a library, package, block, subprogram or record name.

Recommendation

Check to make sure that you have correctly specified the selected name. If a selected name was intended, check to make sure the prefix of the selected name is a valid selection name.

Error Message: $Y0197

Summary

Formal 'name' in port map does not exist in port declaration.

Description

The Compiler has encountered a named association within a component instantiation that does not match the port declaration for the specified component or lower-level entity.

Recommendation

Check to make sure that the named association has been correctly entered.

Also check the component declaration to ensure that the lower- level entity has been properly declared.

Error Message: $Y0198

Summary

Only the last entry in a group template declaration can include a <>.

Description

The Compiler has encountered an invalid specification of a group template declaration. When used in a group template declaration, only the last entry of the group can be a box (<>).

Recommendation

Check to make sure that you have specified a legal group template declaration and modify the entries accordingly.

200-series Error Messages

Error Message: $Y0200

Summary

Value of **when** expression is outside range of values of selected expression.

Description

The Compiler has encountered a **when** expression that does not match the possible values specified in the selected expression.

Recommendation

Check to make sure that the **when** expressions specified in the selected assignment are non-overlapping and fall into the range of possible values for the selection.

Error Message: $Y0201

Summary

Value of **when** expression is outside range of values of **case** expression.

Description

The Compiler has encountered a **when** expression that does not match the possible values specified in the **case** condition expression.

Recommendation

Check to make sure that the **when** expressions specified in the **case** condition expression are non-overlapping and fall into the range of possible values for the **case** statement.

Error Message: $Y0202

Summary

Operands have types that are incompatible with the operator 'operator'.

Description

The Compiler has encountered an expression that is not legal due to incompatibilities between the operand types and the operator used.

Recommendation

Check to make sure that the operand types have the required operations defined for them. If the operand types do not support the operator you are using, you can use a type conversion function to convert the operands to the required types, or write your own overloaded operator.

Error Message: $Y0203

Summary

'name' has not been declared.

Description

The Compiler has encountered an identifier that has not been declared, or has been declared but is not visible here.

Recommendation

Check to make sure that the indicated identifier has been declared and is visible where it is being referenced. If the identifier has been declared within a package, make sure the declaration has been made visible with a **use** statement.

Also check to make sure the name has not been hidden by another declaration.

Error Message: $Y0204

Summary

Operands of 'name' have incompatible types.

Description

The Compiler has encountered an expression that is not legal due to incompatibilities between the operand types and the operator being used.

Recommendation

Check to make sure that the operand types have the required operations defined for them. If the operand types do not support the operator you are using, you can use a type conversion function to convert the operands to the required types.

Error Message: $Y0205

Summary

Parameter associated with formal 'name' must be a Variable.

Description

The Compiler has encountered a subprogram parameter that must be a variable, but has not been declared as a variable. The actual and formal parameters of the subprogram do not match.

Recommendation

Check to make sure the subprogram actual parameters have been properly entered and that they match the subprogram formal parameters.

Error Message: $Y0206

Summary

Parameter associated with formal 'name' must be a Signal.

Description

The Compiler has encountered a subprogram parameter that must be a signal, but has not been declared as a signal. The actual and formal parameters of the subprogram do not match.

Recommendation

Check to make sure the subprogram actual parameters have been properly entered and that they match the subprogram formal parameters.

Error Message: $Y0207

Summary

Formal parameter 'name' and its actual parameter have incompatible types.

Description

The Compiler has encountered an actual parameter to a subprogram that is not legal due to incompatibilities between the actual parameter type and the formal parameter type.

Recommendation

Check to make sure that the actual and formal parameters have compatible types. If the types are different, you may be able to use a type conversion function to convert the operands to the required types.

Error Message: $Y0208

Summary

Block guard expression must be type boolean.

Description

The Compiler has encountered an invalid block guard expression in a **block** statement. An implicit guard signal is boolean type.

Recommendation

Check to make sure that the guard expression has a boolean type.

Error Message: $Y0209

Summary

Range of an integer type declaration must be some integer type.

Description

The Compiler has encountered an invalid range specification in an integer type declaration. The range must specify a valid integer range.

Recommendation

Check to make sure that the range has been correctly specified. Make sure the range is specified using integer values.

Error Message: $Y0210

Summary

Unable to determine type of attribute prefix.

Description

The Compiler has encountered an ambiguous prefix of an attribute. The type of the object prefix of the attribute is unknown.

Recommendation

Check to make sure that the attribute is being used in the intended way. You may be able to simplify the description and remove the type ambiguity by introducing an intermediate signal or variable of the correct type.

Error Message: $Y0211

Summary

Prefix of attribute 'attribute must not be a record.

Description

The Compiler has encountered an invalid use of the indicated attribute.

Recommendation

Check to make sure that the attribute prefix is of the intended type and that the correct attribute is being used.

Error Message: $Y0212

Summary

Parameter of an array attribute must be a universal integer.

Description

The Compiler has encountered an incorrect use of an array attribute.

Recommendation

Check to make sure that the array attribute parameter is a universal integer value.

Error Message: $Y0213

Summary

Operand has a type that is incompatible with the operator 'operator'.

Description

The Compiler has encountered an expression that is not legal due to incompatibilities between the operand types and the operator being used.

Recommendation

Check to make sure that the operand types have the required operations defined for them. If the operand types do not support the operator you are using, you can use a type conversion function to convert the operands to the required types.

Error Message: $Y0214

Summary

Exponent is negative, left operand must be a floating point type.

Description

The Compiler has encountered an invalid use of an exponent. When the exponent of an expression is negative, the operand must be a floating point object or literal.

Recommendation

Check to make sure that the left operand is a floating type value, or use a type conversion to convert the value to floating point.

Error Message: $Y0215

Summary

No parameter associated with formal parameter 'name'.

Description

The Compiler has encountered an incorrect use of a procedure or function. One or more of the required actual parameters are missing.

Recommendation

Check to make sure that you have specified all required parameters to the procedure or function.

Error Message: $Y0216

Summary

There are more actual parameters than formal parameters.

Description

The Compiler has encountered an incorrect use of a procedure or function. Too many actual parameters have been specified.

Recommendation

Check to make sure that you have specified the correct number and type of required parameters when invoking the procedure or function.

Error Message: $Y0217

Summary

Expected a Procedure and not a Function.

Description

The Compiler has encountered a function being used when a procedure was expected. Procedures must be used when no return value is expected.

Recommendation

Check to make sure that the subprogram you have invoked is declared as a procedure, or use the subprogram in such a way that the return value is used.

Error Message: $Y0218

Summary

Expected a Function and not a Procedure.

Description

The Compiler has encountered a procedure being used when a return value was required. Procedures do not have return values.

Recommendation

Check to make sure that the subprogram is written as a function, rather than a procedure, or modify the use of the subprogram so that a return value is not required.

Error Message: $Y0219

Summary

Function returns an incompatible type.

Description

The Compiler has encountered an incompatible use of a function. The types required in the expression and the type declared for the function do not match.

Recommendation

Check to make sure that the types are compatible. Use a type conversion function if necessary to convert the returned function value to the appropriate type.

Error Message: $Y0220

Summary

No procedure definition matches 'name'.

Description

The Compiler has encountered a call to a procedure that does not exist, or that is not visible in the current region of the design.

