Metamor

PLD Programming Using VHDL

User's Guide

Version 2.4

Table of Contents - Metamor User's Guide

5 - Programming Sequential Logic

6 - Programming Finite State Machines

7 - Some Common Examples in VHDL

8 - Synthesis of VHDL Types

9 - Managing Large Designs

10 - Logic and Metalogic

11 - XBLOX and LPM

12 - Synthesis Attributes

13 - Synthesis Coding Issues

A - VHDL Quick Reference

B - PREP Examples

C - Error Message Index

D - Compile options

E - VHDL Information Resources

Index - Metamor User's Guide

1 - About This Guide

This guide is intended for the engineer who is familiar with the principles of hardware design, but has little experience in designing with a language-based synthesis system. It describes the general concepts of synthesis, the general organization and usage of VHDL, and provides specific information on how the Metamor tool is used in this environment. It does not attempt to present the VHDL language in depth, but does provide an example-based summary of VHDL syntax that serves as a helpful reference for any user.

Notation Conventions

VHDL is not case-sensitive, so a design description can contain UPPERCASE or lowercase text. In this guide, 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.

Copyright Notice

2 - PLD Programming using VHDL

VHDL for PLD Designers

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, we can describe the design I/O, combinational logic, sequential logic, and state machines. Initially we will only consider signals of type std_logic and std_logic_vector (a 1 dimensional array of std_logic). These types allow us to do logical operations (and, or...) and relational operations (equal, greater than,....).

See Also

- for a summary of the syntax of VHDL:

  A - VHDL Quick Reference

- for more detail on the contents of this section:

  4 - Programming Combinational Logic
  5 - Programming Sequential Logic
  6 - Programming Finite State Machines

- for some VHDL examples:

  7 - Some Common Examples in VHDL

Design I/O

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, we write the following two lines 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**;

We will use these I/O definitions in Combinational Logic.

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. We could describe a two bit adder 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;
```

We can also perform conditional assignment. 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 'dont 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;
```

Further detail on combinational logic is described in 4 - Programming Combinational Logic. Also look at the Seven-Segment Decoder which uses another concurrent statement: the selected signal assignment.

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. We can improve on this by using the rising_edge () function from std_logic_1164. In the following example we add output registers to our 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;
```

A more detailed explanation of register inference occurs in 5 - Programming Sequential Logic.

We can add tristates 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;
```

We can use procedures 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, as was introduced in procedures in Registers and Tri-state. Sequential statements may also occur in processes. We have already seen a D flip-flop described using a concurrent statement.

```
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 ent4 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 we've introduced the process statement as a way to describe state machines, it more generally allows behavioral modeling of both combinational and sequential logic.

There are several examples of state machines in 7 - Some Common Examples in VHDL, these slightly larger examples illustrate the application of process statements.

It is strongly recommended that you read 5 - Programming Sequential Logic and 6 - 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 controling the logic optimize granularity, hierarchical compile, and instantiating silicon specific components. These are described in Using Hierarchy.

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, counter1 in State Machines could be described:

```
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 counter2 ;
```

In this example we used type integer because the "+" operator is defined for integers, but not for std_logic_vectors, which we have been using up to now.

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

```
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_ 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 counter2 ;
```

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.

Further discussion of VHDL types occurs in VHDL Types .

Compiling

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 Metamor 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 we do the following :

a) specify the files(s)
b) specify the top level of this Partition
c) 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.
```

We 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 Metamor 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.

Verbose option

This compile option enables printing of 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.

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. For a more detailed discussion on variables depending on signals see the discussion of metalogic expressions in 10 - Logic and Metalogic.

Downstream Tools

Third-party tools that take the output from the Metamor 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. See Attributes for Downstream Tools.

How to be Happy

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. This guide attempts to provide you with the information you will need. With this approach you can expect to be happy with the results.

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.

3 - Introduction to VHDL

VHDL '93

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.

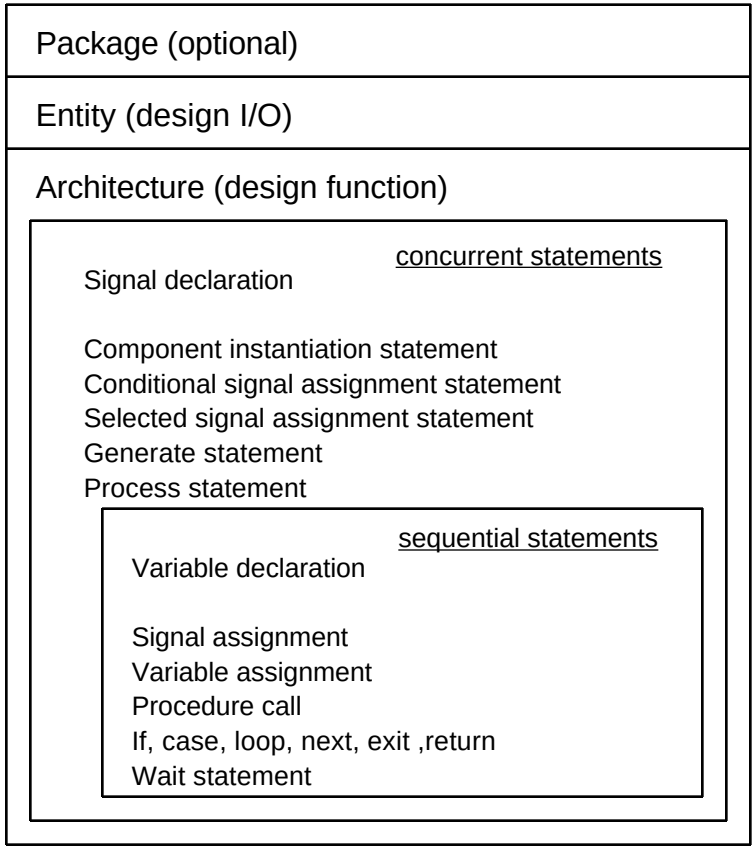
VHDL is a large language and it provides many features. This guide does not attempt to describe the full language -- rather it introduces enough of the language to enable useful design.

Metamor 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 Metamor's VHDL support are listed in Unsupported Constructs, Ignored Constructs , and Constrained Constructs.

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

Structure of a VHDL Design Description

The basic organization of a VHDL design description is shown in the following figure:



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. For more information see Managing Large Designs.

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 **signal**s; 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**. **Signal**s 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.

Later we will discuss sequential *logic* (logic with memory elements such as flip-flops). 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 subsections: 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 as discussed in 9 - Managing Large Designs.

```
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 us 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 we use the **process** statement 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 forms 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. Combinatorial logic is discussed in 4 - Programming Combinational Logic, sequential logic in 5 - Programming Sequential 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**;

For an example see Fifo. For more info see VHDL Design Libraries and Metamor VHDL Libraries.

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 8 - Synthesis of VHDL Types.

My model simulates, but....

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 ? ha ! 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. Metamor assumes the encoding representation for enumerated types unless the encoding is explicitly specified using the attribute 'enum_encoding'.

4 - Programming Combinational Logic

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 Metamor 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 guide.

Logical Operators

VHDL provides the following logical operators:

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

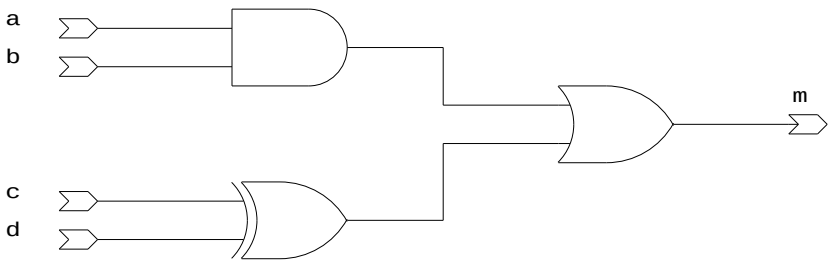
These operators are defined for the types bit, boolean and arrays of bit or boolean (for example, bit_vector). The compilation of logic is fairly direct from the language construct, to its implementation in gates, as shown in the following examples:

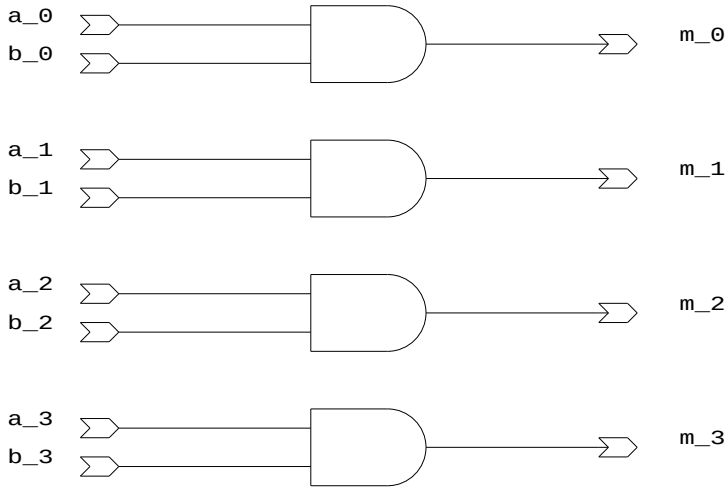
```
entity logical_ops_1 is
    port (a, b, c, d: in bit;  m: out bit);
end logical_ops_1;

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;
```


a
b
c
d
m


a_0
b_0
m_0
a_1
b_1
m_1
a_2
b_2
m_2
a_3
b_3
m_3

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.

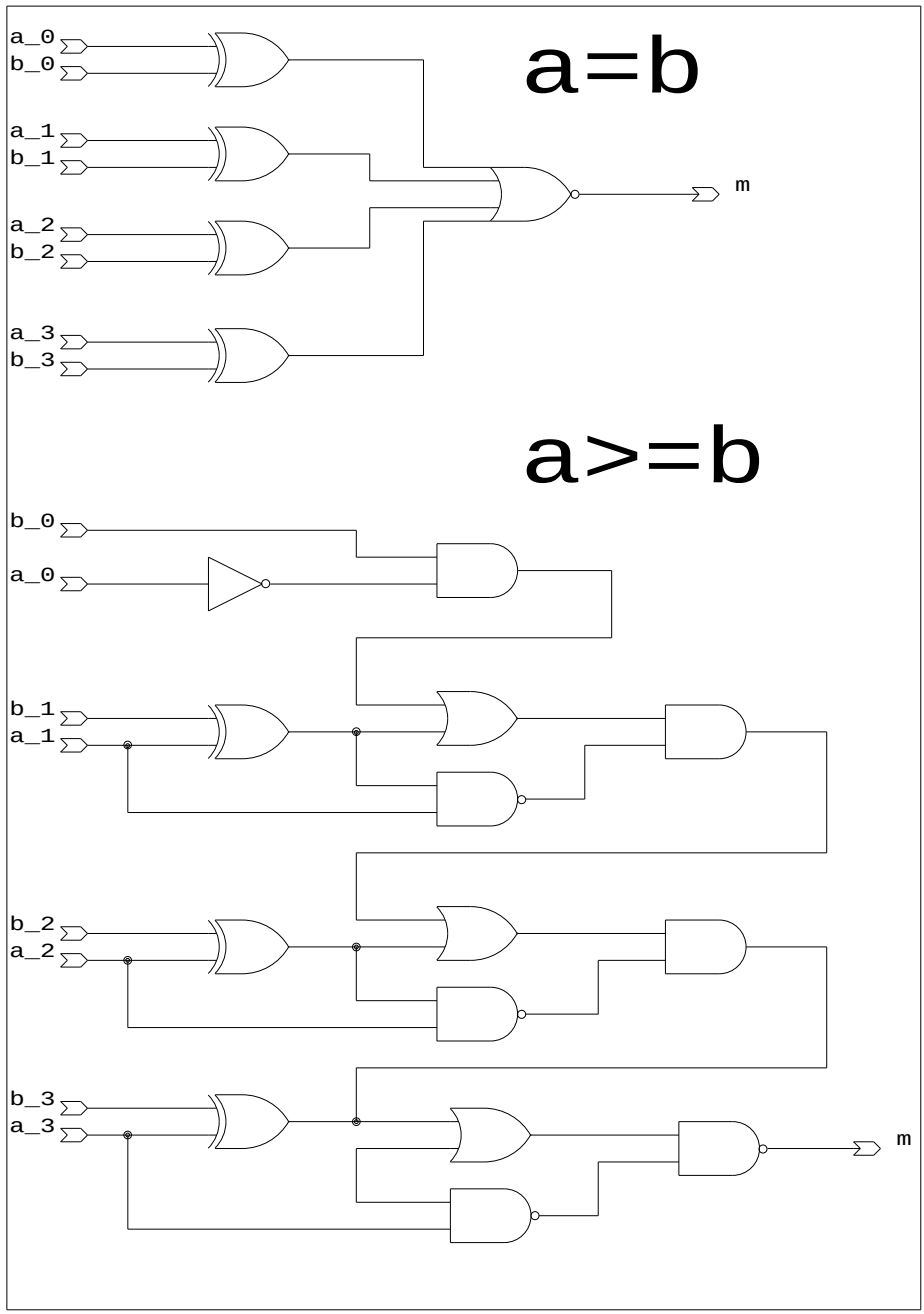
The simple comparisons, equal and not equal, are cheaper to implement (in terms of gates) than the ordering operators. To illustrate, the first example below uses an equal operator and the second uses a greater-than-or-equal-to operator. As you can see from the schematic, the second example uses more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


a=b
a_0
b_0
a_1
b_1
a_2
b_2
a_3
b_3
m
a>=b
b_0
a_0
b_1
a_1
b_2
a_2
b_3
a_3
m

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. Metamor does make special optimizations, however, when the right hand operator is a constant and an even power of 2.

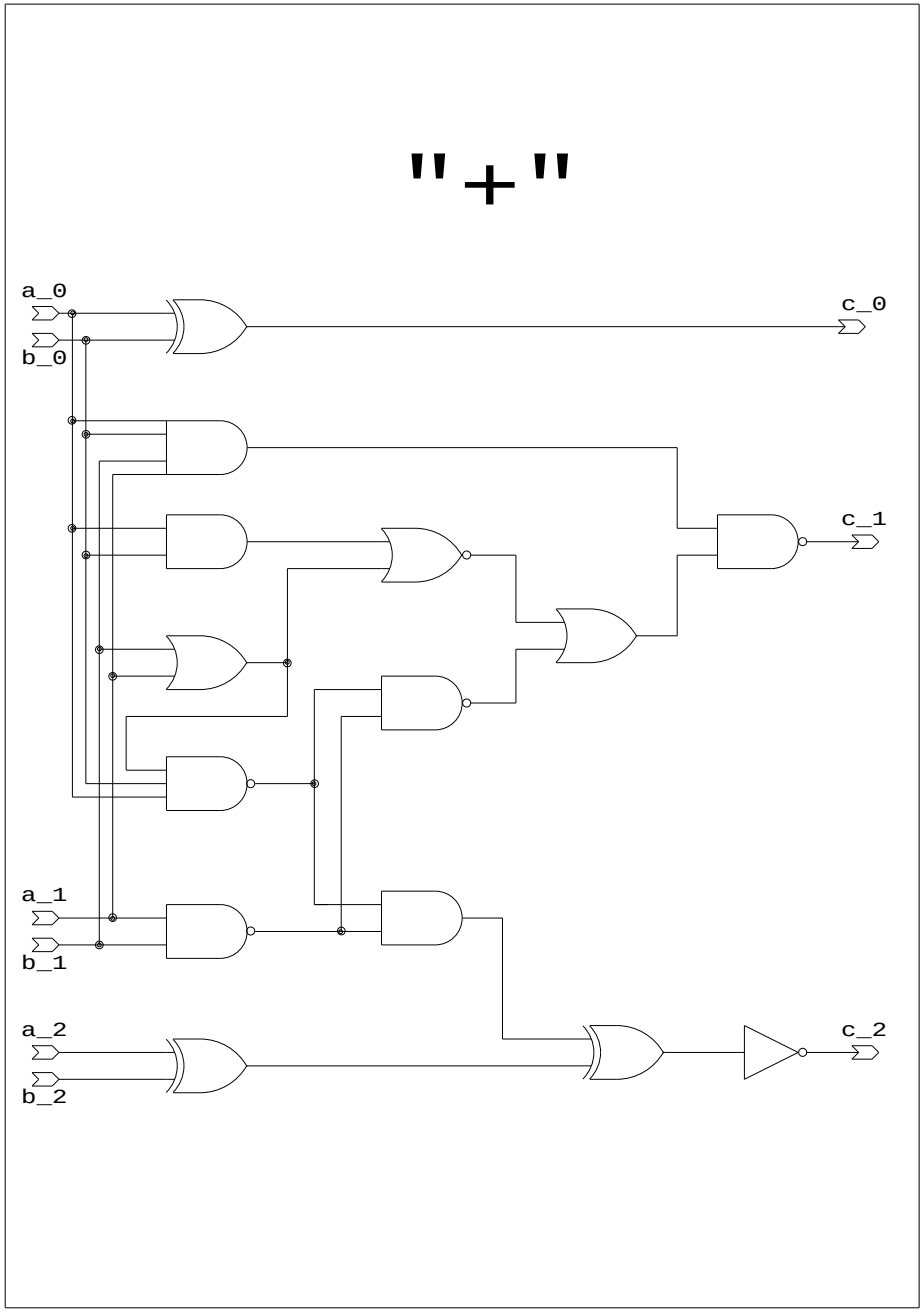
The absolute (**abs**) operator is inexpensive to implement. The ** operator is only supported when its arguments are constants.

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;
```


"+"
a_0
b_0
c_0
c_1
a_1
b_1
a_2
b_2
c_2

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 <= cwhen b"00",
        m <= dwhen b"01",
        m <= awhen b"10",
        m <= bwhen 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. See 5 - Programming Sequential Logic.

