comp.lang.vhdl

Frequently Asked Questions And Answers (Part 1): General

Preliminary Remarks

This is a monthly posting to comp.lang.vhdl containing general information. Please send additional information directly to the editor:

edwin@ds.e-technik.uni-dortmund.de (Edwin Naroska)

Corrections and suggestions are appreciated. Thanks for all corrections.

There are three other regular postings: part 2 lists books on VHDL, part 3 lists products and services (PD+commercial), part 4 contains descriptions for a number of terms and phrases used to define VHDL.

Table of Contents

FAQ comp.lang.vhdl (part 1): General

0. General Information/Introduction

0.1 The Group - Why and What

The newsgroup comp.lang.vhdl was created in January 1991. It's an international forum to discuss ALL topics related to the language VHDL which is currently defined by the IEEE Standard 1076/2002. Included are language problems, tools that only support subsets etc. but NOT other languages such as Verilog HDL. This is not strict - if there is the need to discuss information exchange from EDIF to VHDL for example, this is a topic of the group. The group is unmoderated. Please think carefully before posting - it costs a lot of money! (Take a look into your LRM for example or try to search Google Groups - if you still cannot find the answer, post your question, but make sure, that other readers will get the point).

0.2 What Is VHDL

VHDL-1076 (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language) is an IEEE Standard since 1987. It is "a formal notation intended for use in all phases of the creation of electronic systems. ... it supports the development, verification, synthesis, and testing of hardware designs, the communication of hardware design data ..." [Preface to the IEEE Standard VHDL Language Reference Manual] and especially simulation of hardware descriptions. Additionally, VHDL-models are a DoD requirement for vendors.

Today many simulation systems and other tools (synthesis, verification and others) based on VHDL are available. The VHDL users community is growing fast. Several international conferences organized by the VHDL Users Groups(s) have been held with relevant interest. Other international conferences address the topic as well.

0.3 Before Posting

- Read the 4 FAQ's - they possibly answer your questions
- Question about the language: try to find out in your LRM
- Search Google Groups. E.g., Google Groups is a good information source if you are looking for a specific VHDL model. Note, if you prepend your search string with '~g (comp.lang.vhdl)' Google Groups will search comp.lang.vhdl postings only.
- Please do **not** post homework questions to comp.lang.vhdl! Usually, this kind of queries are easily spotted (see list below) and will not receive any (useful) answers. However, if you should run into a problem during your homework you are of course welcome to post problem related questions.
  So, please do not ask for VHDL source code for
  - vending machines
  - traffic light controllers

    - ...
- If you are new to newsgroups you may also read with "How To Ask Questions The Smart Way" by Eric Steven Raymond (http://www.catb.org/~esr/faqs/smart-questions.html; please note that Eric does **not** reply to VHDL related questions) to ensure that your posting meets the newsgroup "netiquette".

0.4 Major Contributors to this FAQ

The basic version of this FAQ was created by Tom Dettmer. Georg Staebner converted Part 4 to HTML. Special thanks to Paul Menchini for contributing/revising major parts of Section 4.2 as well as Section 4.3!

0.5 Disclaimer

These articles (FAQ Part 1 to 4) are provided as is without any express or implied warranties. While every effort has been taken to ensure the accuracy of the information contained in this article, the author/editor/contributors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

1. Abbreviations

AFD:
: Accellera Designers Forum

AHDL:
: Analog Hardware Description Language

BBS:
: Bulletin Board System

DoD:
: USA Department of Defense

FAQ:
: Frequently Asked Questions

IEEE:
: The Institute of Electrical and Electronics Engineers. In case of VHDL, they defined the standard 1076

LRM:
: Language Reference Manual

TISSS:
: Tester Independent Support Software System

VASG:
: VHDL Analysis and Standardization Group

VFE:
: VHDL Forum Europe

VHDL:
: VHSIC Hardware Description Language

VHSIC:
: Very High Speed Integrated Circuits - A program of the DoD

VI:
: VHDL International

VIUF:
: VHDL International Users Forum

VUG:
: VHDL Users Group, see below

2. Contacts and Archives

2.1 Official Contacts

Accellera

Accellera's (http://www.accellera.org/) goal is to improve designers' productivity, the electronic design industry needs a methodology based on both worldwide standards and open interfaces. Accellera was formed in 2000 through the unification of Open Verilog International and VHDL International to focus on identifying new standards, development of standards and formats, and to foster the adoption of new methodologies.

Accellera's mission is to drive worldwide development and use of standards required by systems, semiconductor and design tools companies, which enhance a language-based design automation process. Its Board of Directors guides all the operations and activities of the organization and is comprised of representatives from ASIC manufacturers, systems companies and design tool vendors.

The contact address is

Accellera
15466 Los Gatos Boulevard
PMB 071, Suite 109
Los Gatos, CA 95032
Phone: (408) 358-9510
Fax: (408) 358-3910

VHDL International

VHDL International and Open Verilog International have merged into a new organization, ACCELLERA (see above).

VHDL-AMS, 1076.1 Working Group

The purpose of 1076.1 Working Group (WG) is to develop analog extensions to VHDL, i.e. to enhance VHDL such that it can support the description and simulation of circuits that exhibit continuous behavior over time and over amplitude. As of summer 1993 the IEEE Computer Society, through its Standards Activity Board (SAB), has approved the 1076.1 WG under PAR1076.1.

```
1076.1 Executive Committee
   Working Group Chair & Secretary:
           Alain Vachoux
           Swiss Federal Institute of Technology
           Integrated Systems Center
```

```
            CH-1015 Lausanne, Switzerland
            Phone: +41 21 693 6984
            Fax: +41 21 693 4663
            Email: alain.vachoux@epfl.ch
      Working Group Vice-Chair:
            Ernst Christen
            Analogy Inc.
            P.O. Box 1669
            9205 SW Gemini Drive
            Beaverton, OR 97075-1669, USA
            Phone: (503) 520-2720
            Fax: (503) 643-3361
            Email: christen@analogy.com
   1076.1 Mailing List
      Reflectors (information to all members of the mailing list):
            1076-1@epfl.ch                  European address
            ahdl1076@cadence.com            US address
      Submit new names to be put on the mailing list to
            1076-1-request@epfl.ch
      Submit to 1076.1 Executive Committee only:
            1076-1-exec@epfl.ch
   1076.1 Repositories:
      ftp://vhdl.org/pub/analog/ftp_files/
```

See Section 4.6 and http://www.vhdl.org/analog/ for further information.

2.2 VHDL International Users Forum/Accellera Designers Forum

The Accellera Designers Forum (ADF) is a professional organization of individuals active or interested in Electronic Design Automation. ADF is a duly recognized part of Accellera and its members are individual members of Accellera.

WWW: http://www.eda.org/adf/

2.3 Archives

Archives:

- The FAQ is also available by ftp on vhdl.org /pub/comp.lang.vhdl/FAQ* see VHDL International for details on accessing the server. Further, you can find this FAQ at http://vhdl.org/comp.lang.vhdl/.
- The ACM SIGDA (Special Interest Group Design Automation) offers a large amount of info and, besides other stuff, the FAQ of this group and an archive dating back to 1992. These services, and many others, are available via the ACM SIGDA Internet Server Project - http://kona.ee.pitt.edu/

Archives for this newsgroup are at
http://www.sigda.acm.org/Archives/NewsGroupArchives/ or try to search Google Groups for comp.lang.vhdl news.

3. VHDL on the Web

Here are some useful links on the web related to VHDL. If you discover an interesting server not mentioned in this list or a link is broken please send a note to the editor.

3.1 Tutorials

- An Introductory VHDL Tutorial, Green Mountain Computing Systems: http://www.gmvhdl.com/VHDL.html
- Notes on VHDL Synthesis and VHDL simulation, Electrical Engineering Department Mississippi State University: http://www.ece.msstate.edu/~reese/EE4743/ and http://www.ece.msstate.edu/~reese/EE8993/
- Introduction to VITAL ’95 by Steve Schulz: http://vhdl.org/vi/vital/wwwpages/steves/
- VHDL-Modelling And Synthesis Of The DLXS RISC Processor by Martin Gumm, University of Stuttgart: ftp://ftp.informatik.uni-stuttgart.de/pub/vhdl/vlsi_course/
- Doulos High Level Design Web site; A Hardware Engineers Guide to VHDL: http://www.doulos.com/hegv/index.htm
- VHDL Synthesis Tutorial from APS: http://www.associatedpro.com/aps/x84lab/
- Interactive VHDL Tutorial from Aldec, Inc.: http://www.aldec.com/Registration/Evita_VHDL_Download.HTM
- VHDL Tutorial by Ulrich Heinkel, Thomas Bürner and Martin Padeffke (in English and German): http://www.vhdl-online.de/~vhdl/TUTORIAL/
- VHDL-FSM-Tutorial by Martin Padeffke: http://www.vhdl-online.de/FSM/
- VHDL Verification Course by Stefan Doll: http://www.stefanVHDL.com/
- An online VHDL language guide by Altium Limited: http://www.acc-eda.com/vhdlref/index.html

3.2 VHDL Models

The following links point to some servers containing "non commercial" VHDL models. Note, there may be some limitations and restrictions concerning the use of this software.

- Free Model Foundary (FMF): http://www.eda.org/fmf/wwwpages/Welcome.html
- The Hamburg VHDL archive: http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
- RASSP www site: http://www.eda.org/rassp/
- Doulos High Level Design Web site; Monthly-updated Original Models (developed by Doulos): http://www.doulos.com/fi/
- A pipelined version of the DLX may be found at: ftp://ftp.informatik.uni-stuttgart.de/pub/vhdl/DLXS-P.beta/ ; for further information check out http://www.informatik.uni-stuttgart.de/ipvr/ise/projekte/dlx/
- EDIF LPM - Library of Parameterized Modules: http://www.edif.org/edif/lpmweb/
- ERC32 Home page at ESTEC includes ERC32 (a fully functional, timing accurate model of

a radiation-tolerant SPARC V7 processor version) and LEON-1 (a synthesizable SPARC compatible (integer) processor): http://www.estec.esa.nl/wsmwww/erc32/ or http://www.gaisler.com/

- Micron Technology, Inc. (memories): http://www.micron.com/mti/
- U.C.I. HLSynth92: http://www.ics.uci.edu/pub/hlsynth/HLSynth92/
- U.C.I. HLSynth95: http://www.ics.uci.edu/pub/hlsynth/HLSynth95/
- A model of the DLX processor from P. Ashenden and an appropriate assembler: http://www.ashenden.com.au/designers-guide/DG-DLX-material.html
- A VHDL synthesizable model for the MICROCHIP PIC 16C5X microcontroller by Tom Coonan: http://www.mindspring.com/~tcoonan/
- VHDL Library of Arithmetic Units developed by R. Zimmermann: http://www.iis.ee.ethz.ch/~zimmi/arith_lib.html
- CMOSexod: http://www.cmosexod.com/
- The OPENCORES.ORG project: http://www.opencores.org

Here are some links to commercial model sites:

- Design And Reuse (searchable database of components from various vendors): http://www.design-reuse.com/
- 4i2i Communications Ltd: http://www.4i2i.com/
- CAST, Inc.: http://www.cast-inc.com/
- Comit Systems, Inc.: http://www.comit.com/
- CorePool: http://www.corepool.com/
- Denali Software, Inc.: http://www.denalisoft.com/
- INICORE: http://www.inicore.com
- Logic Innovations : http://www.logici.com/
- Oxford Semiconductor Ltd: http://www.oxsemi.com/
- PALMCHIP: http://www.palmchip.com/
- Synopsys, Inc.: http://www.synopsys.com/
- VAutomation, Inc.: http://www.vautomation.com/
- inSilicon, Inc.: http://www.vchips.com/
- Millogic: http://www.millogic.com/index.htm
- Sierra Circuit Design, Inc.: http://www.teleport.com/~scd
- Silicore Corporation: http://www.silicore.net/
- Integrated Silicon Systems Ltd: http://www.iss-dsp.com/
- Alatek (synthesizable cores and behavioral models): http://www.alatek.com/
- Dolphin: http://www.dolphin.fr/
- Digital Core Design: http://www.dcd.com.pl/

For other commercial model vendors see FAQ part 3 products & services.

3.3 Magazines

- EE-Times: http://www.eet.com/
- EEdesign: http://www.eedesign.com/
- EDN Magazine: http://www.ednmag.com/

3.4 VHDL Sites

- EDA Industry Working Groups homepage: http://www.eda.org/
- The Hamburg VHDL archive: http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
- Doulos High Level Design Web site: http://www.doulos.com/
- RASSP WWW site: http://www.eda.org/rassp/
- Accellera: http://www.accellera.org/
- Leroy's Engineering Web Site: http://www.interfacebus.com or http://www.interfacebus.com/frames.html
- VHDL-online, University of Erlangen-Nürnberg: http://www.vhdl-online.de/
- Design Automation Cafe: http://www.dacafe.com/
- VHDL info pages of the Microelectronics Department (University of Ulm, Germnay): http://mikro.e-technik.uni-ulm.de/vhdl/vhdl_infos.html

See also FAQ part 3 products & services for other VHDL vendor sites.

4. Frequently Asked Questions

There’s not much until today - but I included those questions I’ve often heard from beginners. If someone feels, that a point should be included, please let me know.

4.1 About Changes to the VHDL Standard

According to IEEE rules every five years a standard has to be reproposed and accepted. This can include changes. Because VHDL is relatively young, the restandardization in 1992 included changes as well as the revision from 2000 and 2002 (actually, no new VHDL standard was produced in 1997).

Changes in VHDL93 include: groups, shared variables, inclusion of foreign models in a VHDL description, operators, support of pulse rejection by a modified delay model, signatures, report statement, basic and extended identifiers, more syntactic consistency.

A summary of the changes made in VHDL-2000 were presented by Paul Menchini at a recent IHDL conference on "Whats New: VHDL-2000" (coauthored by J. Bhasker). The presentation slides are available from http://users.aol.com/hdlfaq/vhdl2001-foils.pdf.

An overview of changes to VHDL planned by the VHDL-200X, IEEE 1164, and IEEE 1076.3 working groups were presented at the MAPLD 2003 conference. The slides of the presentation "VHDL-200X & The Future of VHDL" can be downloaded from http://www.synthworks.com/papers/. Futher info about VHDL-200X are available from http://www.eda.org/vhdl-200x/.

4.2 Language Related Questions

This chapter tries to answer some questions about language details which appear more or less regular on the net.

Paul Menchini contributed/revised major parts of this section. Special thanks to Paul for spending a lot of time and effort on this task!

4.2.1 USE of Library Elements?

Often users believe, they can use names of libraries /= work by simply inserting a use clause in the source. The analyzer responds with error messages. Insert a library clause before the use clause and all should work fine (see your LRM or FAQ Part 4 - B.74 for details).

4.2.2 Component Instantiation and Default Component Binding

VHDL provides three methods for instantiating a component into an architecture:

- Instantiating a component

  Prior to using a component, its interface must be defined in a component declaration (see FAQ Part 4 - B.135, FAQ Part 4 - B.184, and FAQ Part 4 - B.107). Note that no entity corresponding to the component need exist at the time of analysis of the instantiation, as the component merely defines a local, instantiable interface to the entity (see FAQ Part 4 - B.132). A good analogy is that of a socket on a board. The socket is placed and wired up, but nothing yet populates the socket. (The socket becomes populated during elaboration.)

  However, to successfully simulate a design, all sockets must be populated. One way to do so is with a configuration declaration. A configuration declaration is a separate design unit that describes the binding of entity interfaces (and architectures) to component instances (see FAQ Part 4 - B.45). Consider the following entity-architecture pair:

```
entity Test is
   port (S1, S2 : in bit;  Q : out bit);
end Test;

architecture Structure of Test is
   signal A, B, S : bit_vector(0 to 3);

   component Nor2
      port (I1, I2: in  bit;
            O:      out bit);
   end component;

begin
   Q1: Nor2 port map (I1 => S1,
                      I2 => S2,
                      O  => Q);

   Add: for i in 0 to 3 generate
           Comp: Nor2 port map (I1 => A(i),
                                I2 => B(i),
                                O  => S(i));
   end generate;
end Structure;
```

  The architecture has five component instances. In our board- level analogy, the architecture represents a board with five sockets, each of which is supposed to take a two-input NOR gate. A configuration declaration for "Test(Structure)" binding these five instances is:

```
configuration Config1 of Test is
   use WORK.MyNor2; -- make entity "MyNor2" visible

   -- configure architecture "Structure" of entity "Test"
   for Structure
      for Q1: Nor2 use -- configure instance "Q1"
         entity MyNor2(ArchX);
      end for;

      -- configure instances created by for...generate
      for Add
         -- configure all instances of component "Nor2"
         for all: Nor2 use
            entity work.MyNor2(ArchY);
         end for;
      end for;
   end for;
 end Config1;
```

The component declaration can be thought as a definition of a "socket", while the configuration determines for each instantiated socket the "IC" (entity) which is plugged into it. Note that the configuration may also include an additional layer of "wiring" between component ports (socket) and entity ports (IC). Hence, for example, the name of component ports might differ from the corresponding entity port names. In these cases the configuration must establish a connection (mapping) between component and entity ports. (Such mapping is not part of the above component declaration.)

In certain circumstances, no configuration declaration is necessary, as a set of default binding rules can be used to correctly bind entities (and architectures) to component instances. This use requires that the corresponding entity has already been successfully analyzed and is directly visible (see FAQ Part 4 - B.74) at the point of instantiation. (Note that the requirement that the entity to be bound must have already been analyzed makes this approach not possible in a top-down design methodology.)

Further, the interface of the entity and the component must match. For example, let’s say you have an entity interface BAR in library FOO. The following example will allow the default binding rules to work:

```
entity E is
end E;

library FOO;
use FOO.BAR; -- make entity BAR visible
architecture A of E is
   component BAR
         ...
```

```
   end component;
begin
   L: BAR ... ;
end A;
```

Provided that the ports of the entity BAR are identical in number to those of the component BAR, and provided that, for each port of the entity BAR, there is a corresponding port of the component BAR that matches in name, type (and subtype), and has a compatible mode, entity "E" may be compiled and simulated without providing a configuration.

(Mode compatibility is best achieved by having identical modes on the two ports; however, certain mismatches are allowed.)