Recommendation

Check to make sure that the procedure has been declared properly and that the declaration is visible. If the procedure was declared in a package, you must include a **use** statement prior to the current design unit to make the declaration visible.

Procedures may be overloaded; check that the number and type of the actual arguments match one of the formal declarations of the procedure.

Error Message: $Y0221

Summary

No function definition matches 'name'.

Description

The Compiler has encountered a call to a function that does not exist, or that is not visible in the current region of the design.

Recommendation

Check to make sure that the function has been declared properly and that the declaration is visible. If the function was declared in a package, you must include a **use** statement prior to the current design unit to make the declaration visible. Functions may be overloaded, check that the number and type of the actual arguments match one of the formal declarations of the function.

Error Message: $Y0222

Summary

No actual associated with formal 'name'.

Description

The Compiler has encountered an incorrect use of a procedure or function. One or more of the required actual parameters are missing.

Recommendation

Check to make sure that you have specified all required parameters to the procedure or function.

Error Message: $Y0223

Summary

More than one association specified for formal parameter 'name'.

Description

The Compiler has encountered an incorrect use of a procedure or function. One or more of the formal parameters has been incorrectly referenced in a named association, or there is more than one actual parameter associated.

Recommendation

Check to make sure that you have specified all required parameters to the procedure or function.

Error Message: $Y0224

Summary

The aggregate has an incompatible type in this context.

Description

The Compiler has encountered an illegal use of an aggregate. One or more aggregate elements are not of the correct type.

Recommendation

Check to make sure that the aggregate is of the correct format for the intended usage. The type of an aggregate is determined from the context, check that the type of the aggregate is clear in this context.

Error Message: $Y0225

Summary

The string has an incompatible type in this context.

Description

The Compiler has encountered an illegal use of a string literal. An element of the string is not of the correct type.

Recommendation

Check to make sure that the string is of the correct format for the intended usage and that the type of the elements of the string can be distinguished in this context. The type of a string is determined from the context, check that the type of the string is clear in this context.

Error Message: $Y0226

Summary

The bit string has an incompatible type in this context.

Description

The Compiler has encountered an illegal use of a bit string literal. An element of the bit string is not of the correct type.

Recommendation

Check to make sure that the string is of the correct format for the intended usage as a bit string, and that the type of the elements of the string can be distinguished in this context. The type of a bit string is determined from the context, check that the type of the bit string is clear in this context.

Error Message: $Y0227

Summary

The direction of the slice is not the same as the direction of the prefix.

Description

The Compiler has encountered an index range that does not match the direction of the array prefix.

Recommendation

Check to make sure that the declaration of the array matches (in terms of direction, either **to** or **downto**) the range specified in the array slice.

Error Message: $Y0228

Summary

Unable to resolve overloaded procedure 'name'.

Description

The Compiler has encountered a procedure that has two or more possible declarations, but is unable to determine which procedure declaration is intended due to ambiguous parameter types.

Recommendation

Check to make sure that the parameter types are clearly specified. Introducing intermediate variables or signals can help to resolve ambiguous types.

Also check that the overloaded procedure declarations do not have arguments with the same type profile. Overloaded procedures are resolved based on the type of the arguments.

Error Message: $Y0229

Summary

Unable to determine type of attribute parameter.

Description

The Compiler has encountered an attribute parameter that is of an ambiguous type.

Recommendation

Check to make sure that the type of the attribute parameter is clearly distinguished.

Error Message: $Y0230

Summary

Prefix of attribute 'attribute must be a scalar type.

Description

The Compiler has encountered an illegal use of an attribute. The indicated attribute is only allowed for scalar (integer, real, physical or enumerated) types.

Recommendation

Check to make sure that the attribute is being applied to a scalar type.

Error Message: $Y0231

Summary

Prefix of attribute 'attribute must be an array.

Description

The Compiler has encountered an illegal use of an attribute. The indicated attribute is only allowed for array data types.

Recommendation

Check to make sure that the attribute is being applied to an array data type.

Error Message: $Y0232

Summary

Attribute parameter value exceeds dimensionality of array.

Description

The Compiler has encountered an illegal use of a parameterized attribute. The value of the attribute parameter must fall within the range of the prefix array.

Recommendation

Check to make sure the attribute parameter matches the corresponding array type declaration.

Error Message: $Y0233

Summary

An **If** statement condition expression must be type boolean.

Description

The Compiler has encountered a condition expression in an **if** statement. The condition expression used in an **if** statement must evaluate to a Boolean (True or False) value.

Recommendation

Check to make sure that the expression will evaluate to a Boolean value. If you are testing a binary value (such as a bit type signal), you should use the relational operator '=' to create a Boolean result.

Error Message: $Y0234

Summary

Wait until expression must be type boolean.

Description

The Compiler has encountered an invalid **until** expression in a **wait** statement. The expression used in a **wait until** statement must evaluate to a Boolean (True or False) value.

Recommendation

Check to make sure that the expression will evaluate to a Boolean value. If you are testing a binary value (such as a bit type signal), you should use the relational operator '=' to create a Boolean result.

Error Message: $Y0235

Summary

Select expression must be an integer type, enumerated type, or an array.

Description

The Compiler has encountered an invalid use of a **select** expression. The **select** expression must be an integer, enumerated type or array.

Recommendation

Check to make sure that you have specified a valid **select** expression and that the expression evaluates to an integer, enumerated type or array.

Error Message: $Y0236

Summary

Case expression must be an integer, enumerated type, or an array.

Description

The Compiler has encountered an invalid use of a **case** expression. The **case** expression must be an integer, enumerated type or array.

Recommendation

Check to make sure that you have specified a valid **case** expression and that the expression evaluates to an integer, enumerated type or array.

Error Message: $Y0237

Summary

Select expression must not be a multi-dimensional array.

Description

The Compiler has encountered an invalid use of a **select** expression. The **select** expression must be an integer, enumerated type or single-dimension array.

Recommendation

Check to make sure that you have specified a valid **select** expression and that the expression evaluates to an integer, enumerated type or single-dimension array.

Error Message: $Y0238

Summary

Case expression must not be a multi-dimensional array.

Description

The Compiler has encountered an invalid use of a **case** expression. The **case** expression must be an integer, enumerated type or single-dimension array.

Recommendation

Check to make sure that you have specified a valid **case** expression and that the expression evaluates to an integer, enumerated type or single-dimension array

Error Message: $Y0239

Summary

Unable to determine type of **With** expression from context.

Description

The Compiler has encountered a **with** expression with an unknown type.

Recommendation

Check to make sure that the **with** expression has been clearly specified. If necessary, introduce one or more intermediate signals to clearly distinguish the types of the expression elements.

Error Message: $Y0240

Summary

Unable to determine type of **Case** expression from context.

Description

The Compiler has encountered a **case** expression with an unknown type.

Recommendation

Check to make sure that the **case** expression has been clearly specified. If necessary, introduce one or more intermediate signals to clearly distinguish the types of the expression elements.

Error Message: $Y0241

Summary

The type of a **With** expression must be locally static.

Description

The Compiler has encountered a **with** expression that has a type that is not locally static.

Recommendation

Check to make sure that the type of the **with** expression is locally static.

Error Message: $Y0242

Summary

The type of a **Case** expression must be locally static.