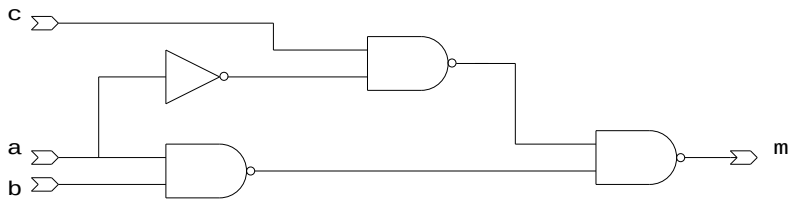
VHDL requires that all the possible conditions be represented in the condition of a case statement. To ensure this, use the **others** clause at the end of a case statement to cover any unspecified conditions.

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

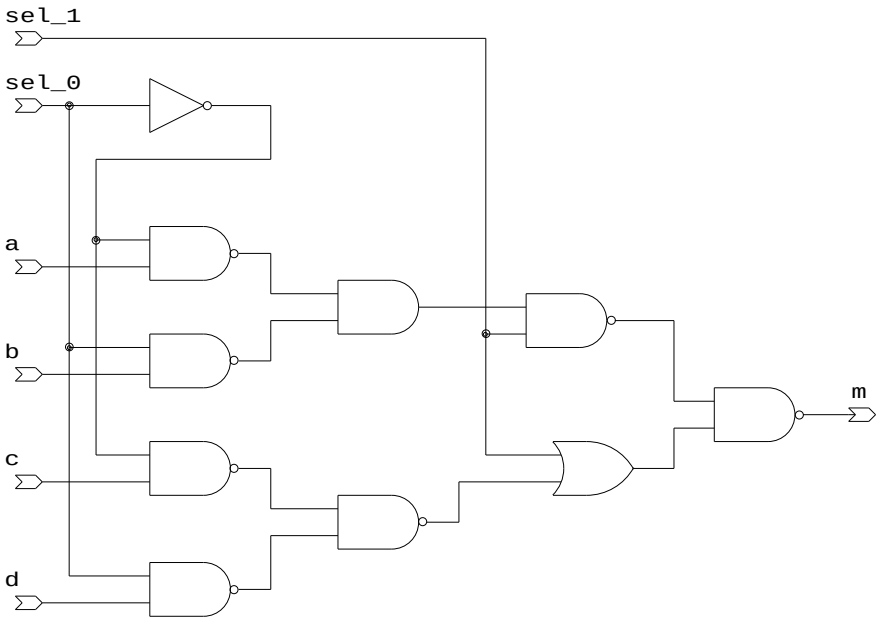
```
entity control_stmts is
    port (sel: bit_vector (0 to 1);  a,b,c,d : bit;  m: out bit);
end control_stmts;

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;
```

if



case



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;
```

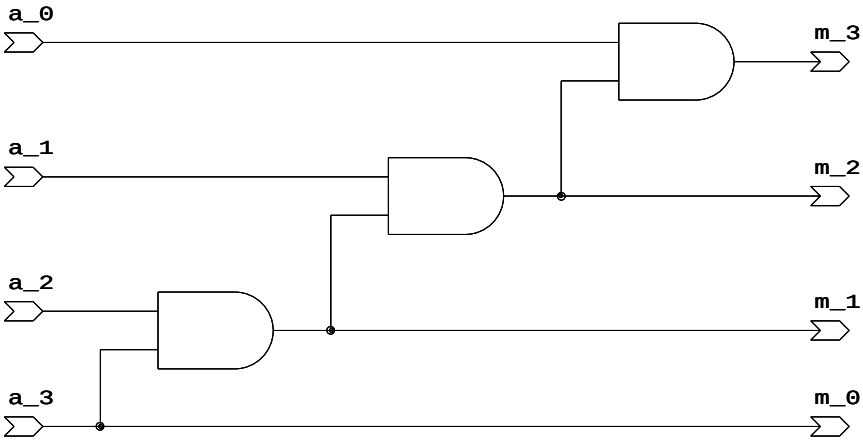
Loop statements may be terminated with an **exit** statement, and specific iterations of the loop statement terminated with a **next** statement. When simulating, an **exit** or **next** may be used to speed up simulation time. For synthesis, where each loop iteration replicates logic, there is probably no speedup. In addition, the **exit** or **next** may synthesizes 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.

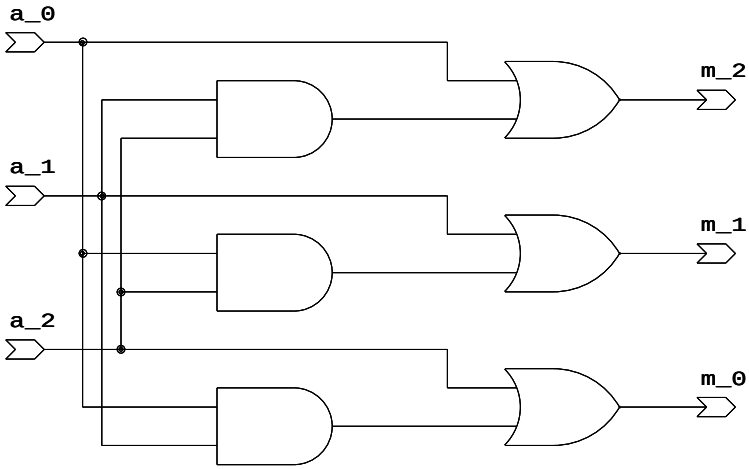
```
entity subprograms is
    port (a: bit_vector (0 to 2);  m: out bit_vector (0 to 2));
end subprograms;

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
```

loop



subprogram



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
```

Tristates

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. Note that a constraint of Metamor requires all tristate buffers driving this bus to be in the same architecture.

```
architecture example3 of tbuf3 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;
```

5 - Programming Sequential Logic

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). These techniques are expanded upon in 7 - Some Common Examples in VHDL where we will show examples of more complex machines.

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 as shown in Registers and Tri-state.

Alternatively the designer may choose to describe a design in a more behavioral form as show in the examples in 7 - Some Common Examples in VHDL.

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 we had assigned 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.

We could describe our latch as transparent low:

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

Or more concisely:

```
if not 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.

See also: D - Compile options.

Synchronous Set/Reset

To add the behavior of synchronous set or reset we 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. We 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 we 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 statements

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

6 - Programming Finite State Machines

Introduction

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.

You can find additional examples in 7 - Some Common Examples in VHDL.

Feedback Mechanisms

There are two ways to create feedback -- using signals and using variables. With the addition of feedback you can build state machines. This will be discussed later in this section.

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 Metamor user attribute 'critical' (as discussed in 2 - PLD Programming using VHDL).

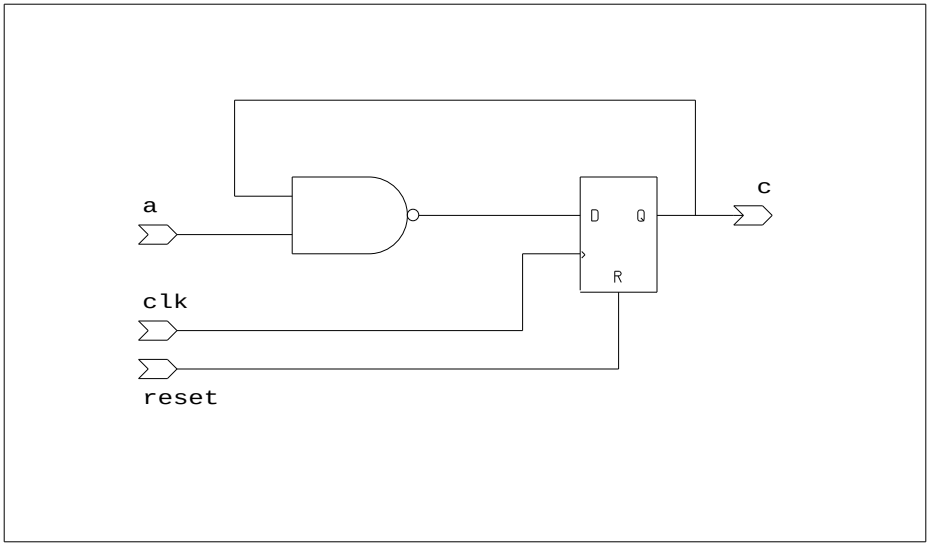
Feedback on signals

```
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:

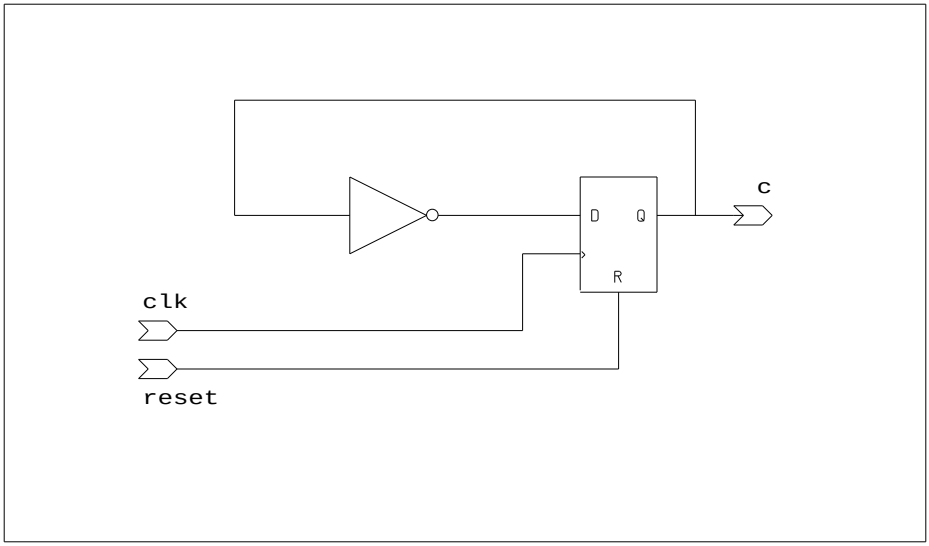
```
use work.my_functions.all; -- package containing
                           -- user function rising_edge
architecture example of some_entity is
begin
   process(clk,reset)
   begin
      if reset = '1' then
         c <= '0';
      elsif rising_edge(clk)
         c <= a and c;
      end if;
   end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



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.



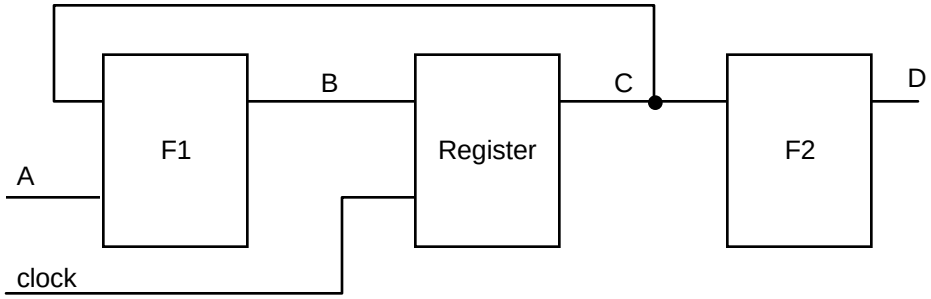
```
process
    variable v: bit;
begin
    wait until clk = '1';
    if reset = '1' then
        v <= '0';
    else
        v := not v; --v is used before it is assigned
        c <= v;
    end if;
end process;
```

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 Metamor constraint), variables inside subprograms never persist over time and never specify registers.

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:

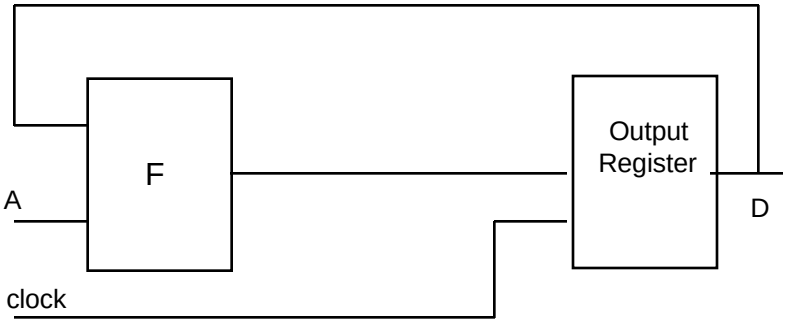
```
architecture moore2 of system is
    signal c: some_type;
begin
    process (c)-- combinational logic
    begin
        d <= F2(c);
    end process;

    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

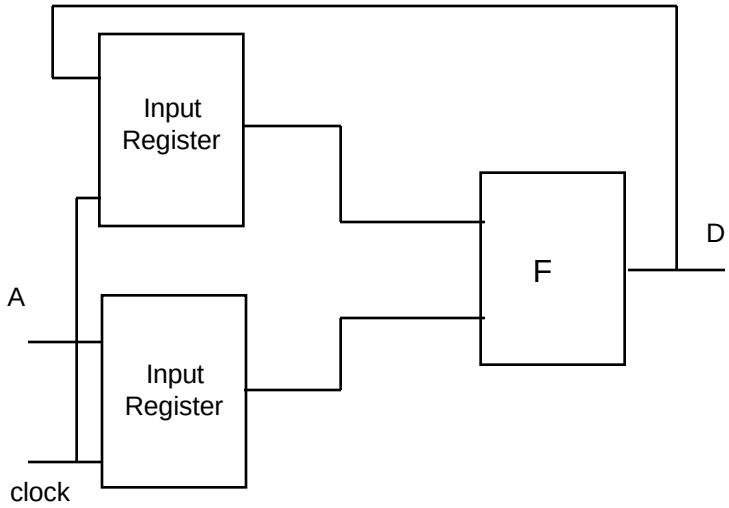
If the system timing requires no logic between the registers and the output (the shortest output propagation delay is desired), the following architecture could be used:




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

Input Registers

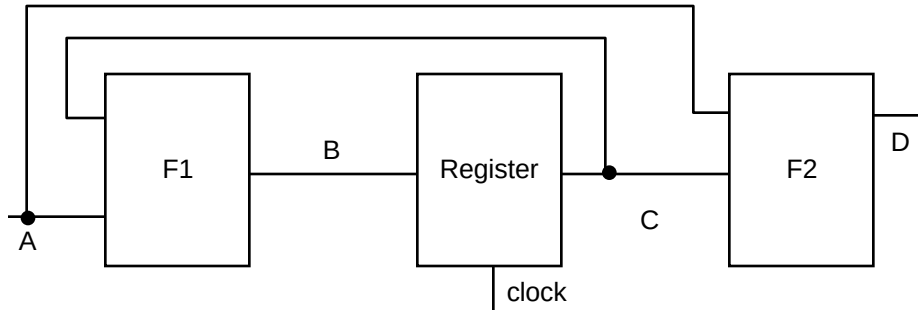
If the system timing requires no logic between the registers and the input (if a short setup time is desired), the following architecture could be used:




```
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;
```

7 - Some Common Examples in VHDL

Seven-Segment Decoder

```
package seven is
    subtype segments is bit_vector (0 to 6);
    type bcd is range 0 to 9;
end seven;

use work.seven.all;
entity decoder is
    port (input: bcd;  drive: out segments);
end decoder;

architecture simple of decoder is-- segment
                                  --encoding:
begin                             --      0
    with input select       --      ---
          drive <= b"1111110" when 0,--   5 |    | 1
                   b"0110000" when 1,--      ---  <- 6
                   b"1101101" when 2,--   4 |    | 2
                   b"1111001" when 3,--      ---
                   b"0110011" when 4,--       3
                   b"1011011" when 5,
                   b"1011111" when 6,
                   b"1110000" when 7,
                   b"1111111" when 8,
                   b"1111011" when 9,
                   b"0000000" when others ;-- just in case
end simple;
```

Craps Game

```
package gamble is
    type dice is range 0 to 12;
end gamble;

use work.gamble.all;
entity craps is
    port (roll, new_game: boolean;  number: dice;  win,
                                    loss: out boolean);
end craps;
architecture no_cheating of craps is
begin
    process (roll, new_game,number)
        variable first_roll: boolean;
        variable point: dice;
        constant snake_eyes: dice := 2;
    begin
        if newgame then-- async reset
            first_roll := true;
            win <= false;
            loss <= false;
            point := 0;
        elsif roll and roll'event then-- clock
            if first_roll then
                first_roll := false;
                if number = 7 or number = 11 then
                    win  <= true;
                elsif number = snake_eyes then
                    loss <= true;
                else
                    point := number;
                end if;
```

```
            else
                if number = 7 or number = 11 then
                    loss <= true;
                elsif number = point then
                    win  <= true;  -- else roll again
                end if;
            end if;
        end if;
    end process;
end no_cheating;
```

This is an example of a state machine with asynchronous reset ("new_game") clocked by the input "roll". Output signals "win" and "loss" are registered and reset, state bit "first_roll" is registered and set, and the 4 state bits for "point" are reset. The input number is used on the rising edge of "roll".

Blackjack

```
entity  blackjack is
port    (new_card,clk , new_game: boolean;
                card : integer range 2 to 11 ;
                say_card,say_hold,say_bust : out boolean) ;
end blackjack;
architecture no_cheating of blackjack is
   type play is (hit_me,got_im,test16,test21,hold,bust);
   signal action : play ;
begin
   state_machine : process
      variable total :  integer range 0 to 31;
      variable ace :  boolean;
   begin
      wait until clk;
      if new_game then
         action <= hit_me.
         total := 0;
         ace := false;
      else
         case action is
         when hit_me =>
            if new_card then
               total  := total + card;
               ace   := (card = 11) or ace;
               action :<= got_im;
            end if;
         when got_im =>
            if not new_card then
               action :<= test16;
            end if;
```

```
            when test16 =>
                if total > 16 then
                    action :<= test21;
                else
                    action :<= hit_me;
                end if;
            when test21 =>
                if total < 21 then
                    action :<= hold;
                elsif ace then
                    total := total - 10;
                    action :<= test16;
                else
                    action :<= bust;
                end if;
            when others => null;
        end case;
    end process;

    -- decode outputs

    say_card <= action = hit_me;
    say_bust <= action = bust;
    say_hold <= action = hold;

end;
```

Traffic Light Controller

```
entity tlc is
    port (clk, reset, farm_traffic: boolean;
        farmroad_green_on,
        farmroad_yellow_on,
        farmroad_red_on,
        highway_green_on,
        highway_yellow_on,
        highway_red_on: out boolean);
end tlc;
architecture a_classic of tlc is
    type states is (highway_green, highway_yellow,
                    farmroad_green,
                    farmroad_yellow);
    type time is range 0 to 100;
    signal short_timer, long_timer: time;
    signal tlc_state: states;
    signal set_timer, short_timeout, long_timeout: boolean;

    procedure timer (signal set, clk, set_timer : boolean;
                     signal timer : inout time;
                     constant load : time;
                     signal timeout : out boolean) is
    begin
        if set or set_timer then -- gated reset !!
            timer <= load;
            timeout <= false;
        elsif timer = 0 then
            timout <= true;
        else
            timer <= timer - 1;
        end if;
    end;
```

```
begin

    timer ( reset, clk, set_timer, short_timer, 30, short_timeout);
    timer ( reset, clk, set_timer, long_timer, 100, long_timeout);

    process ( clk )
    begin
        if clk'event and clk then
            if reset then
                tlc_state <= highway_green;
                set_timer <= false;
            else
                set_timer <= false;
                case tlc_state is
                when highway_green  =>
                    if farm_traffic and long_timeout then
                        tlc_state <= highway_yellow;
                        set_timer <= true;
                    end if;

                when highway_yellow =>
                    if short_timeout then
                        tlc_state <= farmroad_green;
                        set_timer <= true;
                    end if;

                when farmroad_green =>
                    if not farm_traffic or long_timeout then
                        tlc_state <= farmroad_yellow;
                        set_timer <= true;
                    end if;
```

```
            when farmroad_yellow =>
                if short_timeout then
                    tlc_state <= highway_green;
                    set_timer <= true;
                end if;
            end case;
        end if;
    end if;
end process;
    -- Decode states and drive lights.
farmroad_green_on  <=  tlc_state = farmroad_green;
farmroad_yellow_on <=  tlc_state = farmroad_yellow;
farmroad_red_on    <=  tlc_state = highway_green or
                       tlc_state = highway_yellow;
highway_green_on   <=  tlc_state = highway_green;
highway_yellow_on  <=  tlc_state = highway_yellow;
highway_red_on     <=  tlc_state = farmroad_green or
                           tlc_state = farmroad_yellow;
end a_classic;
```

A Simple ALU

```
package alu_types is
    type ops is (add, nop, load, loadc, op_and, op_or, shl, shr);
    subtype data_path is bit_vector (0 to 7);
end alu_types;

use work.alu_types.all;
entity alu is
    port (a: buffer data_path;  instr: ops;
                    clk: boolean;  b: data_path);
end alu;

architecture simple of alu is
begin
    process
   function "+" (a, b: bit_vector) return bit_vector is
            variable sum: bit_vector (0 to a'high);
            variable c: bit:= '0';
        begin
            for i in 0 to a'high 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;
    return sum;
        end;
        function shiftl(a: bit_vector) return bit_vector is
            variable shifted: bit_vector (0 to a'high);
        begin
            -- Shift_left and shift_right are coded differently
            -- only for the sake of example.
            for i in 0 to a'high -1 loop
                shifted(i + 1) := a(i);
            end loop;
            return shifted;
        end;
```

```
        function shiftr(a: bit_vector) return bit_vector is
            constant highbit: integer := a'high;
            variable shifted: bit_vector (0 to highbit);
        begin
            -- Shift_left and shift_right are coded differently
            -- only for the sake of example.
            shifted(0 to highbit - 1) := a(1 to highbit);
            return shifted;
        end;

    begin
        wait until clk;
        case instr is
            when add => a <= a + b;  -- Uses "+" function.
            when nop => null;        -- A null statement,
            when op_and =>a <= a and b;
            when op_or  => a <= a or b;
            when shl => a <= shiftl(a);
            when shr => a <= shiftr(a);
            when load => a <= b;
            when loadc  => a <= not b;
        end case;
    end process;
end simple;
```

Hello

This first design compiles to a simple waveform generator with one input (the clock) 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'. The design has four flip-flops that make up the counter.

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

architecture simple of hello is
    constant char_sequence:string := "hello world";
    signal step:integer range 1 to char_sequence'high;
begin

    counter : process
    begin
        wait until clock;
        if reset then
            step <= 1;
        else
            if step = char_sequence'high then
                step <= 1;
            else
                step <= step +1;
            end if;
        end if;
    end process;

    decoder : char <= char_sequence(step);
end simple;
```

This example is a variant of the previous design; in this case the outputs are registered as well. The state is maintained in the counter so there are 11 flip-flops. Note how the counter "step" is initialized (set / reset) by the signal "reset", but the wraparound for the counter (step = char_sequence'high) will be loaded using the data input.

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

architecture second of hello is
begin
    process
        constant char_sequence:string := "hello world";
        variable step:integer range 1 to char_sequence'high;
    begin
        wait until clock;
        if reset then
            step := 1;
        elsif step = char_sequence'high then
            step := 1;
        else
            step := step +1;
        end if;
        char <= char_sequence(step);
    end process;
end second;
```

This final implementation is not optimal as it specifies two more flip-flops that are not really required (it is, however, what the code specifies). The third architecture no longer uses a counter, but uses the output as current state. The situation is complicated by the repeated characters, therefore, we use a variable to keep track of these. This design has 9 flip-flops.

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

architecture third of hello is
begin
   process
    type rpt_char is ( first, second, third, fourth );
    variable rpt : rpt_char;
    begin
        wait  until clock;
        case char is
        when 'h' => char <= 'e';
        when 'e'=> char <= 'l';
        when 'l' => if rpt = first then
                            char <= 'l';
                            rpt := second;
                    elsif rpt = second then
                            char <= 'o';
                            rpt := third;
                    else
                            char <= 'd';
                            rpt := first;
                    end if;
```

```
        when 'o' => if rpt = third then
                            char <= '  ';
                            rpt := fourth;
                    else
                            char <= 'r';
                            rpt := first;
                    end if;
        when '  ' => char <= 'w';
        when 'w' => char <= 'o';
        when 'r' => char <= 'l';
        when others => char <= 'h';
        end case;
        if reset then      -- yet another way to implement reset
           char <= 'h';
        end if;
    end process ;
end third ;
```

Fifo

```
-- A Parameterized Fifo
-- uses 'one hot' std_logic_vector counters
-- shows techniques for more efficient implementation
-- based on designers knowledge that counters are one hot.

library ieee;
use ieee.std_logic_1164.all;

entity fifo4 is
    - change size of fifo by changing vales of generics
    generic(FifoDepth : integer := 16;
            FifoWidth : integer := 5);
    port(D      : std_logic_vector(FifoWidth - 1 downto 0);
        Rst     : std_logic;
        Oe      : std_logic;
        Ldclk   : std_logic;
        Unclk   : std_logic;
        Empty   : buffer std_logic;   -- these outputs are fed back
        Full    : buffer std_logic;   -- to flip-flop clock enable
        Full_2  : out std_logic;
        Empty_2: out std_logic;
        Q       : out std_logic_vector(FifoWidth - 1 downto 0));
end fifo4;

architecture dataflow of fifo4 is
    -- types
    subtype data_path is std_logic_vector(0 to FifoWidth -1);
    subtype fifo_ptr is std_logic_vector(1 to FifoDepth);
    type fifo_type is array (fifo_ptr'range) of data_path;

    -- constant
    constant tristate  : data_path := (others => 'Z');
    constant dont_care : data_path := (others => '-');
```

```
    -- signals
    signal Fifo : fifo_type;
    signal ReadPtr, WritePtr : fifo_ptr;
    signal Qint : data_path;

    -- rotate, no logic when 'r' is a constant
    function "rol"(l: std_logic_vector; r: integer) return
                          std_logic_vector is
    begin
        return To_StdLogicVector( To_bitvector(l) rol r );
    end  ;

    -- the ring counter, fast but register greedy, it is 'one hot'
    -- uses std_logic_vector "rol" above
    procedure ring_counter (signal rst , clk, clkena : std_logic;
                    signal count : inout std_logic_vector) is
        constant one_hot : fifo_ptr :=  ( 1 => '1' , others => '0');
    begin
        if Rst = '0' then
            count <= one_hot;
        elsif rising_edge(clk) then
        if clkena = '1' then
                count <= count rol 1;
            end if;
        end if;
    end;

-- the ring counters are coded 'one hot',
-- so we write a faster compare,
-- because the predefined "/=" doesnt know
-- std_logic_vector is one hot
-- in this case the (l'length /= r'length) test
-- is never true in the fifo example
```

```
function "/="(l,r: std_logic_vector) return std_logic is
    -- normalize the ranges because
    -- loop below assumes the same range
    alias ll: std_logic_vector (l'length downto 1) is l ;
    alias lr: std_logic_vector (r'length downto 1) is r ;
begin
    if l'length /= r'length then
        return '1';
    end if;
    for i in ll'reverse_range loop
        -- if same hot bit then args are equal, so /= returns zero
        if (ll(i) AND lr(i)) = '1' then
            return '0';
        end if;
    end loop;
    return '1';
end;

begin
    -- FifoDepth flip-flops with clock enable,
    -- low bit preset, rest are reset
    read_ring_counter:
        ring_counter (Rst, Unclk, Empty, ReadPtr);

    -- FifoDepth flip-flops with clock enable,
    -- low bit preset, rest are reset
    write_ring_counter:
        ring_counter (Rst, Ldclk, Full, WritePtr);
```

```
read_fifo:
process(ReadPtr, Fifo)
begin
    Qint <= dont_care;  -- because use of ReadPtr is one hot
    for i in ReadPtr'range loop
        if ReadPtr(i) = '1' then
            Qint <= Fifo(i);
        end if;
    end loop;
end process read_fifo;

-- (FifoWidth * FifoDepth) flip-flops with clock enable
write_fifo:
process (Ldclk)
begin
        if rising_edge(Ldclk) then
            if Full = '1' then
                for i in WritePtr'range loop
                if WritePtr(i) = '1' then
                    Fifo(i) <= D;
                end if;
                end loop;
            end if;
        end if;
end process write_fifo;
```

```
    -- FifoWidth tristate buffers
    Q <= Qint when Oe = '1' else tristate;

    -- active low control signals ,
    -- the "/=" and "rol" are the overloaded functions defined above

    Full       <=  ReadPtr/=  WritePtr rol 1;
    Full_2  <=  ReadPtr/=  WritePtr rol 3;
    Empty<=  ReadPtr/=  WritePtr;
    Empty_2 <=  ReadPtr rol 2  /=  WritePtr;

end dataflow;
```

8 - Synthesis of VHDL Types

Introduction

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 Metamor, 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 Metamor to perform additional logic optimizations. Subtypes use the element encodings of their base, and types define additional "don't care" conditions. Dont care may be explicitly specified using 'enum_encoding' as described in the next section. See also: Std_logic_ll64.

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 "dont 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 whitespace. 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 dont care, and the encoding 'M' represents a metalogic value.

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.

An example of the use of enum_encoding is the PREP 4: Using enum_encoding implementation.

Std_logic_ll64

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 Metamor) 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 dont care logic optimization if evaluation results in assigning a metalogic value or dont care value.

These semantics are designed for compatibility with simulation; if an 'X' propagates in simulation, there may be dont 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 dont care value.

An example of the use of std_logic is in PREP 4: Using std_logic_1164 .

One Hot Encoding

User defined encoding may be used to specify one hot encoding. For instance, in Prep 4 the enumerated type 'state_type' could be redefined as one hot simply by changing the enum_encoding attribute. There are two possible forms:

```
type state_type is (st0,st1,st2,st3,st4,st5,st6,st7,st8,
                    st9,st10,st11,st12,st13,st14,st15);
-- 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. Dont 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 dont care specification. The recommended style is to use enumerated types and enum_encoding.

