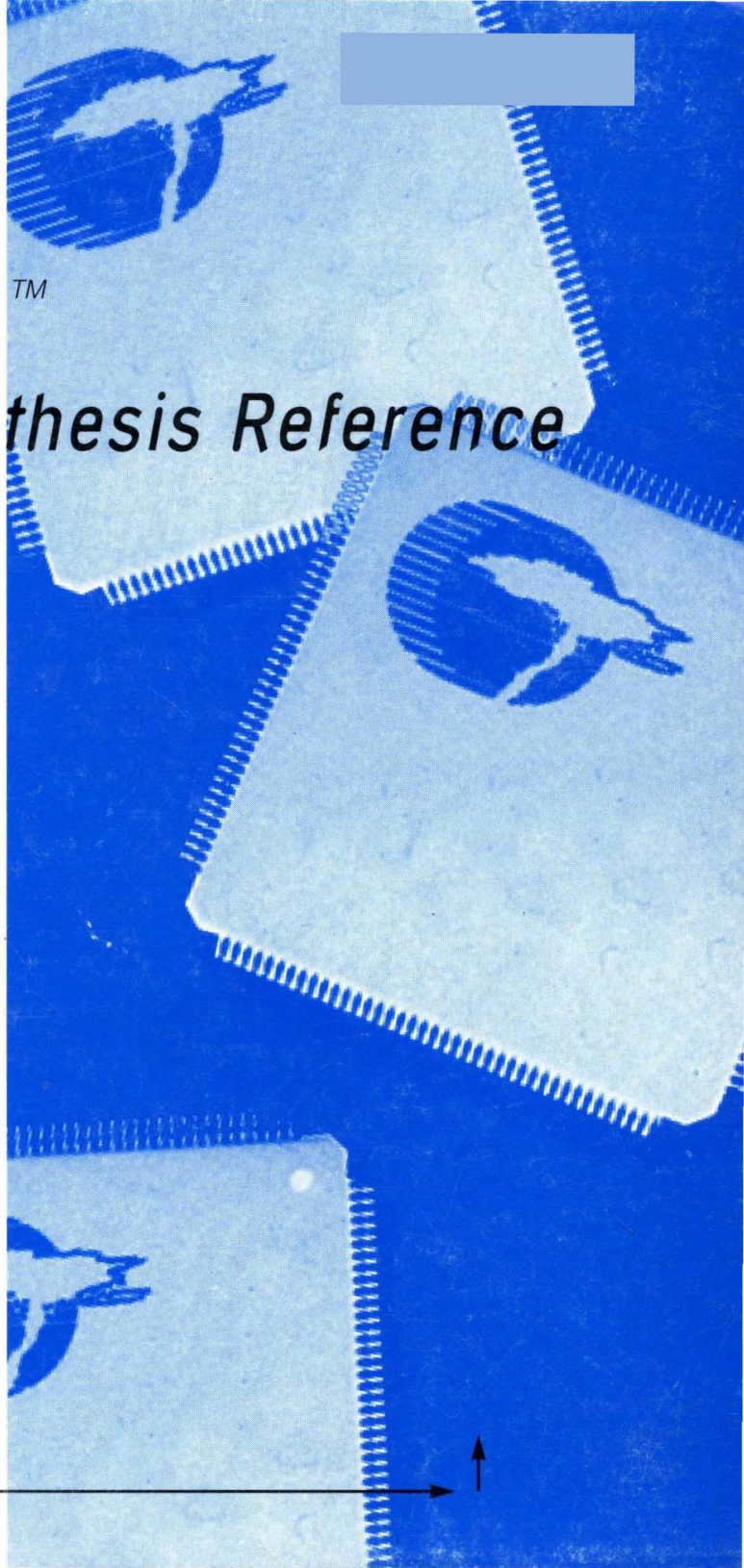


WARP™

VHDL Synthesis Reference

CYPRESS

Warp™

VHDL Development System

Warp Synthesis Compiler Manual

Cypress Semiconductor
3901 North First Street
San Jose, CA 95134
(408) 943-2600
JANUARY 1995

Cypress Software License Agreement

1. LICENSE. Cypress Semiconductor Corporation ("Cypress") hereby grants you, as a Customer and Licensee, a single-user, non-exclusive license to use the enclosed Cypress software program ("Program") on a single CPU at any given point in time. Cypress authorizes you to make archival copies of the software for the sole purpose of backing up your software and protecting your investment from loss.

2. TERM AND TERMINATION. This agreement is effective from the date the diskettes are received until this agreement is terminated. The unauthorized reproduction or use of the Program and/or documentation will immediately terminate this Agreement without notice. Upon termination you are to destroy both the Program and the documentation.

3. COPYRIGHT AND PROPRIETARY RIGHTS. The Program and documentation are protected by both United States Copyright Law and International Treaty provisions. This means that you must treat the documentation and Program just like a book, with the exception of making archival copies for the sole purpose of protecting your investment from loss. The Program may be used by any number of people, and may be moved from one computer to another, so long as there is **No Possibility** of its being used by two people at the same time.

4. DISCLAIMER. THIS PROGRAM AND DOCUMENTATION ARE LICENSED "AS-IS," WITHOUT WARRANTY AS TO PERFORMANCE. CYPRESS EXPRESSLY DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTY OF MERCHANTABILITY OR

FITNESS OF THIS PROGRAM FOR A PARTICULAR PURPOSE.

5. LIMITED WARRANTY. The diskette on which this Program is recorded is guaranteed for 90 days from date of purchase. If a defect occurs within 90 days, contact the representative at the place of purchase to arrange for a replacement.

6. LIMITATION OF REMEDIES AND LIABILITY. IN NO EVENT SHALL CYPRESS BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM PROGRAM USE, EVEN IF CYPRESS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. CYPRESS'S EXCLUSIVE LIABILITY AND YOUR EXCLUSIVE REMEDY WILL BE IN THE REPLACEMENT OF ANY DEFECTIVE DISKETTE AS PROVIDED ABOVE. IN NO EVENT SHALL CYPRESS'S LIABILITY HEREUNDER EXCEED THE PURCHASE PRICE OF THE SOFTWARE.

7. ENTIRE AGREEMENT. This agreement constitutes the sole and complete Agreement between Cypress and the Customer for use of the Program and documentation. Changes to this Agreement may be made only by written mutual consent.

8. GOVERNING LAW. This Agreement shall be governed by the laws of the State of California. Should you have any question concerning this agreement, please contact:

   Cypress Semiconductor Corporation
   Attn: Legal Counsel
   3901 N. First Street
   San Jose, CA 95134-1599

   408-943-2600

Table of Contents

Chapter 1

Introduction

Introduction

1.1. Overview of *Warp* Synthesis Compiler

The *Warp*[1] synthesis compiler is a state-of-the-art VHDL compiler for designing with Cypress PLDs, CPLDs, and pASIC380 FPGAs.

Warp utilizes a subset of IEEE 1076 VHDL as its Hardware Description Language (HDL) for design entry. *Warp* accepts VHDL text input, then synthesizes and optimizes the design for the target hardware. It then outputs a JEDEC map for programming PLDs and CPLDs, or a .QDF netlist for the place and route and eventual programming pASIC380 FPGAs (Figure 1-1).

The JEDEC map that *Warp* produces when targeting PLDs and CPLDs can be used to program parts with a device programmer. It can also be used as input to the Nova functional simulator. Nova is an interactive, graphical simulator that allows you to examine the behavior of synthesized designs.

The .QDF file that *Warp* produces when targeting pASIC380 FPGAs can be used as input to the SpDE Toolkit. The SpDE Toolkit is a collection of interactive, graphical tools that perform logic optimization, placement, and routing of pASIC designs.

FPGA support is available in Warp2+ and Warp3 only.