Description

The Compiler has encountered a **case** expression that has a type that is not locally static.

Recommendation

Check to make sure that the type of the **case** expression is locally static.

Error Message: $Y0243

Summary

Loop range must be an integer type or an enumerated type.

Description

The Compiler has encountered an invalid range in a **loop** statement. Loops ranges must be integer or enumerated types.

Recommendation

Check to make sure that you have correctly specified the loop range.

Error Message: $Y0244

Summary

Assert condition must be type 'boolean'.

Description

The Compiler has encountered an invalid condition in an **assert** statement. The condition of an **assert** statement must evaluate to a Boolean type.

Recommendation

Check to make sure that you have correctly specified the **assert** statement. If the **assert** expression is an object name, check to make sure the object has been declared as type Boolean. If necessary, use the '=' comparison operator to create a Boolean expression.

Error Message: $Y0245

Summary

Assert severity must be type 'severity_level'.

Description

The Compiler has encountered an invalid use of the **assert severity** statement. The severity value must be specified using the type severity_level (note, warning, error, failure).

Recommendation

Check to make sure that you have correctly specified the value of the **assert severity**.

Error Message: $Y0246

Summary

Assert report must be type 'string'.

Description

The Compiler has encountered an invalid use of the **assert report** statement. The **report** keyword must be followed by a valid string.

Recommendation

Check to make sure that you have correctly specified the **assert report** string.

Error Message: $Y0247

Summary

Shift or rotate right operand must be type 'integer'.

Description

The Compiler has encountered an invalid use of the shift or rotate operator. Only integer values are allowed as the shift distance (right operand).

Recommendation

Check to make sure that you have correctly specified the shift operation. Check also to make sure the right operand evaluates to an integer type.

Error Message: $Y0248

Summary

Unable to resolve overloaded function 'name'.

Description

The Compiler has encountered an overloaded function that cannot be resolved due to ambiguous parameter types or other conditions.

Recommendation

Check to make sure that the parameters of the function are clearly distinguished in terms of their types.

Also check that the overloaded function declarations do not have arguments with the same type profile. Overloaded functions are resolved based on the type of the arguments.

Error Message: $Y0249

Summary

Others is illegal here because aggregate is associated with an unconstrained array.

Description

The Compiler has encountered an illegal use of the **others** choice. The aggregate being specified includes an unconstrained array.

Recommendation

Check to make sure that an unconstrained array was actually intended.

Error Message: $Y0250

Summary

Record aggregate contains too many elements.

Description

The Compiler has encountered a record aggregate that is invalid, due to too many elements being specified.

Recommendation

Check to make sure that the declaration of the record matches its use in the record aggregate.

Error Message: $Y0251

Summary

Others must represent at least one element.

Description

The Compiler has encountered an **others** clause that does not represent any possible elements.

Recommendation

Check to make sure that the **others** clause is actually needed.

Error Message: $Y0252

Summary

Others represents choices of record elements of different types.

Description

The Compiler has encountered an invalid use of an **others** clause. The record elements specified by the **others** clause do not match.

Recommendation

Check to make sure that the **others** clause is being used correctly.

Error Message: $Y0253

Summary

Record aggregate contains unknown named association.

Description

The Compiler has encountered a record aggregate that includes a named association that is not valid.

Recommendation

Check to make sure that the named associations have been properly entered and all names used in the association are valid.

Error Message: $Y0254

Summary

Operands of 'operator' have incompatible lengths.

Description

The Compiler has encountered an invalid expression using the indicated operator. The operands of the expression do not match.

Recommendation

Check to make sure that the correct operands have been specified and that they have the type and length required.

Error Message: $Y0255

Summary

Array name and range index value number is out of range of the bounds range.

Description

The Compiler has encountered an array index that is invalid. The index is outside the range of the array declaration. For arrays with more than one index the number of the invalid index is also displayed.

Recommendation

Check to make sure that the declaration of the array matches the use of the array.

Error Message: $Y0256

Summary

Duplicate association in array aggregate, it is a duplicate of association on line 'number'.

Description

The Compiler has encountered an array aggregate association that has already been specified.

Recommendation

Check to make sure that the array association has been correctly entered. Check the duplicate association indicated for more information.

Error Message: $Y0257

Summary

Duplicate choice in selected signal assignment, it is a duplicate of choice on line 'number'.

Description

The Compiler has encountered a choice in a selected signal assignment that has already been specified.

Recommendation

Check to make sure that the choice has been correctly entered.

Error Message: $Y0258

Summary

Duplicate choice in **case** statement, it is a duplicate of choice on line 'number'.

Description

The Compiler has encountered a choice in a **case** statement that has already been specified.

Recommendation

Check to make sure that the choice has been correctly entered.

Error Message: $Y0259

Summary

Choice **Others** is required when selected signal assignment expression is a universal integer.

Description

The Compiler has encountered an incomplete selected signal assignment. The use of a universal integer has resulted in an **others** choice being required.

Recommendation

Check to make sure that an **others** choice has been provided, or do not use a universal integer.

Error Message: $Y0260

Summary

Choice **Others** is required when **case** statement expression is a universal integer.

Description

The Compiler has encountered a **case** statement that does not include a required **others** choice due to the use of a universal integer.

Recommendation

Check to make sure that a universal integer is really what you intend in the **case** expression. Add an **others** choice to the **case** statement to cover the unspecified conditions.

Error Message: $Y0261

Summary

Missing choice in selected signal assignment.

Description

The Compiler has encountered a selected signal assignment that does not include all possible choices. Selected signal assignments must include all possible choices.

Recommendation

Check to make sure that you have included all possible choices in the selected signal assignment, or add the **others** choice to define a default choice.

Error Message: $Y0262

Summary

Missing choice in **case** statement.

Description

The Compiler has encountered an incompletely specified **case** statement. **Case** statements must cover all possible input choices, or include the **others** choice to provide a default choice.

Recommendation

Check to make sure that all possible choices are included in the **case** statement, or add an **others** choice.

Error Message: $Y0263

Summary

The value of the choice is outside the range of array elements.

Description

The Compiler has encountered a choice in a **case** statement that does not fall in the range of possible values specified in the selection expression.

Recommendation

Check to make sure that the selection expression and choices in the **case** statement are compatible.

Error Message: $Y0264

Summary

Elements of an array aggregate must be either all positional or all named.

Description

The Compiler has encountered an array aggregate that is composed of both positional association and named association for its elements. Aggregates that use named association for any of their elements must use named association for all elements.

Recommendation

Check to make sure that you have not inadvertently omitted the named association for one or more elements of the aggregate.

Error Message: $Y0265

Summary

Too few elements in array aggregate.

Description

The Compiler has encountered an array aggregate that does not match the usage. The number of elements in the array aggregate is incorrect.

Recommendation

Check to make sure that source and destination array aggregates match, in terms of the number and types of their elements.

Error Message: $Y0266

Summary

Too few elements in string.

Description

The Compiler has encountered a string that is not valid in the current context.

Recommendation

Check the format of the string and check to ensure that it matches the intended usage.

Error Message: $Y0267

Summary

Too few elements in bit string.

Description

The Compiler has encountered a bit string that does not match (in terms of size) the objects used in an expression or assignment.

Recommendation

Check to make sure that the bit string contains the correct number of bit characters. If you have entered the bit string using an alternate (non-binary) format, check to ensure that the bit string represents the expected number of bits when analyzed.