Numeric Types

Numeric types consist of integer, floating point, and physical types. Two encoding schemes are used by Metamor 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.

9 - Managing Large Designs

Using Hierarchy

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.

The 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 Metamor 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 logicoptimize granularity, hierarchical compiles, and silicon specific components.

Controlling the logic optimize granularity

The domain of the Metamor 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. Compile methodology is discussed in Compiling.

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 Hierarchical compile). 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 stupid of child is
begin
    c <= a and b;
end stupid;


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

architecture family of parent is
    signal w, r: bit;
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;
```

A more powerful method of instantiation using components is described in the following section.

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 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. Metamor's 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" (LRM, Section 11.2). The library "work" is a special case, as it is an alias for the current library.

In Metamor, 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 Metamor 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;
```

There are two mechanisms for associating VHDL libraries with source files. The first assumes a library statement directly specifies a file name, the second uses a compile option to associate one or more files with a library name. Power users will probably want to use the second mechanism. The first mechanism provides a simple default support for libraries.

Direct association

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

Alias association

A library is defined as a list of files by a compile option. The library alias compile option allows a library to be defined as containing a list of files in the order thay are to be analyzed. See D - Compile options.

Common uses are to add files such as the Synopsys library to the library IEEE:

```
IEEE  <path>\ieee.vhd <path>\synopsys.vhd
```

or to place a package shared between separtely compiled design units in the library WORK:

```
WORK my_pack.vhd
```

There are three special cases. Aliases of the library "std" are ignored. Also the file metamor.vhd must be in a library named "metamor" ; and the file ieee.vhd must be in a library named "ieee." It is not good practice to list unused files because large libraries may use significant amounts of memory.

Metamor VHDL Libraries

The library files supplied with Metamor 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	Metamor specific package 'attributes' Metamor 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'
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.

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 VHDL Design Libraries 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)

See VHDL Design Libraries for information on alias association.

metamor.attributes

Declarations of the metamor specific synthesis 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 Metamor compiler. Vlbit based designs may (or may not) require some modification; this is described below.

VIbit designs may make use of register inference conventions that are different from those used by Metamor. The case to look for is preset/reset, which is specified in a wait statement along with the clock. Using Metamor, 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. When run in verbose mode, the compiler reports register types, and number of instances.

ieee.std_logic_arith
ieee.std_logic_unsigned

These packages are versions of the Synopsys packages that have been optimized for use with the Metamor 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. When run in verbose mode the compiler reports register types, and number of instances.

These packages are in the file synopsys.vhd (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.

See VHDL Design Libraries for information on alias association.

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 Metamor 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 Metamor 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. Metamor 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 Metamor. 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 Metamor. .

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.

10 - Logic and Metalogic

Introduction

An 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 dont care) as a metalogic value.

This section is <u>not for beginners !</u>

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 (see LRM) 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, as shown in the following example.

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. We will see that 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 we can see that the variable 'w' depends only on the initial value (w: integer := 1;) and the current value of 'w' (w := w + w). We can say 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 we add 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 dont care (-) for logic optimization. We add these metalogic values as alternatives to logic values (0,1) within the tools. These metalogic values may have different meanings to different tools.

Unknowns allows us 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 we know that the design description does not represent logic.

Dont 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 dont cares. This alternative is to describe the design using multi-valued enumerated types in place of arrays of booleans. Compare 'state_type' in the two versions of the PREP 4 design: PREP 4: Using enum_encoding and PREP 4: Using std_logic_1164. The two descriptions produce equally efficient results.

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 we make use of the attribute 'enum_encoding' to describe which elements describe logic values and which describe metalogic values (see Std_logic_ll64). Metamor 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 Metamor's logic minimization.

When using std_logic, the metalogic values 'U' 'X' 'W' and '-' have one meaning to a simulation tool and another (dont care) to a synthesis tool. Within Metamor, metalogic values are not simply thrown away, but are treated in expressions as dont 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. 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 !

11 - XBLOX and LPM

Macrocells

Macrocells are components whose behavior is defined outside of VHDL by some other (downstream) tool. Examples of macrocells include Xilinx XBLOX macros, LPM macros, or a target hardware specific macrocell such as a micro controller. The Metamor compiler handles macrocells in a manner similar to Hierarchical Compilation described in 9 - Managing Large Designs.

To declare a macrocell simply add the attribute Macrocell (value true) to the component declaration.

```
component compare--component declaration
    port (a, b: bit_vector(4 downto 0);  c: out bit);
end component;
attribute macrocell of compare :
    component is true;-- attribute macrocell
```

Usage is exactly like Hierarchical Compilation with one exception; there is no requirement that the component match an entity because no such entity exists (the behavior of a macrocell is defined by some other tool). You may instantiate this macrocell as you would any other component. For example:

```
U1: compare port map (a, w, r);
```

The compiler will issue a run time message:

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

This is not an error, simply a note that this component's behavior is not defined in VHDL , it will be defined by the macrocell 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 LPM, XBLOX and Open Abel 2 output formats. This is described in the following section.

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 verbose command line option will enable the compiler to print the names of inferred macrocells. See D - Compile options.

LPM and XBLOX

The LPM and XBLOX specifications allow extended macrocell support. :

- Macrocells may be parameterized. This means that each instance of a particular macrocell may describe different amounts of logic.
- Libraries of component declarations are provided (see section VHDL Design Libraries)
- Macrocell are inferred. This means that the compiler automatically recognizes some VHDL statements and expressions as the equivalent macrocell.

See also: D - Compile options

Macrocell Instantiation

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 XBLOX default. And, in the usual VHDL manner, we may use named association 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 XBLOX 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 (see 5 - Programming Sequential Logic) 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;
```

12 - Synthesis Attributes

Predefined 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. A list of the supported attributes and their definitions is contained in A - VHDL Quick Reference. Some examples are :

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

If we declare a subtype of type integer

```
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. Metamor uses this capability to define attributes for synthesis. The declaration of these attributes may be found in the system library 'metamor'. 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;
```

Attribute 'critical'

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, Metamor 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 factoring function as specified by the user. It is also used to specify signals that will have net attributes for downstream tools.

For example, look at the top level of the of PREP 4: Using std_logic_1164 implementation. Critical is used here 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 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.

Attribute 'enum_encoding'

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. For further information see User Defined Encoding and One Hot Encoding . Also see Don't Cares for more on enum_encoding.

Attribute part_name

Metamor allows designers to pass place and route information to fitters, or netlists. This information has no meaning to Metamor, 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 in the Metamor library as :

```
attribute part_name : string;
```

The value may be specified as follows:

```
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.

Attribute pinnum

Metamor allows designers to pass place and route information to fitters, or netlists. This information has no meaning to Metamor, 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 in the Metamor library 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
                           -- bit 0 of 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(0) get "a3"
   attribute pinnum of d : signal is  "w1,W2,w99"; -- descending order
                           -- c(27) gets "w1"
   attribute pinnum of e : signal is  "2";         -- single bit

end special_attributes;
```

Attribute property

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

The property attribute is used to pass an arbitary 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 in the Metamor library 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 we concatenate strings and a new line 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;
```

Atribute Xilinx_BUFG

This attribute is ignored if the compiler output format is not XNF. If the output format is XNF 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;
```

Atribute Xilinx_GSR

This attribute is ignored if the compiler output format is not XNF. If the output format is XNF, this attribute is used to mark a net that uses the global set or reset resource. It has the same behavior as a STARTUP symbol.

The attribute must be declared as :

```
attribute Xilinx_GSR : 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_GSR : boolean;

    -- mark port RST as using GSR routing resource
    -- use this OR use startup symbol below
    attribute Xilinx_BUFG of RST : signal is true;

end prep7;


architecture behavior of prep7 is
    -- Xilinx 4k startup
    component STARTUP
        port (gsr,gts,clk : in std_logic := '0';
            q2,q3,q1q4,donein : out std_logic);
    end component;
begin
    -- Instantaiate startup OR use Xilinx_GSR as above
    U1 : STARTUP port map (gsr => rst);
end;
```

If design units are separately compiled and linked with XNFMERGE and one unit contains a startup symbol, the units that do not contain the startup symbol should use the Xilinx_GSR attribute.

Attribute foreign

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 described in 9 - Managing Large Designs.

Attribute array_to_numeric

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 bit_vector to integer as shown in 10 - Logic and Metalogic. This particular conversion specifies no logic but is slow to compile. This aspect is also discussed in some detail in 10 - Logic and Metalogic.

Metamor provides an attribute, 'array_to_numeric', 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 a array formal parameter returning a numeric type when the parameter and the return value have the same synthesis encoding. See 8 - Synthesis of VHDL Types for a discussion of 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.

Attribute macrocell

When a component instance has no entity bound to it the macrocell attribute is used to specify a different naming convention for the component's multi-bit formal ports. Examples of multi-bit types are integer and std_logic_vector. Examples of single bit types are bit and std_logic. The macrocell attribute is required for parameterized macrocells such as XBLOX and LPM.

The modified naming convention is "NameNumber" with no other 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 compatibility with certain downstream 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.

Attribute Ungroup

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.

Attribute Inhibit_buf

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 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;
```

Attributes for Downstream Tools

The Metamor compiler makes use of user defined attributes to pass information to downstream tools. The Metamor 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 tables show how these VHDL attributes (attached to specific VHDL objects) are propagated to EDIF, XNF, OpenAbel 2, CUPL, 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

XNF	part_name	pinnum	property	name/ value(4)
Entity, top level	PART		file(5)	
Port, top level (2)		EXT	EXT, or SIG	EXT, or SIG
Entity, lower level (3)				
Port, lower level (3)				
Signal, inferring flip-flop, latch, or tristate			SYM	SYM
Signal with attribute 'critical' true			SIG	SIG
Component (unbound) (3)			SYM	SYM
Component (unbound) instance label (3)			SYM	SYM

Open Abel 2	part_name	pinnum	property	name/value
Entity, top level	DEVICE		PROPERTY	
Port, top level		PINS	PINS	

CUPL	part_name	pinnum	property	name/value
Entity, top level	DEVICE		file (5)	
Port, top level		PIN	PIN	

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 Metamor compiler is allowed to retain the signals in the output netlist with the Metamor Critical attribute.

```
architecture x of y is
    signal c : std_logic;
    --allow Metamor 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 net (XNF only)
```

```
    -- 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 TMN is attribute type string
    attribute TMN  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;
    ....
```

13 - Synthesis Coding Issues

Introduction

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.

See Also

How to be Happy

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 fanin 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;
```

An example where this is not the expected behavior may be hidden in your code .....

Some of the predefined VHDL operations also imply long signal paths, see 4 - Programming Combinational Logic.

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 need 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. And 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. Metamor supplies some tuned packages; study these before attempting your own port.

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 array_to_numeric attribute documented in 12 - Synthesis Attributes.

Further discussion on why this function is slow to compile may be found in 10 - Logic and Metalogic.

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
    ecode_signal <= address = "101010";
else
    decode_signal <= false;
end if;
```

You can use the verbose compile option, which will log 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 (from the Fifo 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 ;
```

We code for this case 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;
```

You can use the verbose compile option. It will log 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. Other examples in this section show unintended latches. 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.

A - 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*.

Lexical Elements

- comments from -- to end of line
- characters 'a' 'Z' ':'
- strings "hi there"
- bit strings b"0101"o"05"x"5"
- integers 123_4562E22#0101#
- identifiers , a letter followed by letters, numbers, or underbar : hellohello7h_e_l_l_o
- extended identifiers , any characters delimited by backslash

Reserved Words

The following words are reserved in standard VHDL (regardless of case) and cannot serve as user-defined identifiers:

abs
access
after
alias
all
and
architecture
array
assert
attribute

begin
block
body
buffer
bus

case
component
configuration
constant

disconnect
downto

else
elsif
end
entity
exit

file
for
function

generate
generic
group
guarded

if
impure
in
inertial
inout
is

label
library
linkage
literal
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others

out

package
port
posponed
procedure
process
pure

range
record
register
reject
rem
report
return
rol
ror

select
severity
shared
signal
sla
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 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 statements :

```
-- 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 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

-- see Constrained Constructs

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	hierarch y path string
E'instance_name	hierarch y and binding string

VHDL constructs

The following is a partial list of VHDL constructs. Some constructs are constrained in their usage. For a list of these and unsupported constructs see the following section. (This list format is based on the VHDL 1076 LRM chapters.)

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

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

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

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 Metamor 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 statement

- 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 cannot be recursive.
- Formal part of a named association may not be a function call.
- 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). See also 10 - Logic and Metalogic.

- 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 set of values as the equivalent integer type.
- While loop and unconstrained loop execution completion must depend only on metalogic expressions.

B - PREP Examples

PREP 1

```
 package   typedef   is
   subtype byte   is bit_vector (7  downto 0);
end;

use work.typedef.all;

entity data_path   is
   port    (clk,rst,s_l : boolean;
       s0, s1 : bit;
       d0, d1 ,d2, d3 : byte;
       q :  out byte);
end data_path;

architecture instance  of data_path   is
   signal reg,shft : byte;
begin
   process (clk,rst)
   begin
       if rst  then                  -- async reset
          reg <= x"00";
          shft   <= x"00";
       elsif clk  and clk'event  then    -- clock shft  and reg
          case s0 & s1   is           -- mux
          when b"00" => reg <= d0;
          when b"10" => reg <= d1;
          when b"01" => reg <= d2;
          when b"11" => reg <= d3;
          end  case;
```

```
if s_l then                -- conditional shift
                shft <= shft rol 1;
            else
                shft <= reg;
            end if;
        end if;
    end process;
    q <= shft;
end;

use work.typedef.all;

entity prep1 is
    port  (CLK,RST,S_L : boolean;
           S0, S1 : bit;
        ID : bit_vector(23 downto 0);
           IPD : byte;
        Q :  out byte);
end prep1;

architecture top_level of prep1 is
    component data_path
        port  (clk,rst,s_l : boolean;
               s0, s1 : bit;
               d0, d1 ,d2, d3 : byte;
            q :  out byte);
    end component;
```

```
begin
   first : data_path  port  map (CLK,RST,S_L,
                                 S0,S1,
                                 IPD,
                                 ID(7  downto 0),
                                    ID(15  downto 8),
                                    ID(23  downto 16),
                                    Q);
end;
```

PREP 2

```
 package  typedef  is
   subtype byte  is integer  range 0 to 2**8 -1;  -- 8 bit
end;

use work.typedef.all;

entity prep2  is
   port(CLK,RST,SEL,LDCOMP,LDPRE : boolean;
          DATAa , DATAb : byte;
       DATAc :  out byte);
end prep2;

architecture behavior  of prep2  is
   procedure reg ( signal clk,rst,ld : boolean; signal d : byte;
                        signal q :  out byte)  is
   begin
      if rst  then
         q <= 0;
      elsif clk  and clk'event  then
         if ld  then
            q <= d;
         end  if;
      end  if;
   end;
```

```
procedure counter ( signal clk,rst,ld : boolean; signal d : byte;
                    signal q :  inout byte)   is
    begin
    if rst  then
       q <= 0;
    elsif clk  and clk'event  then
       if ld  then
          q <= d;
       else
          q <= q + 1;
       end  if;
    end  if;
 end;

 signal bus1,bus2,bus3,bus4 : byte;
 signal load : boolean;
begin

 reg( CLK, RST, LDPRE, DATAb,  bus1);-- upper register
 reg( CLK, RST, LDCOMP, DATAb, bus2); -- lower register

 counter( CLK, RST, load,  bus3, bus4);-- counter register

 bus3 <= bus1  when sel   else DATAa;    -- mux
 load <= bus2 = bus4;                    -- compare
 DATAc <= bus4;
end behavior;
```

PREP 3

```
 package   typedef   is
    subtype byte   is bit_vector (7  downto 0);
end;

use work.typedef.all;

entity state_machine   is
    port (clk,rst : boolean;
        inn : byte;
        outt :  out byte);
end state_machine;

architecture behavior  of state_machine   is
begin
    process(clk,rst)
         type state_type   is (start,sa,sb,sc,sd,se,sf,sg);
         variable current_state : state_type;
    begin
        if rst  then
            current_state := start;
            outt <= x"00";
        else
            if clk  and clk'event  then
                case current_state   is
```