1. Warp is a Trademark of Cypress Semiconductor Corporation

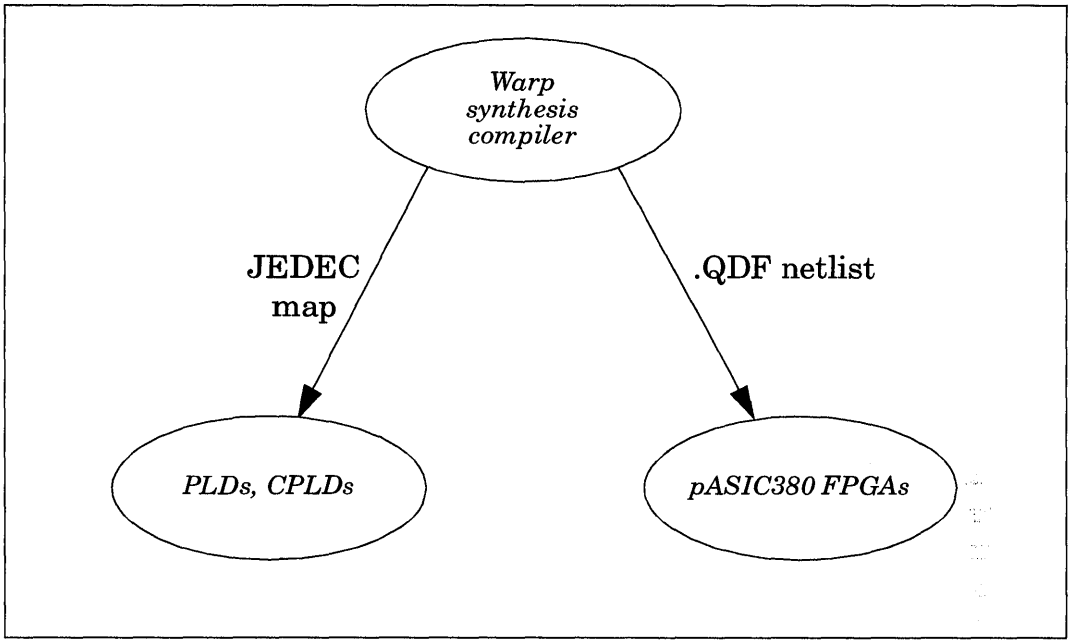


Figure 1-1. The Warp synthesis compiler produces JEDEC maps for programming PLDs and CPLDs, and .QDF netlists for placing and routing pASIC380 FPGAs.

1.2. *Warp* Synthesis Compiler Capabilities

The *Warp* synthesis compiler utilizes a VHDL subset geared for synthesis of designs onto PLDs, CPLDs, and pASIC380 FPGAs.

Some highlights of the *Warp* synthesis compiler:

- VHDL is an open, non-proprietary language, and a *de facto* standard for describing electronic systems. It is mandated for use by the DoD, and supported by every major CAE vendor.
- VHDL allows designers to describe designs at different levels of abstraction. Designs can be entered as descriptions of behavior (high level of abstraction), as state tables and boolean entry descriptions (intermediate level), or at gate level (low-level of abstraction).
- *Warp* supports numerous data types, including enumerated types, integer types, and user-defined types, among others.
- *Warp* supports the for...generate loop construct for structural descriptions, providing a powerful, efficient facility for describing replication in low-level designs.
- *Warp* incorporates state-of-the-art optimization and reduction algorithms, including automatic selection of optimal flip-flop type (D type/T type).
- While users can specify the signal-to-pin mapping for their designs, *Warp* can also map signals from the design to pins on the target device automatically, making it easy to re-target designs from one device to another.
- *Warp* can automatically assign state encodings (e.g., gray code, one-hot, binary) for efficient use of device resources.

- *Warp* supports all Cypress PLD, CPLD, and FPGA/pASIC families, including the FLASH370, pASIC380, and C34X (compatible with the MAX5000 series) families.

FPGA support is available in Warp2+ and Warp3 only.

1.3. About This Manual

This section describes the contents of the remainder of this manual.

Section 2 of this manual describes *Warp*'s command line interface.

Section 3 describes *Warp*'s graphical user interface.

Section 4 describes the fundamental elements of VHDL, as implemented in *Warp*.

Section 5 provides a comprehensive reference to the VHDL statements and other constructs implemented in *Warp*.

Section 6 describes how *Warp* synthesizes various VHDL constructs.

Appendix A lists and provides brief explanations for the various error messages that *Warp* produces.

Appendix B is a glossary of *Warp*/VHDL terminology.

Appendix C provides a Backus-Naur Form (BNF) listing of *Warp*'s implementation of VHDL.

Appendix D is a bibliography of books and articles about VHDL and design synthesis.

Chapter 2

Using *Warp* from a Command Line

Using *Warp* from a Command Line

2.1. *Warp* Command Syntax

On Sun workstations, you can run *Warp* by typing the warp command from a shell window. On IBM PC's and compatibles running Windows, you can run *Warp* by typing the **warp** command in the "Command Line" box in response to the File/Run menu item in the Windows File Manager. On IBM PC's and compatibles running DOS, you can run *Warp* by typing the **warp** command at the command line prompt, or by including the **warp** command in a batch file and typing the name of the batch file at the command line prompt.

This chapter documents the **warp** command and its options.

Syntax

```
warp [filename]
     [-d device]
     [-b filename]
     [-a[library] filename[, filename...]]
     [-e max-#-of-errors]
     [-f{d | f | o | p | t}]
     [-h]
     [-l[library]]
     [-o 0|1|2]
     [-p package-name]
     [-q]
     [-r[library] filename]
     [-s[library] path]
     [-w max-#-of-warnings]
     [-xor2]
```

The **warp** command runs the *Warp* synthesis compiler.

Typing **warp** with no arguments brings up a help screen showing the available options for the warp command.

Typing **warp** followed by the name of a file compiles the named file and, if compilation is successful, synthesizes the design.

Note that, when using the *warp* command line interface on a Sun workstation, the command and its options are case-sensitive. On an IBM PC or compatible computer, they aren't.

2.2. *Warp* Command Options

Numerous options control the execution of the warp command from the command line. This section documents warp's command-line options.

The **warp** command options you will use most frequently are **-d**, **-b**, and **-a**. These three options are described first, followed by the remaining options in alphabetical order.

Note that, when using the *warp* command line interface on a Sun workstation, the command and its options are case-sensitive. On an IBM PC or compatible computer, they aren't.

The -d Option

The **-d** option specifies a target device for synthesis. If this option is not included on the command line, *Warp* targets devices in the following order:

1. it searches for a part_name attribute in the file being compiled/synthesized, and targets the device specified by that attribute. If no part_name attribute is found, then
2. it searches for an architecture that identifies a device as a top-level entity, and targets that device. If no such architecture is found, then
3. it uses the last device targeted by a previous *Warp* run from the same directory.
4. otherwise, an error is returned.

Example:

```
warp -d c371 myfile.vhd
```

The command above compiles and synthesizes a source file named myfile.vhd, targeting a CY7C371.

Allowable arguments for the **-d** option consist of the letter "C" followed by a part identifier, usually consisting of the three rightmost digits of the part's name (e.g., C335, C371, etc.) Notable exceptions to this rule are the arguments C22V10 and C22VP10, which target a PAL22V10 and PAL22VP10, respectively.

Each time the **-d** option is used in a warp command, it creates a subdirectory within the current directory in which compilation results are stored, if such a subdirectory does not already exist. The name of this directory consists of the letters "lc" followed by the part identifier used in the argument to the **-d** option (e.g., an argument of "C371" creates an "LC371" subdirectory, etc.). This subdirectory becomes the work library for the *Warp* run.

In addition, the **-d** option causes *Warp* to look for a library in a subdirectory of the warp directory (default: /warp). This subdirectory is named /lib/lc*device-name*. This library has the same root name as the **-d** option's argument, followed by the extension .VHD (e.g., the path to the C22V10 library is /WARP/LIB/LC22V10/C22V10.VHD).

When *Warp* interprets the **-d** option on the command line, it creates a subdirectory for the specified device if one does not already exist within the current directory, compiles the appropriate library file(s) for the device within the new subdirectory, assigns the path of the new subdirectory to the "work" logical name, and writes or revises the WARP.RC file (if necessary) to reflect the new path to the work library.

The -b Option

The **-b** option specifies a VHDL source file to compile. All packages referenced within the file are also compiled. If compilation is successful, this option causes *Warp* to synthesize the design, producing a JEDEC file.

The **-b** option assumes that the file to be compiled has an extension of .VHD, unless a different extension is specified on the command line.

The **-b** option is assumed if a filename is included on the command line and no other option is present.