Additionally, an instantiated component may be configured using a configuration specification. The specification must be placed into the declarative part of the same block (architecture, block statement or generate statement) that contains the instance. Taking a previous example, we can rewrite is to use configuration specifications as follows:

```
entity Test is
   port (S1, S2 : in bit;  Q : out bit);
end Test;

architecture Structure of Test is
   signal A, B, S : bit_vector(0 to 3);

   component Nor2
      port (I1, I2: in  bit;
            O:      out bit);
   end component;

   -- configure Q1 with:
   --      entity interface MyNor2, and
   --      architecture body Archx,
   --      both found in library WORK
   for all: Nor2 use use WORK.MyNor2(ArchX);
begin
   Q1: Nor2 port map (I1 => S1,
                      I2 => S2,
                      O  => Q);

   Add: for i in 0 to 3 generate
      -- configure all four instances of Comp the same
      -- way Q1 has been configured:
      for all: Nor2 use WORK.MyNor2(ArchX);
   begin
      Comp: Nor2 port map (I1 => A(i),
```

```
                                I2 => B(i),
                                O  => S(i));
      end generate;
   end Structure;
```

Note, the above example works only with VHDL'93 compliant tools. To write configuration specifications within a generate statement in a manner that works for both VHDL'93 and VHDL'87, rewrite the generate statement as follows (see also Section 4.2.3):

```
Add: for i in 0 to 3 generate
   B: block
      -- configure all four instances of Comp the same
      -- way Q1 has been configured:
      for all: Nor2 use WORK.MyNor2(ArchX);
   begin
      Comp: Nor2 port map (I1 => A(i),
                           I2 => B(i),
                           O  => S(i));
   end block;
end generate;
```

- Instantiating an entity (VHDL'93 only)

  A model may also directly instantiate an entity interface (and possibly one of its architectures). However, this approach requires that the corresponding entity interface (and, if instantiated, the architecture) be previously analyzed (a bottom-up approach). An example is:

```
entity E is
end E;

library FOO;
architecture A of E is
begin
   L: entity FOO.BAR(Arch) ... ;
end A;
```

- Instantiating a configuration (VHDL'93 only)

  Finally, an entire subtree of a design hierarchy may be directly instantiated. The requirements are similar to those for the direct instantiation of the entity. All entities, architectures, packages, package bodies, and configuration declarations making up the sub-hierarchy must have been previously analyzed. The following example shows how to instantiate the sample configuration given above:

```
entity E is
end E;

architecture A of E is
begin
   L: configuration Work.Config1 ... ;
end A;
```

4.2.3 GENERATE Usage and Configuration

The generate statement (FAQ Part 4 - B.105) is a concurrent statement (FAQ Part 4 - B.44) that contains other concurrent statements. Two forms exist: for and if generate. An example which uses both is:

```
First: if i=0 generate
  Q: adder port map (A(0), B(0), Cin, Sum(0), C(0));
end generate;
Second: for i in 1 to 3 generate
    Q: adder port map (A(i), B(i), C(i-1), Sum(i), C(i));
end generate;
```

The components are addressed (e.g., for specification):

First.Q, Second(1).Q, Second(2).Q, and Second(3).Q

An external configuration specification might look like:

```
for First -- First.Q
  for Q: adder use entity work.adder(architectureX);
  end for;
end for;

for Second(1) -- Second(1).Q
  for Q: adder use entity work.adder(architectureX);
  end for;
end for;
for Second(2) -- Second(2).Q
  for Q: adder use entity work.adder(architectureY);
  end for;
end for;
for Second(3) -- Second(3).Q
  for Q: adder use entity work.adder(architectureZ);
  end for;
end for;
```

Note: that form is used in an external configuration. If you need it inside the architecture you

have to insert a block (FAQ Part 4 - B.136) in the generate statement and place your configuration specification within that block (see 4.2.6).

If you have a VHDL’93 compliant tool, things are easier. The generate statement has a declarative part in which you can directly place any needed configuration specifications. An example:

```
First: if i=0 generate
  for all: adder use entity work.adder(architectureX);
begin
  Q: adder port map (A(0), B(0), Cin, Sum(0), C(0));
end generate;
Second: for i in 1 to 3 generate
  for all: adder use entity work.adder(architectureY);
begin
    Q: adder port map (A(i), B(i), C(i-1), Sum(i), C(i));
end generate;
```

4.2.4 Aggregates/Arrays Containing a Single Element

The question is often, whether

```
-- array type "one_element" contains a single element
subtype one_element IS bit_vector(1 TO 1);

type single IS record
        a : integer;
     end record;

signal o: one_element;
signal s: single;
...
s <= 1; -- first illegal try to assign a value to
        -- the record
s <= (1); -- second try, also not legal in VHDL

o <= ’1’; -- first illegal try to assign a value to
          -- the array
o <= (’1’); -- second try, also illegal
```

is valid VHDL? It isn’t. "Aggregates containing a single element association must always be specified using named association in order to distinguish them from parenthesized expressions." says the LRM. Therefore, "(1)" is simply a parenthesized expression (equal to "(((1)))").

```
s <= (a => 1); -- ok
o <= (1 => '1'); -- ok;
o <= (others => '1'); -- also ok, because the compiler
                      -- can derive from the context that
                      -- there is only a single element
```

is valid. See FAQ Part 4 - B.7 for more information on aggregates.

4.2.5 Operations With Array Aggregates

Often operations between vectors (e.g., bit_vector, std_logic_vector, unsigned, signed) are required where at least one of the arguments is a constant vector. An example is:

```
signal Sig : bit_vector(7 downto 0);
...
process
   if Sig = "00000000" then -- ok
   ...
end process;
```

The compare operation will return true if each of the bits of "Sig" is equal to '0'.

While in this example the constant vector is defined as a bit string literal (see also FAQ Part 4 - B.33), it would be more convenient to define the zero vector using array aggregates (FAQ Part 4 - B.7), as shown in the next example:

```
if Sig = (others => '0') then -- illegal!!!
...
```

The code given above fails to compile as the arguments for vector operators are usually defined as unconstrained arrays and VHDL does not require the size of the left and right operator argument to be equal. Hence, it is impossible for the compiler to determine the bounds of the constant value.

There are several solutions to this problem:

- Use attributes to constraint the aggregate as follows:

  ```
  if Sig = (Sig'range => '0') then ... -- ok
  ```

  Note that attributes may be also used to build more complex bit patterns. For example, the expressions:

  ```
  Sig = (Sig'high downto Sig'low + 1 => '0', Sig'low => '1')
  Sig = (Sig'low + 1 to Sig'high => '0', Sig'low => '1')
  ```

  are both equivalent to:

```
Sig = "10000000"
```

Note that in this case the range direction (see also FAQ Part 4 - B.16 and FAQ Part 4 - B.64) of the aggregate is not derived from "Sig" or from the directions given in the aggregate expressions. It is determined by the range direction of the corresponding operator parameter. As the operator is defined as

```
function "="(l, r : bit_vector) return boolean
```

the range direction is derived from the predefined type "bit_vector", which in turn has an ascending range (FAQ Part 4 - B.16). Consequently, "Sig'low => '1'" in the aggregate expression given above will set the first (leftmost) bit of the bit vector to '1'.

- Qualify the aggregate expression with an appropriate constrained array subtype (see also Section 4.2.18). An example is:

```
subtype byte is bit_vector(7 downto 0);
...
if Sig = byte'(others => '0') then ...
```

Similar to the first solution, more complex bit patterns can be constructed here with this approach. However, now the range direction of the aggregate is derived from the type that is used to qualify the aggregate! Hence, in the next example the aggregate is equal to "00000001":

```
subtype byte is bit_vector(7 downto 0);
...
Sig = byte'(byte'high downto byte'low + 1 => '0',
            byte'low => '1')
```

- Define a constant with an appropriate value. Note that the bounds of the constant vector must be either defined by its subtype indication or must be determinable from the initial value. However, it is recommended to constrain the constant using an appropriate type or subtype (FAQ Part 4 - B.235).

```
constant All_One : bit_vector(7 downto 0) := (others => '1');
...
if Sig = All_One then ...
```

Note that the assignment symbol (signal or variable assignment) is not a VHDL operator. Hence, it is not possible to overload it (FAQ Part 4 - B.174). Furthermore, the size of the right-hand side of an assignment operation must match the size of its target. If the right-hand side is unconstrained, this requirement allows the constraint to "cross over" from the target and determine the bounds and direction of the right-hand side. Consequently, the following signal assignment statement is legal:

```
Sig <= (others => '0'); -- ok
```

Another important issue when building aggregates is that a non locally static expression for a choice is allowed for an aggregate with a single choice only. I.e., the following code is illegal as "num" is not locally static:

```
variable num : integer;
...
Sig <= (num => '1', others => '0'); -- illegal!!!
```

The compiler cannot determine which bit of the aggregate is set to '1' as "num" may vary during runtime. To achieve the desired behavior a process may be used:

```
variable num : integer;
...
p: process (num)
begin
   Sig <= (others => '0');
   Sig(num) <= '1';
end process;
```

4.2.6 How to Attach Attributes Inside of Generate

The '87 LRM is a bit confused as to whether the generate statement forms a declarative region, so it does not make provision for a declarative part in the syntax. However, it was decided that it indeed does form a declarative region (FAQ Part 4 - B.56), so in the '93 LRM we also included the syntax extension necessary for a declarative part. For '87, however, there is no such declarative part. The canonical method is to use an embedded block statement:

```
G1:for i in DataIn'Low to DataIn'High generate
   b: block                                  -- PJM addition
        attribute foo of RECV: label is 42; -- PJM addition
      begin                                  -- PJM addition
        RECV:CHVQA; [simplified-PJM]
      end block b;                           -- PJM addition
end generate ;
```

cited from an article of Paul Mechini.

4.2.7 Notes on Range Directions

Consider the following code

```
signal r3, r4: Bit_Vector(3 downto 0);
...
r3(0 to 1) <= r4(0 to 1);
```

Note the inverse range directions (FAQ Part 4 - B.193) of the slices (FAQ Part 4 - B.222) and the arrays "r3" and "r4". The 87 LRM does not clarify, if this is legal, but the official interpretation of the 87 language (VASG) says that the directions of slices must match the direction of the array being sliced. So it is not legal code. This decision is reflected in the VHDL’93 LRM.

Another related problem is

```
signal r3, r4: Bit_Vector(3 downto 0);
...
r3(0 downto 3) <= r4(0 downto 3);
```

As these slices are null slices (FAQ Part 4 - B.167) the code is legal. An array value of no elements (i.e., having a ’LENGTH of 0, FAQ Part 4 - B.165) can be assigned to an array having no elements, as long as their types are the same.

4.2.8 Integer - Time Conversion

The following example converts integer (FAQ Part 4 - B.134) to time (FAQ Part 4 - B.182) and time to integer.

```
architecture convert of test is
   signal a, c : integer := 20;
   signal b : time := 1 ns;
begin
   process
   begin
      wait for 1 fs;
      a <= a + 1;
      b <= a * 1 fs;
      wait for 1 fs;
      c <= b / 1 fs;
   end process;
end;
```

Please note that, depending on the actual value of signal b and the internal representation of time and integer values, the divide operation may overflow. E.g., some simulators use 32 bits to represent integers and 64 bits to store time values. In this case, the division operation may produce results that cannot be represented within 32 bits. E.g., 1 ms / 1 fs equals to 1E12, which cannot be stored as a 32-bit integer. Hence, choose the divisor carefully, based on the expected time values and the required resolution, in order to prevent overflows (e.g., 1 ms / 1 ps = 1E9 does not overflow 32 bits).

4.2.9 "Don’t Cares" in VHDL

The names given to the states (be it X, Z, don’t care, even 0 and 1) of an enumeration type (FAQ Part 4 - B.85) have only the meaning brought by the subprograms of the package that defines the type. For example, 0 is 0 only because of the way "and", "or", etc., are written. Z is ’high impedance’ and X is ’conflict’ only because of the way the package, and particularly the resolution function is written. As for the "don’t care", it has no particular meaning for simulation, and the STD_LOGIC package does not provide any semantics for it: it’s a normal state.

For example in a case statement like

```
variable address : std_logic_vector(5 downto 0);
...
case address is
   when "-11---" => ...
   when "-01---" => ...
   when others => ...
end case;
```

an ’address’ value of "111000" or "101000" will match the ’others’ clause! Here is a solution to this problem:

```
if (address(4 downto 3)="11") then ...
elsif (address(4 downto 3)="01") then ...
else ...
end if;
```

Another solution is to use the "std_match" functions defined in the numeric_std package (see Section 4.8):

```
if (std_match(address, "-11---") then ...
elsif (std_match(address, "-01---") then ...
else ...
end if;
```

Partially extracted from an article by Jacques Rouillard.

4.2.10 How to Open and Close Files

The answer depends on which version of the language you’re using. In VHDL’87, files cannot be opened and closed under model control. Instead the file object must be re-elaborated (see FAQ Part 4 - B.81). The following example shows how access a file via a procedure:

```
-- VHDL’87 example!
-- Note the file is opend when get_file_data is
-- called and closed when it returns!
procedure get_file_data(file_name : in string) is
```

```
   type int_file_type is file of integer;
                         -- see FAQ Part 4 - B.99
   file file_identifier : int_file_type is in file_name;
                          -- open file
begin
   ... -- read in the file
   return; -- note, the file is closed now!
end get_file;
...

get_file_data("file1"); -- reads in the file named "file1"
```

Additionally, in VHDL’93 it is possible to open and close files under model control via file_open(...) and file_close(...).

Note that the syntax rule for file declaration in VHDL’87 and VHDL’93 are different. Moreover, the VHDL’87 syntax rule is not a subset of the corresponding VHDL’93 syntax rule.

File declartion in VHDL’87:

```
type integer_file is file of integer;

-- The following two declarations open a file for reading
file file_ident1 : integer_file is "a_file_name1";
file file_ident2 : integer_file is in "a_file_name2";

-- The next declaration opens a file for writing
file file_ident3 : integer_file is out "a_file_name3";
```

File declaration in VHDL’93:

```
type integer_file is file of integer;

-- The following two declarations open a file for reading
file file_ident1 : integer_file is "a_file_name1";
file file_ident2 : integer_file open read_mode is
                                    "a_file_name2";

-- The next declaration opens a file for writing
file file_ident3 : integer_file open write_mode is
                                     "a_file_name3";

-- The next declaration opens a file for appending
file file_ident4 : integer_file open append_mode is
                                      "a_file_name4";

-- Finally, in VHDL’93 it is possible to declare a file
```

```
-- identifier without associating it with a file name
-- directly. Use the (implicitly declared) procedures
-- file_open(...) and file_close(...) to open/close
-- the file.
file file_ident5 : integer_file;
...
file_open(file_ident5, "a_file_name5", read_mode); -- opens
                          -- "a_file_name5" for reading
file_close(file_ident5); -- closes file
```

4.2.11 How to Read/Write Binary Files

The file formats for binary files read or written by VHDL are not standardized. However, each simulator should be able to read its own generated binary files. This gives you two choices:

- Write a pre/post-processor to convert binary files into text files and vice versa. Text files can be read/written by VHDL using the TEXTIO package.
- Analyze the binary, which is generated by the simulator and then in turn generate binary according to that format. Usually, binary files written by VHDL start with a header followed by the actual data part. The header contains some release and/or type information. However, there is at least one simulator which doesn't create a header.

Partially extracted from an article posted by Wolfgang Ecker.

To read raw binary files (e.g. bitmaps) a solution was suggested by Edward Moore which should work at least with Modelsim: Open a file of type 'character', which in Modelsim translates to one byte of i/o. Use the READ procedure to read each character and the 'POS attribute to convert from character to integer. Note, this technique does not work with simulators that use only 7 bits to represent characters.

4.2.12 How to Use Package Textio for Accessing Text Files

While a simulator should always be able to read back its own generated binary files it is often not possible to share binary files between different simulators. A portable way to access data from files is provided by the package IEEE.TextIO, which implements mechanisms to read and write data values in ASCII format. In detail, it defines subprograms to perform formatted I/O operations for the data types bit, bit_vector, boolean, character, integer, real, string and time.

The package TextIO introduces two new types: "line" and "text":

```
type line is access string; -- line is a pointer to "string"
type text is file of string;
```

Further, two procedures, readline and writeline, are declared that respectively read and write an

entire line from or to a file. Finally, TextIO provides a set of overloaded procedures named "read" and "write" to respectively read or write data values of a specific data type from or to a line. In detail, separate read and write functions are defined for each of the data types bit, bit_vector, boolean, character, integer, real, string and time. The following code shows the declaration of procedures readline and writeline as well as the corresponding read and write procedures for type bit:

```
procedure readline(file F : text; L : out line);
procedure writeline(file F : text; L : inout line);

procedure read(L : inout line; value: out bit;
               good : out boolean);
procedure read(L : inout line; value: out bit);

procedure write(L : inout line; value : in bit;
                justified : in side := right;
                field : in width := 0);
```

The parameter "good" of the first read procedure returns true if the operation was successful (and false otherwise) while the second read procedure generates a run-time error in case of an error occuring while reading. (Usually, an appropriate message is printed to the screen in this event.) The optional parameters "justified" and "field" of the write procedure controls the alignment (left or right) and (minimum) number of characters which are used to print the corresponding data value (i.e., remaining characters not required to print the data value are filled with blanks). The write procedure for type real has an additional parameter named "digits", which specifies the number of digits following the decimal point.