Error Message: $Y0268

Summary

Too many elements in array aggregate.

Description

The Compiler has encountered an array aggregate that does not match the usage. The number of elements in the array aggregate is incorrect.

Recommendation

Check to make sure that source and destination array aggregates match, in terms of the number and types of their elements.

Error Message: $Y0269

Summary

Too many elements in string.

Description

The Compiler has encountered a quoted string that is illegal for the current expression or assignment.

Recommendation

Check to make sure that the string is of the correct format for the intended usage.

Also check to ensure that you have not omitted the terminating quote character.

Error Message: $Y0270

Summary

Too many elements in bit string.

Description

The Compiler has encountered a bit string that does not match (in terms of size) the objects used in an expression or assignment.

Recommendation

Check to make sure that the bit string contains the correct number of bit characters. If you have entered the bit string using an alternate (non-binary) format, check to ensure that the bit string represents the expected number of bits when analyzed.

Error Message: $Y0271

Summary

Value assigned to target is outside range of values in target subtype.

Description

The Compiler has encountered an invalid assignment to an object. The value on the right-hand side is outside of the possible values allowed for the left-hand side. The possible values are defined by the subtype of the left-hand side as specified in its declaration.

Recommendation

Check to make sure that the target of the assignment is a type compatible with the assigned value.

Check the declaration of the subtype to ensure that it specifies the required range.

Error Message: $Y0272

Summary

Too many elements in record aggregate.

Description

The Compiler has encountered an invalid aggregate. The record aggregate specified has too many elements for the record type.

Recommendation

Check to make sure that the record type declaration is compatible with the aggregate you have specified.

Error Message: $Y0273

Summary

The actual signal associated with a signal parameter must be denoted by a static signal name.

Description

The Compiler has encountered an invalid actual argument to a subprogram or component. Parameters of kind **signal** must be specified with static signal names, rather then expressions.

Recommendation

Check to make sure that the actual parameter is compatible with a parameter of kind **signal**, or modify the subprogram so that it does not require parameter kind **signal**.

Error Message: $Y0274

Summary

Aggregate type must be array or record.

Description

The Compiler has encountered an invalid aggregate specification. The result of the aggregate must be an array or record type.

Recommendation

Check to make sure that the aggregate has been properly specified.

Error Message: $Y0275

Summary

Parameter of attribute 'succ equals prefix'base'high.

Description

The Compiler has encountered an invalid use of the 'succ attribute. The parameter of the 'succ attribute does not have a successor.

Recommendation

Check to make sure that the declaration of the base type is compatible with the use of the 'succ attribute parameter.

Error Message: $Y0276

Summary

Parameter of attribute 'pred equals prefix'base'low.

Description

The Compiler has encountered an invalid use of the 'pred attribute. The parameter of the 'pred attribute does not have a predecessor.

Recommendation

Check to make sure that the declaration of the base type is compatible with the use of the 'pred attribute parameter.

Error Message: $Y0277

Summary

Parameter of attribute 'leftof equals prefix'base'left.

Description

The Compiler has encountered an invalid use of the 'leftof attribute. The parameter of the 'leftof attribute does not have a predecessor.

Recommendation

Check to make sure that the declaration of the base type is compatible with the use of the 'leftof attribute parameter.

Error Message: $Y0278

Summary

Parameter of attribute 'rightof equals prefix'base'right.

Description

The Compiler has encountered an invalid use of the 'rightof attribute. The parameter of the 'rightof attribute does not have a successor.

Recommendation

Check to make sure that the declaration of the base type is compatible with the use of the 'rightof attribute parameter.

Error Message: $Y0279

Summary

Parameter of attribute 'val is too large.

Description

The Compiler has encountered an attribute value that is too large.

Recommendation

Check to make sure that the attribute parameter is a valid integer value.

Error Message: $Y0280

Summary

Parameter of attribute 'val is too small.

Description

The Compiler has encountered an attribute value that is too small.

Recommendation

Check to make sure that the attribute parameter is a valid integer value.

Error Message: $Y0281

Summary

Subtype range is not within the range of the base type.

Description

The Compiler has encountered an invalid specification of a subtype. Subtype declarations must specify ranges of values that are within the range of the specified base type.

Recommendation

Check to make sure that the correct base type has been referenced, and that the range of the subtype falls within the range of the base type.

Error Message: $Y0282

Summary

Too many choices in **case** statement.

Description

The Compiler has encountered an invalid **case** statement. There are too many choices provided for the possible values of the **case** condition expression.

Recommendation

Check to make sure that you have not specified **case** choices that overlap and that you have not duplicated the same choice in two different **case** choices.

Error Message: $Y0283

Summary

Select expression is an array which must be of a character type.

Description

The Compiler has encountered an invalid array specification in a selection expression. The expected selection expression must be a character array.

Recommendation

Check to make sure that the selection expression is a valid array type.

Error Message: $Y0284

Summary

Case expression is an array that must be of a character type.

Description

The Compiler has encountered an invalid array specification in a **case** expression. The expected **case** expression must be a character array.

Recommendation

Check to make sure that the **case** expression is a valid array type.

Error Message: $Y0285

Summary

Unable to determine type of array.

Description

The Compiler has encountered an array specification that cannot be resolved to a known type.

Recommendation

Check to make sure that the array type is clearly distinguished.

Error Message: $Y0286

Summary

Attempt to index non-array.

Description

The Compiler has encountered an index operation on an object that is not an array type. Only array types may be indexed.

Recommendation

Check to make sure that the object being indexed is declared as an array. Use a type conversion function to convert the object to a valid array type if necessary.

Error Message: $Y0287

Summary

Array index has an incompatible type.

Description

The Compiler has encountered an invalid array index. An array index must be either an integer, enumerated or physical type.

Recommendation

Check to make sure that the array index has been correctly specified.

Also check the declaration of the array index object to ensure it is an integer, enumerated or physical type.

Error Message: $Y0288

Summary

Array index must be a scalar type.

Description

The Compiler has encountered an invalid array index. The array index must be a scalar type.

Recommendation

Check to make sure that the array index has been correctly specified.

Also check the declaration of the array index object to ensure it is a scalar type.

Error Message: $Y0289

Summary

A **Next** statement condition expression must be type boolean.

Description

The Compiler has encountered an invalid condition expression in the **next** statement of a **loop**. The conditions expression used in a **next** statement must evaluate to a Boolean (True or False) value.

Recommendation

Check to make sure that the expression will evaluate to a Boolean value. If you are testing a binary value (such as a bit type signal), you should use the relational operator '=' to create a Boolean result.

Error Message: $Y0290

Summary

An **Exit** statement condition expression must be type boolean.

Description

The Compiler has encountered an invalid condition expression in the **exit** statement of a loop. The conditions expression used in an **exit** statement must evaluate to a Boolean (True or False) value.

Recommendation

Check to make sure that the expression will evaluate to a Boolean value. If you are testing a binary value (such as a bit type signal), you should use the relational operator '=' to create a Boolean result.

Error Message: $Y0291

Summary

A **while loop** condition expression must be type boolean.

Description

The Compiler has encountered an invalid condition expression in a **while loop**. The conditions expression used in a **while loop** must evaluate to a Boolean (True or False) value.