Example:

```
warp myfile.vhd
```

The command above compiles a file named myfile.vhd. If compilation is successful, the file will be synthesized, producing an output file called myfile.jed.

The -a Option

The **-a** option analyzes one or more files and adds them to the work library or to a different, user-specified library. To specify a library other than "work", follow the **-a** option immediately (i.e., without an intervening space) by the name of the library. This library is referenced by its logical name, and must have been previously created by means of the **-l** option.

The **-a** option assumes that the file to be compiled has an extension of .VHD, unless a different extension is specified on the command line.

Example:

```
warp -a file1 file2 -b myfile.vhd
```

The command above compiles two files named file1.vhd and file2.vhd. If those two files compile successfully, *Warp* will then compile myfile.vhd. If compilation is successful, myfile.vhd will be synthesized, producing an output file called myfile.jed.

```
warp -amylib file1 file2 -b myfile.vhd
```

This command is identical to the previous, except that results from the compilation of file1.vhd and file2.vhd will be written into a subdirectory called mylib.

For more information about libraries and their use, see Section 5.13, "LIBRARY" and Section 5.22, "USE".

The -e Option

The -e option specifies the maximum number of non-fatal errors that can occur on a single *Warp* run before *Warp* quits.

The -f Option

The **-f** option enables certain global fitter options. **-f** must be followed (without an intervening space) by one of the arguments 'd', 't', 'o', 'f', 'h', 'l', or 'p'. (Multiple uses of the **-f** option are allowed on a single line.) Arguments 'd', 't', and 'o' are mutually exclusive. The meanings of these arguments are as follows:

- 'd' forces registered equations to a 'D' registered form (i.e., forces use of D-type flip-flops). For some devices, this may result in a non-minimal solution for an output register. This is the default for the **-f** option.
- 't' forces the use of T-type flip-flops for registered equations. For some devices, this may result in a non-minimal solution for an output register. If the target PLD does not support a physical 'T' flip-flop, the equation is converted to a 'D' registered form using the formula 'D = T XOR Q'. Use of this option may lead to fitter errors if the target device cannot support either a physical 'T' flip-flop or product-term programmable XOR function.
- 'o' tells the fitter to optimize the *Warp*-generated design to either D-type or T-type flip-flops, whichever produces the

smaller equation set. If the target PLD does not support a physical 'T' flip-flop, the equation is converted to a 'D' registered form using the formula 'D = T XOR Q'.

- 'f' tells the fitter to ignore any user-specified pin assignments and assign pins itself instead.

  Note: when you run *Warp* using the "-ff" option, *Warp* always assign pins itself, overriding any pin assignments made in the source file (e.g., by the use of the "pin_numbers" attribute).

- 'h' writes out the JEDEC output file in hexadecimal format. This can effect a considerable (i.e., 4X) savings in storage space for JEDEC files.

- 'l' allows the fitter to perform three-level logic factoring instead of just two-level (sum of products) factoring. This is recommended for the 7c34x device family. For pASIC architectures, it allows common sub-expressions between different signals to be shared, thereby reducing fanout. This option is recommended for VHDL designs targeting pASIC devices, and is only applicable for non-frag based signals. (See also the descriptions for the synthesis_off and dont_touch attributes, elsewhere in this chapter.)

- 'p' logically reduces output signals via Espresso during the optimization process. This option selects the output polarity that produces the minimum number of product terms.

The 'f' and 'p' arguments can be used in conjunction with the 'd', 'o', or 't' arguments, e.g., "-fo -ff -fp".

Example:

```
warp -b myfile.vhd -fo -ff -fp
```

The command above compiles and synthesizes a file named myfile.vhd. During synthesis, *Warp* is directed to optimize the design to use either D- or T-type flip-flops ("-fo"), ignore any pin assignments in the file and assign pins itself ("-ff"), and optimize output polarity ("-fp").

The -h Option

The **-h** ("help") option lists the available options, their syntax, and meanings. Executing *warp* with this option is the same as executing *warp* with no command line options.

Example:

```
warp -h
```

The command above prints the `warp` command's available options, syntax, and meanings.

The -l Option

The **-l** option lists the contents of the "work" library (default), or of any user-specified library. To specify a library other than "work", follow the **-l** option immediately (i.e., without an intervening space) by the name of the library. The listing of library contents includes the type and name of each design unit and the name of the file in which the unit is found.

Example:

```
warp -l
```

The command above lists the contents of the "work" library.

```
warp -lmylib
```

The command above lists the contents of library "mylib".

The -o Option

The **-o** option specifies the level of optimization to perform on *Warp* output:

- An argument of '0' provides minimal optimization.
- An argument of '1' (default) provides more optimization.
- An argument of '2' runs the industry-standard Espresso optimizer, giving the most thorough optimization possible. However, running *Warp* with Espresso takes much more time than running it without Espresso, so use the **-o** option only when you think it is necessary to fit the design onto the target PLD. Running *Warp* with this option is highly recommended, however, when targeting pASICs.

Example:

```
warp -o2 myfile.vhd
```

The command compiles and synthesizes a file named myfile.vhd. *Warp* is directed to use the highest level of optimization possible.

The -q Option

The **-q** ("quiet") option suppresses the printing of status messages during compilation. This leads to a less cluttered screen when compilation and synthesis are finished.

Example:

```
warp -q myfile.vhd
```

This command compiles and synthesizes a file named myfile.vhd, quietly.

The -r Option

The **-r** option removes design units compiled from one or more files from the "work" library, or from a user-specified library. To specify a library other than "work", follow the **-r** option immediately (i.e., without an intervening space) by the name of the library.

Example:

```
warp -r file1.vhd
```

This command removes the design units compiled from file file1.vhd from the work library.

```
warp -rmylib file1.vhd
```

This command removes the design units compiled from file file1.vhd from library mylib.

The -s Option

The **-s** option pairs a library name with a path. The name of the library and its path are written into the warp.rc file in the current directory. To use a library other than "work" with a VHDL description, follow the **-s** option immediately (i.e., without an intervening space) by the name of the library.

Example:

```
warp -smylib /usr/myname/mydir
```

This command pairs the library name "mylib" with path "/usr/myname/mydir".

The -w Option

The -w option specifies the maximum number of warnings that can appear as a result of a single *Warp* run before *Warp* quits.

The -xor2 Option

The **-xor2** option passes along any XOR operators found in the design to the fitter for PLD's/CPLD's, and to SpDE for pASIC's. If this option is disabled, any XOR operators contained within the design are flattened, and it would be up to the fitter or to SpDE to detect the XOR contained within the equation. Always use the -xor2 option when targeting pASICs.

Example:

```
warp -d c382a -xor2 myfile.vhd
```

This command compiles and synthesizes a file named myfile.vhd, preserving any XOR operators in the input file.

2.3. *Warp* Output

A *warp* run produces numerous output files, of which the following are important to the user: .JED files for targeting PLDs, .QDF files for targeting pASIC380 FPGAs, and .RPT files to report on compilation results.

A successful **warp** run produces two output files in the current directory:

- *filename*.**JED**

  or

- *filename*.**QDF**

  and

- *filename*.**RPT**

The .JED file is a fuse map that can be used by a PLD programmer. It is also used as input to the Nova simulator.

The .QDF file, which can be produced only when targeting pASIC380 FPGAs, can be used as input to the SpDE place and route tool.

The .RPT file is an ASCII text file that contains fitter statistics and informational, warning, and error messages from the *Warp* run, as well as pinout information for the synthesized design.

FPGAs are only available with Warp2+ and Warp3.

Chapter 3

Using Warp with Galaxy

Using *Warp* with Galaxy

3.1. Introduction

Galaxy is Cypress Semiconductor Corporation's graphical user interface (GUI) for its *Warp* synthesis compiler.

Warp is Cypress Semiconductor Corporation's name for its VHDL compilation and synthesis software. *Warp* accepts VHDL source files as input. The primary output of a *Warp* run is a .JED or a .QDF file. The .JED file can be used as input to a PLD programmer. The .QDF file can be used as input to SpDE for the place and route of pASICs. *Warp* can compile objects (e.g., components, type and function declarations, etc.) into a VHDL library, which can be used by numerous designs.