As mentioned before the subprograms read and write do not directly access files but instead read and write data from or to a line. This approach allows the same file to be accessed simultaneously from several processes. Hence, in order to write data to a file, a line is first created and preloaded with the corresponding text. Then, it is atomically written to the file via writeline. Similarly, reading data from a file is performed as a sequence of readline and read operations. Note that a single line may contain several data values, as shown in the following example:

```
use std.TextIO.all;

entity top is
end top;
architecture arch of top is

   type data_set is record
      bvec : bit_vector(0 to 7);
      int : integer;
   end record;
```

```
   type data_set_vec is array (natural range <>) of data_set;

   procedure read_from_file(file_name : string;
                            dr : out data_set_vec) is
      file data_file : text open read_mode is file_name;
      variable L : line;
      variable i : integer := dr'low;
   begin
      while not endfile(data_file) loop
         -- note that bvec and int are read from the
         -- SAME line
         readline(data_file, L);
         read(L, dr(i).bvec);
         read(L, dr(i).int);
         deallocate(L); -- just to make sure that no memory
                        -- is lost
         i := i + 1;
      end loop;
   end read_from_file;

   procedure write_to_file(file_name : string;
                           dw : data_set_vec) is
      file data_file : text open write_mode is file_name;
      variable L : line;
   begin
      for i in dw'range loop
         -- note that bvec and int are written to the
         -- SAME line
         write(L, dw(i).bvec);
         write(L, dw(i).int);
         writeline(data_file, L);
      end loop;
   end write_to_file;

   signal data : data_set_vec(0 to 10) :=
         ( 1 => ((others => '1'), 0),
           3 => ((others => '1'), -1),
           others => ((others => '0'), 2));
begin

 p: process
    variable data_read : data_set_vec(0 to 10);
 begin
    -- write data to file
    write_to_file("testfile.txt", data);
    -- then, read it back from file
    read_from_file("testfile.txt", data_read);
```

```
        -- compare read with written data
        assert data_read /= data
           report "Read data is ok!"
           severity note;
        wait; -- end process
     end process;

  end arch;
```

Note that all procedures modify their parameter L (and hence modify the string referred to by L):

- Procedure Read automatically removes those characters from L that were used to determine the data value.
- Procedure Write appends the new data to the line referenced by L (L may be resized or reallocated by this operation).
- Procedure Readline automatically deallocates its line parameter L before allocating new memory to store the next line of the file (however, not all simulators actually do this; hence, performing an explicit deallocate operation on L before calling readline ensures that no memory is lost).
- After writing the line to the file, procedure Writeline removes all characters from L; hence, L will point to a null string after Writeline has been called.

Note that directly assigning to variables of type line may introduce memory leaks. E.g., the following procedure will leak memory each time it is called:

```
use std.TextIO.all;
...
procedure memory_leak is
   variable L : line;
begin
   L := new string(1 to 16);
   L.all := "this will create";
   -- the next assignment creates a memory leak as
   -- L is not deallocated!
   L := new string(1 to 12);
   L.all := "memory leaks";
   -- as assigning null to L does not automatically
   -- deallocates memory the following line also
   -- leaks memory
   L := null;
   L := new string(1 to 1);
   -- further, returning without deallocating L will
   -- most probably create another memory leak!
end procedure;
```

In addition, TextIO contains a function Endfile, which is a predicate that indicates whether the

next call to Readline will fail. (That is, as long as Endfile returns false, the next Readline on the same file will succeed.)

Further, two special files "output" and "input" are defined in IEEE.TextIO to support console I/O operations. In conjunction with the readline, writeline, read, and write procedures, they may be used to print text to the screen or to read data from the keyboard.

Note that the TextIO package is for simulation only! Hence, it cannot be synthesized. If you want to feed source code that includes some TextIO related statements to your synthesis tool, then use appropriate synthesis commands to switch off synthesis for the offending statements (e.g., with Synopsys synthesis tools you may use embedded *"synthesis off / synthesis on"* pseudo-comments).

4.2.13 Signal Drivers

Each concurrent statement (see FAQ Part 4 - B.44) that executes (namely, processes, concurrent signal assignment statements, and concurrent procedure calls) has a separate driver (see FAQ Part 4 - B.78) for the longest static prefix (see FAQ Part 4 - B.151) of each signal that is target of a signal assignment statement within the concurrent statement (Concurrent asserts execute, but are always passive; that is, they contain no drivers as they never assign to signals).

It does not matter whether or not a specific signal assignment statement will actually be executed. For example, the following process "p" contains two drivers, one each for signals "s1" and "s2":

```
signal s1, s2 : integer;
...
p: process (s1, s2)
begin
  s1 <= 1;
  if false then -- note, the condition always evaluates to
  false
    s2 <= 1; -- this line actually will be never executed!
  end if;
end process;
```

Hence, should there be another process driving signal "s2", the model contains multiple drivers for an unresolved signal (see FAQ Part 4 - B.204), which is an error.

Note that in the case of a process, there is a maximum of one driver per signal driven, no matter how many assignments to that signal appear within the process. For example, the following process has two drivers, one each for each of the signals Q and QBar:

```
process (Clk, Clr)
begin
  case To_X01( Clr ) is
    when '1' =>
```

```
        Q <= '0';
        QBar <= '1';

      when '0' =>
        if rising_edge( Clk ) then
          Q <= D;
          QBar <= not D;
        end if;

      when 'X' =>
        Q <= 'X';
        QBar <= 'X';
    end case;
  end process;
```

The initial value of a driver is defined by the default value associated with the signal (see FAQ Part 4 - B.58). Because in the first example no explicit default value is given for "s1" and "s2", the initial value of the drivers for both signals will be set to the left bound of the integer type (usually -2147483647).

Further, VHDL needs to be able to statically (that is, during static elaboration) determine all drivers of a signal, in order to create a static network topology. A driver is created for the longest static prefix of each target signal. During elaboration the compiler analyzes the target of each signal assignment statement to determine the smallest portion of the signal that can be statically determined as being driven by the concurrent statement. For example, the following model is erroneous, as both the process "p" and the concurrent signal assignment both drive "sig(3)", an unresolved signal.

```
architecture behave of test is
  signal sig : bit_vector(0 TO 7);
  constant c : integer := 3;
begin
  p: process (sig)
  begin
    for i in 1 to 1 loop
      sig(i) <= '1'; -- signal assignment statement
    end loop;
  end process;

  sig(c) <= '1'; -- concurrent signal assignment driving
                 -- "sig(3)"
end behave;
```

In this example, the longest static prefix of the target of the assignment statement "sig(i) <= '1'" is the entire signal "sig", since "sig" is a static signal name and "i" is a loop constant and hence not static. Consequently, "p" has a driver for the entire signal "sig", although actuality only

"sig(1)" will be driven by the process. Further, the longest static prefix of the concurrent signal assignment is "sig(3)", since "c" is a statically elaborated constant equal to 3. Hence, an error message should be generated to the effect that several processes are driving "sig(3)".

Note that the longest static prefix of a target of a signal assignment is determined at elaboration time. For example,

```
entity test is
  generic (g : integer range 0 to 7);
end test;
architecture behave of test is
  signal sig : bit_vector(0 to 7);
begin
  sig(2) <= '1'; -- concurrent signal assignment #1
                 -- driving "sig(2)"
  sig(g) <= '1'; -- concurrent signal assignment #2
                 -- driving "sig(g)"
end behave;
```

The longest static prefix of "sig(g)" is the element of "sig" determined by the generic parameter "g" (and NOT all elements of signal "sig"), since "g" is constant during elaboration. Hence, an error will only occur if "g" is set to 2 during elaboration:

```
-- this component instantiation will produce no error

comp_ok: entity test(behave) generic map(0);

-- the following instantiation is erroneous since
-- both concurrent signal assignment statements of
-- component "comp_error" are driving "sig(2)"
-- (which is not of a resolved type).
comp_error: entity test(behave) generic map(2);
```

There are situations where it is useful to drive a signal by several processes; for example, when the signal represents a tristate bus. However, VHDL does not build in any resolution mechanisms, so this situation requires that a resolution function (see FAQ Part 4 - B.202) be associated with the multiply driven signal. The following simple example shows how to use the resolved type "std_logic" (defined in the package "std_logic_1164") to drive a signal with two concurrent signal assignments:

```
library IEEE;
use IEEE.std_logic_1164.all;
architecture test_arch of test is
  -- "std_logic" is a resolved multi-value logic type
  -- defined in the package "std_logic_1164".
  signal source1, source2, target : std_logic;
```

```
  signal control : bit;
begin
  -- depending on the value of "control" either the value
  -- of "source1" or "source2" is assigned to "target".
  target <= source1 when control = '1' else 'Z'; -- driver #1
  target <= source2 when control = '0' else 'Z'; -- driver #2
end test_arch;
```

4.2.14 Procedures and Drivers

Procedures may contain signal assignment statements. In this case, the driver or drivers (see FAQ Part 4 - B.78) corresponding to these assignments are not associated with the procedure, but with the process(es) calling the procedure. As stated in Section 4.2.13, VHDL needs to be able to statically determine all drivers of a signal. Hence, unless the procedure is declared within a process (and therefore callable only by the process), the procedure must drive only signals passed as parameters to the procedure. This restriction allows the elaborator to determine the signals that are driven by a given process, so that the drivers for the process can be identified during elaboration.

For example,

```
architecture behave of test is
  signal s1, s2 : std_logic;

  -- "global" may be called by several processes.
  -- Consequently, it must drive only signals passed
  -- as parameters.
  procedure global(signal proc_sig : out std_logic) is
  begin
    proc_sig <= '0';
  end global;

begin
-- "p1" has a driver for "s1" and "s2"
p1: process (...)
  -- "local" can be called by "p1" only. Hence it can
  -- directly write to signal "s1"
  procedure local is
  begin
    s1 <= '0';
  end local;
begin
  local; -- creates a driver for "s1"
  global(s2); -- creates a driver for "s2"
```

```
end process;

-- "p2" has a driver for "s1"
p2: process (...)
begin
  global(s1); -- created a driver for "s1"
end process;
end test_arch;
```

4.2.15 Case Statement

For each case statement the compiler must verify that alternatives defined by the type of the selection expression are covered exactly once by the set of choices. (This condition is also required of selected signal assignment statements.) A necessary condition of this check is that the choices must be locally static (see FAQ Part 4 - B.147 and Section 4.2.40); i.e., they must be determinable at compilation time and thereafter fixed. The following example meets these conditions:

```
subtype my_int is integer range 0 to 2;
signal sig : my_int;
constant const_a : my_int := 0;
...
case sig is
   when const_a => ...
   when 1 => ...
   when 1+1 => ...
end case;
```

while this example fails to compile as it violates the above requirements:

```
entity test is
  generic (gen_b : bit := '1');
end test;
architecture erroneous of test is
  signal sig : bit;
  variable var_a : bit := '0';
begin
  ...
  case sig is
     when var_a => ... -- error: "var_a" is not locally
                       -- static!
     when gen_b => ... -- error: "gen_b" is not locally
```

```
                          -- static!
  end case;
  ...
end erroneous;
```

If you find that you must use non-locally static choices in a case statement, you must instead use a if-then-else chain. For example, the above architecture can be rewritten correctly as:

```
architecture correct of test is
  signal sig : bit;
  variable var_a : bit := '0';
begin
  ...
  if sig = var_a then
    ...
  elsif sig = gen_b then
    ...
  end if;
...
end correct;
```

4.2.16 How to Monitor Signals

VHDL does not grant direct access to signals at arbitrary locations in the design hierarchy. Hence, you cannot directly read or write signals at the top-level interface from the testbench.

There are three solutions to this problem:

- Add appropriate ports to the affected components to pull the signals up to the top level hierarchy. For example, consider the following model:

  ```
  entity test is
    port (...);
  end test;
  architecture behave of test is
    signal local : bit; -- a locally declared signal
  begin
    ...
  end behave;
  ```

  To monitor the signal named "local" of the component, add an out port to the interface and drive it with the value to be monitored:

```
entity test is
  port (...; monitor : out bit);
end test;
architecture behave of test is
  signal local : bit; -- a locally declared signal which
                      -- shall be monitored
begin
  ...
  monitor <= local;
end behave;
```

- Another method is to assign the values to be monitored to package-resident signals. Package-resident signals can be read and written from anywhere in the design hierarchy. For example, create a package:

```
package monitor_signals is
  signal monitor : bit;
end monitor_signals;
```

and add a corresponding statement to drive the monitor signal from your model:

```
use work.monitor_signals.all; -- make the "monitor"
                              -- signal visible
entity test is
  port (...);
end test;
architecture behave of test is
  signal local : bit; -- a locally declared signal
                      -- which shall be monitored
begin
  ...
  monitor <= local; -- copy the signal value to the
                    -- monitor signal
end behave;
```

In a similar way the monitor signal can now be read in each level of the design hierarchy, including the testbench.

- Some simulators provide access to arbitrary signals in the design hierarchy via internal simulator functions or special simulator control commands. An animation describing how to monitor signals using the Modelsim VHDL simulator from Model Technology is available from http://www.model.com/support/technote/index.html. A similar solution for NC-VHDL from Cadence is available from http://in.geocities.com/srinivasan_v2001/technical/nc_signal_spy.htm. However, these methods are simulator dependent and hence *not* portable.

4.2.17 Resolving Ambiguous Procedure/Function/Operator Calls

VHDL uses the parameter and result type profiles of functions and procedures to uniquely determine the subprogram to call. (Enumeration literals also have parameter and result type profiles.) In detail, it uses:

- the subprogram name,
- parameter types (in order), and the return type

to identify the subprogram (see also FAQ Part 4 - B.177 and faq4ref(parameter and result type profile, B.178)). For example, consider the following two packages, "A" and "B":

```
package A is
  function func(p1 : in integer; p2 : in bit := '1')
                                          return integer;
  procedure proc(p1: inout bit);
  function "="(p1 : bit; p2 : bit) return bit;
end package;

package B is
  -- note, "func" of both packages differ in their result
  -- type only
  function func(p1 : in integer; p2 : in bit) return bit;
  -- "proc" has the same parameter profile than "proc"
  -- of package "A"
  procedure proc(x1: inout bit);
  function "="(p1 : bit; p2 : bit) return bit;
end package;
```

The subprograms "func" and "proc" may be used as follows

```
use work.A.all; -- make declarations of package "A" visible
use work.B.all; -- make declarations of package "B" visible
architecture behave of test is
  signal int_signal : integer;
  signal bit_signal : bit;
begin
  -- calls "func" from package "A":
  int_signal <= func(int_signal, bit_signal);

  -- calls "func" from package "A":
  int_signal <= func(int_signal);

  -- calls "func" from package "B" because the result type of
  -- "B.func" matches the type of "bit_signal"
```

```
  bit_signal <= func(int_signal, bit_signal);

  -- calls "proc" from package "A"
  proc(p1 => bit_signal);

  -- calls "proc" from package "B"
  proc(x1 => bit_signal);
end behave;
```

There are situations where it is not possible for the compiler to uniquely select a subprogram. Expanded names can often be used to identify the correct subprograms in such cases (see also FAQ Part 4 - B.90):

```
use work.A.all; -- make declarations of package "A" visible
use work.B.all; -- make declarations of package "B" visible
architecture behave of test is
  signal int_signal : integer;
  signal bit_signal : bit;
begin
  -- use selected names to identify which subprogram to call
  work.A.proc(bit_signal); -- calls "proc" from package "A"
  work.B.proc(bit_signal); -- calls "proc" from package "B"

   -- selected names can be use to identify operators as well
   -- calls "=" from "A":
   bit_signal <= work.A."="(bit_signal, bit_signal);

   -- calls "=" from "B":
   bit_signal <= work.B."="(bit_signal, bit_signal);
end behave;
```

4.2.18 How to Resolve Type Ambiguities in Expressions

VHDL is a strongly typed language. Hence, the compiler does not perform any implicit type conversions or attempt to "guess" the type of an expression. VHDL provides two mechanisms to change or fix the type of an expression:
1. "Type conversion" may be used to change the type of an expression (see also FAQ Part 4 - B.243). In VHDL, type conversions are allowed only between types that are "closely related" (see also FAQ Part 4 - B.40). Two types are closely related if and only if one of the following conditions hold:

- A type is closely related to itself.
- Two scalar types are closely related if they are abstract numerical types. The abstract numeric types are the integer types and the floating point types.

- Two array types are closely related if and only if they have the same number of dimensions, the array element types are equal, and the corresponding index types are closely related.

Note, during conversion some information might be lost. E.g., converting a real to an integer will round. A type conversion is coded by enclosing the expression to be converted in parenthesis and prepending the target type name. The following example shows how to convert between values of the types integer and real:

```
variable int : integer;
variable float : real;
...
int := integer(float); -- converting real to integer
real := real(int); -- converting integer to real
```

2. "Qualified expressions" may be used to explicitly identify the type, and possibly the subtype, of an expression. Such qualification is a "hint" to the compiler, informing it as to the desired interpretation of the expression being qualified. Hence, such qualification is legal only if the expression can be legally interpreted as a value of the qualifying type. In contrast to type conversion, no information loss can occur.

The syntax of a qualified expression is similar to the syntax of a type conversion. The difference is that a "tick" (the ’’’ character) is inserted between the type name and the parentheses surrounding the expression, as shown in the next example:

```
-- note, both enumeration type declarations have some common
-- enumeration items
type color1 is (violet, blue, green, yellow, red);
type color2 is (black, blue, green, red);

-- this array type has an index type "color1". Without
-- using type qualification this example will not compile
-- because the compiler cannot determine the type
-- of "blue" (might be "color1" or "color2").
type a1 is array(color1’(blue) to color1’(red));

-- this array type has an index type "color2". Without
-- using type qualification this example will not compile
type a2 is array(color2’(blue) to color2’(red));
```

A common use for qualified expressions is in disambiguating the meaning of strings and aggregates (FAQ Part 4 - B.7), whose type can only be determined from context. For example, the textio package has write procedures for both strings and bit vectors, so the following example is ambiguous:

```
variable L: Std.TextIO.Line;
...
Std.TextIO.Write( L, "1001" );
```

The problem here is that the string "1001" can be interpreted either as a character string or as a bit vector. Since textio contains write procedures for both strings and bit vectors, the VHDL analyzer cannot determine which write procedure to call, so an error is generated.

The fix is simple: Use a qualified expression to disambiguate the string. Either of the following lines may be substituted for the ambiguous line above:

```
Std.TextIO.Write( L, String'("1001") );
```

or

```
Std.TextIO.Write( L, Bit_Vector'("1001") );
```

For many typical conversion problems that cannot be solved by VHDL "type conversion" or "type qualification" a set of appropriate conversion functions were defined in various packages. Hence, before writing your own routine check out the corresponding packages (e.g., a table of conversion functions defined in ieee.std_logic_1164 and ieee.numeric_std is available from http://www.ce.rit.edu/pxseec/VHDL/Conversions.htm).