```
when start =>
    if inn = x"3c"  then
        current_state := sa;
        outt <= x"82";
    else
        outt <= x"00";
    end  if;
when sa =>
    if inn = x"2a"  then
        current_state := sc;
        outt <= x"40";
    elsif inn = x"1f"  then
        current_state := sb;
        outt <= x"20";
    else
        outt <= x"04";
    end  if;
when sb =>
    if inn = x"aa"  then
        current_state := se;
        outt <= x"11";
    else
        current_state := sf;
        outt <= x"30";
    end  if;
when sc =>
    current_state := sd;
    outt <= x"08";
when sd =>
    current_state := sg;
    outt <= x"80";
when se =>
    current_state := start;
    outt <= x"40";
```

```
            when sf =>
                current_state := sg;
                outt <= x"02";
            when sg =>
                current_state := start;
                outt <= x"01";
            end case;
        end if;                    -- clocked logic
    end if;                        -- reset logic
  end process;
end behavior;

entity prep3 is
  port (CLK,RST : boolean;
    INN : byte;
    OUTT : out byte);
end prep3;

architecture top_level of prep3 is
  component state_machine
    port (clk,rst : boolean;
      inn : byte;
      outt : out byte);
  end component;
begin
  one : state_machine port map (CLK,RST,INN, OUTT);
end;
```

PREP 4: Using enum_encoding

```
library metamor;
use metamor.attributes.all;
package encode1 is
    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
    "00101 00000 10000 00100 10100 " &
    "01100 01000 10101 10001 11000 " &
    "10011 00011 00001 01101 01001 11001";

    type byte is (o0,o1,o2,o3,o4,o5,o6,o7,o8,o9,
                o10,o11,o12,o13,o14,o15);
    attribute enum_encoding of byte : type is
    "00000000 00000110 00011000 01100000 1------0 -1----0- " &
    "00011111 00111111 01111111 11111111 -1-1-1-1 1-1-1-1- " &
    "11111101 11110111 11011111 01111111";
end ;

use work.encode1.all;
entity prep4 is
    port (clk,rst : boolean;
        i : bit_vector(7 downto 0);
        o :  out byte);
end prep4;
```

```
architecture instance  of prep4 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 x"00"          => machine <= st0;
            when x"01" to x"03" => machine <= st1;
            when x"04" to x"1f" => machine <= st2;
            when x"20" to x"3f" => machine <= st3;
            when  others        => machine <= st4;
            end  case;
          when st1 =>
            if I(1  downto 0) = b"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 b"00" => machine <= st6;
    when b"01" => machine <= st8;
    when b"10" => machine <= st9;
    when b"11" => machine <= st1;
    end  case;
when st7 =>
    case I(7  downto 6)   is
    when b"00"  => machine <= st3;
    when b"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 = x"40"  then
                machine <= st15;
            else
                machine <= st8;
            end  if;
        when st12 =>
            if i = x"ff"  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 x"00"          => machine <= st14;
            when x"01" to x"3f" => machine <= st12;
            when  others        => machine <= st10;
            end  case;
        when st15 =>
            if (I(7) = '1')  then
                case I(1  downto 0)   is
                when b"00" => machine <= st14;
                when b"01" => machine <= st10;
                when b"10" => machine <= st13;
                when b"11" => machine <= st0;
                end  case;
            end  if;
        end  case;
    end  if;
end  process;
```

```
    with machine  select
        O <= o0 when st0,
             o1 when st1,
             o2 when st2,
             o3 when st3,
             o4 when st4,
             o5 when st5,
             o6 when st6,
             o7 when st7,
             o8 when st8,
             o9 when st9,
             o10 when st10,
             o11 when st11,
             o12 when st12,
             o13 when st13,
             o14 when st14,
             o15 when st15;
end;
```

```
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_machine is
    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;
```

```
entity arith is
   port(CLK,MAC,RST: boolean; A,B: integer range 0 to 15;
                  Q: in buffer integer range 0 to 255);
end arith;

architecture behavior of arith is
   signal P: integer range 0 to 255;
begin
   P <= A * B;                        -- Product of A and B
   process(RST,CLK)
   begin
      if (RST) then                      -- Reset
         Q <= 0;
      else
         if (CLK and CLK'event) then       -- Clock (edge
                                triggered)
            if (MAC) then
               Q <= P + Q;
            else
               Q <= P;
            end if;
         end if;
      end if;
   end process;
end behavior;
```

```
entity prep5_4  is
   port(CLK,MAC,RST: boolean; A,B: integer  range 0 to 15;
                  Q: in buffer integer  range 0 to 255);
end prep5_4;

architecture structure  of prep5_4  is
   signal QXinteger  range 0 to 255; -- Q  output from #1
   signal QYinteger  range 0 to 255;  -- Q  output from #2
   signal QZinteger  range 0 to 255;  -- Q  output from #3
   signal QX_Low,QX_High: integer  range 0 to 15;
                           -- A,B inputs  to #2
   signal QY_Low,QY_High: integer  range 0 to 15;
                           -- A,B inputs to #3
   signal QZ_Low,QZ_High: integer  range 0 to 15;   -- A,B to #4

   component arith
      port(CLK,MAC,RST: boolean; A,B: integer  range 0 to 15;
                  Q: in out integer  range 0 to 255);
   end  component;

begin
   one:  arith  port  map (CLK,MAC,RST,  A  ,  B  ,QX);
                                       -- Instance #1
   QX_Low <= QX rem 16;                    -- slice low nibble
   QX_High <= QX / 16;                    -- slice high nibble
   two:  arith  port  map (CLK,MAC,RST,QX_Low,QX_High,QY);
                                       -- Instance #2
   QY_Low <= QY rem 16;                    -- slice low nibble
   QY_High <= QY / 16;                    -- slice high nibble
   three: arith  port  map (CLK,MAC,RST,QY_Low,QY_High,QZ);
                                       -- Instance #3
   QZ_Low <= QZ rem 16;                    -- slice low nibble
   QZ_High <= QZ / 16;                    -- slice high nibble
   four:  arith  port  map (CLK,MAC,RST,QZ_Low,QZ_High, Q);
                                       -- Instance #4
end structure;
```

PREP 6

```
entity prep6  is
   port(CLK,RST: boolean; D: in   integer  range 0 to 65535;
                          Q: buffer integer  range 0 to 65535);
end prep6;

architecture behavior  of prep6  is
begin
   process(RST,CLK)
   begin
      if (RST)  then                  -- Reset
         Q <= 0;
      else
         if (CLK  and CLK'event)  then
                                 -- Clock (edge triggered)
            Q <= Q + D;              -- Add D to accumulator Q
         end  if;
      end  if;
   end  process;
end behavior;
```

PREP 7

```
entity prep7  is
   port(CLK,RST,LD,CE: boolean; D: integer  range 0 to 65535;
                        Q: buffer integer  range 0 to 65535);
end prep7;

architecture behavior  of prep7  is
begin
   process(RST,CLK)
   begin
      if RST  then                    -- Async Reset
         Q <= 0;
      else
         if  CLK  and CLK'event  then
               -- Clock (edge triggered)
            if LD or CE  then
               -- load dominates clock enable, OR clkena pin
               if LD then             -- sync load
                  Q <= D;
               else
                  Q <= Q + 1;
               end if;
            end  if;
         end  if;
      end  if;
   end  process;
end behavior;
```

PREP 9

```
package   typedef   is
   subtype byte   is bit_vector(7  downto 0);
end ;

use work.typedef. all;

entity prep9   is
   port    (clk,rst,as,ce : boolean; al,ah : byte;
              be :  out boolean; q :  out byte);
end prep9;

architecture only_level  of prep9   is
   procedure decoder( signal clk,rst,as : boolean;
                      signal al,ah : byte;
                   signal be :  out boolean;
                   signal q :  out byte)   is
   begin
      if rst  then
         q <= x"00";
         be <= false;
      elsif clk  and clk'event  then
         if as  then
            be <= false;
            case ah & al   is
            when x"f000" to x"ffff"     => q <= x"80";
            when x"efff"  downto x"e800" => q <= x"40";
            when x"e7ff"  downto x"e400" => q <= x"20";
            when x"e3ff"  downto x"e300" => q <= x"10";
            when x"e2ff"  downto x"e2c0" => q <= x"08";
            when x"e2bf"  downto x"e2b0" => q <= x"04";
            when x"e2af"  downto x"e2ac" => q <= x"02";
            when x"e2ab"                 => q <= x"01";
            when  others                 => q <= x"00";
               be <= true;
            end  case;
```

```
            else
                q <= x"00";
                be <= false;
            end  if;
        end  if;
    end;