Recommendation

Check to make sure that the expression will evaluate to a Boolean value. If you are testing a binary value (such as a bit type signal), you should use the relational operator '=' to create a Boolean result.

Error Message: $Y0292

Summary

Unable to resolve the types of the operands of 'name'.

Description

The Compiler has encountered an ambiguous expression in which the argument types could not be resolved.

Recommendation

Check to make sure that the argument types are clearly distinguished. Introduce intermediate signals or variables if necessary to clearly distinguish the types of literal values.

Error Message: $Y0293

Summary

Too few elements in Group.

Description

The Compiler has encountered a group declaration that does not match the size of the group template declaration.

Recommendation

Check to make sure that the group declaration and group template declaration are compatible.

Error Message: $Y0294

Summary

Too many elements in Group.

Description

The Compiler has encountered a group declaration that does not match the size of the group template declaration.

Recommendation

Check to make sure that the group declaration and group template declaration are compatible.

Error Message: $Y0295

Summary

The prefix of a signature must be a subprogram or enumeration literal.

Description

The Compiler has encountered a signature prefix that is invalid. A signature prefix must be either a subprogram (function or procedure) name or an enumeration literal.

Recommendation

Check to make sure that the signature prefix is a valid function or procedure name, or is an enumeration literal.

Error Message: $Y0296

Summary

The signature does not match the 'description'.

Description

The Compiler has encountered a subprogram signature that does not match the specified subprogram.

Recommendation

Check to make sure that the type specified in the signature matches the return value of the specified function or procedure.

Error Message: $Y0297

Summary

A signature is required here because the 'description' is overloaded.

Description

The Compiler has encountered an ambiguous use of an overloaded operator. The context of the operation does not provide enough information to distinguish between two or more possible operator functions.

Recommendation

Check to make sure that the data types used for the operands are clear and unambiguous. Add a signature if necessary to clearly identify the operator function. Introducing intermediate signals or variables can often solve problems with ambiguous types and operations.

400-series Error Messages

Error Message: $Y0400

Summary

Signal 'name' has multiple drivers.

Description

The Compiler has encountered a signal that is being driven in more than one process.

Recommendation

Check to make sure that the signal is not assigned in more than one process.

Note that it is legal VHDL to have a signal with multiple drivers if the signals type is a resolved type (i.e. has a resolution function) such as 'std_logic' (but not 'std_ulogic'). It is a synthesis constraint, however, that resolution functions are ignored so that no type is a resolved type. In this case you must recode your design so that it does not depend upon the resolution function.

Error Message: $Y0401

Summary

No Block label matches configuration label 'name'.

Description

The Compiler has encountered an invalid configuration. The indicated block label cannot be found.

Recommendation

Check to make sure that the block label has been correctly entered in the configuration.

Error Message: $Y0402

Summary

No component matches Configuration for 'name'.

Description

The Compiler has been unable to find the indicated component in the current design. The configuration statement or declaration is invalid.

Recommendation

Check to make sure that the component name has been correctly specified in the configuration.

Check also to make sure that the component has been properly referenced in the design and that the design unit in which the component has been referenced is included in the current compile.

Error Message: $Y0403

Summary

Generate range is unconstrained.

Description

The Compiler has encountered a generate range that is invalid. Generate ranges must not be unconstrained.

Recommendation

Check to make sure that the generate range specified is correct and is properly constrained.

Error Message: $Y0404

Summary

Component has more than one binding.

Description

The Compiler has encountered a problem in the specified binding of a component. Two or more component configurations are in conflict.

Recommendation

Check to make sure that duplicate component bindings are not specified.

Error Message: $Y0405

Summary

Next or **Exit** is not inside loop with matching label.

Description

The Compiler has encountered an invalid **next** or **exit** statement. The loop label specified in the **next** or **exit** statement is not valid.

Recommendation

Check to make sure that the loop label specifies a valid loop and that the **next** or **exit** statement is inside the specified loop.

Also check to make sure you have not inadvertently terminated the loop with a misplaced **end loop** statement.

Error Message: $Y0406

Summary

The array index is illegal for a null array.

Description

The Compiler has encountered an array index for a null array. Null arrays do not have any members and therefore cannot be indexed.

Recommendation

Check to make sure that the array has been declared as intended and that the index is valid.

Error Message: $Y0408

Summary

Design contains no entity.

Description

The Compiler was unable to find a valid entity in the input design files.

Recommendation

Check to make sure that you have correctly specified the input source files and that one or more valid entities exist in the design.

Error Message: $Y0420

Summary

Result of 'operator' exceeds maximum possible value.

Description

The Compiler has encountered an operation that will produce an overflow result.

Recommendation

Check to make sure that the operator and operands have been correctly specified.

Also check the range of the data type being used.

Error Message: $Y0421

Summary

Result of 'operator' exceeds minimum possible value.

Description

The Compiler has encountered an operation that will produce an underflow result.

Recommendation

Check to make sure that the operator and operands have been correctly specified. Also check the range of the data type being used.

Error Message: $Y0422

Summary

Divide by zero.

Description

The Compiler has encountered an operation that will produce an undefined result. A divisor specified in the expression is zero.

Recommendation

Check to make sure that the operator and operands have been correctly specified.

Check to ensure that the divisor is non-zero.

Error Message: $Y0430

Summary

Description 'name' was not declared as static.

Description

The Compiler has encountered a non-static expression in a context where only a static expression is valid.

Recommendation

Check to make sure that the expression specified is a static expression.

Error Message: $Y0431

Summary

Unconstrained range in **CASE** statement choice.

Description

The Compiler has encountered an invalid choice in a **case** statement. The range specified must be constrained.

Recommendation

Check to make sure that the **case** statement has been correctly specified and that all choices specify constrained expressions.

Error message: $Y0432

Summary

Selected prefix is not a record.

Description

The Compiler has encountered an invalid used of a record attribute. The prefix is not a record type.

Recommendation

Check to make sure that you are specifying a valid record type of object in the attribute specification. If you did not intend to use a record attribute, check to make sure you are specifying the correct attribute.

Error Message: $Y0434

Summary

Description 'name' value is non-constant.

Description

The Compiler has encountered a constant declaration that does not specify a constant value.

Recommendation

Check to make sure that the constant value is correctly specified, or use a signal declaration if a non-constant value is required.

Error Message: $Y0435

Summary

Generic value illegal for its type.

Description

The Compiler has encountered an invalid use of a generic. The value of generic must be within the range of the subtype of the generic.

Recommendation

Check to make sure that you have correctly specified the generic value.

Also check to make sure the generic has been correctly specified in the lower-level design unit.

Error Message: $Y0436

Summary

Constant value illegal for its type.

Description

The Compiler has encountered a constant (scalar) value that is not legal for the type used. This error is most likely the result of specifying a numeric value that is outside the valid range of numeric types.

Note that the value of a constant must be within the range of the subtype of the constant.

Recommendation

Check to make sure that the constant has been entered in the format required for the type.

Also check to ensure that you have specified a value that is in the legal range for the type.

Error Message: $Y0437

Summary

Expected signal, variable, or constant but not a description.

Description

The Compiler has encountered a named item (such as a design unit name, subprogram name, type or block name) when a signal, variable or constant name was required.

Recommendation

Check to ensure that you have used the correct object name.