4.2.19 How to Use Bit Strings as Argument to the To_StdLogicVector Function

An easy way to initialize a std_logic_vector or std_ulogic_vector in VHDL is to define the bit pattern via a bit string. However, when using the To_StdLogicVector function to convert a bit string into a std_logic_vector VHDL'87 and VHDL'93 compliant compilers behave differently. For example:

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all; -- See Section 4.11

-- This will NOT compile with a VHDL'93 compliant compiler!
constant c : std_logic_vector(31 downto 0)
        := To_StdLogicVector(x"0080_0000");
```

While the example shown above is valid in VHDL'87, the function call to To_StdLogicVector is ambiguous in VHDL'93. In VHDL'87, bit strings (such as x"0080_0000") can only be of type Bit_Vector. However, in '93, bit strings can be of any one-dimensional array type that includes the values '1' and '0'. So, in '93, there are three possible interpretations of the bit string in the above example:

- As a bit_vector.
- As a std_ulogic_vector.
- As a std_logic_vector.

Since both the following functions exist, the expression is ambiguous in ’93:

```
function To_StdLogicVector (P: Bit_Vector)
        return std_logic_vector;

function To_StdLogicVector (P: std_ulogic_vector)
        return std_logic_vector;
```

The expression is not ambiguous in ’87 since there’s only one interpretation of the bit string.

A universal way (i.e., one that works in both versions of VHDL) to create a constant is:

```
constant c: std_logic_vector(31 downto 0)
        := To_StdLogicVector(Bit_Vector’(x"0080_0000"));
```

Here, type qualification is used to select the desired interpretation of the bit string. Therefore, the call to To_StdLogicVector is unambiguous in VHDL’93.

Of course, for this particular example, it may be easier to use an aggregate and avoid this whole issue:

```
constant c: std_logic_vector(31 downto 0)
        := (22 => ’1’, others => ’0’);
```

Note, while all simulators should handle the above constructs, synthesis tools still vary widely in their ability to handle aggregates and conversions.

4.2.20 Conflicting Compare Operators

As described in Section 4.11, not all packages that are usually stored in library IEEE are really standardized. E.g., while std_logic_1164 is a standard package approved by the IEEE, package std_logic_unsigned is provided by Synopsys but not supported by IEEE. Unfortunately, Synopsys introduced a flaw into std_logic_unsigned (as well as into package std_logic_signed) that is flagged by some compilers. An example where this error may show up is shown below:

```
use ieee.std_logic_1164.all;     -- IEEE package
use ieee.std_logic_unsigned.all; -- Synopsys package
...
variable v1, v2 : std_logic_vector(3 downto 0);
...
if v1 = v2 then  -- error! Neither the implicit "="
                 -- operator (defined in std_logic_1164 nor the
                 -- explicit operator (defined in std_logic_unsigned)
                 -- are directly visible here.
   ...
```

In std_logic_1164, type STD_LOGIC_VECTOR is defined along with an appropriate compare

operator, "=", that is automatically (and implicitly) created by the type declaration of STD_LOGIC_VECTOR. This implicitly defined operator conflicts with an explicitly defined compare operator "=" introduced in std_logic_unsigned. Due to the visibility rules of VHDL, both operators become invisible after the second use clause. As a result, the subsequent compare operation, "v1 = v2", is invalid as no appropriate (directly visible) "=" operator is found at this point in the source code (see also Section 4.2.37).

While some compilers ignore this conflict and automatically (and improperly) choose the explicit operator declaration (from package std_logic_unsigned), other tools (properly) flag an error. There are several solutions to overcome this problem:

- Choose the operator by selection telling the compiler which operator from which package shall be called (see also Section 4.2.37). E.g.,

```
if ieee.std_logic_unsigned."="(v1, v2) then
   ...
```

- Often, compilers provide a switch to prioritize explicit declarations. (Note, this is an extra-language capability.)
- If possible, use ieee.numeric_std instead of the non-standard Synopsys packages. However, note that numeric_std requires STD_LOGIC_VECTORs to be converted to SIGNED or UNSIGNED before applying any arithmetic operations on them (see also Sections 4.2.41, 4.8.1 and 4.11). (A better solution is to operate internally on SIGNED or UNSIGNED vectors as appropriate and convert only as necessary at the interfaces.

Note that the reason for this conflict is that the operator function "=" from std_logic_unsigned is not located in the same package as the type STD_LOGIC_VECTOR (which resides in package std_logic_1164). To prevent these kind of problems, type definitions and their associated explicit operators should be located in the same package (same declarative region). Then, the compiler will always choose the explicit operator.

4.2.21 How to Convert Between Enumeration and Integer Values

An enumeration type declaration defines a type as an ordered set of enumeration literals (FAQ Part 4 - B.85). Each enumeration literal has a unique "position number," which is an integer value. (Values of integer and physical types also have position numbers.) The position number of the leftmost enumeration literal is 0, the next literal has the position number 1, etc. For example,

```
 -- In the following type, the enumeration literal "violet"
 -- has the position number 0, "blue" has the position
 -- number 1, "green" has 2, "yellow" has 3, and "red" has
 -- the position number 4.
type color is (violet, blue, green, yellow, red);
```

Two predefined attributes exist to convert between values and their associated position numbers (FAQ Part 4 - B.77). The predefined attribute ’pos, when given a type and the value, will provide the position number. The predefined attribute ’Val, when given a type and a position number, will return the value corresponding to the position number (FAQ Part 4 - B.24). For example, given the above type color, the following assertions never fail:

```
assert color’pos(violet) = 0;
assert color’val(0) = violet;
```

(The type must be provided since enumeration literals may be overloaded.) For example, the following assertion does not fail:

```
assert bit’pos(’0’) /= character’pos(’0’);
```

Some other examples:

```
signal hat : color := blue;
signal int : integer := 0;
...

-- this is equal to "int <= 1"
int <= color’pos(green);

-- assigns the position number of enumeration
-- value associated with the value of "hat"
int <= color’pos(hat);

-- this is equal to "hat <= violet"
hat <= color’val(0);

-- assigns the enumeration value associated
-- with position "int"
hat <= color’val(int);
```

Note that the predefined type character is an enumeration type. Hence, attributes ’pos and ’val may be used to convert character to and from their integer (ASCII) equivalent:

```
signal char : character := ’a’;
signal int : integer := 32;
...
-- converts character ’b’ to integer (ASCII)
int <= character’pos (’b’);

-- converts character stored in char to
-- integer (ASCII)
int <= character’pos (char);
```

```
-- converts integer stored in int to
-- character
char <= character'val (int);
```

4.2.22 How to Convert Between ASCII and Characters

See Section 4.2.21.

4.2.23 How to Convert Between Scalar Values and Strings

In VHDL'93 a mechanism to convert between strings and scalar types is implemented. The attribute (FAQ Part 4 - B.24) "T'image(...)" converts a scalar value into its string representation, while "T'value(...)" is used to transform a string into the corresponding value of type T, where T is the name of a scalar type or subtype. T'image(...) is often used to report the value of an scalar object on the screen during simulation, as shown in the following example:

```
type color is (violet, blue, green, yellow, red);

process (...)
  variable var : color;
  variable int : integer;
  variable str : string := "yellow";
begin
  -- "color'image(var)" may be used to output the
  -- value of variable "var" on the screen
  report "The value of var is" & color'image(var);

  -- "color'value(str)" returns the value
  -- associated with the string "str"
  var := color'value(str);

  -- "integer'image(int)" returns the textual
  -- representation of "int"
  report " while the value of int is " & integer'image(int);

  -- the following code sequence will store the
  -- value 123 into "int"
  str := "123";
  int := integer'value(str);
end process;
```

However, VHDL'87 does not provide this attribute. On such a system the appropriate "write" function from the textio package may be used to plot the value into a string (see also Section 4.2.12.).

Another option to convert enumeration values to their corresponding string representation in VHDL'87 is to create an array of strings where each array element stores the string representation of an enumeration item. For example,

```
type color is (violet, blue, green, yellow, red); -- the type
-- next, define an array of strings index by "color"
-- and create a table of "color names"
type color_table is array (color) of string(1 to 7);
constant color2str : color_table :=
   ( "violett", "blue ", "green ", "yellow ", "red ");
...
variable color_obj : color;
...
-- color2str usage
report "value of color_obj is " & color2str(enum_obj);
```

4.2.24 How to Print Integer/Floatingpoint Values to the Screen

In VHDL'93 the value of integer or floatingpoint signals/variables can be easily printed to the screen by using the predefined VHDL attribute "image" and the report statement. In VHDL'83 the textio package may be used. See Section 4.2.23 for further info.

4.2.25 How to Convert Bit/Std_Logic_Vectors to Strings

As mentioned in the previous section the predefined attribute "image" is only applicable on scalars. Hence, bit_vectors or std_logic_vectors cannot be directly converted to strings using build-in VHDL mechanisms. However, there are packages listed in Section 4.10 that provide this functionality.

4.2.26 How to Convert Between Integer and Bit/Std_Logic-Vectors

There are two IEEE-standard packages that provide functionality to convert between integers and either bit_vectors or std_logic_vectors:

- "IEEE.numeric_bit" includes functions to convert bit-based vectors
- "IEEE.numeric_std" includes functionality to convert std_logic_vectors

Both packages (see Section 4.8 on how to get these packages) define two new vector types: SIGNED and UNSIGNED. SIGNED vectors represent two's-complement integers, while UNSIGNED vectors represent unsigned-magnitude integers. Each package includes four conversion functions:

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
-- Result subtype: NATURAL.  Value cannot be negative since
-- parameter is an UNSIGNED vector.
-- Result: Converts the UNSIGNED vector to an INTEGER.

function TO_INTEGER (ARG: SIGNED) return INTEGER;
-- Result subtype: INTEGER
-- Result: Converts a SIGNED vector to an INTEGER.

function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
-- Result subtype: UNSIGNED(SIZE-1 downto 0)
-- Result: Converts a non-negative INTEGER to an UNSIGNED
-- vector with the specified size.

function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return
SIGNED;
-- Result subtype: SIGNED(SIZE-1 downto 0)
-- Result: Converts an INTEGER to a SIGNED vector of the
-- specified size.
```

The first two functions may be used to convert a UNSIGNED or SIGNED vector to an integer. For example,

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...
variable int : integer;
variable unsigned_vec : UNSIGNED(0 to 7);
variable signed_vec : SIGNED(0 to 7);
...
int := TO_INTEGER(unsigned_vec);
int := TO_INTEGER(signed_vec);
```

Note that a value of type bit_vector or std_logic_vector must be converted to SIGNED or UNSIGNED before calling TO_INTEGER. This conversion (see also Section 4.2.18) is necessary in order to determine the interpretation of the most-significant bit of the vector:

```
variable int : integer;
variable bvec : bit_vector(0 to 7);
...
int := TO_INTEGER(UNSIGNED(bvec)); -- bvec is treated as a
                                   -- unsigned magnitude
                                   -- representation of an
                                   -- integer value
```

```
int := TO_INTEGER(SIGNED(bvec)); -- bvec is treated as a
                                 -- two's-complement
                                 -- representation of an
                                 -- integer value
```

TO_UNSIGNED and TO_SIGNED may be used to convert a integer value into a corresponding signed or unsigned vector, as shown in the following example. The second parameter to each function determines the size of the resulting vector:

```
variable int : integer := 1;
variable unsigned_vec : UNSIGNED(0 to 7);
variable signed_vec : SIGNED(0 to 7);
...
unsigned_vec := TO_UNSIGNED(int, unsigned_vec'Length);
signed_vec := TO_SIGNED(int, signed_vec'Length);
```

A SIGNED or UNSIGNED value can be transformed into a std_logic_vector using explicit conversion (see also Section 4.2.18 and Section 4.2.27). The conversion from std_logic_vector (or std_ulogic_vector) to bit_vector is done using function To_bitvector (from package ieee.std_logic_1164):

```
variable bvec : bit_vector(0 to 7);
variable slvec : std_logic_vector(0 to 7);
...
slvec := std_logic_vector(TO_UNSIGNED(int, slvec'Length));
bvec := to_bitvector(slvec);
```

Note: Synopsys has produced three packages: std_logic_arith, std_logic_signed, and std_logic_unsigned that are intended to provide functionality similar to numeric_bit and numeric_std. In particular, the same two types, SIGNED and UNSIGNED are defined. Moreover, the packages include functions to convert between vectors and integers. These packages are typically even installed in the library IEEE. However, these packages are NOT standard, and different vendors have different and mutually incompatible versions. Also, there are naming clashes when some of these packages are used together. So, it is recommended that numeric_bit or numeric_std be used in preference to these non-standard packages.

See Section 4.11 for a more detailed discussion on arithmetic packages for bit_vectors and std_logic_vectors.

4.2.27 How to Convert Between bit_vector, std_logic_vector, std_ulogic_vector, signed and unsigned

In the package ieee.std_logic_1164 a set of functions to convert between bit_vectors, std_logic_vectors and std_ulogic_vectors are defined. The functions to transform either std_logic_vectors or std_ulogic_vectors to bit_vectors are:

```
function To_bitvector(s : std_logic_vector; xmap : BIT := '0')
         return bit_vector;

function To_bitvector(s : std_ulogic_vector; xmap : BIT := '0')
         return bit_vector;
```

The std_(u)logic values '0' and 'L' are translated to bit value '0' while '1' and 'H' are mapped to '1'. The translation of the remaining std_logic values ('U', 'X', 'Z', '-' and 'W') is determined by parameter xmap (whose default is '0').

The following two functions are provided to convert from std_logic_vector and std_ulogic_vector to bit_vector:

```
function To_StdLogicVector(b : bit_vector)
         return std_logic_vector;

function To_StdULogicVector(b : bit_vector)
         return std_ulogic_vector;
```

An example showing the usage of these functions is:

```
variable slv_vec  : std_logic_vector(0 to 7);
variable sulv_vec : std_ulogic_vector(0 to 7);
variable bvec     : bit_vector(0 to 7);
...
slv_vec  := To_stdlogicvector(bvec);
sulv_vec := To_stdulogicvector(bvec);
bvec     := To_bitvector(slv_vec);
bvec     := To_bitvector(sulv_vec);
```

Note that the types std_logic_vector, std_ulogic_vector, signed and unsigned are all closely related to each other (see FAQ Part 4 - B.40 and Section 4.2.18). Hence, the explicit conversion can be used to transform the types as needed--no conversion functions are, in fact required (although they are provided by the packages std_logic_1164 and numeric_std). An example showing the use of explicit conversion is:

```
variable slv_vec  : std_logic_vector(0 to 7);
variable sulv_vec : std_ulogic_vector(0 to 7);
variable uns_vec  : unsigned(0 to 7);
variable sgn_vec  : signed(0 to 7);
...
slv_vec  := std_logic_vector(sulv_vec);
sulv_vec := std_ulogic_vector(slv_vec);
slv_vec  := std_logic_vector(uns_vec);
```

```
slv_vec  := std_logic_vector(sgn_vec);
uns_vec  := unsigned(slv_vec);
sgn_vec  := signed(sulv_vec);
uns_vec  := unsigned(sgn_vec);
```

Conversion from bit_vector to either signed or unsigned takes place in two steps. First, the value must be converted to a std_logic_vector and then to the target type (see also Section 4.2.26):

```
variable bvec    : bit_vector(0 to 7);
variable uns_vec : unsigned(0 to 7);
variable sgn_vec : signed(0 to 7);
...
bvec    := to_bitvector(std_logic_vector(uns_vec));
sgn_vec := to_unsigned(to_stdlogicvector(bvec));
```

4.2.28 Reduction Operators for Bit-Vectors

There is no predefined VHDL operator to perform a reduction operation on all bits of vector (e.g., to "or" all bits of a vector). However, the reduction operators can be easily implemented:

```
signal a : bit;
signal a_vec : bit_vector(0 to 10);
...
-- this concurrent assignment performs an "or"
-- reduction on "a_vec"
a <= ’0’ when (a_vec = (a_vec’range => ’0’)) else ’1’;

-- while this calculates an "and" reduction
a <= ’1’ when (a_vec = (a_vec’range => ’1’)) else ’0’;
```

Note that these approaches may not produce the same results as a chain of or/and gates if the input vectors are of type std_(u)logic_vector and contain other values than ’0’ or ’1’ (e.g., ’X’ or ’Z’). For example, the or-reduction approach will assign ’1’ to the output signal if at least one element of the input vector is ’X’. However, if all other elements are ’0’ the result should actually be ’X’.

A more general method (which also handles ’X’ values correctly) is to loop through the bits in the manner shown for an or-reduction operator on std_logic_vectors:

```
function or_reduce( V: std_logic_vector )
                                  return std_ulogic is
  variable result: std_ulogic;
begin
  for i in V’range loop
    if i = V’left then
```

```
      result := V(i);
    else
      result := result OR V(i);
    end if;
    exit when result = '1';
  end loop;
  return result;
end or_reduce;
...
b <= or_reduce( b_vec );
```

Finally, a package including various reduce operator functions (and_reduce, or_reduce, xor_reduce, ...) can be downloaded from http://www.vhdl.org/vhdlsynth/vhdl/reduce_pack.vhd.

4.2.29 Gray Code Counter Model

The following model implements a simple Gray code counter with adjustable counter width (SIZE). For a more sophisticated model see Section 4.10.

```
entity gray_counter is
  generic (SIZE : Positive range 2 to Integer'High);
  port (clk : in bit;
        gray_code : inout bit_vector(SIZE-1 downto 0));
end gray_counter;

architecture behave of gray_counter is
begin

gray_incr: process (clk)
   variable tog: bit_vector(SIZE-1 downto 0);
begin
  if clk'event and clk = '1' then
    tog := gray_code;
    for i in 0 to SIZE-1 loop
      tog(i) := '0';
      for j in i to SIZE-1 loop
        tog(i) := tog(i) XOR gray_code(j);
      end loop;
      tog(i) := NOT tog(i);
      for j in 0 to i-1 loop
        tog(i) := tog(i) AND NOT tog(j);
      end loop;
    end loop;
    tog(SIZE-1) := '1';
    for j in 0 to SIZE-2 loop
      tog(SIZE-1) := tog(SIZE-1) AND NOT tog(j);
    end loop;
```

```
    gray_code <= gray_code XOR tog;
  end if;
end process gray_incr;

end behave;
```

Based on a posting by Rajkumar.

4.2.30 Is There a printf() Like Function in VHDL?

The easiest way to implement a similar functionality in VHDL is by using the image attribute of VHDL-93. See Section 4.2.21 for further information.

For a package providing C-style formatted printing see Section 4.10.

4.2.31 How to Code a Clock Divider

The following example will divide the clock frequency of the "ClkIn" signal by "Modulus" and output it on "ClkOut". It produces a symmetric output waveform if "Modulus" is even, otherwise it stays low for one input clock longer than it stays high (for a VHDL model with 50%-duty-cycle for odd divisor rates see http://www.e-insite.net/ednmag/archives/1997/081597/17di_01.htm; the architecture of some "unusual" clock dividers is shown in http://www.xilinx.com/xcell/xl33/xl33_30.pdf).