Galaxy is the name of the graphical user interface for *Warp*. Galaxy gives you a graphical way to:

- select VHDL source files for compilation or synthesis;
- choose whether to compile selected files into a library, or synthesize them to program an actual device;
- select a target device for synthesis;
- choose synthesis options, such as the type of flip-flops used, degree of logic optimization, etc.

This chapter tells you how to use Galaxy to run *Warp*. It assumes that you are already familiar with common user interface operations for your platform, such as the use of scroll bars, menu buttons, opening and closing windows, etc.

3.2. Starting Galaxy

To start Galaxy on a Sun workstation, type "galaxy" on the command line of a shell window. To start Galaxy while running Windows on an IBM PC or compatible computer, double-click on the Galaxy icon in the Cypress window. This brings up the Galaxy window.

The Galaxy window (Figure 3-1) consists of a menu bar with four items across the top, a scroll bar along the right side, and a large, mostly blank, text area.

Menu Bar

The four menu items are File, Edit, Tools, and Font. Under each of these items are menus for selecting related actions. The menus are ordered so that the most common operation is at the top. The contents of each menu are described in greater detail later in this manual.

Text Area

As compilation and synthesis proceed, *Warp* writes various messages into the text area. As the text area fills up, new messages are written at the bottom of the text area, while old text scrolls off the top. You can view this information using the scroll bar located at the right of the text area.

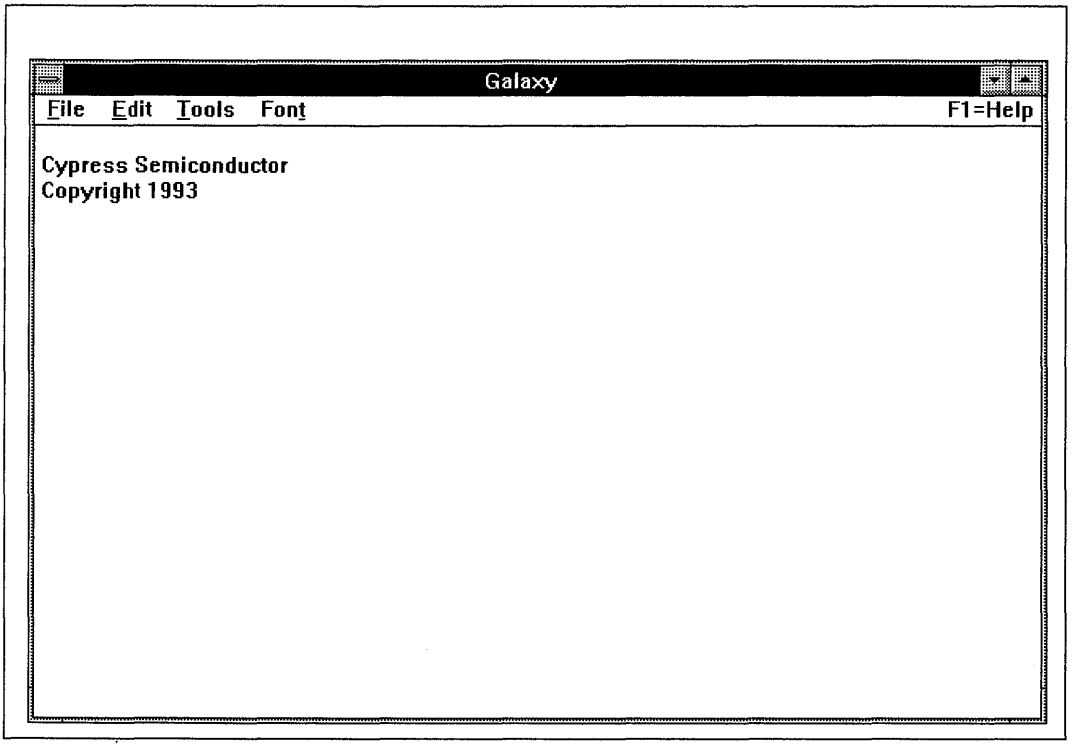


Figure 3-1. Galaxy Window.

3.3. Galaxy Window Menu Items

The menu items in the Galaxy window bring up menus that control *Warp* operation and control the appearance of messages and output from *Warp*.

Table 3-1 summarizes the operation of the menu items available in the Galaxy dialog box.

Table 3-1.
Galaxy Dialog Box Menu Items

Menu Item	Submenu Item	Meaning
File	Open...	brings up the Warp VHDL dialog box.
	Save Transcript File...	brings up dialog box to save contents of text area to a file.
	Work Area List	lists contents of work library (for the last device processed) in the text area.
	Work Area Remove...	brings up dialog box to specify file whose contents will be removed from work library.
	Exit	exits Galaxy.
	About...	displays software version and copyright message
Edit	Copy	copies selected text from text area to clipboard.
	Clear	clears text area.

Table 3-1. (Continued)
Galaxy Dialog Box Menu Items

Menu Item	Submenu Item	Meaning
Tools	Run Warp Menu...	brings up Warp VHDL dialog box.
	Nova Functional Simulator	runs Nova functional simulator.
Font	Courier	specifies Courier font for text area.
	Helvetica	specifies Helvetica font for text area.
	Times	specifies Times font for text area.
	System	specifies System font for text area.
	Fixed	specifies Fixed font for text area.
Style	Normal	specifies Normal type for text area.
	Bold	specifies Boldface type for text area.
	Italic	specifies Italic type for text area.
	8 point	specifies 8 point type for text area.
	10 point	specifies10 point type for text area.
	12 point	specifies12 point type for text area.
	14 point	specifies14 point type for text area.
	18 point	specifies18 point type for text area.
	24 point	specifies24 point type for text area.

3.3. Galaxy Window Menu Items

3.3.1. File Menu

The File menu contains items to bring up the Warp VHDL dialog box, save a transcript of a Galaxy session, view the contents of the work library, remove objects from the work library, exit Galaxy, and display the *Warp* version number.

The File menu in the Galaxy dialog box (Figure 3-2) contains the following items:

- **Open...** brings up the Warp VHDL dialog box.
- **Save Transcript File...** brings up a dialog box that lets you specify a file in which to save the current contents of the text area.
- **Work Area List** writes the contents of the work library (for the last device targeted by a *Warp* run) to the text area.
- **Work Area Remove...** brings up a dialog box that allows you to specify a file whose contents will be removed from the work library.
- **Exit** exits the Galaxy user interface.
- **About** displays the *Warp* version number and a copyright message.

Each of these items is discussed in greater detail on the following pages.

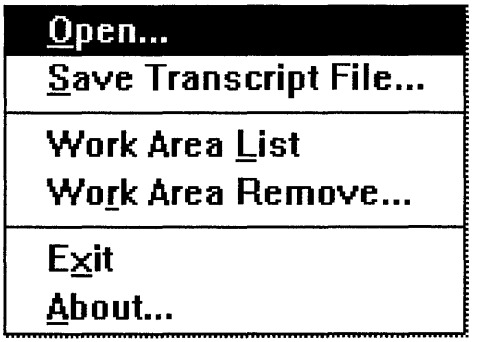


Figure 3-2. File Menu.

3.3. Galaxy Window Menu Items

3.3.1. File Menu

3.3.1.1. Open...

To select *Warp* input files, or to specify options for a *Warp* run, bring up the *Warp* VHDL Files dialog box, using **File/Open...** or **Tools/Run Warp Menu...**.

The *Warp* VHDL Files dialog box (Figure 3-3) is the "command center" for compilation and synthesis. To bring up the *Warp* VHDL Files dialog box, select **Open...** from the File menu or **Run Warp Menu...** from the Tools menu.

From the *Warp* VHDL Files dialog box, you can:

- change the current directory;
- selected files to be compiled or synthesized;
- select whether to compile-only or compile and synthesize selected files;
- bring up an editor to modify VHDL source files;
- bring up the *Warp* Options dialog box to specify further *Warp* operation details.

The elements of the *Warp* VHDL Files dialog box are discussed in greater detail in Section 3.4, "The Warp VHDL Files Dialog Box."