Also check to make sure that you have not inadvertently used the same name for a block, loop or process label as you have used for a signal, variable or constant.

Error Message: $Y0438

Summary

Loop range is unconstrained.

Description

The Compiler has encountered a loop with an unconstrained range in the iteration specifier. Unconstrained loops are not supported.

Recommendation

Check to make sure that the iteration range has been properly specified.

Error Message: $Y0440

Summary

Subprogram call actual parameter is an unconstrained array.

Description

The Compiler has encountered an invalid use of a function or procedure. The parameters specified for functions or procedures must be either a signal, variable, or constant, or an expression that results in a value of the appropriate type. Subprogram parameters that are arrays must be constrained.

Recommendation

Check to make sure that the function or procedure is being used properly, and that all parameters specified when using the subprogram are valid.

Error Message: $Y0441

Summary

Actual parameter associated with **OUT** formal parameter 'name' is an expression.

Description

The Compiler has encountered an invalid use of a procedure. Parameters specified as type **out** in a procedure must be either a signal, variable, or constant. Expressions are not allowed as actual parameters when the procedure parameter is of mode **out**.

Recommendation

Check to make sure that the procedure is being used properly, and that all parameters specified when using the subprogram are valid. If the parameter indicated is intended to be a procedure input, then change the parameter's mode from **out** to **in**.

Error Message: $Y0442

Summary

Function 'name' has no **return** statement.

Description

The indicated function has not been provided with a **return** statement. All functions must be provided with a value prior to exiting. This value must be specified using a **return** statement.

Recommendation

Check to make sure that the function is provided with a **return** statement and that there is no possibility of the **return** statement to be skipped as a result of a conditional expression.

Error Message: $Y0450

Summary

Expected a static expression here.

Description

The Compiler has encountered a non-static expression when a static value or expression was required. A static expression is an expression whose value cannot be determined at the time of compilation.

Recommendation

Check to make sure that the expression used is static.

Error Message: $Y0451

Summary

Named association missing from record aggregate.

Description

The Compiler is unable to determine the correct mapping of record elements in an aggregate due to the lack of a named association.

Recommendation

Check to make sure that each item in the record aggregate is provided with a named association, or use positional association and do not omit any record elements.

Error Message: $Y0452

Summary

Named association missing from array aggregate.

Description

The Compiler is unable to determine the correct mapping of array elements in an aggregate due to the lack of a named association.

Recommendation

Check to make sure that each item in the array aggregate is provided with a named association, or use positional association and do not omit any array elements.

Error Message: $Y0453

Summary

Record aggregate has missing element(s).

Description

The Compiler is unable to determine the correct mapping of record elements in an aggregate due to the lack of one or more record elements being specified.

Recommendation

Check to make sure that each item in the record aggregate is provided, or use named association to specify the aggregate.

Error Message: $Y0454

Summary

Description 'name' has a type that is an unconstrained array.

Description

The Compiler has encountered an unconstrained array being used where an unconstrained array is not allowed.

Recommendation

Check to make sure that you have properly specified the array and provided a constraint range if necessary.

Error Message: $Y0460

Summary

Combinational Feedback using variable 'name'.

Description

The Compiler has determined that the indicated variable will require combinational feedback to produce the specified behavior. Combinational feedback is specified whenever a variable's value is read prior to its having been set in a process or subprogram.

Recommendation

Check to make sure that you have used the variable correctly. If you did not intend to generate a combinational feedback loop, be sure you have assigned a value to the variable before attempting to use it in an expression.

Error Message: $Y0470

Summary

Constraint: Unexpected use of 'Z' or NULL, unable to infer a tristate.

Description

The Compiler has encountered a use of the 'Z' or null value that appears to be for describing an output enable, but enable logic following the conventions of the Synthesis Compiler has not been specified.

Recommendation

Check to make sure that an enable expression has been specified using a conditional signal assignment or an **if** statement. Also check that an **if** statement describing a tristate is a simple **if** statement and not embedded within another **if** statement or a **case** statement.

Error Message: $Y0480

Summary

Constraint: The name library does not contain a description.

Description

The Compiler has encountered the use of a logic element that does not exist in the named gate library (not VHDL library) and is unable to re-synthesize to some logic element that does exist in the library. The logic element description will be some form of flip-flop, latch, or tristate. The element does not exist in the target gate library because it has no realization in the target silicon. A common example of a structure that may cause this error is a flip-flop with both set and reset.

Recommendation

Change the VHDL source code desription so it does not describe this logic element.

500-series Error Messages

Error Message: $Y0500

Summary

Constraint: A **Wait** statement may only be the first statement in a Process.

Description

The Compiler has encountered a **wait** statement used in an unsupported manner. **Wait** statements are only supported as the first statement in a process.

Recommendation

Check to make sure that the **wait** statement is the first statement in the process. If you are attempting to describe registered behavior with an asynchronous reset, you should use a sensitivity list and an **if-then** statement to described the reset and clock logic.

Error Message: $Y0501

Summary

Constraint: Process contains more than one **Wait** statement.

Description

The Compiler has encountered more than one **wait** statement being used in a process. Only one **wait** statement may be used in a process and that **wait** statement must be the first statement in the process.

Recommendation

Check to make sure that the **wait** statement is the first statement in the process and do not attempt to use more than one **wait** statement in any one process. If you are trying to describe a system with multiple clocks, you will have to use multiple processes.

Error Message: $Y0502

Summary

Constraint: Formal part may not be a function call.

Description

The Compiler has encountered an unsupported named association in a subprogram or component instantiation. The Compiler does not allow the formal parts of subprograms and component instantiations to be function calls.

Recommendation

Check to make sure that the formal part of the subprogram or component is not a function call.

Error Message: $Y0503

Summary

Constraint: Signal attribute 'attribute is not supported.

Description

The Compiler has encountered the use of an unsupported attribute. This attribute has no meaning for synthesis and must not be used.

Recommendation

Check to make sure that the correct attribute is being used, or remove the attribute.

Error Message: $Y0504

Summary

Constraint: **WAIT** statement in a procedure is not allowed.

Description

The Compiler has encountered a **WAIT** statement used within a procedure. **WAIT** statements may only be used within processes and may only be used as the first statement in a process.

Recommendation

If you are attempting to describe registered logic in a procedure, use the **if-then** synthesis convention for describing flip-flop logic.

Also check to ensure that you are not inadvertently attempting to synthesize a test bench.

Error Message: $Y0505

Summary

Constraint: Expected a static expression, description 'name' is not allowed here.

Description

The Compiler has encountered the unsupported use of a non-static expression. Non-static expressions are those that depend on the value of a signal or port and cannot be evaluated during compilation.

Recommendation

Check to make sure that the expression is static and does not depend on the value of a signal or port.

Error Message: $Y0506

Summary

Constraint: Expected a static expression, Constant 'name' is not static.

Description

The Compiler has encountered the unsupported use of a non-static expression. Non-static expressions are those that depend on the value of a signal or port and cannot be evaluated during compilation.

Recommendation

Check to make sure that the expression is static and does not depend on the value of a signal or port.

Error Message: $Y0507

Summary

Constraint: '**' is supported only for constant operands.

Description

The Compiler has encountered an unsupported use of the '**' (exponentiation) operator. Only constant exponent values are allowed.