```
entity ClockDivider is
  generic(Modulus: in Positive range 2 to Integer'High);
  port(ClkIn: in bit;
       Reset: in bit;
       ClkOut: out bit);
end ClockDivider;

architecture Behavior of ClockDivider is
begin
  process (ClkIn, Reset)
    variable Count: Natural range 0 to Modulus-1;
  begin
    if Reset = '1' then
      Count := 0;
      ClkOut <= '0';
    elsif ClkIn = '1' and ClkIn'event then
      if Count = Modulus-1 then
        Count := 0;
      else
        Count := Count + 1;
      end if;
      if Count >= Modulus/2 then
        ClkOut <= '0';
```

```
      else
        ClkOut <= '1';
      end if;
    end if;
  end process;
end Behavior;
```

4.2.32 How to Stop Simulation

In VHDL, simulation normally stops when there are no more pending events (see FAQ Part 4 - B.88) anywhere in the system. Since, with the exception of clocks, it is often the case that there are only a finite number of events coded in a test bench, a simulation will naturally come to an end when all input events are exhausted, provided that clocks do not run indefinitely. An easy way to stop a clock is to use a clock enable signal. For example, here is a small design entity that generates a clock that will run only for a certain period of time:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity ClkGen is
  port( ClkPd: in Time range 0 ns to Time'High;
        RunTime: in TIme range 0 ns to Time'High;
        ClkOut: out std_ulogic);
end ClkGen;

architecture Behavior of ClkGen is
  signal ClkEna: std_ulogic := '1';
  signal IntClk: std_ulogic := '0';
begin
  ClkEna <= '0' after RunTime;
  IntClk <= ClkEna and not IntClk after ClkPd/2;
  ClkOut <= IntClk;
end Behavior;
```

This model generates a clock with a 50% duty cycle with period ClkPd that runs only for the interval 0 ns <= T <= RunTime.

Sometimes this method is not convenient. For example, in the event of a catastrophic error, simulation should come to an end immediately. In these circumstances, it may be possible to use the following method. However, there are some tool dependencies involved in its use.

A concurrent or sequential assert statement (see FAQ Part 4 - B.18) may be used to stop simulation under model control. For instance a concurrent assert statement to stop simulation might look like

```
assert not STOP_CONDITION
  report "Simulation stopped"
  severity failure;
```

where "STOP_CONDITION" is a Boolean expression which becomes true when the simulation should stop. Note that "STOP_CONDITION" is re-evaluated only when a signal included in the expression changes its value.

The assert statement may be also embedded into a process. However, in this case the statement and the corresponding stop condition will be evaluated only when the normal rules of process execution and sequential code flow dictate. An example process which stops simulation after a fixed simulation time limit has been reached is:

```
stop_sim: process
begin
  wait for 1 ms; -- stop simulation after 1 ms
  assert false
    report "End simulation time reached"
    severity failure;
end process;
```

A severity condition of failure is used in the example assert in order to maximize the possibility that the simulation will stop when the STOP_CONDITION is met. However, whether the simulation actually stops depends on the simulator being used, and possibly on the switches used to control the simulator. Some simulators have settings that allow one to suppress the display of assert messages whose severity is less than a certain value; other switches may exist to prevent halting on assertion failures whose severity is less than a certain value. The proper setting of these switches, if they exist in the tool used, is essential to the correct functioning of this approach.

Finally, some simulators also provide means to control execution using scripting engines (e.g., Tcl/Tk). However, solutions based on these features are simulator dependent and hence not portable.

4.2.33 Ports of Mode Buffer

Ports of mode *buffer* (see FAQ Part 4 - B.183 and FAQ Part 4 - B.157) can be both read and written. However, there is only a single source allowed to drive any net containing a buffer port, and that source must be internal to the port. While this restriction enables detection of unintended "multiple driver" errors during compilation (see also Section 4.2.13), a flaw in their definition makes them hard to use in the context of other designs. The following example is illegal in VHDL’87 and VHDL’93 because a port of mode out or inout must not be connected with a port of mode buffer:

```
entity SRLatch is
  port( S, R: in bit;
        Q, QBar: buffer bit); -- "Q" and "Qbar" are of
                              -- mode buffer!
end SRLatch;

architecture Structure of SRLatch is
  component Nor2
    port( I1, I2: in bit;
          O: out bit); -- "O" is of mode out
  end component;
begin
  Q1: Nor2 port map (I1 => S, -- ok
                     I2 => QBar, -- ok
                     O => Q); -- illegal

  Q2: Nor2 port map (I1 => R, -- ok
                     I2 => Q, -- ok
                     O => QBar); -- illegal
end Structure;
```

The component instantiation statements in this example are illegal because port "O" of "Nor2" is of mode "out" and hence cannot be associated with a buffer port. So, for the moment, the use of buffer ports may require that multiple libraries of standard cells be defined, one with out ports, the other with buffer ports. In most situations, this extra effort is not justified and therefore the use of buffer ports is discouraged.

However, the unnecessary restrictions on buffer ports were removed in VHDL 2000, so that buffer ports are more useful now (if your tool supports VHDL 2000).

4.2.34 Multi-Dimensional Arrays

There is no upper bound on the dimensionality of arrays in VHDL.

Arrays (see FAQ Part 4 - B.15) with two or more dimensions can be implemented in VHDL easily using a type declaration. For example, the type declaration:

```
type two_dim_array is array(0 to 63, 0 to 7) of bit;
```

declares a two dimensional array with element type bit. A single array element is addressed as follows:

```
signal sig : two_dim_array;
...
sig(0,0) <= sig(63,7); -- assign bit at index (63,7) to (0,0)
```

While a fully compliant VHDL tool can handle arrays with an arbitrary number of dimensions,

there are synthesis tools which accept only one- dimensional arrays. However, usually these tools are able to synthesize arrays where each element itself is an array. For example, instead of the two-dimensional array of type bit shown above, one may implement an array of bit_vectors (a one-dimensional array whose element type is itself a one-dimensional array--an "array of arrays"):

```
subtype elem is bit_vector(0 to 7);
type array_of_bitvec is array(0 to 63) of elem;
```

Or even simpler:

```
type array_of_bitvec is array(0 to 63) of bit_vector(0 to 7);
```

Note that a one-dimensional array whose element type is itself a one-dimensional array is indexed differently than the corresponding two-dimensional array. For example, a single bit of the array or arrays shown above is accessed using a sequence of two index values, each enclosed in braces:

```
signal sig : array_of_bitvec;
...
sig(0)(0) <= sig(63)(7); -- assign bit at index (63,7) to (0,0)
```

Notice that the first index corresponds to the outermost type declaration. Arrays are "row major" in VHDL, which is to say that "the last index varies fastest" between adjacent elements.

Another advantage of this array of arrays technique is that an entire row can now be accessed in a whole. For example, the following code assigns row number 63 to row number 0 using a single assignment statement:

```
signal sig : array_of_bitvec;
...
sig(0) <= sig(63); -- assign bit 0 to 7 of row 63 to row 0
```

4.2.35 Multi-Dimensional Array Literals

In the previous section, we show that multi-dimensional arrays are different from arrays of arrays; in particular, we noted that elements are accessed using a different syntax.

However, there is one area where there is no difference between multi- dimensional arrays and arrays of arrays: when constructing literal (see FAQ Part 4 - B.144) values, there is no syntactical difference.

For example, consider the following set of type declarations, which might be part of a simple, multi-valued logic type system:

```
type MVL is ('X', '0', '1', 'Z');

type resolveTableType1 is array (MVL, MVL) of MVL;

type MVL_Vector is array(MVL) of MVL;
type resolveTableType2 is array (MVL) of MVL_Vector;
```

Note that resolveTableType1 is a two-dimensional array of MVLs, while resolveTableType2 is a one-dimensional array of a one-dimensional array of MVLs.

However, objects of either type are initialized in an identical manner:

```
constant resTable1: resolveTableType1 :=
        ( -- 'X' '0' '1' 'Z'
            ('X','X','X','X'), -- 'X'
            ('X','0','X','0'), -- '0'
            ('X','X','1','1'), -- '1'
            ('X','0','1','Z') -- 'Z'
        );

constant resTable2: resolveTableType2 :=
        ( -- 'X' '0' '1' 'Z'
            ('X','X','X','X'), -- 'X'
            ('X','0','X','0'), -- '0'
            ('X','X','1','1'), -- '1'
            ('X','0','1','Z') -- 'Z'
        );
```

The first constant is of the two-dimensional array type, while the second constant is of the one-dimensional array type whose element is of another one-dimensional array. However, both are initialized as if they are of the latter type!

Notice that the literal initializing either array is a four-element aggregate (see FAQ Part 4 - B.7). Each element of the outer aggregate is itself a four- element sub-aggregate (see FAQ Part 4 - B.232). The elements of the sub-aggregates in all cases is a single MVL value.

Finally, notice that, once again, these arrays are row major. That is, to get to the element that is in the second row and the third column, use the following references:

```
resTable1('0','1')
```

in the case of the first array type, or

```
resTable2('0')('1')
```

in the case of the second array type.

4.2.36 Conditional Compilation

The generate statement combined with generic parameters or constants may be used to achieve a behavior similar to "conditional compilation" in C. An example is:

```
entity test is
  generic (switch : boolean);
end if;

architecture struct of test is
begin
   -- instantiate "adder1" if "switch" is true
   First: if switch generate
       Q: adder1 port map (...);
   end generate;
   -- instantiate "adder2" if "switch" is false
   Second: if not switch generate
       Q: adder2 port map (...);
   end generate;
end behave;
```

Component "adder1" will be included into the design only if the generic parameter "switch" is true. Otherwise, "adder2" is instantiated. Note that the generate statement is "executed" at elaboration time (see FAQ Part 4 - B.81), the entire architecture has to be analyzed by the compiler regardless of the generate parameter value (note that the values of generic parameters are unknown at compile time). Thus, syntactically or semantically incorrect code inside of generate statements will be flagged, regardless of whether the statements are ever elaborated. (As a consequence, both instantiation statements in the example above must be legal.) Further, the generate statement cannot be used to configure the interface of an entity; i.e., to add or remove ports depending on the value of a generic parameter. However, the sizes of array ports can be controlled through the use of generics.

In order to achieve the same functionality as the "#if" and "#ifdef" constructs of C/C++, the macro processor of an ordinary C compiler may be used. Usually, C compiler have a special command line switch to stop compilation after the preprocessing stage (e.g., "-E" for the GNU C/C++ compiler). Hence, VHDL source code can be augmented with C macros or "#ifdef...#endif" statements and then preprocessed with a C compiler before it is compiled with a VHDL compiler. A "make" tool may help to automate this two-stage approach.

Another solution is to use a special macro processor, like "m4", which is available for most Unix platforms as well as for Windows (e.g., from http://www.cs.colorado.edu/~main/mingw32/; see also FAQ Part 3, Section 1.5).

4.2.37 Remarks on Visibility of Declarations

VHDL controls visibility of declarations using the notion of the scope of a declaration. The name of a declaration is visible only within the scope of the declaration (see FAQ Part 4 - B.212). (Note that a given declaration may be hidden (see FAQ Part 4 - B.116) by another declaration in certain circumstances.) The scope of a declaration extends from the beginning of the declaration to the end of the innermost declarative region (see FAQ Part 4 - B.56) containing the declaration. (E.g., a constant declared immediately within a given architecture is visible from the end of the constant declaration to the end of the architecture and also within any configuration declaration configuring that architecture.) An example is:

```
process (sig)
   constant A : integer := 100;
   -- "A" becomes visible here and means this constant.
   ...

   procedure Test is
   -- "Test" becomes visible here and means this procedure.

      constant B : bit := '1';
      -- "B" becomes visible here and means this constant.
   begin
      ...
   end Test; -- "B" is no longer visible

begin
   ...
end process; -- "Test" and "A" are no longer visible
```

A declaration may be hidden from direct visibility (see FAQ Part 4 - B.74) by another declaration, but it is always visible by selection, as shown in the following example:

```
P: process (Sig)
   constant A : integer := 100;
   -- "A" becomes visible here and means this integer
   constant.

   procedure Test is
      constant A : bit := '1';
      -- "A" becomes visible here and means this bit constant.
      -- Hence, "A" no longer means the integer constant.
      -- However, the integer constant may be referred to as
      "P.A"
      variable Var : bit := A; -- "A" is of type bit
   begin
      if P.A > 50 then
```

```
        Var := not A;
      end if;
   end Test; -- The bit "A" is no longer visible.  Thus, "A"
             -- once again means the integer constant.

   variable Var : integer := A; -- "A" is of type integer
begin
   Var := A + 1; -- "A" is of type integer
end process; -- "A" is no longer visible
```

In the above example the integer constant A was "visible by selection" as "P.A" within the procedure Test, even though it is not directly visible within this procedure. Note that the declarations within packages are always visible by selection, as are the contents of libraries.

Use clause work by providing direct visibility of declarations that are visible by selection. Such declarations must be within libraries or packages. Note that there are cases where use clauses can cause conflicts in visibility, either between two declarations that are visible by selection, or between one directly visible declaration and another declaration that is visible by selection. For example:

- A declaration made potentially visible by an use clause cannot hide a declaration within the immediate scope of the second declaration:

```
package Pack is
   constant A : integer := 100; -- A is of type integer
end package Pack;

entity Test is
end Test;

architecture Struct of Test is
   constant A : bit := '1';
   use WORK.Pack.all;

   -- "WORK.Pack.A" is not directly visible here,
   -- although it is still visible by selection as
   -- "WORK.Pack.A"
   signal Sig : bit := A; -- hence, this declaration is ok
begin
end Struct;
```

- If at a given point two or more conflicting declarations (with the same name) has been made potentially directly visible by different use clauses then none of the declarations are directly visible. An exception to this rule are enumeration literals and subprograms. Note that at a given point in the code the order of the corresponding use clauses does not have an affect on which declaration is directly visible.

Because of this mechanism the following code will not compile:

```
package Pack1 is
   constant A : integer := 100; -- "A" is of type integer
end package Pack1;

package Pack2 is
   constant A : bit := '1'; -- "A" is of type bit
end package Pack2;

entity Test is
end Test;

architecture Arch of Test is
   use WORK.Pack1.all;
   use WORK.Pack2.all;

   -- neither "WORK.Pack1.A" nor "WORK.Pack2.A" is
   -- directly visible

   signal Sig1 : integer := A; -- ILLEGAL! "WORK.Pack1.A" is
                               -- not directly visible!
   signal Sig2 : bit := A; -- ILLEGAL! "WORK.Pack2.A" is
                           -- not directly visible either!
begin
end Arch;
```

A simple solution to bypass visibility conflicts is to use selected names (see FAQ Part 4 - B.213) to address a declaration within a specific package or declarative region (see FAQ Part 4 - B.56). This approach works because the names continue to be visible by selection. The following code demonstrates how to apply this technique on the example shown above:

```
architecture Arch of Test is
   signal Sig1 : integer := WORK.Pack1.A; -- ok
   signal Sig2 : bit := WORK.Pack2.A; -- ok
begin
end Arch;
```

4.2.38 Difference between std_logic and std_ulogic

The type std_ulogic is an enumeration type defined in the package IEEE.std_logic_1164. The type std_logic is a subtype of std_ulogic. Both are intended to model scalar logical values in ASICs and FPGAs.

As the ’u’ in meant to convey, std_ulogic is an unresolved type. That is, it may be driven by a maximum of one source. Conversely, std_logic is a resolved (sub)type (see FAQ Part 4 - B.204), which means it may be driven by any number of sources. Resolved (sub)types have resolution functions associated with them--the resolution function (see FAQ Part 4 - B.202)) associated with the subtype std_logic, called resolved, is also defined in the package IEEE.std_logic_1164.

The fact that std_logic is a subtype of std_ulogic means that operations and assignments on values and objects of type std_logic and std_ulogic may be freely intermixed and no type conversion (see FAQ Part 4 - B.243) is necessary. An example showing this usage of both types is:

```
library IEEE;
use IEEE.std_logic_1164.all;
architecture Arch of Test is
   signal unres_sig : std_ulogic := ’0’; -- unresolved signal
   signal res_sig : std_logic := ’0’; -- resolved signal
begin
   -- this concurrent signal assignment is a source for
   unres_sig
   unres_sig <= not res_sig;

   -- this process is a source for res_sig
   p: process (res_sig)
   begin
      res_sig <= not unres_sig;
   end process;

   -- this concurrent signal assignment
   -- is a second source for res_sig
   res_sig <= ’1’ when control = ’1’ else ’Z’;
end Arch;
```

Since "res_sig" has multiple sources (see FAQ Part 4 - B.223), it must be of a resolved type. However, since "unres_sig" has but a single source, it may be of either a resolved or unresolved type.

In addition to the scalar logical types std_logic and std_ulogic, there are vector types defined in IEEE.std_logic_1164. One, a resolved type, is called std_logic_vector. It is defined as a one-dimensional, unconstrained array of std_logic elements. The other, an unresolved type, is called std_ulogic_vector and is defined as a one-dimensional, unconstrained array of std_ulogic elements. Unlike the case of the scalar types std_logic and std_ulogic (where std_logic is a subtype of std_ulogic) the types std_logic_vector and std_ulogic_vector are distinct types. (This difference comes about because of the details of VHDL’s type system.) Consequently, care must be taken when intermixing std_logic_vector and std_ulogic_vector values and objects in expressions and assignments.