    signal q1,q2 : byte;
begin
    one : decoder(clk,rst,as,al,ah,be,q);
--  two : decoder(clk,rst,ld,ce,q1,q2);
--  q <= q2;
end;
```

C - Error Message Index

Words in *italics* will be substituted in the actual error message. The information in this section is 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. Note that the workarounds listed are only suggestions; it is not possible for the language compiler to know your intention for using a particular set of VHDL language statements.

To view an error message, select a number below:

001

Unexpected end of file.

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).

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.

002

Syntax error near '*operator*'.

The compiler has encountered an unexpected sequence of characters or language tokens. The error is associated with the indicated VHDL operator.

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.

003

Syntax error near '*name*'.

The compiler has encountered an unexpected sequence of language elements. The error is associated with the indicated identifier name.

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.

004

Based literal format is incorrect.

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.

Check the syntax of the literal to make sure it conforms to the requirements of the specified number base.

005

Unexpected non-graphic character found.

The compiler has encountered a character that is not a part of the defined VHDL character set.

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.

006

An identifier may not begin with the special character '*character*'.

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. 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.

007

Unable to open file '*name*'.

The compiler has encountered an error when attempting to open the indicated file.

Check to ensure that the indicated filename is correctly spelled, and exists in the current directory or the directory indicated in the file path.

008

A '*name*' must not contain a new line character.

The compiler has encountered a newline character in a quoted string or an extended name. Strings and extended identifiers in VHDL must not contain newline characters.

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.

009

A '*name*' must not contain a CR character.

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.

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 substrings with a carriage return character.

010

A '*name*' must not contain a non-graphic character.

The compiler has encountered an illegal character in a quoted string or extended identifier.

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.

011

A bit string must not contain a new line character.

The compiler has encountered a newline 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 '_'.

Check to make sure that you have placed a terminating quote character on the end of the bit string.

012

Bit string delimiters do not match.

The compiler has encountered an unexpected character at the end of a bit string.

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.

013

Illegal binary value '*character*' in bit string.

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 '_'.

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.

014

A Bit string must not have '_' as its first element.

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.

Check to make sure that the bit string does not begin with a '_' character.

015

A bit string must not contain consecutive under bars '__'.

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.

Check to make sure that the bit string does not include extraneous '_' characters.

016

A bit string must not have '_' as its last element.

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. Check to make sure that the bit string does not end with an extraneous '_' character.

017

Illegal octal value '*character*' in bit string.

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 '_'.

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.

018

Illegal hex value '*character*' in bit string.

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 '_'.

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.

019

Based literal contains illegal character '*character*'.

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 '_' for an if the base specifier is 8).

Check to make sure that the based numeric literal is in a valid numeric format that matches the base specification.

020

Literal Base must not be greater than 16.

The compiler has encountered an invalid based numeric literal. The literal base specification must be in the range of 2 to 16. Check to make sure that the literal has a valid base specification.

021

Literal Base must not be less than 2.

The compiler has encountered an invalid based numeric literal. The literal base specification must be in the range of 2 to 16.

Check to make sure that the literal has a valid base specification.

022

Illegal literal format, missing 'E'.

The compiler has encountered a floating point literal that is incorrectly specified.

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.)

023

A number must not contain '_*character*'.

The compiler has encountered in 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'.)

Check to make sure that there are no invalid characters used in the numeric literal, and that the '_' character is used properly.

024

A number must not have '*character*' as its last character.

The compiler has encountered in invalid character at the end of a numeric literal.

Check to make sure that there are no missing or additional delimiters (such as white space or newline) at the end of the number.

025

An identifier may not contain consecutive under bars '__'.

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. In addition, the '_' character must not be used consecutively.

Check to make sure that there are no invalid characters used in the numeric literal, and check to make sure there are no extraneous consecutive '_' characters in the numeric literal.

026

An identifier may not contain a '*character*'.

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.

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.

027

An identifier may not have '_' as its last character.

The compiler has encountered in 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.

Check to make sure that the '_' character is not used as the last character in the identifier.

028

A character literal must not contain a non-graphic character.

The compiler has encountered a quoted character that is not a graphic character.

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.

029

'mm': unknown command option '*name*'.

The compiler software has been invoked with an unknown command option.

Check the compiler documentation for information about compiler options and option formats.

030

Unable to create a temporary file.

The compiler has encountered a system error while attempting to write a file to the disk.

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.

031

Unable to open a temporary file.

The compiler has encountered a system error while attempting to open an existing temporary file.

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.

032

Unable to write to a temporary file.

The compiler has encountered a system error while attempting to write a file to the disk.

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.

033

Out of memory.

The compiler has encountered a system error while attempting to allocate memory.

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. (Select the Help About menu item from the Windows Program Manager -- or Help About Windows 95 in any Windows 95 folder window -- to determine the amount of free memory available.)

If your design is very large, you should consider partitioning it into multiple, smaller design modules and synthesize those modules independently.

034

Disk is full.

The compiler has encountered a system error when attempting to write a file to the disk.

Check to make sure that you have adequate disk space.

035

Software security protection check failed.

The compiler was unable to find the software security device.

Check to make sure that the software security device is connected properly before running the software.

036

Design too large for Demonstration version.

The compiler is operating in demonstration mode. In this mode, you are restricted in the size of design that can be processed.

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.

082

Unable find package 'standard' in the file 'std.vhd'.

The compiler has encountered a problem in the standard library file, std.vhd.

Check to make sure the std.vhd file has not become corrupted. If necessary, re-install the std.vhd file from the installation disk.

083

Entity '*name*' does not exist in the design.

The compiler was unable to find the indicated entity in the specified input source files.

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 on the command line, and that the desired top-level entity exists.

084

Architecture '*name*' does not exist in the design.

The compiler was unable to find the indicated architecture in the specified input source files.

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 on the command line, and that the desired top-level architecture exists.

085

Input file and output file have the same name.

The compiler has determined that the input and output file names you have specified are the same.

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.

086

Incorrect version of library STD.

The compiler has encountered a problem in the standard library file, std.vhd. The version of the file is not correct.

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.

087

Incorrect version of library METAMOR.

The compiler has encountered a problem in the standard library file, metamor.vhd. The version of the file is not correct.

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.

088

Install error, file *name* is missing.

The compiler has encountered a missing file, this file is part of the product and must be present.

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 product.

089

Install error, file *name* is incorrect version.

The compiler has encountered a file that is part of the product but is from another version of the product. This file is incompatible and must be replaced.

Re-install the product.

100

A *description* '*name*' is used in an expression as a primary, expected a signal, a variable, or a constant.

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.

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.

101

'*name*' has not been declared as a '*description*'.

The compiler has encountered a component instantiation that does not reference a known component, entity or configuration.

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.

102

Mode conflict associating actual '*name*' with formal '*name*'.

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.

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.

103

No actual is specified for generic '*name*''.

The compiler has encountered an incomplete generic mapping. The actual generic value is missing in the generic map.

Check to make sure that all required generic parameters have been specified, or add a default value to the declaration of this generic.

104

Port '*name*' has mode IN, is unconnected and has no explicit default value.

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.

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.

105

Port '*name*' has mode *description* has a type that is unconstrained and may not be unconnected.

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.

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.

106

Block specification must be an Architecture, Block label, or Generate label.

The compiler has encountered a configuration that references an invalid design unit type or other unknown label.

Check to make sure that the block specification in the configuration specifies a valid architecture, block label or generate label.

107

'*name*' is not an Entity.

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.

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.

108

'*name*' is not a Type or Subtype.

The compiler expected a type or subtype name, but has instead encountered an identifier that is not a type or subtype.

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.

109

A Return statement in a procedure must not return an expression.

The compiler has encountered a return statement within a procedure that includes a return value. Return values are not allowed in procedures.

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.

110

A Return statement in a function must return an expression.

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.

Check to make sure that all return statements within the function have valid return values.

111

Operator function has too few parameters.

The compiler has encountered a operator function (overloaded operator) that does not have the required number of parameters for the specified operator.

Check to make sure that the number of function parameters matches the requirements of the specified operator.

112

Operator function has too many parameters.

The compiler has encountered an operator function (overloaded operator) that does not have the required number of parameters for the specified operator.

Check to make sure that the number of function parameters matches the requirements of the specified operator.

113

Cannot type convert a NULL, an aggregate, or a string literal.

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.

Check to make sure that the explicit type conversion is being used for closely related types, or use a type conversion function.

114

'*name*' has no Architecture named '*name*'.

The compiler was unable to find the specified architecture name.

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.

115

'*name*' is already declared as a *description*.

The compiler has encountered a duplicate declaration for the indicated identifier name.

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.

116

Name at end of *description* does not match *description* name.

The compiler has encountered a mismatched name at the end of a design unit, subprogram or other end terminated section of the design.

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.

117

Block configuration must be an Architecture.

The compiler has encountered an invalid binding of a block with an architecture in a configuration statement or declaration.

Check to make sure that the name specified in the block configuration is an architecture, and that the architecture specified exists in the design.

119

Unable to determine the range of a non-scalar type.

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.

Check to make sure that a range is actually needed, or rewrite the design so that a scalar data type is used.

120

Illegal subtype constraint.

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.

Check to make sure that the subtype and base type are compatible with the constraint specified.

121

'*name*' is not an array.

The compiler expected an array object or literal.

Check to make sure that the object or literal you are specifying is an array type.

123

Attempt to select element of an object whose type is not a record.

The compiler has encountered an invalid use of a record field specifier. The object referenced in the statement is not a record type.

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.

124

'*name*' does not conform to declaration in package.

The compiler has encountered in invalid declaration in a package body. The declaration for the indicated name must match the declaration in the corresponding package.

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.

125

'*name*' is not a Physical Unit.

The compiler has encountered an apparent use of a physical type literal that does not specify a valid physical type unit.

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.

126

description '*name*' may not be a prefix for .ALL.

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.

Check to make sure that you have specified a valid package, library, entity or architecture name in the statement.

127

Physical unit prefix must be a number.

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.

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.

128

Range is not within the range of the base type.

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.

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.

129

Illegal NULL in expression, NULL must be in a simple assignment.

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.

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.

130

Others must be the last choice in a selected signal assignment.

The compiler has encountered an illegal use of the choice others. Others is only allowed as the last choice in a series of choices.

Check to make sure that the others choice is at the end of the series of choices.

131

Others must be the last choice in a case statement.

The compiler has encountered an illegal use of the choice others. Others is only allowed as the last choice in a case statement.

Check to make sure that the others choice is at the end of the case statement.

132

Others must be the only choice in a selected alternative.

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 => .....).

Check to make sure that the others choice is the only choice in the selected alternative. You can probably remove the or'd choice.

133

Others must be the only choice in a case alternative.

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 => .....).

Check to make sure that the others choice is the only choice in the selected alternative. You can probably remove the or'd choice.

134

The label at end of the *description* does not match *description* label.

The compiler has encountered an end statement that references a concurrent statement label other than expected.

Check to make sure that you have terminated the concurrent statement with the correct label.

135

An Exit statement must be within a loop statement.

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.

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.

136

An Exit statement specifies a label that is not a Loop label.

The compiler has encountered an exit statement that specifies an invalid loop label.

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.

137

A Next statement specifies a label that is not a Loop label.

The compiler has encountered a next statement that specifies an invalid loop label.

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.

138

A Return statement must be within a Function or Procedure.

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.

Check to make sure that the return statement is being properly used within a function or procedure.

139

A passive process may not contain a signal assignment.

The compiler has encountered a process that is passive (such as one entered in the entity declaration) and has one or more signal assignments.

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.

140

Process has a sensitivity list and a wait statement.

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.

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.

141

Illegal NULL in concurrent signal assignment.

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.

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.

142

Missing block guard expression or signal 'guard'.

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.

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.

143

'guard' is not a signal.

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.

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.

144

Signal 'guard' is not type 'boolean'.

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.

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.

145

Target of un-guarded assignment is guarded.

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.

Check to make sure that you have specified the guarded keyword for all assignments to guarded signals.

146

'*name*' is not a Procedure.

The compiler expected to encounter a procedure name, but the name specified is not a procedure.

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.

147

Positional association must not follow named association.

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.

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.

148

Attribute '*name* has not been declared.

The compiler has encountered an attribute name that has not been declared.

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.

149

Attribute not defined for this object.

The compiler has encountered an attribute use that is not defined for the object the attribute is being applied to.

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.

150

Prefix for attribute 'base must be a type or subtype.

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.

Check to make sure that the type specified is a subtype.

151

Attribute '*attribute* must not have a parameter.

The compiler has encountered an invalid use of a predefined attribute. The indicated attribute does not have a parameter.

Check to make sure that you are using the correct attribute. If necessary, add the appropriate attribute parameter.

152

Attribute 'base must be the prefix of another attribute.

The compiler has encountered an incorrect use of the predefined 'base attribute.

'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.

153

Attribute '*attribute* may not have a parameter if the prefix is a scalar type.

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.

Check to make sure that the attribute is being used correctly.

154

Prefix of attribute '*attribute* must be a discrete type.

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).

Check to make sure that the attribute is being used correctly, and that the prefix is an enumeration type or integer.

155

Attribute '*attribute* must have a parameter.

The compiler has encountered an invalid use of a predefined attribute. The indicated attribute requires a parameter.

Check to make sure that you are using the correct attribute. Add an attribute parameter if necessary.

157

Signal attribute prefix is not a signal.

The compiler has encountered a predefined attribute being used incorrectly. The indicated attribute requires a prefix that is a signal identifier.

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.

158

In an aggregate, positional associations must occur before named associations.

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.

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.

159

Choice Others must only occur once in an aggregate.

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.

Check to make sure that you have only provided one others choice in the aggregate.

160

Choice Others must be last element of an aggregate.

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.

Check to make sure that you have only provided one others choice in the aggregate, and that it is the last choice.

161

Choice Others must be the only choice in an aggregate element association.

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 => .....).

Check to make sure that the others choice is the only choice in the selected alternative. You can probably remove the or'd choice.

162

Unable to *action* library file '*name*'.

The compiler was unable to perform an action (read or write) on the specified library file.

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.

163

'*name*' is a *description* and not a *description*.

The compiler has encountered an unexpected use of the indicated identifier.

Check to make sure that the identifier has been entered correctly, and is the expected type of object, design unit, loop, block, or subprogram.

164

'*name*' is not a user defined attribute.

The compiler has encountered an invalid use of the indicated identifier. The name used is not declared as a user defined attribute.

Check to make sure that 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.

165

Subtype has more dimensions than base type.

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.

Check to make sure that the correct base type has been referenced, and check to make sure that the dimensions of the subtype are compatible with the range of the base type.

166

Subtype index is incompatible with base type index.

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.

Check to make sure that the correct base type has been referenced, and check to make sure that the index of the subtype is compatible with the range of the base type.

167

Base type of subtype must not be a record.

The compiler has encountered an invalid specification of a subtype. The base type of a subtype may not be a record.

Check to make sure that the correct base type has been referenced, and check to make sure that the base type is not a record type.

168

Base type for index constraint must be an array.

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.

Check to make sure that the correct base type has been referenced.

169

Base type of subtype must be an unconstrained array.

The compiler has encountered an invalid specification of an unconstrained array subtype. The base type of a subtype must be an unconstrained array.

Check to make sure that the correct base type has been referenced, or constrain the array subtype with a valid range.

170

'*name*' was declared outside of the function in which it is used.

The compiler has encountered an object being referenced within a function, but that object was not declared within the function.

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.

171

A function may not contain a wait statement.

The compiler has encountered a wait statement within a function. Functions may not include wait statements.

Note that wait statements are legal within procedures, but are not supported for synthesis.

172

A function must be completed by a return statement.

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 .

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.

173

The subtype indication given in the full declaration of '*name*' must conform to that given in the deferred constant declaration.

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.

Check to make sure that the same subtype indications of the deferred and full constant declaration are compatible.

174

Range of a physical type must be an integer.

The compiler has encountered a physical type declaration that includes non-integer units.

Check to make sure that the physical type declaration includes a valid base unit, and that subsequent units are defined using an integer multiplier.

175

Subprogram declaration '*name*', does not have a corresponding body.

The compiler was unable to find a function or procedure body corresponding to the subprogram declaration indicated.

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.

176

Deferred constant declaration '*name*', is not declared in a package body.

The compiler was unable to find a full constant declaration corresponding to the deferred constant declaration indicated.

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.

177

Return statement must be within a Function or Procedure.

The compiler has encountered a return statement that is not within a function or procedure body.

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.

178

Next statement must be within a Loop statement.

The compiler has encountered a next statement that is not within a function or procedure body.

Check to make sure that you are using the next statement correctly in the 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.

179

Reserved word UNAFFECTED may only appear as a waveform in a concurrent signal assignment.

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.)

Check the proper use of the unaffected keyword and modify the design.

180

Attempt to assign to a port with mode IN.

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.

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.

181

Attempt to assign to a port with mode LINKAGE.

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.

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.

182

Attempt to assign to implicit signal 'guard'.

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.

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.

183

Attempt to assign to an alias of a port with mode IN.

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.

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.

184

Attempt to assign to an alias of a port with mode LINKAGE.

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.

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.

185

A *description* cannot be the target of a Signal assignment statement.

The compiler has encountered an invalid use of a signal assignment (<=). The target of a signal assignment must be a signal or port.

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.

186

A *description* cannot be the target of a Variable assignment statement.

The compiler has encountered an invalid use of a variable assignment (:=). The target of a variable assignment must be a variable.

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.

187

Expected a static expression, *description* '*name*' is illegal here.

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.

Check to make sure the expression is static.

188

Signal '*name*' is not readable as it has mode OUT.

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.

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 overspecification.

189

'*name*' is not a static signal name.

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.

Check to make sure that the object or expression is static.

190

Enumerated type contains duplicate element '*name*'.

The compiler has encountered two or more identical enumerated values in an enumerated type declaration.

Check to make sure that you have not incorrectly typed in the enumerated values for the type. Remove or rename the duplicate entries.

191

Array type is not constrained.

The compiler has encountered an unsupported use of an unconstrained array type.

Check to make sure that all array types in your design are provided with valid array bounds (ranges).

192

Function parameter must be mode IN.

The compiler has encountered an incorrect use of a function parameter.

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.

193

Package Body with no Package of same name.

The compiler has encountered a package body that does not correspond to any package in the design.

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.

194

Unable to determine type of array index.

The compiler has encountered an array index that is of an unknown type.

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.

195

Array must have an index constraint.

The compiler has encountered an array without an index constraint, in a context where unconstrained arrays are not allowed.

Check to make sure that the array is provided with an index constraint.

196

Prefix of a selected name cannot be a slice name.

The compiler has encountered an invalid selected name. The prefix of a selected name must be a library, package, block, subprogram or record name.

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.

197

Formal '*name*' in port map does not exist in port declaration.

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.

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.

198

Only the last entry in a group template declaration can include a <>.

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 (<>).

Check to make sure that you have specified a legal group template declaration, and modify the entries accordingly.

200

Value of when expression is outside range of values of selected expression.

The compiler has encountered a when expression that does not match the possible values specified in the selected expression.

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.

201

Value of when expression is outside range of values of case expression.

The compiler has encountered a when expression that does not match the possible values specified in the case condition expression.

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.

202

Operands have types that are incompatible with the operator '*operator*'.

The compiler has encountered an expression that is not legal due to incompatibilities between the operand types and the operator used.

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.

203

'*name*' has not been declared.

The compiler has encountered an identifier that has not been declared, or has been declared but is not visible here.

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.

204

Operands of '*name*' have incompatible types.

The compiler has encountered an expression that is not legal due to incompatibilities between the operand types and the operator being used.

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.

205

Parameter associated with formal '*name*' must be a Variable.

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.

Check to make sure the subprogram actual parameters have been properly entered, and that they match the subprogram formal parameters.

206

Parameter associated with formal '*name*' must be a Signal.

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.

Check to make sure the subprogram actual parameters have been properly entered, and that they match the subprogram formal parameters.

207

Formal parameter '*name*' and its actual parameter have incompatible types.

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.

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.

208

Block guard expression must be type boolean.

The compiler has encountered an invalid block guard expression in a block statement. An implicit guard signal is boolean type.

Check to make sure that the guard expression has a boolean type.

209

Range of an integer type declaration must be some integer type.

The compiler has encountered an invalid range specification in an integer type declaration. The range must specify a valid integer range.

Check to make sure that the range has been correctly specified. Make sure the range is specified using integer values.

210

Unable to determine type of attribute prefix.

The compiler has encountered an ambiguous prefix of an attribute. The type of the object prefix of the attribute is unknown.

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.

211

Prefix of attribute '*attribute* must not be a record.

The compiler has encountered an invalid use of the indicated attribute.

Check to make sure that the attribute prefix is of the intended type, and that the correct attribute is being used.

212

Parameter of an array attribute must be a universal integer.

The compiler has encountered an incorrect use of an array attribute.

Check to make sure that the array attribute parameter is a universal integer value.

213

Operand has a type that is incompatible with the operator '*operator*'.

The compiler has encountered an expression that is not legal due to incompatibilities between the operand types and the operator being used.

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.

214

Exponent is negative, left operand must be a floating point type.

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.

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.

215

No parameter associated with formal parameter '*name*'.

The compiler has encountered an incorrect use of a procedure or function. One or more of the required actual parameters are missing.

Check to make sure that you have specified all required parameters to the procedure or function.

216

There are more actual parameters than formal parameters.

The compiler has encountered an incorrect use of a procedure or function. Too many actual parameters have been specified.

Check to make sure that you have specified the correct number and type of required parameters when invoking the procedure or function.

217

Expected a Procedure and not a Function.

The compiler has encountered a function being used when a procedure was expected. Procedures must be used when no return value is expected.

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.

218

Expected a Function and not a Procedure.

The compiler has encountered a procedure being used when a return value was required. Procedures do not have return values.

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.

219

Function returns an incompatible type.

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.

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.

220

No procedure definition matches '*name*'.

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.