Recommendation

Check to make sure that the exponent value specified is a constant value.

Error Message: $Y0508

Summary

Constraint: Assign to array element must have constant array index.

Description

The Compiler has encountered the unsupported use of a non-constant array index in an assignment to a multi-dimensional array.

Recommendation

The Compiler requires that the array target of an assignment be referenced using only constant index values if you are trying to index more that one dimension of the array.

Check to make sure that the index argument used in the target of the assignment is a constant value.

Error Message: $Y0509

Summary

Constraint: Array slice must have constant range.

Description

The compiler has encountered a non-constant range specification in an array slice. The Compiler required that array slices be specified using constant range values.

Recommendation

Check to make sure that the array slice is specified using a constant range. You can use a **loop** or **generate** statement to specify non-constant array slices if necessary.

Error Message: $Y0510

Summary

Constraint: Recursive Hierarchy instantiation.

Description

The Compiler has encountered a recursive instantiation of a component, entity, or configuration. There is no practical synthesis equivalent to recursive hierarchy instantiation.

Recommendation

Check to make sure that you have not inadvertently created recursion in your design by specifying the wrong component, entity, or configuration name.

Error Message: $Y0511

Summary

Constraint: Recursive Subprogram call.

Description

The Compiler has encountered a recursive reference to a function or procedure. There is no practical synthesis equivalent to recursive subprogram specifications.

Recommendation

Check to make sure that you have not inadvertently created recursion in your design by specifying the wrong function or procedure name.

Error Message: $Y0512

Summary

Constraint: Literal value exceeds maximum positive value.

Description

The Compiler has encountered a numeric value that is larger than the maximum allowed.

Recommendation

Check to make sure that you have correctly specified the numeric literal value.

Error Message: $Y0513

Summary

Constraint: Literal value exceeds minimum negative value.

Description

The Compiler has encountered a negative numeric value that is smaller than the maximum allowed.

Recommendation

Check to make sure that you have correctly specified the numeric literal value.

Error Message: $Y0514

Summary

Constraint: Literal fractional part truncated.

Description

The Compiler has encountered a floating point literal value that includes a fractional part. Floating point numbers are only supported as integer values in synthesis and any fractional part is truncated.

Recommendation

Check to make sure a floating point value was actually intended.

Error Message: $Y0515

Summary

Constraint: Attribute 'event is misused in this process, some combinational logic depends upon the 'event.

Description

The Compiler has encountered an unsupported use of the 'event attribute. 'Event is only supported in **wait** statements (entered as the first statement of a process), or in **if-then** conditional expressions in processes or subprograms to specify edge-triggered (flip-flop) behavior.

Recommendation

Check to make sure that you have followed the documented synthesis conventions for specifying registered logic. If the cause of the error is not clear to you then split the process into two processes, one for the combinational logic and one for the sequential logic.

Error Message: $Y0516

Summary

Constraint: Attribute 'stable is misused in this process, some combinational logic depends upon the 'stable.

Description

The Compiler has encountered an unsupported use of attribute 'stable in the design. 'stable is not recommended for synthesizable designs and is only supported in **wait** statements.

Recommendation

Check to make sure that you have specified the correct attribute. Use the 'event attribute to describe edge-triggered flip-flop logic. If the cause of the error is not clear to you then split the process into two processes, one for the combinational logic and one for the sequential logic.

Error Message: $Y0517

Summary

Constraint: Access types are not supported.

Description

The Compiler has encountered an unsupported use of an access type. Access types are not supported in synthesis.

Recommendation

Rewrite your design so that access types are not required.

Error Message: $Y0518

Summary

Constraint: File types are not supported.

Description

The Compiler has encountered an unsupported use of the type file. File types are not supported in synthesis.

Recommendation

Check to ensure that you are not inadvertently compiling a test bench, rather than a synthesizable design description.

Rewrite your design so that file types are not required.

Error Message: $Y0519

Summary

Constraint: File Declaration is not supported.

Description

The Compiler has encountered an unsupported use of a file declaration. File declarations are not supported in synthesis.

Recommendation

Rewrite your design so that file declarations are not required.

Error Message: $Y0520

Summary

Constraint: Allocator New is not supported.

Description

The Compiler has encountered an unsupported use of the memory allocation feature, new. New is not supported in synthesis.

Recommendation

Check to ensure that you are not inadvertently compiling a test bench, rather than a synthesizable design description.

Rewrite your design so that new is not required.

Error Message: $Y0521

Summary

Constraint: Waveform truncated.

Description

The Compiler has encountered a waveform specification that includes more than one entry. Waveforms are not supported for synthesis unless they consist of only a single entry.

Recommendation

Re-specify the design so a waveform is not required, or simply ignore the error message.

Error Message: $Y0524

Summary

Constraint: Attribute 'attribute parameter is non-constant.

Description

The Compiler has encountered an attribute that is only supported when applied to constant values.

Recommendation

Check to make sure that the target of the attribute parameter is a constant value.

Error Message: $Y0525

Summary

WARNING: Signal 'name' must be in the Process sensitivity list or is an input to a flip-flop that was not inferred because the flip-flop is incorrectly specified.

Description

The Compiler has determined that the indicated signal is an asynchronous input to the current process and must therefore be included in the process sensitivity list. This may result in a mismatch between simulation before synthesis and simulation after synthesis, but will not terminate synthesis execution.

Recommendation

Check to make sure that the indicated signal was intended to be an asynchronous input to the process.

If the signal was intended as an asynchronous input, add that signal name to the sensitivity list. If the signal was not intended to an asynchronous input, check to ensure that all output signals referencing the indicated signals as an input have been properly and completely specified.

Take special care to ensure that unwanted latches have not been inadvertently specified.

Error Message: $Y0526

Summary

WARNING: Flip-flop 'name' has missing preset or reset.

Description

The Compiler has determined that the behavior of the indicated registered signal is ambiguous without a reset or preset being provided. This warning occurs when the missing preset or reset specifies a flip-flop with a gated clock, when the other flip-flops inferred in the process do not have gated clocks. By convention synthesis tools implement a flip-flop without a gated clock in this case. This may result in a mismatch between simulation before synthesis and simulation after synthesis, but will not terminate synthesis execution.

Recommendation

Check to make sure that the indicated signal is either provided with preset or reset logic, or has been described in such a way that its behavior is unambiguous for all possible input conditions.

Error Message: $Y0527

Summary

Constraint: Shared Variable Declarations not supported.

Description

The Compiler has encountered an unsupported use of shared variables. Shared variables are not support in synthesis.

Recommendation

Rewrite your design so that shared variables are not required.

Error Message: $Y0528

Summary

Constraint: An operator symbol (description) is not supported here.

Description

The Compiler has encountered an unsupported use of an operator in the context of an alias.

Recommendation

Rewrite the design section so the operator is not required, or do not use an alias in this context.

Error Message: $Y0529

Summary

Constraint: Design contains no top level Output, Buffer, or Inout ports.

Description

The Compiler has encountered a design that has no top-level output ports.

Recommendation

Check to make sure you are not inadvertently compiling a test bench. Also check to make sure you have correctly specified the mode of all entity ports.

Error Message: $Y0530

Summary

Constraint: Hierarchy name must not contain a white space.

Description

The Compiler has encountered an unsupported hierarchical name. Hierarchical names may not include white space characters.