Ideally, all signals in a model should be declared using the unresolved types std_ulogic or std_ulogic_vector (depending on whether a scalar or vector signal is needed), except those signals that are to be multiply driven; for example, tristate busses. In that case, the types std_logic and std_logic_vector should be used as appropriate.

If this policy is adhered to, then signals that are multiply driven in error will be caught by either the analyzer or elaborator, depending on the exact circumstances. For example, consider the following model:

```
library IEEE;
use IEEE.std_logic_1164.all;
architecture Arch of Test is
   component comp
      port (a : in std_ulogic);
   end component;

   signal sig : std_ulogic := '0'; -- unresolved signal
begin
   -- a source for signal sig
   sig <= not sig;

   c: comp port map( a => sig );
end Arch;
```

The component comp contains a port of mode in. Hence, the instance c of this component does not create a source for the signal sig.

Should the mode of the port a be inadvertently changed to out, inout or buffer, the instance c then becomes an additional source for the signal sig. Since we took care to define "sig" to be of an unresolved type, this will be flagged prior to simulation start.

However, if we had instead declared "sig" to be of type std_logic, it would now be legal to multiply drive "sig", and no error would be reported by the tools. Instead, careful examination of waveforms and inspection of the model would be required to locate the problem.

(Sections 4.2.13 and 4.2.14 contain additional information on signal sources and drivers.)

Unfortunately, at least until recently, not all tools provided good support for std_ulogic and std_ulogic_vector. As a consequence, before deciding whether to follow our recommendation, you must take a look at your tools and determine whether they adequately support std_ulogic and std_ulogic_vector.

Typically, simulators are not the problem, it's the synthesis tools that might lack full support for the unresolved types. Consider a RTL model that uses std_ulogic_vectors as interface (port) signals and an appropriate testbench to stimulate the design. Often, the gate level model that is

generated by the synthesis tool from this RTL description will use std_logic_vectors as port types. As a result, the gate level model cannot directly be connected with the testbench without using type conversion.

4.2.39 VHDL and Synthesis

Synthesis is the automatic process to generate hardware (gates, flipflops, latches and the like) from a formal description (e.g., VHDL). A synthesis tool translates the HDL source code into a net of hardware primitives. The primitives are defined by the target technology (e.g., FPGA or ASIC) and vendor. Ideally, using VHDL for synthesis decouples the design process from the actual target technology, vendor and synthesis tool chain. In this manner, a design expressed in VHDL can, in theory, be easily retargeted to another technology vendor or synthesis tool.

In reality, different synthesis tools will usually generate identical hardware only for simple VHDL models. Tightening the synthesis constraints (e.g., on required operating speed, power consumption or device count) will often require tool-specific modifications to the VHDL source in order to meet the constraints. Because of this fact, to obtain similar results from different synthesis tools, one must typically modify the VHDL source when moving from one tool to another.

While a significant part of the VHDL language can be synthesized, there are constructs that cannot be handled by today's synthesis tools. Some of the restrictions are simply due to a the lack of corresponding counterparts in hardware (e.g., report statements and files). Other restrictions are due to the fact that all statements of a synthesisable VHDL description must ultimately be statically mapped to a set of corresponding hardware primitives. Currently, each synthesis tool supports a different subset of the VHDL standard. This situation should be temporary, as there is an effort underway to develop a layered set of synthesis interoperability standards (see http://www.eda.org/siwg/ for further information).

The following list discusses a number of issues related to synthesis that show up frequently in comp.lang.vhdl:

- **After clauses.**

  After clauses in signal assignment statements are not synthesised as there is no hardware primitive that has a fixed but settable delay. (Usually the delay varies significantly with temperature, supply voltage, output loading and the like). In the best case they are ignored by the synthesis tool. Hence, the statement

  ```
  sig <= not sig after 10 ns;
  ```

  will not result in an inverter with a 10 ns delay. Most likely, an inverter will be synthesized, but no attempt is made to adhere to the delay specification.

- **Processes sensitive to both edges of a clock.**

  Processes which perform operations on both edges of a clock are usually not synthesisable as there are no flipflops that are sensitive to both edges of a single signal (at least these type of flipflops are currently rarely available). For example, the following process is not synthesisable:

```
p: process (clk)
begin
  counter <= counter + 1;
end process;
```

  The process runs on all edges of clk, so the hardware representing this process would be required to increment "counter" on both clock edges. Such devices do not exist.

  However, the following process is synthesisable as it increments "counter" on rising clock edges only (note that the simulator will actually execute the process on both edges but 'useful' operations are done on rising edges only):

```
p: process (clk)
begin
  if clk'event and clk = '1' then -- select only rising edges
     counter <= counter + 1;
  end if;
end process;
```

- **Non-static loop bounds and slice ranges.**

  Loop bounds and the bounds of array slices (see FAQ Part 4 - B.222) must be static (see FAQ Part 4 - B.226 and Section 4.2.40); i.e. they must be determinable and fixed at elaboration time. Synthesis tools unroll loops and generate hardware from the linearized code, substituting a fixed value for the loop index in each iteration. Such unrolling requires that the synthesis tool know precisely how many times the loop is to be executed and what the value of the loop index is in each iteration. Hence, the loop:

```
for i in 0 to 7 loop
  ...
end loop;
```

  is synthesisable, but the following loop is not:

```
variable var : integer;
  ...
for i in 0 to var loop
  ...
end loop;
```

- **Mixing sequential and combinational logic into one process/FSM coding style.**

  Whether it is better to separate code into processes with synchronous elements in one and processes with combinational logic in the other or combining both parts into a single process is actually a matter of style and taste. An example for a Moore state machine (without the output logic part) designed with two processes is:

```
state_reg:
  process (clk, reset)
  begin
    if reset = '1' then
      current_state <= state1; -- reset action
    elsif rising_edge(clk) then
      current_state <= next_state;
    end if;
  end process;

next_state:
  process (current_state, ctrl)
  begin
    case current_state is
      when state1 =>
        if ctrl = '1' then
          next_state <= state1;
        else
          next_state <= state2;
        end if;
      when state2 =>
        next_state <= state3;
      when state3 =>
        next_state <= state1;
    end case;
  end process;
```

  The same state machine, coded with a single process, is:

```
combined:
  process (clk, reset)
  begin
    if reset = '1' then
      current_state <= state1; -- reset action
    elsif rising_edge(clk) then
      case current_state is
        when state1 =>
          if ctrl = '1' then
            current_state <= state1;
```

```
          else
            current_state <= state2;
          end if;
        when state2 =>
          current_state <= state3;
        when state3 =>
          current_state <= state1;
      end case;
    end if;
  end process;
```

Note that the output logic part of the FSM is not listed here.

The advantages of the two processes style are:

- Some synthesis tools perform better with the two processes approach.
- Many feel that the two process style is more readable. However, this is also a matter of taste and may be dependent on the design details.
- The two process approach is recommended in most VHDL textbooks as well as by many synthesis tool vendors.

The advantages of the single process style are:

- A single process generally simulates faster than two processes. Again, this is tool dependent and may vary with the actual design.
- A single synchronous process is less error prone to missing signals in the sensitivity list as only the clock signal along with any (re)set signal(s) are in the sensitivity list. Note that in order to infer combinational logic from a process, all signals read inside the process must appear in the sensitivity list.

4.2.40 Locally and Globally Static

In VHDL a *locally static* expression can be evaluated at compile time (see also FAQ Part 4 - B.147). Similarly, the value of a *globally static* expression can be determined at elaboration time (see also FAQ Part 4 - B.108 and FAQ Part 4 - B.81); i.e. when the design hierarchy in which it appears is elaborated. In detail, there is a set of rules that determine whether an expression is locally static, globally static or not static. A summary of the most important rules is given in the following (for the complete detailed description see the LRM).

An expression is said to be locally static if it is

- a literal (see FAQ Part 4 - B.144) of any type other than type time (i.e. 10, ’0’, and true are all locally static, while 10 ns is not)
- a constant that is initialized by a locally static expression (i.e., deferred constants are not locally static)
- a call to an implicitly predefined operator which returns a scalar value and whose parameters

are scalar and locally static

An expression is said to be globally static if it is

- a locally static expression (i.e., locally static expressions are also globally static)
- a generic constant (see FAQ Part 4 - B.106) or a generate parameter (see FAQ Part 4 - B.105)
- an array aggregate (see FAQ Part 4 - B.7) whose element associations are globally static (i.e., the expressions and the ranges of its element associations must be globally static)
- a record aggregate (see FAQ Part 4 - B.7) whose element associations are globally static
- a slice (see FAQ Part 4 - B.222) of a globally static array, provided that the range of the slice is globally static
- a call to a pure function (see FAQ Part 4 - B.191) whose parameters are all globally static

In addition, a globally static expression, subtype, type, or range must appear in a statically elaborated context.

For example, consider the following model:

```
entity Test is
  port (port_bit : in bit);
  generic (gen_bit : bit);
end Test;

architecture Structure of Test is
  constant a : bit := '1'; -- locally static
  constant b : bit := '1' and '0'; -- locally static
  constant c : bit := gen_bit; -- globally static

  constant d : bit_vector(0 to 3) := "0000"; -- locally static
  constant e : bit_vector(0 to 1) := d(0 to 1); -- globally s.
  constant f : bit := vec(0); -- globally static

  pure function test1 (constant p : bit) return bit is
  begin
    return not p;
  end test;
  constant g : bit := test1('1'); -- globally static

  impure function test return bit is
  begin
    return not a;
  end test;
  constant h : bit := test2; -- NOT static
```

```
begin
  ...
end Structure;
```

Note that constants "e" and "f" are globally but not NOT locally static because slices of an array or references to array elements are not locally static (even if the array is locally static). Constant "g" is globally (and NOT locally) static because the initial value contains a call to a user defined (pure) function.

The choices of a case statement are required to be locally static (see also Section 4.2.15). This enables compile time checking to ensure that all alternatives are actually covered by the choices. Hence, the following will NOT compile:

```
entity test is
  generic (gen : bit_vector(0 to 1) := "00");
end test;

architecture erroneous of test is
  signal sig : bit_vector(0 to 1);
  constant a : bit_vector(0 to 3) := "0100";
  constant b : bit_vector(0 to 1) := a(0 to 1);-- globally s.
  constant c : bit_vector(0 to 1) := '1' & '0';-- globally s.
  constant d : bit_vector(0 to 1) := a(1) & a(1);-- globally s.
begin
  ...
  case sig is
    when gen => ... -- error: "gen" is not locally static!
    when b => ... -- error: "b" is not locally static!
    when c => ... -- error: "c" is not locally static!
    when d => ... -- error: "d" is not locally static!
  end case;
  ...
end erroneous;
```

VHDL does not require the compiler to evaluate slices (or references to array elements) at compile time. Hence, "b" and "d" cannot be used as choices. This raises some difficulties when constant arrays or records are used to increase readability of the source code. An example is:

```
type rec is record is
  value : integer;
  flag : boolean;
end record;

constant c_rec : rec := (value => 32, flag => true);
constant c_value : integer := rec.value; -- globally s.
constant c_flag : boolean := rec.flag; -- globally s.
```

Due to the LRM rules, constants "c_value" and "c_flag" are only globally static and hence cannot be used as choices within a case statement. Fortunately, "reversing" the constant definitions resolves the problem:

```
type rec is record is
  value : integer;
  flag : boolean;
end record;

constant c_value : integer := 32; -- locally static!
constant c_flag : boolean := true; -- locally static!
constant c_rec : rec := (value => c_value, flag => c_flag);
```

4.2.41 Arithmetic Operations on Bit-Vectors

VHDL has no predefined arithmetic operators for bit_vectors or std_logic_vectors predefined. This situation exists as VHDL does not assume the interpretation to be applied to such vectors.

Instead, the following data types can be used in synthesizable designs to represent numbers:

- **INTEGER**: Type INTEGER (see FAQ Part 4 - B.134) can be used for objects that do not overflow. However, they must be constrained properly to prevent the synthesis tool from inferring 32- bit wide buses or registers (unless, of course, a full 32 bits are needed).

  ```
  -- synthesis tool will use 3 bits to represent
  -- signals "int1" and "int2"
  signal int1, int2 : integer range 0 to 7;
  ...
  int1 <= int2 + int2;
  int2 <= int1 + 1;
  ```

  As VHDL does not provide any mechanisms to directly set or extract single bits of a scalar object manipulation of integers at bit level is tedious. (Again, this apparent lack stems from the fact that VHDL does not assume any particular representation of integers.)

- **SIGNED/UNSIGNED**: The types SIGNED and UNSIGNED are vector types based on either of the scalar types bit or std_logic (type std_logic is defined in the package "IEEE.std_logic_1164") and hence support bit-level manipulation.

  The types based on bits are found in the package IEEE.numeric_bit, and the types based on std_logic are found in IEEE.numeric_std. Of the two, the numeric_std types are the more commonly used:

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

  Each package defines these types along with a set of logical and mathematical operators. SIGNED vectors represent two's- complement integers, while UNSIGNED vectors represent unsigned- magnitude integers. In each case, the MSB is to the right.

  The operators defined in these packages allow mathematical operations on SIGNED and UNSIGNED vectors and on mixed type operations with either SIGNED or UNSGINED on the one hand and INTEGERs on the other. However, note that UNSIGNED and SIGNED vectors cannot be directly mixed in the same expression. Fortunately, as both types (as well as std_logic_vectors) are closely related to each other (see FAQ Part 4 - B.40), predefined conversion functions (see FAQ Part 4 - B.50 and Section 4.2.18) can be applied to change the type of a vector as required. Note that the bit pattern of a vector is not changed during conversion. I.e., an UNSIGNED number will become negative after conversion to SIGNED if the most significant bit is set.

  An example illustrating the above is:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
...
signal udata1, udata2, udata3 : UNSIGNED(0 to 3);
signal sdata1, sdata2, sdata3 : SIGNED(3 downto 0);
signal slv : std_logic_vector(3 downto 0);
...
udata3 <= udata1 + udata2; -- arithmetic operation "+"
sdata1 <= sdata2 and sdata3; -- logical operation "and"

sdata1(2) <= sdata2(1); -- set/read single bit

sdata2 <= SIGNED(udata1); -- "udata1" must be converted to
SIGNED
sdata3 <= SIGNED(slv); -- "slv" must be converted to SIGNED
slv <= std_logic_vector(sdata2); -- "sdata1" must be converted
                                 -- to std_logic_vector
```

Methods to convert between integer on the one hand and bit_vector, std_(u)logic_vector, SIGNED, or UNSIGNED on the other are described in Section 4.2.26.

There are also other numeric packages provided by different vendors. Some even provide arithmetic operations for std_logic_vectors. However, it is recommended that you use "numeric_std" whenever possible (see Section 4.11 for further information).

Finally, VHDL also provides floating point numbers (see FAQ Part 4 - B.100). However, as they are currently not synthesizable (this is going to change; see the Floating-Point HDL Packages Home Page at http://www.eda.org/fphdl/) their usage is restricted to testbenches or similar model types.

4.2.42 VHDL'93 Generates Different Concatenation Results from VHDL'87

Certain VHDL tools report that the VHDL'93 concatenation operator generates different results from the same operator in VHDL'87. This section describes the differences. The operator "&" may be used to concatenate one-dimensional arrays (or their elements) to form a new array consisting of the elements of the left operand followed by the elements of the right operand. An example is

```
constant c1 : bit_vector(0 to 3) := "1101";
constant c2 : bit_vector(0 to 3) := "0010";

-- value of "c3" is "11010010"
constant c3 : bit_vector(0 to 7) := c1 & c2;

-- value of "c4" is "11101"
constant c4 : bit_vector(0 to 4) := '1' & c1;

-- value of "c5" is "01"
constant c5 : bit_vector(0 to 1) := '0' & '1';
```

Note that the concatenation operator may be also used to concatenate an array element with an array (see "c4") or two array elements (see "c5").

Due to a flaw in the definition of VHDL'87 the result of a concatenation may produce illegal array bounds. In VHDL 87 the left bound of the result is the left bound of the left operand, unless it is a null array (an array of length 0; see also FAQ Part 4 - B.165), in which case it is the left bound of the right operand. The direction of the result is derived from the left and right operand in a similar fashion. Consider the following sequence

```
type r is 0 to 7;
type r_vector is array (r <> range) of bit;

constant k1 : r_vector(1 downto 0) := "10";
constant k2 : r_vector(0 to 1) := "01";

constant k3 : r_vector := k2 & k1;
constant k4 : r_vector := k1 & k2;
```

According to the VHDL’87 rules for concatenation, the left bound of constant "k3" is 0 and the right bound is 3. Similarly, the left bound of "k4" is 1 while the right bound is -2. Obviously, the right bound of "k4" is illegal as it is not within the range defined by type "r".

To fix this problem in VHDL’93 the corresponding rules has been modified: if both operands are null arrays, then the result of the concatenation is the right operand. Otherwise, the direction and bounds of the result are determined from the the index subtype of the base type of the result. The direction and the left bound of the result are taken from the index subtype while the right bound is calculated based on the left bound, the direction and the length of the array.

In the previous example the index subtype of the base type of "k3" and "k4" is "r". Hence, the left bounds of "k3" and "k4" are set to 0 and the directions are "to". Finally, the right bounds of both constants equals to 0 + length of array - 1 = 0 + 4 - 1 = 3:

	k3’left	k3’right		k4’left	k4’right
VHDL’87	0	3		1	-2
VHDL’93	0	3		0	3

Note that the resulting sequence of bits is the same in either case. The only difference between the operators is in the construction of the index bounds of the result.

So, as long as the index values of the result don’t go out of bounds, and the code using the concatenation does not depend on specific index values of the result (which is a bad idea in any case), the differences between the VHDL’87 and VHDL’93 concatenation operators are of no concern. In fact, the differences are precisely to avoid the condition where the index values needlessly go out out bounds.

4.2.43 rising_edge(clk) versus (clk’event and clk=’1’)

In general, there are two ways in VHDL to describe edge sensitive elements (we assume that signal "clk" is of type std_logic):

- The expression

  ```
  clk'event and clk = '1'
  ```

  returns true if signal "clk" has a transition and its new value is '1'. If the previous value of "clk" is '0', then this expression correctly detects a rising edge. However, any transition that ends at '1' will be considered as a rising edge. Hence, changes from 'H' (weak '1') to '1' will return true while transitions from '0' to 'H' evaluate to false. As a result, "clk'event and clk = '1'" is only safe if "clk" toggles between '0' and '1'.
-