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.

221

No function definition matches '*name*'.

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.

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.

222

No actual associated with formal '*name*'.

The compiler has encountered an incorrect use of a procedure or function. One or more of the required actual parameters are missing.

Check to make sure that you have specified all required parameters to the procedure or function.

223

More than one association specified for formal parameter '*name*'.

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 parameters associated.

Check to make sure that you have specified all required parameters to the procedure or function.

224

The aggregate has an incompatible type in this context.

The compiler has encountered an illegal use of an aggregate. One or more aggregate elements are not of the correct type.

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.

225

The string has an incompatible type in this context.

The compiler has encountered an illegal use of a string literal. An element of the string is not of the correct type.

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.

226

The bit string has an incompatible type in this context.

The compiler has encountered an illegal use of a bit string literal. An element of the bit string is not of the correct type.

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.

227

The direction of the slice is not the same as the direction of the prefix.

The compiler has encountered an index range that does not match the direction of the array prefix.

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.

228

Unable to resolve overloaded procedure '*name*'.

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.

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.

229

Unable to determine type of attribute parameter.

The compiler has encountered an attribute parameter that is of an ambiguous type.

Check to make sure that the type of the attribute parameter is clearly distinguished.

230

Prefix of attribute '*attribute* must be a scalar type.

The compiler has encountered an illegal use of an attribute. The indicated attribute is only allowed for scalar (integer, real, physical or enumerated) types.

Check to make sure that the attribute is being applied to a scalar type.

231

Prefix of attribute '*attribute* must be an array.

The compiler has encountered an illegal use of an attribute. The indicated attribute is only allowed for array data types.

Check to make sure that the attribute is being applied to an array data type.

232

Attribute parameter value exceeds dimensionallity of array.

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.

Check to make sure the attribute parameter matches the corresponding array type declaration.

233

An If statement condition expression must be type boolean.

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.

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.

234

Wait until expression must be type boolean.

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.

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.

235

Select expression must be an integer type, enumerated type, or an array.

The compiler has encountered an invalid use of a select expression. The select expression must be an integer, enumerated type or array.

Check to make sure that you have specified a valid select expression, and that the expression evaluates to an integer, enumerated type or array.

236

Case expression must be an integer, enumerated type, or an array.

The compiler has encountered an invalid use of a case expression. The case expression must be an integer, enumerated type or array.

Check to make sure that you have specified a valid case expression, and that the expression evaluates to an integer, enumerated type or array.

237

Select expression must not be a multi dimensional array.

The compiler has encountered an invalid use of a select expression. The select expression must be an integer, enumerated type or single-dimension array.

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.

238

Case expression must not be a multi-dimensional array.

The compiler has encountered an invalid use of a case expression. The case expression must be an integer, enumerated type or single-dimension array.

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.

239

Unable to determine type of With expression from context.

The compiler has encountered a with expression with an unknown type.

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.

240

Unable to determine type of Case expression from context.

The compiler has encountered a case expression with an unknown type.

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.

241

The type of a With expression must be locally static.

The compiler has encountered a with expression that has a type that is not locally static.

Check to make sure that the type of the with expression is locally static.

242

The type of a Case expression must be locally static.

The compiler has encountered a case expression that has a type that is not locally static.

Check to make sure that the type of the case expression is locally static.

243

Loop range must be an integer type or an enumerated type.

The compiler has encountered an invalid range in a loop statement. Loops ranges must be integer or enumerated types.

Check to make sure that you have correctly specified the loop range.

244

Assert condition must be type 'boolean'.

The compiler has encountered an invalid condition in a assert statement. The condition of an assert statement must evaluate to a Boolean type.

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.

245

Assert severity must be type 'severity_level'.

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).

Check to make sure that you have correctly specified the value of the assert severity.

246

Assert report must be type 'string'.

The compiler has encountered an invalid use of the assert report statement. The report keyword must be followed by a valid string.

Check to make sure that you have correctly specified the assert report string.

247

Shift or rotate right operand must be type 'integer'.

The compiler has encountered an invalid use of the shift or rotate operator. Only integer values are allowed as the shift distance (right operand).

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.

248

Unable to resolve overloaded function '*name*'.

The compiler has encountered an overloaded function that cannot be resolved due to ambiguous parameter types or other conditions.

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.

249

Others is illegal here because aggregate is associated with an unconstrained array.

The compiler has encountered an illegal use of the others choice. The aggregate being specified includes an unconstrained array.

Check to make sure that an unconstrained array was actually intended.

250

Record aggregate contains too many elements.

The compiler has encountered a record aggregate that is invalid, due to too many elements being specified.

Check to make sure that the declaration of the record matches its use in the record aggregate.

251

Others must represent at least one element.

The compiler has encountered an others clause that does not represent any possible elements.

Check to make sure that the others clause is actually needed.

252

Others represents choices of record elements of different types.

The compiler has encountered an invalid use of an others clause. The record elements specified by the others clause do not match.

Check to make sure that the others clause is being used correctly.

253

Record aggregate contains unknown named association.

The compiler has encountered a record aggregate that includes a named association that is not valid.

Check to make sure that the named associations have been properly entered, and all names used in the association are valid.

254

Operands of '*operator*' have incompatible lengths.

The compiler has encountered an invalid expression using the indicated operator. The operands of the expression do not match.

Check to make sure that the correct operands have been specified, and that they have the type and length required.

255

Array index is out of range of array.

The compiler has encountered an array index that is invalid. The index is outside the range of the array declaration.

Check to make sure that the declaration of the array matches the use of the array.

256

Duplicate association in array aggregate, it is a duplicate of association on line *number*

The compiler has encountered an array aggregate association that has already been specified.

Check to make sure that the array association has been correctly entered. Check the duplication association indicated for more information.

257

Duplicate choice in selected signal assignment, it is a duplicate of choice on line *number*.

The compiler has encountered a choice in a selected signal assignment that has already been specified.

Check to make sure that the choice has been correctly entered.

258

Duplicate choice in case statement, it is a duplicate of choice on line *number.*

The compiler has encountered a choice in a selected signal assignment that has already been specified.

Check to make sure that the choice has been correctly entered.

259

Choice Others is required when selected signal assignment expression is a universal integer.

The compiler has encountered an incomplete selected signal assignment. The use of a universal integer has resulted in an others choice being required.

Check to make sure that an others choice has been provided, or do not use a universal integer.

260

Choice Others is required when case statement expression is a universal integer.

The compiler has encountered a case statement that does not include a required others choice due to the use of a universal integer.

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.

261

Missing choice in selected signal assignment.

The compiler has encountered a selected signal assignment that does not include all possible choices. Selected signal assignments must include all possible choices.

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.

262

Missing choice in case statement.

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.

Check to make sure that all possible choices are included in the case statement, or add an others choice.

263

The value of the choice is outside the range of array elements.

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.

Check to make sure that the selection expression and choices in the case statement are compatible.

264

Elements of an array aggregate must be either all positional or all named.

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.

Check to make sure that you have not inadvertently omitted the named association for one or more elements of the aggregate.

265

Too few elements in array aggregate.

The compiler has encountered an array aggregate that does not match the usage. The number of elements in the array aggregate is incorrect.

Check to make sure that source and destination array aggregates match, in terms of the number and types of their elements.

266

Too few elements in string.

The compiler has encountered a string that is not valid in the current context.

Check the format of the string, and check to ensure that it matches the intended usage.

267

Too few elements in bit string.

The compiler has encountered a bit string that does not match (in terms of size) the objects used in an expression or assignment.

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.

268

Too many elements in array aggregate.

The compiler has encountered an array aggregate that does not match the usage. The number of elements in the array aggregate is incorrect.

Check to make sure that source and destination array aggregates match, in terms of the number and types of their elements.

269

Too many elements in string.

The compiler has encountered a quoted string that illegal for the current expression or assignment.

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.

270

Too many elements in bit string.

The compiler has encountered a bit string that does not match (in terms of size) the objects used in an expression or assignment.

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.

271

Value assigned to target is outside range of values in target subtype.

The compiler has encountered in 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.

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.

272

Too many elements in record aggregate.

The compiler has encountered an invalid aggregate. The record aggregate specified has too many elements for the record type.

Check to make sure that the record type declaration is compatible with the aggregate you have specified.

273

The actual signal associated with a signal parameter must be denoted by a static signal name.

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.

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.

274

Aggregate type must be array or record.

The compiler has encountered in invalid aggregate specification. The result of the aggregate must be an array or record type.

Check to make sure that the aggregate has been properly specified.

275

Parameter of attribute 'succ equals prefix'base'high.

The compiler has encountered an invalid use of the 'succ attribute. The parameter of the 'succ attribute does not have a successor.

Check to make sure that the declaration of the base type is compatible with the use of the 'succ attribute parameter.

276

Parameter of attribute 'pred equals prefix'base'low.

The compiler has encountered an invalid use of the 'pred attribute. The parameter of the 'pred attribute does not have a predecessor.

Check to make sure that the declaration of the base type is compatible with the use of the 'pred attribute parameter.

277

Parameter of attribute 'leftof equals prefix'base'left.

The compiler has encountered an invalid use of the 'leftof attribute. The parameter of the 'leftof attribute does not have a predecessor.

Check to make sure that the declaration of the base type is compatible with the use of the 'leftof attribute parameter.

278

Parameter of attribute 'rightof equals prefix'base'right.

The compiler has encountered an invalid use of the 'rightof attribute. The parameter of the 'rightof attribute does not have a successor.

Check to make sure that the declaration of the base type is compatible with the use of the 'rightof attribute parameter.

279

Parameter of attribute 'val is too large.

The compiler has encountered an attribute value that is too large.

Check to make sure that the attribute parameter is a valid integer value.

280

Parameter of attribute 'val is too small.

The compiler has encountered an attribute value that is too small.

Check to make sure that the attribute parameter is a valid integer value.

281

Subtype range is not within the range of the base type.

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.

Check to make sure that the correct base type has been referenced, and check to make sure that the range of the subtype falls within the range of the base type.

282

Too many choices in case statement.

The compiler has encountered an invalid case statement. There are too many choices provided for the possible values of the case condition expression.

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.

283

Select expression is an array which must be of a character type.

The compiler has encountered an invalid array specification in a selection expression. The expected selection expression must be a character array.

Check to make sure that the selection expression is a valid array type.

284

Case expression is an array that must be of a character type.

The compiler has encountered an invalid array specification in a case expression. The expected case expression must be a character array.

Check to make sure that the selection expression is a valid array type.

285

Unable to determine type of array.

The compiler has encountered an array specification that cannot be resolved to a known type.

Check to make sure that the array type is clearly distinguished.

286

Attempt to index non-array.

The compiler has encountered an index operation on an object that is not an array type. Only array types may be indexed.

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.

287

Array index has an incompatible type.

The compiler has encountered an invalid array index. An array index must be either an integer, enumerated or physical type.

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.

288

Array index must be a scalar type.

The compiler has encountered an invalid array index. An array index must be either an integer, enumerated or physical type.

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.

289

A Next statement condition expression must be type boolean.

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.

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.

290

An Exit statement condition expression must be type boolean.

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.

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.

291

A while loop condition expression must be type boolean.

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.

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.

292

Unable to resolve the types of the operands of '*name*'.

The compiler has encountered an ambiguous expression in which the argument types could not be resolved.

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.

293

Too few elements in Group.

The compiler has encountered a group declaration that does not match the size of the group template declaration.

Check to make sure that the group declaration and group template declaration are compatible.

294

Too many elements in Group.

The compiler has encountered a group declaration that does not match the size of the group template declaration.

Check to make sure that the group declaration and group template declaration are compatible.

295

The prefix of a signature must be a subprogram or enumeration literal.

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.

Check to make sure that the signature prefix is a valid function or procedure name, or is an enumeration literal.

296

The signature does not match the '*description*'.

The compiler has encountered a subprogram signature that does not match the specified subprogram.

Check to make sure that the type specified in the signature matches the return value of the specified function or procedure.

297

A signature is required here because the '*description*' is overloaded.

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.

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

Signal '*name*' has multiple drivers.

The compiler has encountered a signal that is being driven in more than one process.

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.

401

No Block label matches configuration label '*name*'.

The compiler has encountered an invalid configuration. The indicated block label cannot be found.

Check to make sure that the block label has been correctly entered in the configuration.

402

No component matches Configuration for '*name*'.

The compiler has been unable to find the indicated component in the current design. The configuration statement or declaration is invalid.

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.

403

Generate range is unconstrained.

The compiler has encountered a generate range that is invalid. Generate ranges must not be unconstrained.

Check to make sure that the generate range specified is correct, and is properly constrained.

404

Component has more than one binding.

The compiler has encountered a problem in the specified binding of a component. Two or more component configurations are in conflict.

Check to make sure that duplicate component bindings are not specified.

405

Next or Exit is not inside loop with matching label.

The compiler has encountered an invalid next or exit statement. The loop label specified in the next or exit statement is not valid.

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.

406

The array index is illegal for a null array.

The compiler has encountered an array index for a null array. Null arrays do not have any members, and therefore cannot be indexed.

Check to make sure that the array has been declared as intended, and that the index is valid.

408

Design contains no entity.

The compiler was unable to find a valid entity in the input design files.

Check to make sure that you have correctly specified the input source files, and that one or more valid entities exist in the design.

420

Result of '*operator*' exceeds maximum possible value.

The compiler has encountered an operation that will produce an overflow result.

Check to make sure that the operator and operands have been correctly specified.

Also check the range of the data type being used.

421

Result of '*operator*' exceeds minimum possible value.

The compiler has encountered an operation that will produce an underflow result.

Check to make sure that the operator and operands have been correctly specified. Also check the range of the data type being used.

422

Divide by zero.

The compiler has encountered an operation that will produce an undefined result. A divisor specified in the expression is zero.

Check to make sure that the operator and operands have been correctly specified.

Check to ensure that the divisor is non-zero.

430

description '*name*' was not declared as static.

The compiler has encountered a non-static expression in a context where only a static expression is valid.

Check to make sure that the expression specified is a static expression.

431

Unconstrained range in CASE statement choice.

The compiler has encountered an invalid choice in a case statement. The range specified must be constrained.

Check to make sure that the case statement has been correctly specified, and that all choices specify constrained expressions.

432

Selected prefix is not a record.

The compiler has encountered an invalid used of a record attribute. The prefix is not a record type.

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.

434

Description '*name*' value is non-constant.

The compiler has encountered a constant declaration that is does not specify a constant value.

Check to make sure that the constant value is correctly specified, or use a signal declaration if a non-constant value is required.

435

Generic value illegal for its type.

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.

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.

436

Constant value illegal for its type.

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.

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.

437

Expected signal, variable, or constant but not a *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.

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.

438

Loop range is unconstrained.

The compiler has encountered a loop with an unconstrained range in the iteration specifier. Unconstrained loops are not supported.

Check to make sure that the iteration range has been properly specified.

440

Subprogram call actual parameter is an unconstrained array.

The compiler has encountered in 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.

Check to make sure that the function or procedure is being used properly, and check to ensure that all parameters specified when using the subprogram are valid.

441

Actual parameter associated with OUT formal parameter '*name*' is an expression.

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'.

Check to make sure that the procedure is being used properly, and check to ensure 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'.

442

Function '*name*' has no return statement.

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.

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.

450

Expected a static expression here.

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.

Check to make sure that the expression used is static.

451

Named association missing from record aggregate.

The compiler is unable to determine the correct mapping of record elements in an aggregate due to the lack of a named association.

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.

452

Named association missing from array aggregate.

The compiler is unable to determine the correct mapping of array elements in an aggregate due to the lack of a named association.

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.

453

Record aggregate has missing element(s).

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.

Check to make sure that each item in the record aggregate is provided, or use named association to specify the aggregate.

454

Description '*name*' has a type that is an unconstrained array.

The compiler has encountered an unconstrained array being used where an unconstrained array is not allowed.

Check to make sure that you have properly specified the array, and provided a constraint range if necessary.

460

Combinational Feedback using variable '*name*'.

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.

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.

470

Constraint: Unexpected use of *'Z' or NULL*, unable to infer a tristate.

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 Metamor synthesis compiler has not been specified.

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.

480

Constraint: The *name* library does not contain a *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.

Change the VHDL source code desription so it does not describe this logic element.

500

Constraint: A Wait statement may only be the first statement in a Process.

The compiler has encountered a wait statement used in an unsupported manner. Wait statements are only supported as the first statement in a process.

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.

501

Constraint: Process contains more than one Wait statement.

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.

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.

502

Constraint: Formal part may not be a function call.

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.

Check to make sure that the formal part of the subprogram or component is not a function call.

503

Constraint: Signal attribute '*attribute* is not supported.

The compiler has encountered the use of an unsupported attribute. This attribute has no meaning for synthesis and must not be used.

Check to make sure that the correct attribute is being used, or remove the attribute.

504

Constraint: WAIT statement in a procedure is not allowed.

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.

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.

505

Constraint: Expected a static expression, *description* '*name*' is not allowed here.

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.

Check to make sure that the expression is static, and does not depend on the value of a signal or port.

506

Constraint: Expected a static expression, Constant '*name*' is not static.

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.

Check to make sure that the expression is static, and does not depend on the value of a signal or port.

507

Constraint: '**' is supported only for constant operands.

The compiler has encountered an unsupported use of the '**' (exponentiation) operator. Only constant exponent values are allowed.

Check to make sure that the exponent value specified is a constant value.

508

Constraint: Assign to array element must have constant array index.

The compiler has encountered the unsupported use of a non-constant array index in an assignment to a multi-dimensional array.

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.

509

Constraint: Array slice must have constant range.

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.

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.

510

Constraint: Recursive Component instantiation.

The compiler has encountered a recursive instantiation of a component. There is no practical synthesis equivalent to recursive component or subprogram specifications.

Check to make sure that you have not inadvertently created recursion in your design by specifying the wrong component or subprogram name.

511

Constraint: Recursive Subprogram call.

The compiler has encountered a recursive reference to a subprogram. There is no practical synthesis equivalent to recursive subprogram specifications.

Check to make sure that you have not inadvertently created recursion in your design by specifying the wrong subprogram name.

512

Constraint: Literal value exceeds maximum positive value.

The compiler has encountered a numeric value that is larger than the maximum allowed.

Check to make sure that you have correctly specified the numeric literal value.

513

Constraint: Literal value exceeds minimum negative value.

The compiler has encountered a negative numeric value that is smaller than the maximum allowed.

Check to make sure that you have correctly specified the numeric literal value.

514

Constraint: Literal fractional part truncated.

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.

Check to make sure a floating point value was actually intended.

515

Constraint: Attribute 'event is not supported here.

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.

Check to make sure that you have followed the documented synthesis conventions for specifying registered logic.

516

Constraint: Attribute 'stable is not supported here.

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.

Check to make sure that you have specified the correct attribute. Use the 'event attribute to describe edge-triggered flip-flop logic.

517

Constraint: Access types are not supported.

The compiler has encountered an unsupported use of an access type. Access types are not supported in synthesis.

Rewrite your design so that access types are not required.

518

Constraint: File types are not supported.

The compiler has encountered an unsupported use of the type file. File types are not supported in synthesis.

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.

519

Constraint: File Declaration is not supported.

The compiler has encountered an unsupported use of a file type. File types are not supported in synthesis.

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.

520

Constraint: Allocator New is not supported.

The compiler has encountered an unsupported use of the memory allocation feature, new. New is not supported in synthesis.

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.

521

Constraint: Waveform truncated.

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.

Re-specify the design so a waveform is not required, or simply ignore the error message.

524

Constraint: Attribute '*attribute* parameter is non-constant.

The compiler has encountered an attribute that is only supported when applied to constant values.

Check to make sure that the target of the attribute parameter is a constant value.

525

Constraint: Signal '*name*' must be in the Process sensitivity list, or it is an input to a flip-flop that was not inferred because the flip-flop is incorrectly specified.

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. An asynchronous input is one that directly causes a change in the process output and need to be in the sensitivity list. An input that is the data input to a flip-flop is synchronous and need not be in the sensitivity list.

Check to make sure that the indicated signal was intended to be an asynchronous input to the process. If the signal is a data input to a flip-flop in this process then it is not asynchronous and the error is in the specification of the flip-flop.

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 flip-flops in the process referencing the indicated signal as an input have been properly and completely specified.

Take special care to ensure that unwanted latches have not been inadvertently specified.

526

Constraint: Flip-flop '*name*' has missing preset or reset.

The compiler has determined that the behavior of the indicated registered signal is ambiguous without a reset or preset being provided. This error occurs when the missing preset or reset would result in a flip-flop with a gated clock, when the other flip-flops inferred in the process do not have gated clocks.

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.

527

Constraint: Shared Variable Declarations not supported.

The compiler has encountered an unsupported use of shared variables. Shared variables are not support in synthesis.

Rewrite your design so that shared variables are not required.

528

Constraint: An operator symbol (*description*) is not supported here.

The compiler has encountered an unsupported use of an operator in the context of an alias.

Rewrite the design section so the operator is not required, or do not use an alias in this context.

529

Constraint: Design contains no top level Output, Buffer, or Inout ports.

The compiler has encountered a design that has no top-level output ports.

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.

530

Constraint: Hierarchy name must not contain a white space.

The compiler has encountered an unsupported hierarchical name. Hierarchical names may not include white space characters.

Check to make sure that the hierarchical name has been correctly entered.

531

Constraint: Tristate buffer '*name*' drives a logic gate, it must drive a port.

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.

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 verbose option may also help as it will report the inference of tristate buffers on a per-process basis.

600

Enum_encoding string may only contain '0' '1' 'Z' '-' 'M' or ' '.

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' or '1-hot'.

Check to make sure that the enum_encoding attribute string has been correctly specified.

601

Each encoding in Enum_encoding must have the same number of characters.

The compiler has encountered an enum_encoding attribute that does not specify the same number of characters (bits) for each enumeration value.

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.

602

Enum_encoding may only be applied to an enumerated type.

The compiler has encountered an enum_encoding attribute that references a non-enumerated type.

Check to make sure that the enum_encoding attribute is being applied to the correct type.

603

Enum_encoding must follow the enumerated type declaration.

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.

Check to make sure that the referenced enumerated type has been properly declared.

604

Too few encodings specified in Enum_encoding.

The compiler has encountered an enum_encoding attribute specification that does include the correct number of encodings.

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.

605

Too many encodings specified in Enum_encoding.

The compiler has encountered an enum_encoding attribute specification that does not include the correct number of encodings.

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.

606

Enum_encoding may not be applied to a subtype of an enumerated type.

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.

Check to make sure the attribute is being applied to an enumerated type.

607

User attribute Critical may only be applied to a Signal.

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.

Check to make sure that the critical attribute has been applied to a signal.

608

'*name*' has a type which is not locally static, a design unit with 'foreign attribute must have ports with locally static types.

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.

Check to make sure that the indicated port name represents a locally static type of object.

609

A design unit with 'foreign attribute may only have ports with mode IN or OUT.

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.

Check to make sure the external module has been referenced using only ports of mode in or out.

D - Compile options

Compile options are specified from within the OEM environment in which the Metamor compiler is found. Please refer to documentation of that tool set first. The following mechanisms exists only to bypass the OEM environment, and should not be considered by the end user for normal use. The availability of output formats referenced here depends upon the OEM version of the compiler you are using.

Compile options may also be set from a file named 'metamor.arg' in the current working directory. This allows the user to set the command line arguments directly. Any settings made by the OEM environment will be overridden by settings in the file 'metamor.arg'.

In metamor.arg, for example, you could set the library alias for the IEEE library using (your file path may vary):