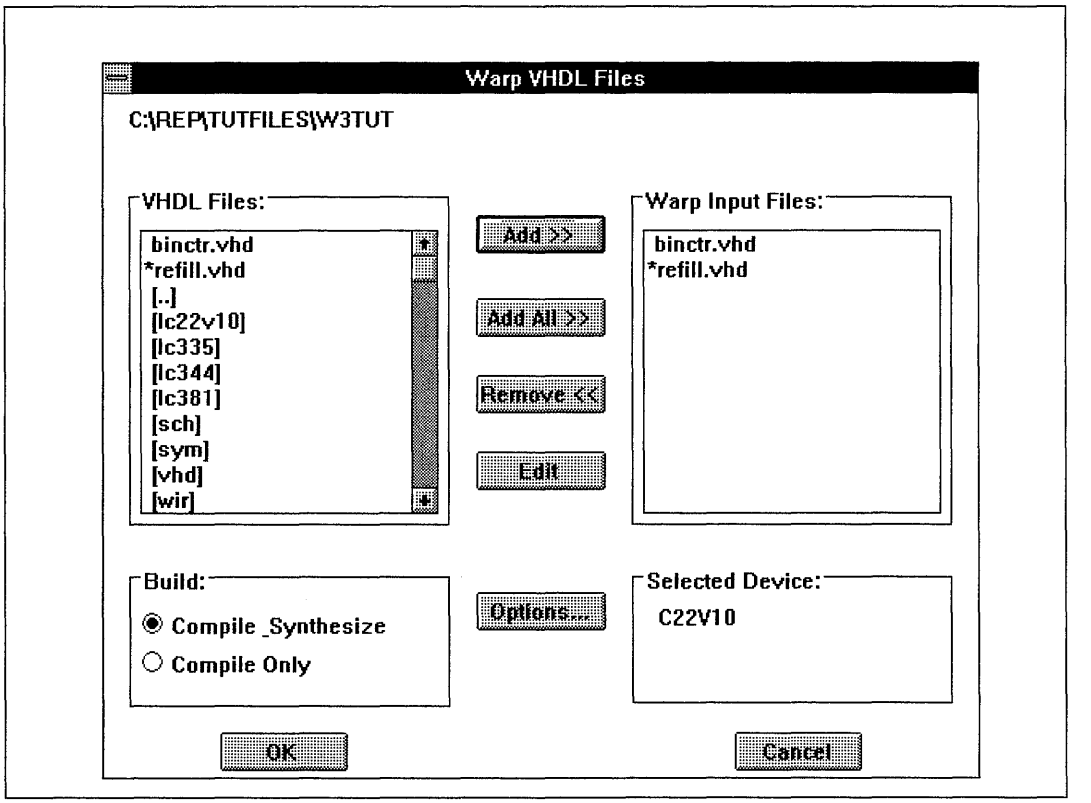


Figure 3-3. Warp VHDL Files Dialog Box.

3.3. Galaxy Window Menu Items

3.3.1. File Menu

3.3.1.2. Save Transcript File...

Save Transcript File... in the File menu brings up a dialog box that lets you specify a file in which to save the current contents of the text area.

Selecting **Save Transcript File...** from the File menu brings up the Save Transcript to File... dialog box (Figure 3-4).

Specifying a File name and selecting Save writes the current contents of the text area to the specified file.

The default File name is warp.log. If you want to use a different name for the log file, enter it on the line labeled File name.

All transcript files are given a .log extension. If the name you type doesn't have a .log extension, .log will be added to the name when the file is written. If the name you type includes a .log extension, nothing more will be added to the file name when the file is written.

The Files window on the dialog box lists any .log files already in your current directory.

The line labeled Path: displays the full path of the directory you are working in.

The Directories window lists the sub-directories of the current directory. To change directories, double click on one of the items in the Directories window. The Path: line updates automatically as you go from one directory to another.

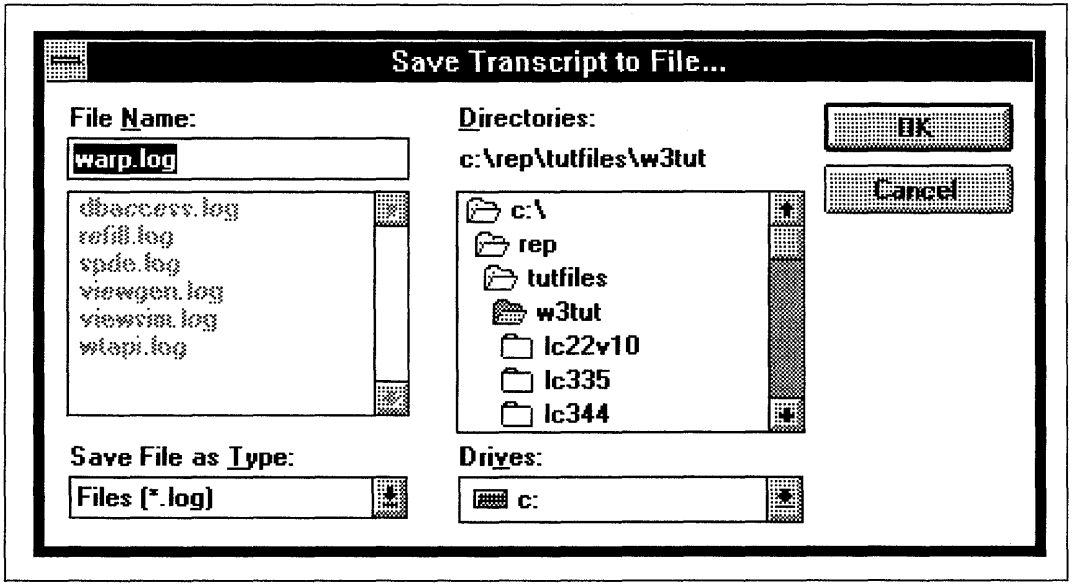


Figure 3-4. Save Transcript to File... Dialog Box.

Once you have specified the transcript file name and its directory, selecting Save closes down the dialog box and writes the contents of the text area to the specified file.

Selecting Cancel closes the dialog box without saving the contents of the text area to the .log file.

3.3. Galaxy Window Menu Items

3.3.1. File Menu

3.3.1.3. Work Area List

Work Area List in the File menu writes the contents of the current work library to the text area.

In VHDL, a library is a collection of previously compiled design elements (packages, components, entities, architectures) that can be referenced by other VHDL descriptions.

In *Warp*, a library is implemented as a directory, containing one or more VHDL files and an index to the design elements they contain.

Work Area List displays the contents of the current work library in the text area. Figure 3-5 shows an example of library contents. First, the name of the library and the directory it resides in is identified. Then, each design element contained in the library is listed, along with the name of its source file.

```
VHDL parser (C:\WARP\bin\vhdlfe.exe V3.5 IR x57)
Contents of library 'work' with directory 'lc22v10':
Unit                                    Design File
—                                       ———
Package 'binctr_pkg'                        binctr.vhd
Entity 'binctr'                         binctr.vhd
Architecture 'archbinctr' of 'binctr'       binctr.vhd
———————————————
WARP done.
```

Figure 3-5. Sample Work Area List Output.

3.3. Galaxy Window Menu Items

3.3.1. File Menu

3.3.1.4. Work Area Remove...

Work Area Remove... in the File menu brings up a dialog box that lets you specify an element to remove from the work library.

Work Area Remove... removes the file containing a specified element from the work library. Selecting **Work Area Remove...** brings up the Work Area Remove dialog box (Figure 3-6).

The dialog box prompts you for the name of an object to remove from the work library. When you type the name of an object and click on "OK," the source file of that object is removed from the library. This means that the named object AND ALL OTHER OBJECTS FROM THE SAME SOURCE FILE can no longer be referenced by any design until the objects are re-compiled and added to the library again.