  ```
  rising_edge (clk)
  ```

  is defined in package std_logic_1164 and returns true if signal "clk" has a rising transition (to detect a falling edge use "falling_edge"). The function is implemented as follows:

  ```
  function rising_edge  (signal s : std_ulogic) return boolean is
  begin
     return (s'event and (To_X01(s) = '1') and
                          (To_X01(s'last_value) = '0'));
  end;
  ```

  It uses the function "To_X01" to compare the current and previous value of "clk" making it more robust against "unusual" clock values. "To_X01" maps 'H' to '1' and 'L' to '0'. Hence, rising_edge also returns true for transitions from '0' to 'H' and returns false for transitions from 'H' to '1' (as well as changes from 'X' to '1').

For synthesis, there is usually no difference between both alternatives as the synthesis tool assumes that transitions on clock signals are "clean" (i.e., clock signals toggle between '0' and '1' only). However, for simulation it is safer to use "rising_edge" (or "falling_edge").

4.3 What do I Need to Generate Hardware from VHDL Models

Usually, VHDL is used to describe digital designs that are either programmed into FPGAs (field-programmable gate arrays) or integrated into ASICs (application-specific integrated circuits) to perform a specific function. In order to build (or, in case of FPGAs, program) a chip, several tools are needed:

- **Simulation tool**

  The first step in designing a circuit that concerns us is describing its intended behavior in VHDL. This initial model may or may not be described in synthesizable VHDL (a subset of the entire VHDL language), but the description must eventually be refined until it uses only synthesizable VHDL prior to synthesis. But, how does one determine whether the described model reflects the intended behavior?

A tool called a simulator is used to simulate the model in order to verify that the description is working as intended. (It is usually not the case that the model is correct when first described.) Note that at this stage the simulator only simulates the VHDL model and not the real circuit. I.e., the behavior of the model and the hardware that is finally generated from it may differ in ways that may be significant. (There are other tools and methods to detect such problems; see below).

In addition to the VHDL model of the circuit to be developed, a "testbench" should also be coded. The purpose of the testbench is to provide an environment that generates appropriate input to the model and that also checks its output. Unlike the circuit model, the testbench does not have to be synthesizable. Hence, all language features can be used to describe it (e.g., file read and write operations to log input to and output of the circuit).

While some FPGA vendors provide "home grown" simulators along with a tool chain to program their devices, these tools typically do not contain a full-language VHDL simulator. In particular, they are not capable of running arbitrary testbenches. Hence, for any real design task a separate VHDL simulation tool is typically required. A list of simulators can be found in Part 3 Section 2 of this document.

- **Synthesis tool**

  Synthesis tools accept a synthesizable VHDL description and generates a netlist consisting of logic primitives (gates, flipflops, latches, etc.). While a significant part of the VHDL language can be synthesized, there are constructs and combinations of constructs that cannot be handled by today's synthesis tools. Some of the restrictions are due to a lack of corresponding counterparts in hardware (e.g., report statements and files). Other restrictions are due to the fact that all statements of a synthesizable VHDL description must ultimately be statically mapped to a set of corresponding hardware primitives. Note that the VHDL subset supported for synthesis differs between synthesis tools. There is, however, a common subset definition, IEEE Std 1076.6, in progress.

  Synthesis is a complex and resource-consuming process, one that might return results not intended by the designer. A common problem is that synthesis tools and (unexperienced) designers might differ in the interpretation of a specific sequence of VHDL statements as hardware. There are two main approaches to determine whether the synthesized netlist matches the functionality of the VHDL description:

  - **Simulate the synthesized netlist (gate-level simulation)**

    In most cases this can be also done using a VHDL simulator. An advantage of this approach is that the testbench can be re-used with the synthesized circuit. The same or similar test patterns can also be applied to the gate level model to verify that it has response identical to the original model.

Unfortunately, gate level simulation can be rather slow. As a consequence, it may not be possible to run the entire suite of tests, so detecting differences via this approach may not be reliable.

- **Use an equivalence checker to formally prove that synthesizable model and netlist are functionally equivalent.**

  Depending on the circuit structure and complexity, this approach might work well or not. It does, however, require the purchase and use of a separate (and expensive) tool.

- **Implementation tools**

  The synthesis tool creates a (more or less) device-independent description of the circuit. Another tool chain is required to map the primitives used in the netlist to resources in the physical device (FPGA or ASIC). This mapping occurs during "implementation". Implementation tools for FPGAs are usually provided by the FPGA vendor.

  In addition to a description that can be downloaded to the FPGA or sent to the appropriate IC foundry (in case of an ASIC design), the implementation tools also generate a detailed timing model from the circuitry. A timing model consists of gate-level primitives augmented with detailed delay information and can be used to verify the timing behavior of the synthesized circuit, in a process called "timing verification." (This model could also be used to verify the the functional behavior of the synthesized circuit, but this use may be inefficient).

  The purpose of timing verification is to ensure that the circuit meets all timing constraints (clock frequency, setup and hold times of flipflops, ...). There are two main techniques for timing verification:

  - **Timing simulation**

    Timing simulation can be also performed by a VHDL simulation tool. Unfortunately, timing simulation is typically very slow and resource intensive. Further, good coverage is very dependent on the stimuli provided; poor choices can lead to timing violations remaining undiscovered.

  - **Static timing analysis**

    When doing synchronous or "mostly" synchronous designs (the main design methodology in use today), timing may be also verified using a static timing analyzer. This tool conservatively analyzes all paths within the design to locate the most critical (longest) one. From this analysis, the maximum clock frequency and other timing properties are derived.

Timing analysis is a conservative approach (i.e., it may report timing violations that are not, in actuality, present in the real device). However, the tradeoff is that such analysis usually requires significantly less computational effort when compared to simulation. Hence, it is the main technique for timing verification (at least in case of synchronous or mostly synchronous designs) in use today. Nevertheless, some designers prefer to run a final timing simulation to double-check the results obtained from static timing analysis. Note that FPGAs vendors usually provide static timing analysis tools along with their implementation tool chain.

While this should give a rough overview on what is needed to design circuits using a VHDL based design flow, it is by no way an exact and exhaustive description of all tools that may be required and all problems that may be encountered. ASIC design flows, especially, are sometimes far more complex than described here. Hence, please refer to vendor documentation or appropriate literature for further information.

4.4 PUBLIC DOMAIN Tools?

Actually as far as I know, there is only few PD software on VHDL. If you know about something, please let us know. See products posting for more detailed information.

4.5 Is There a VHDL Validation Suite Available?

Yes. The latest version of the test validation suite is available via anonymous http://mikro.e-technik.uni-ulm.de/vhdl/vhdl_utilities.html The current suite covers 36% of VHDL-1987. See also FAQ Part 3, Section 1.5

4.6 Status of Analog VHDL (VHDL-AMS, 1076.1)

The IEEE Standard 1076.1-1999 (Analog and Mixed-Signal Extensions to VHDL) has been approved on March 18, 1999 and the 1076.1 Language Reference Manual is available from the IEEE. VHDL 1076.1-1999 is a strict extension of VHDL 1076-1993, i.e. it includes the full VHDL 1076-1993 language. Extensions have been carefully designed to maintain the existing philosophy and to smoothly integrate new language definitions.

There is a free VHDL-AMS (Analog and Mixed Signal) simulator/compiler called SEAMS available. See part 3, Section 3.1 for further information on SEAMS.

4.7 How to Get More Information about VHDL-AMS (1076.1)

The 1076.1 study group is maintaining an email bulletin board for distribution of announcements and as a forum for technical discussions (see above: official contacts).

4.8 Standards and Standard Packages

The latest standard packages are usually available from IEEE (http://standards.ieee.org/). However, they are not free of charge. In general it is a good idea to look at http://www.eda.org/ or http://dz.ee.ethz.ch/support/ic/vhdl/vhdlsources.en.html.

If someone has a partial or complete list, please send it to the author. I will incorporate it here, if possible. Here is, what is currently available:

- The IEEE 1164 std_logic package (std_logic_1164):
  - version 4.2 (note that this is not the latest one):
    http://tech-www.informatik.uni-hamburg.de/vhdl/packages/ieee_1164/std_logic_1164.vhd
- VHDL 1076.3 Bit Logic based synthesis package (numeric_bit; available from the IEEE). Version 2.4 of numeric_bit can be found at:
  http://www.eda.org/rassp/vhdl/models/standards/numeric_bit.vhd
- VHDL 1076.3 Std_Logic_1164 based synthesis package (numeric_std; available from the IEEE). Version 2.4 of numeric_std can be found at:
  http://www.eda.org/rassp/vhdl/models/standards/numeric_std.vhd
- VITAL ’95 Packages: http://vhdl.org/vi/vital/
- IEEE MATH_REAL Package (DRAFT!) containing log, sin,...:
  http://tech-www.informatik.uni-hamburg.de/vhdl/packages/mathpack/mathpack.vhd

4.8.1 Functions and Operators Defined in Package numeric_std

There are two IEEE-standard packages that provide functionality to handle bit vectors as coded numeric values. The types/functions based on bits are found in the package IEEE.numeric_bit, and the types/functions based on std_logic are found in IEEE.numeric_std. Of the two, numeric_std is the more commonly used.

Both packages (see Section 4.8 on how to get these packages) define two new vector types: SIGNED and UNSIGNED. SIGNED vectors represent two’s-complement integers, while UNSIGNED vectors represent unsigned-magnitude integers.

Note: Synopsys produced another set of packages named std_logic_arith, std_logic_signed, and std_logic_unsigned that are intended to provide functionality similar to numeric_bit and numeric_std. In particular, the same two types, SIGNED and UNSIGNED are defined. These packages are typically even installed in the library IEEE. However, these packages are NOT standard, and different vendors have different and mutually incompatible versions. Also, there are naming clashes when some of these packages are used together. So, it is recommended that numeric_bit or numeric_std be used in preference to these non-standard packages. See http://dz.ee.ethz.ch/support/ic/vhdl/vhdlsources.en.html for a comparison of conversion functions defined in std_logic_arith and their corresponding counterparts in numeric_std.

In order to help designers making best use of numeric_std the following tables list operators and functions defined in this package. In particular, the tables list

- function or operator name
- first parameter type
- type of an optional second parameter
- return type
- some additional remarks

In order to save space type names in the table are abbreviated as follows

type name	abbreviation	remarks
integer	int	
natural	nat	integer >= 0; subtype of integer
boolean	bool	
bit_vector	bvec	
std_ulogic	sul	
std_ulogic_vector	sulv	
std_logic	sl	resolved subtype of sul
std_logic_vector	slv	
unsigned	uns	
signed	sgn	

The function and operators are clustered into groups "logical operators", "arithmetic operators", "compare operators", " and rotate functions", "conversion functions" and "miscellaneous functions":

- Logical operators:

operator	1st par	2nd par	return	remarks
and	sgn	sgn	sgn	bitwise and
and	uns	uns	uns	bitwise and
nand	sgn	sgn	sgn	bitwise nand
nand	uns	uns	uns	bitwise nand
or	sgn	sgn	sgn	bitwise or
or	uns	uns	uns	bitwise or
nor	sgn	sgn	sgn	bitwise nor
nor	uns	uns	uns	bitwise nor
xor	sgn	sgn	sgn	bitwise xor
xor	uns	uns	uns	bitwise xor
xnor	sgn	sgn	sgn	bitwise xnor
xnor	uns	uns	uns	bitwise xnor
not	sgn		sgn	bitwise not
not	uns		uns	bitwise not

- Arithmetic Operators:

  Note that the arithmetic operators return "XX...X" if at least one of their parameters contain values other than ’0’, ’1’, ’L’ or ’H.

operator	1st par	2nd par	return	remarks
abs	sgn		sgn	absolute value
-	sgn		sgn	change sign
+	uns	uns	uns	unsigned add; length of result = max(length of 1st par, length of 2nd par)
+	sgn	sgn	uns	signed add (two’s-complement); length of result = max(length of 1st par, length of 2nd par)
+	uns	nat	uns	add an uns with a non-negative int; length of result = length of 1st par
+	nat	uns	uns	add an uns with a non-negative int; length of result = length of 2nd par

+	sgn	int	sgn	add a sgn with an int; length of result = length of 1st par
+	int	sgn	sgn	add a sgn with an int; length of result = length of 2nd par
-	uns	uns	uns	unsigned subtract; length of result = max(length of 1st par, length of 2nd par)
-	sgn	sgn	uns	signed subtract (two's-complement); length of result = max(length of 1st par, length of 2nd par)
-	uns	nat	uns	subtract a non-negative int from an uns; length of result = length of 1st par
-	nat	uns	uns	subtract an uns from a non-negative int; length of result = length of 2nd par
-	sgn	int	sgn	subtract an int from a sgn; length of result = length of 1st par
-	int	sgn	sgn	subtract a sgn from an int; length of result = length of 2nd par
*	uns	uns	uns	unsigned multiply; length of result = length of 1st par + length of 2nd par - 1
*	sgn	sgn	sgn	signed multiply (two's-complement); length of result = length of 1st par + length of 2nd par - 1
*	uns	nat	uns	multiply a non-negative int with an uns; nat is converted to uns (length = length of 1st par) before multiplication; length of result = 2 * (length of 1st par) - 1
*	nat	uns	uns	multiply an uns with a non-negative int; nat is converted to uns (length = length of 2nd par) before multiplication; length of result = 2 * (length of 2nd par) - 1
*	sgn	int	sgn	multiply an int with a sgn (two's-complement); int is converted to sgn (length = length of 1st par) before multiplication; length of result = 2 * (length of 1st par) - 1
*	int	sgn	sgn	multiply a sgn with an int (two's-complement); int is converted to sgn (length = length of 2nd par) before multiplication; length of result = 2 * (length of 2nd par) - 1
/	uns	uns	uns	unsigned divide; resulting uns has same length as 1st par
/	sgn	sgn	sgn	signed divide; resulting sgn has same length as 1st par
/	uns	nat	uns	divide an uns by a non-negative int; resulting uns has same length as 1st par
/	nat	uns	uns	divide a non-negative int by an uns; resulting uns has same length as 2nd par
/	sgn	int	sgn	divide an sgn by an int; resulting sgn has same length as 1st par

/	int	sgn	sgn	divide an int by a sgn; resulting sgn has same length as 2nd par
rem	uns	uns	uns	unsigned rem; resulting uns has same length as 1st par
rem	sgn	sgn	sgn	signed rem; resulting sgn has same length as 1st par
rem	uns	nat	uns	compute 1st par rem 2nd par; resulting uns has same length as 1st par
rem	nat	uns	uns	compute 1st par rem 2nd par; resulting uns has same length as 2nd par
rem	sgn	int	sgn	compute 1st par rem 2nd par; resulting sgn has same length as 1st par
rem	int	sgn	sgn	compute 1st par rem 2nd par; resulting sgn has same length as 2nd par
mod	uns	uns	uns	unsigned mod; resulting uns has same length as 1st par
mod	sgn	sgn	sgn	signed mod; resulting sgn has same length as 1st par
mod	uns	nat	uns	compute 1st par mod 2nd par; resulting uns has same length as 1st par
mod	nat	uns	uns	compute 1st par mod 2nd par; resulting uns has same length as 2nd par
mod	sgn	int	sgn	compute 1st par mod 2nd par; resulting sgn has same length as 1st par
mod	int	sgn	sgn	compute 1st par mod 2nd par; resulting sgn has same length as 2nd par

- Compare operators:

operator	1st par	2nd par	return	remarks
>	uns	uns	bool	compare greater than; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
>	sgn	sgn	bool	compare greater than; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
>	uns	nat	bool	compare greater than; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
>	nat	uns	bool	compare greater than; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
>	sgn	int	bool	compare greater than; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’

>	int	sgn	bool	compare greater than; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
<	uns	uns	bool	compare less than; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
<	sgn	sgn	bool	compare less than; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
<	uns	nat	bool	compare less than; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
<	nat	uns	bool	compare less than; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
<	sgn	int	bool	compare less than; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
<	int	sgn	bool	compare less than; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
<=	uns	uns	bool	compare less equal; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
<=	sgn	sgn	bool	compare less equal; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
<=	uns	nat	bool	compare less equal; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
<=	nat	uns	bool	compare less equal; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
<=	sgn	int	bool	compare less equal; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
<=	int	sgn	bool	compare less equal; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
>=	uns	uns	bool	compare greater equal; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
>=	sgn	sgn	bool	compare greater equal; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
>=	uns	nat	bool	compare greater equal; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
>=	nat	uns	bool	compare greater equal; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
>=	sgn	int	bool	compare greater equal; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
>=	int	sgn	bool	compare greater equal; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’

=	uns	uns	bool	compare equal; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
=	sgn	sgn	bool	compare equal; returns false if parameter contain values other than ’0’, ’1’, ’L’, ’H’
=	uns	nat	bool	compare equal; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
=	nat	uns	bool	compare equal; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
=	sgn	int	bool	compare equal; returns false if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
=	int	sgn	bool	compare equal; returns false if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
/=	uns	uns	bool	compare not equal; returns true if parameter contain values other than ’0’, ’1’, ’L’, ’H’
/=	sgn	sgn	bool	compare not equal; returns true if parameter contain values other than ’0’, ’1’, ’L’, ’H’
/=	uns	nat	bool	compare equal; returns true if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
/=	nat	uns	bool	compare not equal; returns true if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’
/=	sgn	int	bool	compare not equal; returns true if 1st par contains values other than ’0’, ’1’, ’L’, ’H’
/=	int	sgn	bool	compare not equal; returns true if 2nd par contains values other than ’0’, ’1’, ’L’, ’H’

- Shift and rotate functions:

function	1st par	2nd par	return	remarks
shift_left	uns	nat	uns	1st par left by 2nd par bit positions; vacated positions are filled with ’0’
shift_right	uns	nat	uns	1st par right by 2nd par bit positions; vacated positions are filled with ’0’
shift_left	sgn	nat	sgn	1st par left by 2nd par bit positions; vacated positions are filled with ’0’
shift_right	sgn	nat	sgn	1st par left by 2nd par bit positions; vacated positions are filled with leftmost bit of sgn (sign extension)
rotate_left	uns	nat	uns	rotate 1st par left by 2nd par bit positions
rotate_right	uns	nat	uns	rotate 1st par right by 2nd par bit positions
rotate_left	sgn	nat	sgn	rotate 1st par left by 2nd par bit positions
rotate_right	sgn	nat	sgn	rotate 1st par left by 2nd par bit positions

- Shift and rotate operators:

  Note that these *operators* are not compatible with VHDL’87.

operator	1st par	2nd par	return	remarks
sll	uns	int	uns	shifts left 1st par by 2nd par bit positions if 2nd par >=0, otherwise shifts right; vacated positions are filled with ’0’
srl	uns	int	uns	shifts right 1st par by 2nd par bit positions if 2nd par >=0, otherwise shifts left; vacated positions are filled with ’0’
sll	sgn	int	sgn	shifts left 1st par by 2nd par bit positions if 2nd par >=0, otherwise shifts right; vacated positions are filled with ’0’
srl	sgn	int	sgn	shifts right 1st par by 2nd par bit positions if 2nd par >=0, otherwise shifts left; vacated positions are filled with ’0’
rol	uns	int	uns	rotates left 1st par by 2nd par bit positions if 2nd par >=0, otherwise rotates right
ror	uns	int	uns	rotates right 1st par by 2nd par bit positions if 2nd par >=0, otherwise rotates left
rol	sgn	int	sgn	rotates left 1st par by 2nd par bit positions if 2nd par >=0, otherwise rotates right
ror	sgn	int	sgn	rotates right 1st par by 2nd par bit positions if 2nd par >=0, otherwise rotates left

- Conversion functions:

function	1st par	2nd par	return	remarks
To_Integer	uns		nat	convert uns to nat
To_Integer	sgn		int	convert sgn to int
To_Unsigned	nat	nat	uns	convert nat (1st par) to uns; 2nd par defines bit width of result
To_Signed	int	nat	sgn	convert int (1st par) to sgn; 2nd par defines bit width of result

Note that UNSIGNED and SIGNED (as well as std_logic_vectors) are closely related to each other (see FAQ Part 4 - B.40). Hence, predefined conversion functions (see Section 4.2.41, FAQ Part 4 - B.50 and Section 4.2.18) can be applied to change the type of a vector as required. For a table of conversion functions defined in ieee.std_logic_1164 and ieee.numeric_std see also http://www.ce.rit.edu/pxseec/VHDL/Conversions.htm.

- Translation functions:

function	1st par	2nd par	return	remarks
To_01	uns	sl	uns	termwise translation of 1st par; ’1’ or ’H’ are translated to ’1’, and ’0’ or ’L’ are translated to ’0’; if a value other than ’0’|’1’|’H’|’L’ is found (others => 2nd par) is returned
To_01	sgn	sl	sgn	termwise translation of 1st par; ’1’ or ’H’ are translated to ’1’, and ’0’ or ’L’ are translated to ’0’; if a value other than ’0’|’1’|’H’|’L’ is found (others => 2nd par) is returned

- Miscellaneous functions:

function	1st par	2nd par	return	remarks
resize	uns	nat	uns	resize 1st par to the bit width specified by the 2nd par; any new [leftmost] bit positions are filled with '0'
resize	sgn	nat	sgn	resize 1st par to the bit width specified by the 2nd par; any new [leftmost] bit positions are filled with sign bit (left most bit of 1st par); when truncating the sign bit is retained
std_match	sul	sul	bool	compare terms per std_logic_1164 intent; see Section 4.2.9
std_match	slv	slv	bool	compare termwise per std_logic_1164 intent; see Section 4.2.9
std_match	sulv	sulv	bool	compare termwise per std_logic_1164 intent; see Section 4.2.9
std_match	uns	uns	bool	compare termwise per std_logic_1164 intent; see Section 4.2.9
std_match	sgn	sgn	bool	compare termwise per std_logic_1164 intent; see Section 4.2.9

4.9 Where to Obtain the comp.lang.vhdl FAQ

http://vhdl.org/comp.lang.vhdl/ (See also here)

4.10 "Frequently Requested" Models/Packages

Not all models shown here are really frequently requested but I did not find a better place to list them.

- Not synthesizable random number generators:
  - http://www.eda.org/rassp/vhdl/models/math.html
  - http://vhdl.org/vi/vhdlsynth/
  - http://www.janick.bergeron.com/wtb/packages/random1.vhd
  - Package ieee.math_real includes a random generator. While this package is not freely available usually each commercial VHDL compiler comes with a pre-compiled version of it. The random generator procedure is named "UNIFORM" and returns a pseudo random real number with uniform distribution in the interval 0.0 to 1.0:

```
procedure UNIFORM (variable Seed1,Seed2: inout integer;
                   variable X:out real);
```

- Linear feedback registers/LFSR (often used as synthesizable pseudo random bit stream generator; less good for generating random integers):
  - http://www.vhdlcohen.com/
  - See Xlinix application note XAPP 052: http://www.xilinx.com/bvdocs/appnotes/xapp052.pdf
  - A LFSR generator tool: http://www.logiccell.com/~jean/LFSR/
- Arithmetic models/packages

- Math package (log, sin, cos ...; not synthesizable): see Section 4.8
- A synthesizable function that returns the integer part of the base 2 logarithm for a positive number is (uses recursion; posted by Tuukka Toivonen):

```
function log2 (x : positive) return natural is
begin
  if x <= 1 then
    return 0;
  else
    return log2 (x / 2) + 1;
  end if;
end function log2;
```

  Another synthesizable log2 function using a while loop is (original posted by Ray Andraka; modified by editor):

```
function log2 (x : positive) return natural is
  variable temp, log: natural;
begin
  temp := x / 2;
  log := 0;
  while (temp /= 0) loop
    temp := temp/2;
    log := log + 1;
  end loop;
  return log;
end function log2;
```

  Note that these functions are not intended to synthesize directly into hardware, rather they are used to generate constants for synthesized hardware.
- Commercial floating point arithmetic cores from Digital Core Design: http://www.dcd.com.pl/
- An IEEE-754 synthesizable floating point core by Jamil Khatib (alpha status): http://www.geocities.com/SiliconValley/Pines/6639/ip/fpu/
- Floating point arithmetic models from Hiroaki Yamaok: http://flex.ee.uec.ac.jp/~yamaoka/vhdl/index.html
- Floating point models (behavior level): http://www.ics.uci.edu/pub/hlsynth/HLSynth95/HLSynth95/complete/
- Commercial floating point IP cores from Dillon Engineering, Inc.: http://www.dilloneng.com/ipcores/fpoint/index.html
- Synthesizable floating point arithmetic packages from the IEEE 1076.3 working group: http://www.eda.org/fphdl/
- Synthesizable fixed point arithmetic packages written by David Bishop: http://www.vhdl.org/vhdlsynth/proposals/dave_p3.html

  - Another synthesizable fixed point arithmetic package written by Jonathan Bromley: http://www.doulos.co.uk/knowhow/vhdl_models/fp_arith/
  - Free CORDIC core (COordinate Rotation on a DIgital Computer) from OPENCORES.ORG: http://www.opencores.org/cores/cordic/
  - Commercial CORDIC core from Digital Core Design: http://www.digitalcoredesign.com/
- Models of the DLX RISC processor:
  - http://www.ashenden.com.au/designers-guide/DG.html
  - http://www.eda.org/rassp/vhdl/models/processor.html
  - ftp://ftp.informatik.uni-stuttgart.de/pub/vhdl/DLXS-P.beta/
  - A superscalar version of the DLX processor: http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html
- Other processor models:
  - ERC32 (a fully functional, timing accurate model of a radiation-tolerant SPARC V7 processor version): http://www.estec.esa.nl/wsmwww/erc32/
  - LEON-1 (a synthesisable SPARC compatible (integer) processor): http://www.estec.esa.nl/wsmwww/erc32/ or http://www.gaisler.com/
  - Dalton Project (a synthesizable VHDL Model of the 8051 microcontroller): http://www.cs.ucr.edu/~dalton/i8051/
  - Free behavioral model of the 8051 microcontroller (provided by Alatek): http://www.alatek.com/
  - Commercial 8051 compatible cores from Digital Core Design: http://www.dcd.com.pl/
  - T51 mcu, a free 8052 compatible core: http://www.opencores.org/projects/t51/
  - Another 8051 compatible IP Core: http://www.oregano.at/ip/8051.htm
  - A simple (and incomplete) behavioral model of the Intel 80386 microprocessor: http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
  - A simple (and incomplete) behavioral model of the Motorola 68000 microprocessor: http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
  - Commercial 68000 compatible core from Digital Core Design: http://www.dcd.com.pl/
  - Another commercial 68000 compatible core from VLSI Concepts: http://www.vlsi-concepts.com/V68000.html
  - A model of the AMD 2901: http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
  - A structural synthesisable model of the PIC16C5x microcontroller: http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
  - Another synthesisable model of the PIC 16C5X microcontroller: http://www.mindspring.com/~tcoonan/
  - PPX16, a free core compatible with 16C55 and 16F84: http://hem.passagen.se/dwallner/vhdl.html
  - CQPIC, another free PIC16F84 compatible processor core: http://www02.so-net.ne.jp/~morioka/cqpic.htm
  - RISC5x, a PIC compatible core from from OPENCORES.ORG:

http://www.opencores.org/projects/risc5x/
    - Another 6805 core: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/2000_w/vhdl/6805/
    - A 320c25 compatible digital signal processor core from Digital Core Design: http://www.dcd.com.pl/
    - GM HC11 CPU Core, a synthesizable VHDL implementation of the HC11 CPU: http://www.gmvhdl.com/hc11core.html
    - MSL16, a CPU optimised to run Forth programs: http://www.cs.cuhk.edu.hk/~phwl/msl16/msl16.html
    - T80, a free core compatible with Z80: http://hem.passagen.se/dwallner/vhdl.html
    - JAM, a 32bit 5 stage pipelined RISC core with forwarding and hazard handling (basic design is derived from the DLX architecture): http://www.etek.chalmers.se/~e8johan/concert02/index.html
    - XiRISC, an extensible RISC core: http://xirisc.deis.unibo.it/
    - A minimal 8 bit VHDL CPU designed for a 32 macrocell CPLD: http://www.tu-harburg.de/~setb0209/cpu/mcpu.html
    - MicroCore, a simple micro controller core targeting FPGAs: http://www.microcore.org/index.html
    - uP1232, a 8-bit FPGA-based microprocessor core: http://www.dte.eis.uva.es/OpenProjects/OpenUP/index.htm
    - DRAGONFLY microprocessor core: http://www.leox.org/resources/dvlp.html
    - An 8-bit Stack Processor: http://www.compusmart.ab.ca/rc/Papers/8bitprocessor/stackproc.html
    - JOP - Java Optimized Processor: http://www.jopdesign.com/download.jsp
    - Raptor-16 CISC Microprocessor: http://www.spacewire.co.uk/
- Gray code counter:
    - Gray counter with variable width: http://www.isibrno.cz/~ivovi/vhdl.htm
    - See also Section 4.2.29
- I2C-bus interface:
    - http://www.opencores.org/
    - http://www.corepool.com/ (commercial)
    - See Xilinx application note XAPP 333: http://www.xilinx.com/bvdocs/appnotes/xapp333.pdf
    - Digital Core Design: http://www.dcd.com.pl/ (commercial)
- UART models (Universal Asynchronous Receiver/Transmitter):
    - Ben Cohens web site: http://members.aol.com/vhdlcohen/vhdl
    - The RASSP www site: http://www.eda.org/rassp/vhdl/models/bus.html
    - Serial UART Controller from the OPENCORES.ORG project web site: http://www.opencores.org/cores/uart/
- PCI-bus interface:
    - PLD Applications (commercial): http://www.plda.com/

  - Xlinix, Inc. (commercial; for Xilinx FPGAs): http://www.xilinx.com/ipcenter/
  - Phoenix Technologies (commercial): http://www.vchips.com/
- CAN controller core:
  - Hurricane: ftp://ftp.estec.esa.nl/pub/ws/wsd/CAN/can.htm
  - CAN protocol controller from opencores: http://www.opencores.org/projects/can/
- CRC and BCH encoder, Reed Solomon encoder/decoder, Viterbi encoder/decoder, LDPC decoder:
  - Tools to generate (synthesizable) CRC and BCH encoders, Reed Solomon encoders/decoders are available from the MOBIS Forward Error Correction Page: http://www.fokus.gmd.de/research/cc/mobis/products/fec_old/content.html
  - CRC Tool from easics is a free web tool that can be run interactively to generate synthesizable VHDL code for any polynomial and any data input width (within reasonable limits) over the web: http://www.easics.com/webtools/crctool
  - Viterbi encoder/decoder for the (22,8,6) block code: http://www-ee.eng.hawaii.edu/~pramod/ee628/viterbi.html
  - CRC generator tool from Alan Coppola: http://www.nwlink.com/~ajjc
  - Another CRC generator tool from NoBug Consulting, Inc.: http://www.nobugconsulting.ro/crc.php
  - Confluence LDPC Decoder: http://www.opencores.org/projects/cf_ldpc/
- Memory models/cores:
  - Microsystems Prototyping Laboratory Static RAM (SRAM) Generation Tool: http://www.erc.msstate.edu/mpl/distributions/vhdl/tools/sramgen/
  - RAM models from Micron Semiconductor Products, Inc. (SRAM, DRAM, SDRAM, DDR SGRAM, ...): http://www.micron.com/mti
  - The Hamburg VHDL archive (SRAM, DRAM, EPROM): http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
  - Asynchronous FIFO core from Siemens: http://www.is.siemens.de/itps/eda/englisch/main1183832.htm
  - Memory cores from Jamil Khatib (FIFO, dual/single port memories): http://www.geocities.com/SiliconValley/Pines/6639/ip/memory_cores.html or http://www.opencores.org/
  - Memory models from the Free Model Foundry (FMF): http://www.eda.org/fmf/fmf_public_models/ram/
  - Intel flash models: http://appzone.intel.com/toolcatalog/list.asp?architecture=1&tooltype=Modeling/Simulation
  - Samsung Electronics models (graphics memories, flash): http://www.samsungelectronics.com/support_index.html
- Encryption/Decryption
  - Basic DES Crypto Core from OPENCORES.ORG: http://www.opencores.org/projects.cgi/web/basicdes/overview
  - A encryption/decryption core using DES and RSA (University of Stuttgart;

documentation is mostly in German language):
http://www.ra.informatik.uni-stuttgart.de/~stankats/
- VHDL models developed by NSA to evaluate the hardware performance of the AES finalists: http://csrc.nist.gov/encryption/aes/round2/r2anlsys.htm
- Blowfish Implementation in VHDL (by Wesley J. Landaker):
http://blowfishvhdl.sourceforge.net/

- SDRAM-Controller:
  - SDRAM Controller from the OPENCORES.ORG project web site: http://www.opencores.org/cores/sdram/
- Printing
  - Formatted printing: A package (PCK_FIO) providing C-style formatted printing is availabe from http://www.easics.com/webtools/freesics
  - "image_pb.vhd" includes functions to create string representations of vectors (bit_vector, std_ulogic_vector, std_logic_vector, signed, unsigned):
http://members.aol.com/vhdlcohen/vhdl/
  - Package "ieee.std_logic_textio" from Synopsys includes procedures to read/write std_logic, std_ulogic_vector, std_logic_vector values (including from/to hex and octal):
http://members.aol.com/vhdlcohen/vhdl/vhdlcode/stdtxtio.vhd
- Miscellaneous
  - A package that makes use of the ieee.numeric_std package, and provides overloaded functions and operators a la Synopsys Unsigned package (by Ben Cohen):
http://www.vhdlcohen.com/
  - Free FIR Filter core from OPENCORES.ORG: http://www.opencores.org/
  - Free VGA/LCD Controller core from OPENCORES.ORG:
http://www.opencores.org/cores/vga_lcd/
  - A package containing various functions to convert between hex/decimal/octal/binary strings and std_logic(_vector)/natural:
http://www.eda.org/fmf/fmf_public_models/packages/conversions.vhd

4.11 Arithmetic Packages for bit/std_logic-Vectors

The IEEE did not, originally, define a standard set of types and overloaded functions to handle vectors which contained coded numeric values. This meant that individual vendors were free to define their own types and functions.

Synopsys produced three packages - std_logic_arith, std_logic_signed, and std_logic_unsigned. std_logic_signed/std_logic_unsigned operated on type std_logic_vector, and gave an implicit meaning to the contents of the vector. std_logic_arith, however, defined two new types, SIGNED and UNSIGNED, and operated on these types only. Unfortunately, Synopsys decided that these packages should be compiled into library IEEE. Other vendors, including Cadence and Mentor, now produced their own versions of std_logic_arith, which were not the same as Synopsys's. They also required their packages to be placed in library IEEE.

Finally, the IEEE decided to standardize this situation, and produced packages numeric_bit and numeric_std (see Section 4.8 on how to obtain numeric_bit and numeric_std). numeric_bit is based on type bit, numeric_std on on type std_logic. Both packages followed Synopsys in defining new types, SIGNED and UNSIGNED. However, the package functions did not have the same names, or parameters, as the Synopsys functions.

Currently many vendors support numeric_bit/numeric_std. Hence, for maximum portability, avoid using a package called std_logic_signed or std_logic_unsigned, and always use SIGNED or UNSIGNED types (or integers) for arithmetic. If you are using Synopsys, use std_logic_arith, and if you are not using Synopsys, use numeric_std (if it is supported). This is not completely portable, since the functions are still different (for example, TO_UNSIGNED vs. CONV_UNSIGNED), but it is a lot better than using different types in different environments. See http://dz.ee.ethz.ch/support/ic/vhdl/vhdlsources.en.html for a comparison of conversion functions defined in std_logic_arith and their corresponding counterparts in numeric_std.

Partially extracted from an article by Evan Shattock.

4.12 Where Can I Find More Info

Of course, there are the other three parts of this FAQ.

Other good starting points are http://www.vhdl.org/docs/groups.txt or Section 3. VHDL on the Web.

If other resources should be listed here, please let me know.

Part 2: books on VHDL

Authors:

Tom Dettmer, Edwin Naroska