```
-l IEEE
    C:\metamor\vhdl_lib\ieee.vhd
    C:\metamor\vhdl_lib\synopsys.vhd
```

In addition, the path to the directory containing the Metamor library files may be overridden by setting this path as the value of the environment variable METAMOR_LIB.

The file metamor.arg may contain any of the following options delimited by white-space or newline. Some options apply only to specific output formats. It helps to set -x to 0 when debugging metamor.arg.

All formats:

Analyze

-a

Specifies that only analysis be performed. Analysis is VHDL syntax checking, type checking, and static usage checking.

Log File

-g <file_name>

Writes a copy of the window to a file.

Elaborate

-e <entity>

-e <entity(architecture)>

-e <configuration>

Specifies the root (top) of the design to be elaborated. Default is the last configuration, or the last architecture of the last entity to be analyzed.

Device

-d name

Overrides part_name attribute.

Verbose

-v

Specifies verbose mode, which causes debug information about register and macrocell inference to be displayed.

Quiet

-q

Quiet -- turn off progress messages.

Library Alias

-l <libname> <file_list>

Specifies an alias for a VHDL library, overrides the default mapping of VHDL library name to file name, this allows multiple files to be associated with a single VHDL library. Files must be specified in the order they are analyzed.

Exit strategy

-x #

Window exit strategy 0 , 1, 2.

0 : never close the window at the end of a compile

1 : close the window if there were no compile errors

2 : always close the window at the end of a compile

Optimize level

-z #

Optimize level 0 thru 5.

A value of 0 means no optimization effort, a larger value indicates increased optimization effort.

Cupl only:

Clock Enable

-c

Enables the inference of register clock enable. Allows synthesis of a clock enable structure from certain VHDL coding conventions. Does not change the behavior of the design, but allows the compiler to take advantage of a clock enable if it exists in the target hardware.

Reset

-r

Forces all registers with preset to use reset. Transforms registers with asynchronous preset into registers with asynchronous reset. The design behavior remains unchanged. Resisters with both preset and reset are not transformed.

-p #

Sets maximum PLA product terms -- set to zero for FPGAs.

-s #

Sets maximum number of PLA inputs.

Open Abel 2 only:

Clock Enable

-c

Enables the inference of register clock enable. Allows synthesis of a clock enable structure from certain VHDL coding conventions. Does not change the behavior of the design, but allows the compiler to take advantage of a clock enable if it exists in target hardware.

Reset

-r

Force all registers with preset to use reset. Transforms registers with asynchronous preset into registers with asynchronous reset. The design behavior remains unchanged. Resisters with both preset and reset are not transformed.

-I

Force output inverters on registers.

-b

Force no output inverters on registers.

-p #

Max product terms.

-f xblox

Enable inference of XBLOX macrocells.

-f lpm

Enable inference of LPM macrocells.

XNF only (for XactStep 6.0)

-h

Hierarchy, no IBUF or OBUF insertion.

-u #

Automatic BUFG limit, set to zero for no BUFG.

-p #

Xilinx family 2k,3k,4k,4ke,5k,7k,9k.

-f xblox

Enable inference of XBLOX macrocells.

EDIF only

-h

Hierarchy, no automatic input or output buffer insertion.

-f Actel

Write a netlist for Actel Designer 3.0.

-f Altera

Write a netlist for Altera MaxPlusII version 6.01.

-f Lattice

Write a netlist for Lattice PDS+ version 3.0.

E - VHDL Information Resources

VHDL International Users Forum (VIUF) Home Page

http://www.vhdl.org/

IEEE Documents

"IEEE Standard VHDL Language Reference Manual,"
IEEE Std 1076-1993, IEEE Standards, Order Code SH 16840,
ISBN 1-55937-376-8, 1994

"IEEE Standard Multivalue Logic System
for VHDL Model Interoperability (std_logic_1164),"
IEEE Std 1164-1993, 1993

Books on VHDL in English

VHDL
Doug Perry, 390 pages, 2nd editiond ISBN0-07-049434-7
MacGraw-Hill, Inc.

The VHDL Handbook
David Coelho (Vantage Analysis Systems),
ISBN 0-7923-90310-8 Kluwer Academic Publishers, 1989

The VHDL Cookbook
Peter J. Ashenden, University of Adelaide, South Australia.
ftp://ftp.cs.adelaide.edu.au/pub/VHDL-Cookbook (Mac,PC,PS)
ftp://bears.ece.ucsb.edu pub/VHDL
ftp://du9ds4.fb9dv.uni-duisburg.de/pub/cad

Chip Level Modelling in VHDL
J. Armstrong Prentice Hall, 1988, pp. 148

An Introduction to VHDL: Hardware Description and Design
Lipsett, Schaeffer, Ussery Kluwer Academic Publishers, 1989,
320 pp, ISBN 0-7923-9030-x

Applications of VHDL to Circuit Design
edited by Randolph Harr and Alec Stanculescu Kluwer
Academic Publishers, 1991, 256 pp, ISBN 0-7923-9153-5

ASIC System Design with VHDL: A Paradigm,
S. Leung, M.A. Shanblatt, Kluwer
Academic Publishers, 1989, 240 pp, ISBN 0-7923-9032-6

Introduction to HDL-Based Design Using VHDL
Steve Carlson, Synopsys, Inc., 700 East Middlefield
Road, Mountain View, CA 94043 (415)962-5000

Hardware Design and Simulation in VAL/VHDL
Larry M. Augustin, David C. Luckham, Benoit A. Gennart,
Yo Huh and Alec G. Stanculescu Kluwer
Academic Publishers, 1991, 352 pp, ISBN 0-7923-9087-3

Performance and Fault Modeling with VHDL
edited by Joel M. Schoen Prentice Hall,
406pp ISBN 0-13-658816

A VHDL Primer, Revised Edition
J. Bhasker, Prentice Hall,
ISBN 0-13-181447-8 (based on VHDL-93)

VHDL Designer's Reference
Jean Michel Berge, Alain Fonkua, Serge Maginot, Jacques
Roulliard, Kluwer academic publishers, ISBN 0-7923-1756-4

A Guide to VHDL
Stanley Mazor, Patricia Langstraat, Kluwer academic publishers,
ISBN 0-7923-9255-8

VHDL for Simulation, Synthesis and Formal Proofs of Hardware
Jean Mermet, Kluwer academic publishers, ISBN 0-7923-9253-1

VHDL: Analysis and Modelling of Digital Systems
Zainalabedin Navabi, ISBN 0-07-046472-3, Mc Graw Hill,

VHDL Programming with Advanced Topics
Louis Baker, John Wiley & Sons, New York, 1993

Structured Logic Design With VHDL
J.R. Armstrong and F. Gail Gray,
Prentice Hall, ISBN 0-13-855206-1

Digital System Design using VHDL
Chin-Hwa Lee,
CorralTek P.O. 2616, Salinas, CA 93902 (408)484-1726

Analysis and Design of Digital Systems with VHDL
A. Dewey, Addison-Wesley, 1992

Circuit Synthesis with VHDL
R Airiau, JM Berge, V Olive, Kluwer Academic Publishers, 1994,
ISBN 0-7923-9429-1

VHDL '92;
The New Features of the VHDL Hardware Description Language
Berge, Fonkoua, Maginot and Rouillard, Kluwer
Academic Publishers ISBN:0-7923-9356-2,

The Designer's Guide to VHDL
Peter Ashenden, approx 500 pages,
Morgan Kaufman Publishers, ISBN 1-55860-270-4

A Designer's Guide to VHDL Synthesis
Ott, Kluwer Academic Publishers, ISBN 0-7923-9472-0

A Guide to VHDL Syntax
J. Bhasker, Prentice Hall, ISBN 0-13-324351-6, pp 268

VHDL Techniques, Experiments, and Caveats
J. Pick, McGraw-Hill, ISBN 0-07-049906-3

VHDL Coding Styles and Methodologies, an In-depth Tutorial
Ben Cohen, Kluwer Academic Publishers, 1995, 365 pp, Disk
included, ISBN 0-7923-9598-0

VHDL for Logic Synthesis
Andrew Rushton, McGraw-Hill, 1995, ISBN: 0-07-709092-6

Introduction to VHDL
D Hunter, T Johnson, Chapman & Hall, 246x189mm, 496 pages

VHDL Modeling for Digital Design Synthesis
Yu-Chin Hsu, 376p, Kluwer
Academic Publishers, ISBN 0-7923-9597-2

Digital Design & Synthesis with VHDL
Ross, 03/1994 Automata Publishing Company, Cloth Text,
ISBN 0-9627488-3-8 300p

VHDL Buyer's Guide
Steve Wolfe and Fouad Kiamilev, Trade Paper
ISBN 0-934869-14-6 30p,
11/1992 Cad Cam Publishing, Incorporated

Books on VHDL in French

VHDL du langage a la modelisation
R. Airiau, J.M. Berge, V. Olive and J. Rouillard, Presses
Polytechniques et Universitaires Romandes, Lausanne 1990

Books on VHDL in German

Schaltungsdesign mit VHDL
Gunther Lehmann, Bernhard Wunder, Manfred Selz, 317 Seiten,
mit Diskette, Franzis-Verlag, ISBN 3-7723-6163-3, Poing, 1994,

Abstrakte Modellierung digitaler Schaltungen
(VHDL vom funktionalen Modell bis zur Gatterebene)
K. ten Hagen, Springer, ISBN 3-540-59143-5, August 1995

Books on VHDL in Japanese

Transation of: A VHDL Primer
Jayaram Bhasker,
CQ Publishing,ISBN4-7898-3286-4 C3055 P3200E

Index - Metamor User's Guide

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z