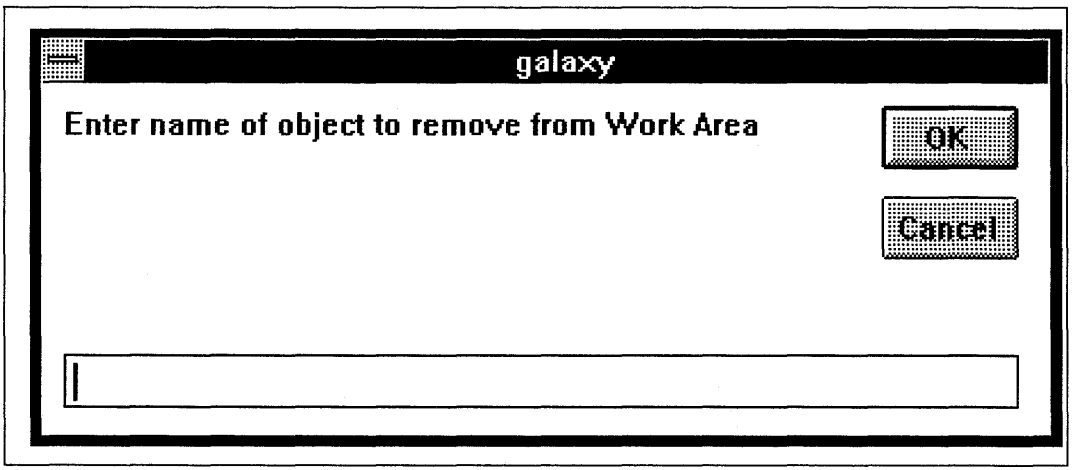


Figure 3-6. Work Area Remove Dialog Box.

3.3. Galaxy Window Menu Items

3.3.1. File Menu

3.3.1.5. Exit

Exit terminates the Galaxy session.

Exit closes the Galaxy window and returns you to your regularly scheduled programming.

3.3. Galaxy Window Menu Items

3.3.1. File Menu

3.3.1.6. About...

About displays the *Warp* version number and a Cypress copyright message.

Figure 3-7 shows the **About...** dialog box. Clicking on "OK" closes the dialog box. Clicking on "Help" brings up help for running *Warp*.

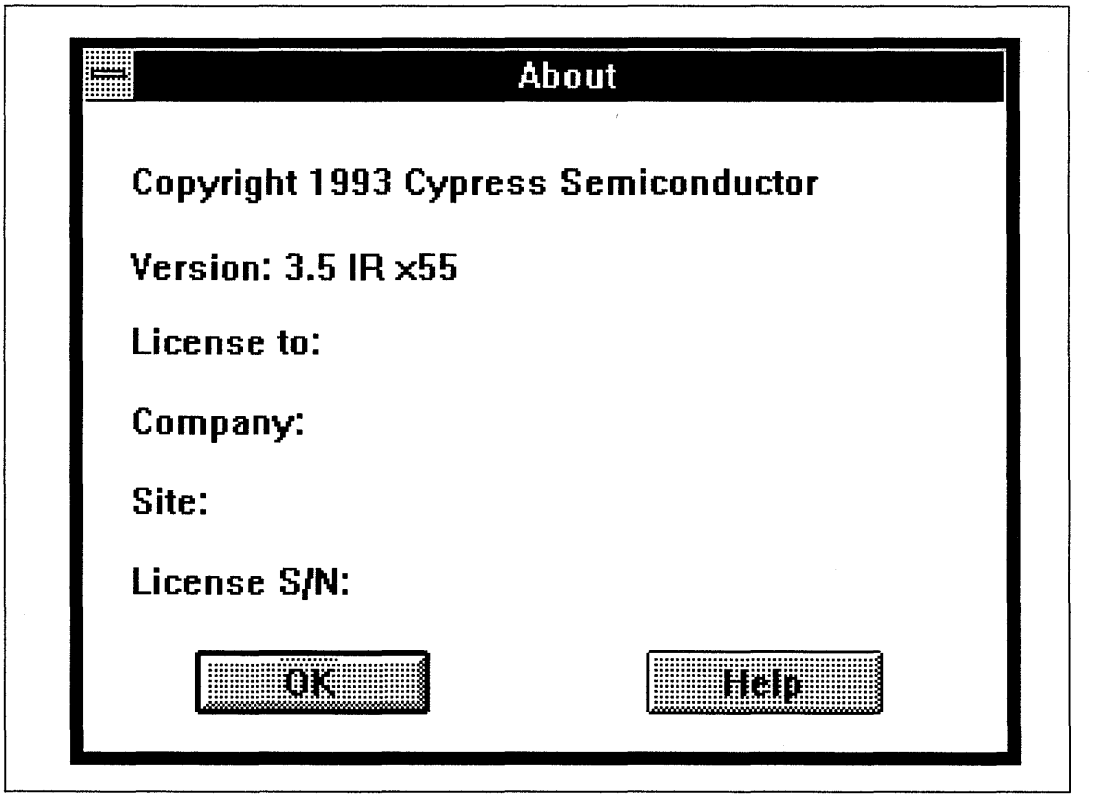


Figure 3-7. About... Dialog Box.

3.3. Galaxy Window Menu Items

3.3.2. Edit Menu

The Edit Menu contains submenus for copying or clearing information displayed in the text area.

Copy copies selected text from the text area to a clipboard. If no text is selected in the text area, the entire contents are copied to the clipboard. If you have an application that reads from a clipboard, this may be a handy option.

Clear erases all text from the text area.

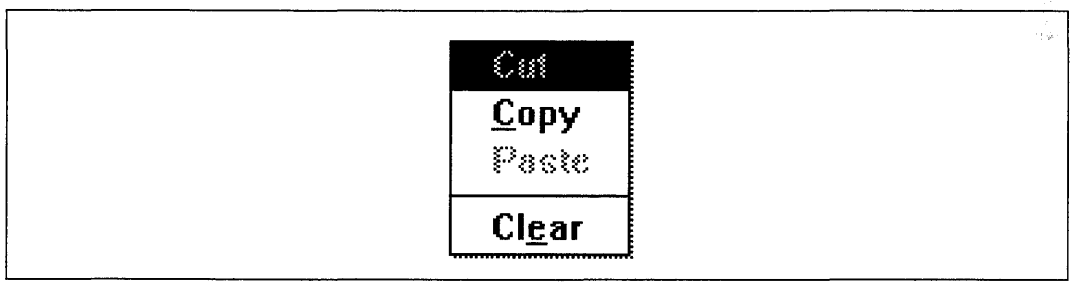


Figure 3-8. Edit Menu.

3.3. Galaxy Window Menu Items

3.3.3. Tools

The Tools Menu contains three items, **Run Warp Menu...**, **Nova Functional Simulator**, and **Tool Versions...**.

Run Warp Menu... brings up the Warp VHDL Files dialog box, which is discussed in greater detail in Section 3.4, "The Warp VHDL Files Dialog Box."

Nova Functional Simulator brings up the Nova simulator, which is discussed in greater detail in Section 4, "Using Nova."

Tool Versions... brings up a scrollable list showing the version numbers of the programs in your *Warp* tool set.

Figure 3-9. Tools Menu.

3.3. Galaxy Window Menu Items

3.3.4. Font

The Font menu controls the font, type size, and appearance of the contents of the text area.

The Font menu (Figure 3-10) contains various items that allow you to control the appearance (font, type size, style) of text in the text area.

When you select a different font, style, or type size, the contents of the entire text area are updated to reflect the change. Future input to the text area is written in the selected font/style/size.