Recommendation

Check to make sure that the hierarchical name has been correctly entered.

Error Message: $Y0531

Summary

Constraint: Tristate buffer 'name' drives a logic gate, it must drive a port.

Description

The Compiler has encountered an unsupported use of tristate logic. The output of a tristate buffer is a logic gate, the buffer must drive a port.

Recommendation

Rewrite the design section so that the tristate buffer drives an output of the design. If you wish to make use of the internal tristate busses available in some FPGAs to build, for example, a small mux, consider instantiating a macrocell.

Note that if the Compiler is unable to provide a 'name' related to the original source description, no 'name' will be reported. In this case you should use the file name and line number from the message to track down the error. The log file may also help as it will report the inference of tristate buffers on a per-process basis.

Error Message: $Y0532

Summary

Constraint: A name operator may not have an argument that contains metalogic value 'U', 'X', 'W', or '-'.

Description

The Compiler has encountered an unsupported use of metalogic values. This usage is the interpretation required by the VHDL 1076.3 synthesis standard.

Recommendation

Error Message: $Y0533

Summary

Constraint: 'name' is global and has not been declared in a package.

Description

The Compiler has encountered an unsupported reference to a signal. The signal was declared outside of the current entity or architecture, but not in a package. The signal is therefore global to the design and not local to a design unit, this is only supported if the signal is declared in a package.

Recommendation

This may occur if the signal is declared in a different entity or architecture and is made visible by a use clause. The usual solution is to declare the signal locally.

600-series Error Messages

Error Message: $Y0600

Summary

Enum_encoding string may only contain the characters '0' '1' 'Z' '-' 'M' or ' ' or the strings “one hot” or “gray”.

Description

The Compiler has encountered an invalid character in the enum_encoding attribute string. The only characters valid in an enum_encoding attribute string are '1', '0', 'Z', 'M', '-' and the space character, or the special strings 'one hot', '1-hot' or 'gray'.

Recommendation

Check to make sure that the enum_encoding attribute string has been correctly specified.

Error Message: $Y0601

Summary

Each encoding in Enum_encoding must have the same number of characters.

Description

The Compiler has encountered an enum_encoding attribute that does not specify the same number of characters (bits) for each enumeration value.

Recommendation

Check to make sure that you have specified all enumeration values with an equal number of characters.

Note that the only characters valid in an enum_encoding attribute string are '1', '0', 'Z', '-', and the space character.

Error Message: $Y0602

Summary

Enum_encoding may only be applied to an enumerated type.

Description

The Compiler has encountered an enum_encoding attribute that references a non-enumerated type.

Recommendation

Check to make sure that the enum_encoding attribute is being applied to the correct type.

Error Message: $Y0603

Summary

Enum_encoding must follow the enumerated type declaration.

Description

The Compiler has encountered an enum_encoding attribute that is out of place. An enum_encoding attribute must be preceded by a valid type declaration.

Recommendation

Check to make sure that the referenced enumerated type has been properly declared.

Error Message: $Y0604

Summary

Too few encodings specified in Enum_encoding.

Description

The Compiler has encountered an enum_encoding attribute specification that does include the correct number of encodings.

Recommendation

Check to make sure there is one attribute encoding specification provided for each symbolic value defined in the type declaration.

Also check to ensure you have separated the enumerated encoding values with spaces. If you have used more than one line in the source file to specify the enum_encoding string, make sure you have concatenated the strings properly using the '&' operator, and have included spaces to delimit each encoding.

Error Message: $Y0605

Summary

Too many encodings specified in Enum_encoding.

Description

The Compiler has encountered an enum_encoding attribute specification that does not include the correct number of encodings.

Recommendation

Check to make sure there is one attribute encoding specification specified for each symbolic valued defined in the declaration for the enumerated type.

Also check to ensure you have separated the enumerated encoding values with spaces. If you have used more than one line in the source file to specify the enum_encoding string, make sure you have concatenated the strings properly using the '&' operator, and have included spaces to delimit each encoding.

Error Message: $Y0606

Summary

Enum_encoding may not be applied to a subtype of an enumerated type.

Description

The Compiler has encountered an invalid use of the enum_encoding attribute. The enum_encoding attribute may only be applied to an enumerated type, and may not be applied to a subtype.

Recommendation

Check to make sure the attribute is being applied to an enumerated type.

Error Message: $Y0607

Summary

User attribute **Critical** may only be applied to a **Signal**.

Description

The Compiler has encountered an invalid use of the special attribute **critical**. The **critical** attribute is used to preserve signals during synthesis and may only be applied to a **signal**.

Recommendation

Check to make sure that the **critical** attribute has been applied to a **signal**.

Error Message: $Y0608

Summary

'name' has a type which is not locally static, a design unit with 'foreign attribute must have ports with locally static types.

Description

The Compiler has encountered an unsupported use of the 'foreign attribute. 'Foreign is used to reference external modules and must be used in conjunction with ports that reference locally static types.

Recommendation

Check to make sure that the indicated port name represents a locally static type of object.

Error Message: $Y0609

Summary

A design unit with 'foreign attribute may only have ports with mode **IN** or **OUT**.

Description

The Compiler has encountered an unsupported mode for an external module port. All ports of external modules specified using 'foreign must be of mode **in** or **out**.

Recommendation

Check to make sure the external module has been referenced using only ports of mode **in** or **out**.

700-series Error Messages

Error Message: $Y0700

Summary

NOTE: Signal 'name' is used but not assigned and is driven by its default value.

Description

The Compiler has encountered a use of a signal that is legal VHDL but is a possible programming error. This message is only reported in the log file and will not terminate execution.

Recommendation

If the intent of the programmer is not to assign to the signal it is more efficient and clearer to replace the signal declaration with a constant declaration. If the intent is to assign to the signal then there is a missing assignment and the signal is assumed to always have its default value. Note that if the named signal is declared as a port in the VHDL source then the message indicates a mismatch between a component port declaration and its entity port declaration.

Error Message: $Y0701

Summary

NOTE: Port 'name' is not assigned and is driven by its default value.

Description

The Compiler has encountered a use of a port that is legal VHDL but is a possible programming error. This message is only reported in the log file and will not terminate execution.

Recommendation

The named output port has no logic associated with it and will be driven always high or always low. The port may be redundant or have a missing assignment.

Error Message: $Y0702

Summary

NOTE: Signal 'name' is assigned but not used.

Description

The Compiler has encountered a use of a signal that is legal VHDL but is a possible programming error. This message is only reported in the log file and will not terminate execution.

Recommendation

The named signal is not used and its assignment is redundant. If the programmers intent was to use this signal then the usage is missing from another part of the architecture. Note that if the named signal is declared as a port in the VHDL source then the message indicates a mismatch between a component port declaration and its entity port declaration.

Error Message: $Y0703

Summary

NOTE: Port 'name' is not used.

Description

The Compiler has encountered a use of a port that is legal VHDL but is a possible programming error. This message is only reported in the log file, and will not terminate execution.

Recommendation

The named input port is unused, the port may be redundant or have a missing usage. The port may be removed from the synthesized netlist by some downstream tools

Revision History

Date	Version No.	Revision
01-Dec-2004	1.0	New product release
10-Jun-2005	1.1	Updated for Altium Designer SP4