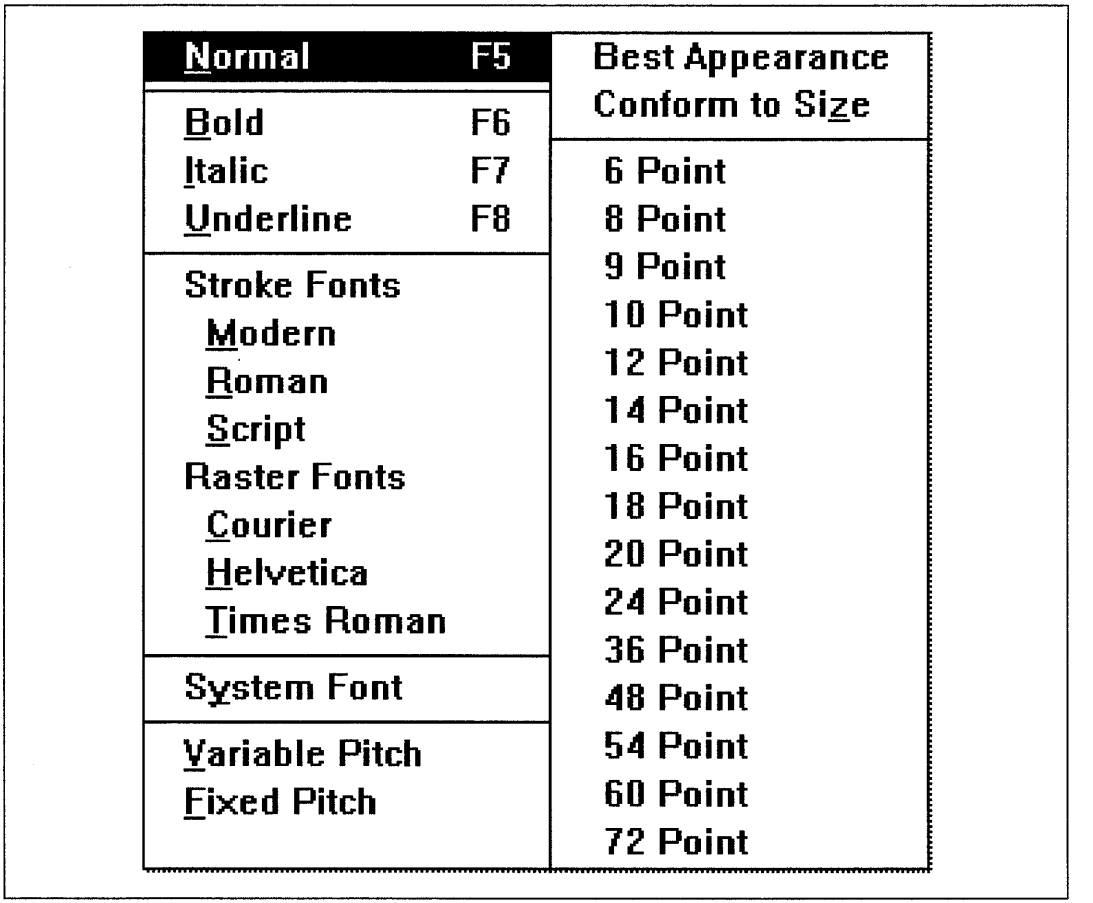


Figure 3-10. Font Menu

3.4. The *Warp* VHDL Files Dialog Box

The *Warp* VHDL Files dialog box is the "command center" for compilation and synthesis. To bring up the *Warp* VHDL Files dialog box, select **Open...** from the File menu or **Run Warp Menu...** from the Tools menu.

From the Warp VHDL Files dialog box (Figure 3-11), you can:

- change the current project (i.e., the directory where *Warp* expects to find VHDL source files, and where output files and subdirectories are written);
- select files to be compiled or synthesized;
- bring up VHDL source files for editing;
- select whether to compile-only or compile and synthesize selected files;
- bring up the Warp Options dialog box to specify further Warp operation details.

Various elements of the Warp VHDL Files dialog box are discussed in greater detail on the next few pages.

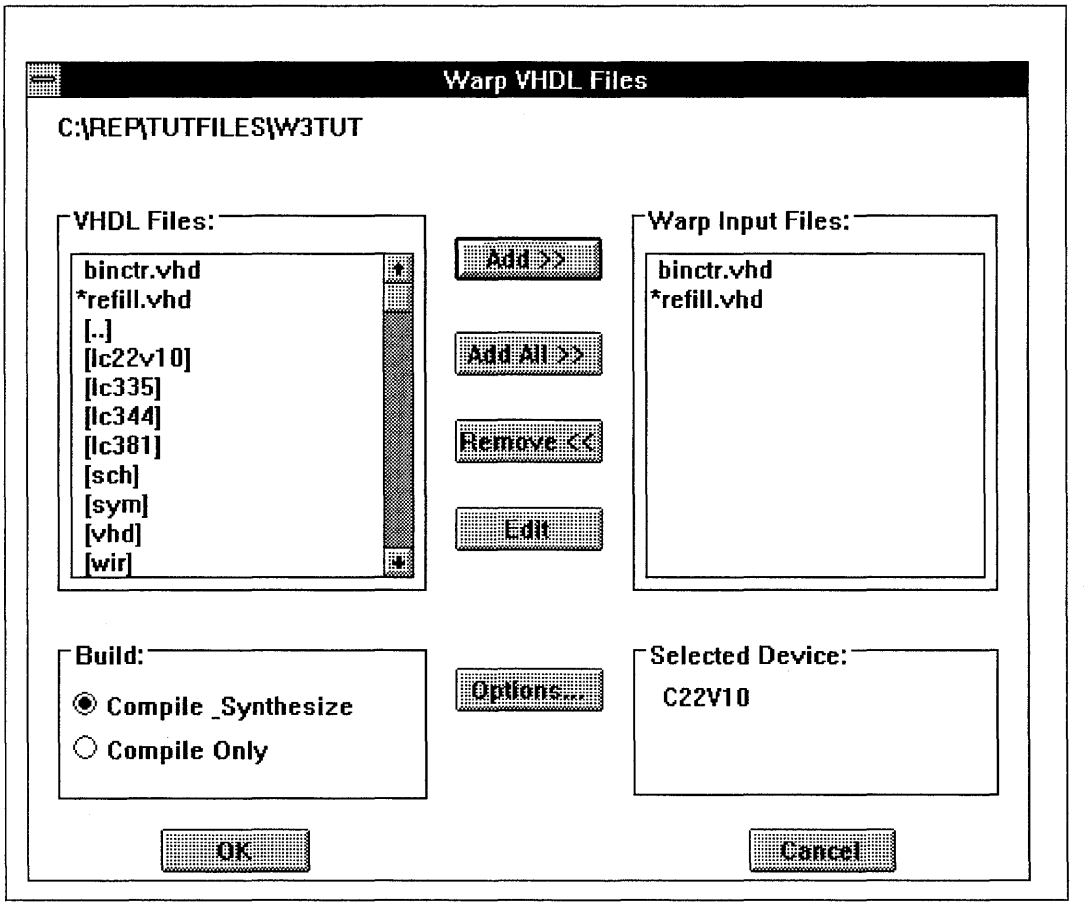


Figure 3-11. Warp VHDL Files Dialog Box.

3.4. The Warp VHDL Files Dialog Box

3.4.1. Selecting Files to Compile, Synthesize, or Edit

You select files for compilation and synthesis from the Warp VHDL Files dialog box, using the mouse, the "VHDL Files:" and "Warp Input Files" windows, and the "Add," "Add All," "Remove," and "Edit" buttons.

The VHDL Files window displays all subdirectories and all files in the current directory with a .VHD extension. Subdirectories are displayed in this window with a trailing slash ("/") character on Sun workstations. On IBM PC or compatible computers, they are surrounded by square brackets ("["..."]").

The Warp Input window lists the names of the files that will be compiled and/or synthesized by the next *Warp* run.

To select a file for compilation or synthesis, highlight the file by clicking on it in the VHDL Files window, then click on the "Add" button. The file name appears in the Warp Input Files window.

To select all .VHD files in the current directory for compilation or synthesis, click on the "Add All" button. All .VHD files in the current directory appear in the Warp Input Files window.

To remove a file from the Warp Input Files window, highlight the file in the window, then click on the "Remove" button. The file's name is removed from the Warp Input Files window. Nothing happens to the actual file, however. It is still available for selection from the VHDL Files window.

To select a VHDL source file for editing, highlight the file by clicking on it in the VHDL Files window, then click on the "Edit" button. The file appears in a text window for editing. By default, the editor used is "textedit" on Sun workstations, "notepad" on PC's and compatibles. You can change this by setting the EDITOR environment variable to the editor you want.

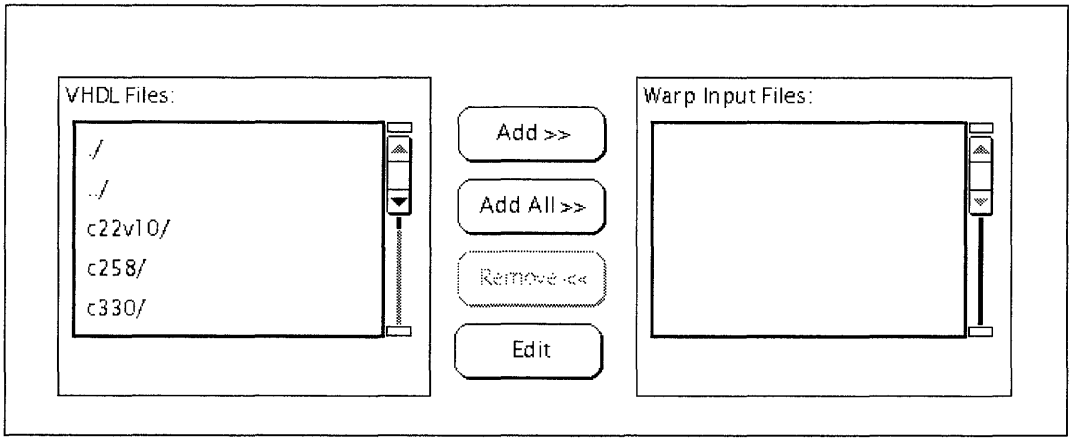


Figure 3-12. VHDL Files Window in the Warp VHDL Files Dialog Box.

To view files in a sub-directory, double-click on the sub-directory in the VHDL Files window, or highlight the sub-directory and click on the "Add" button.

To move to the next higher directory in the directory structure, double-click on the "../" or "[..]" directory in the VHDL Files window, or highlight the directory and click on the "Add" button.

Clicking "OK" runs *Warp* on the files appearing in the "Warp Input Files" window. Clicking "Cancel" exits the Warp VHDL Files dialog box and returns to the Galaxy dialog box.

Important Note About File Order

When synthesizing a design, *Warp* assumes that the last file in the Warp Input Files window is the "top-level" file, i.e., any components or functions referenced in this file have been defined in files earlier in the list.

3.4. The Warp VHDL Files Dialog Box

3.4.2. Compiling or Synthesizing

Warp either compiles VHDL files into a library, or synthesizes a mapping file for use in programming the target device. You control this *Warp* functionality by means of the "Build:" window in the Warp VHDL Files dialog box.

When *Warp* compiles a VHDL file, it checks the file for syntactical correctness, then adds any objects defined by the file to a library for the target device. This library is implemented as a sub-directory of the current directory. The sub-directory is given the name of the target device, and contains an index file and a copy of the compiled file(s).

When *Warp* synthesizes a VHDL file, it produces the following files in the current directory, among others:

- a JEDEC file (.JED extension) for PLDs and PROMs. This file can be used as input to Nova, or as input to a device programmer.
- a QDIF file (.QDF extension) for pASICs. This file can be used as input to the SpDE Place and Route tool.

To compile the files listed in the Warp Input Files window, select "Compile Only" in the "Build:" window (Figure 3-13). Be sure you select a target device if you are compiling files in this project directory for the first time.

To compile and synthesize the files in the Warp Input Files window, select "Compile & Synthesize" in the "Build:" Window.

Clicking on "OK" in the Warp VHDL Files window runs Warp, whether compiling and synthesizing or compiling only. The default is to compile and synthesize.

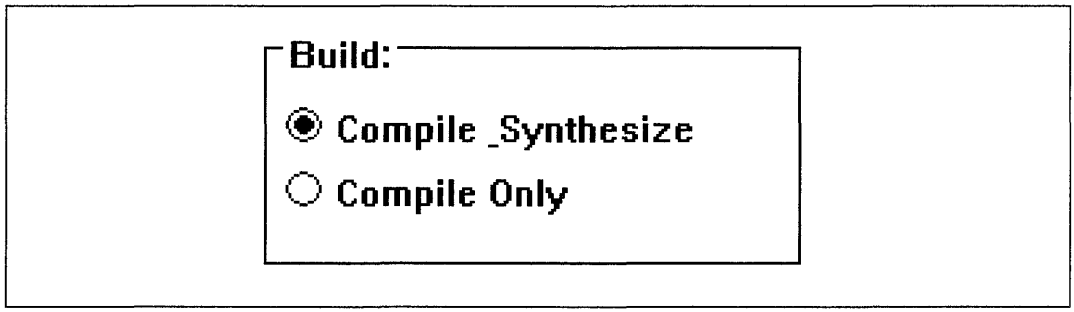


Figure 3-13. "Build" Window in the Warp VHDL Files Dialog Box.

3.4. The Warp VHDL Files Dialog Box

3.4.3. Specifying *Warp* Options

The Warp Option dialog box lets you specify the target device, the level of optimization, the output files created, the type of flip-flops used in synthesis, and other parameters for a *Warp* run.

To bring up the Warp Option dialog box (Figure 3-14), click on the "Options..." button in the Warp VHDL Files dialog box.

The Warp Option dialog box contains the following windows:

- **Devices:** specifies the target device.
- **Package:** specifies the package used by the target device.
- **Optimize:** specifies the degree of optimization used during compilation and synthesis.
- **Output:** specifies which of several possible output files are to be produced.
- **Fitter:** specifies whether D- or T-type flip-flops are used in synthesis; whether to force polarity optimization; whether to assign pins in the fitter; and whether to force logic factoring.
- **Run Options:** specifies whether *Warp* should run in "quiet" mode.

Each of these windows is discussed in greater detail on the following pages.

Clicking on OK returns you to the Warp VHDL Files dialog box, keeping any changes you have made to *Warp* options.

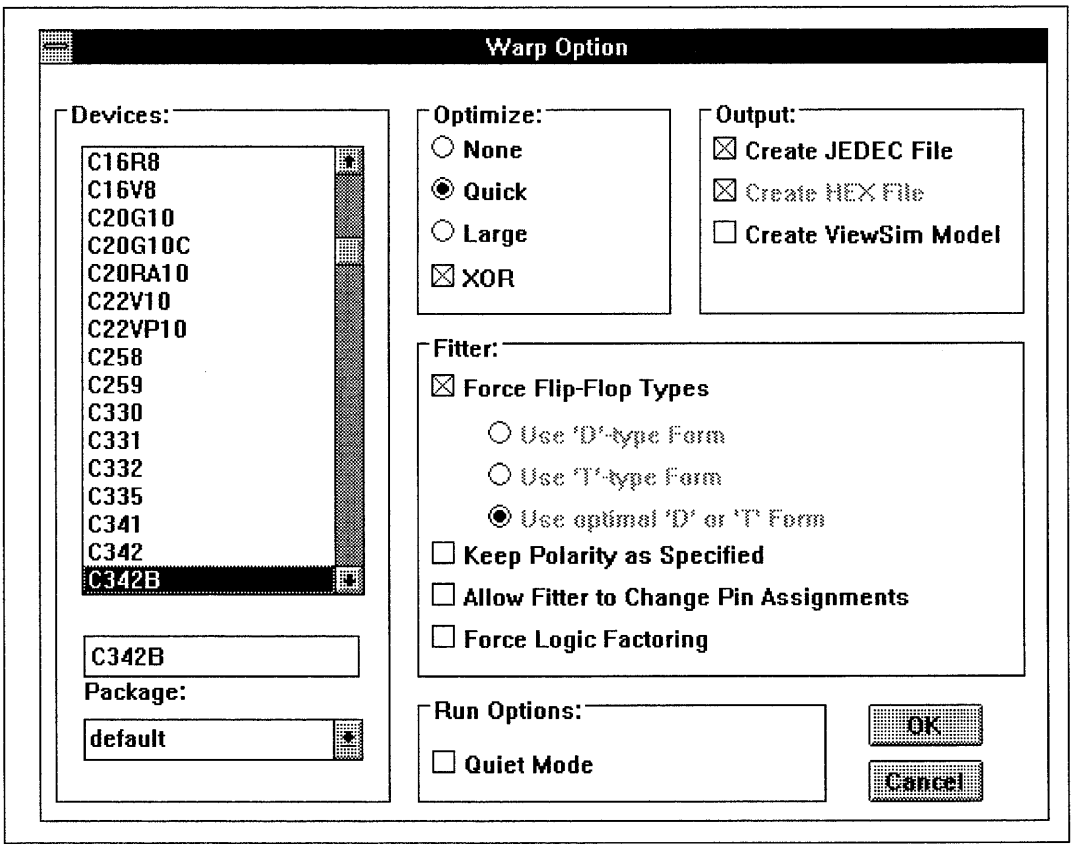


Figure 3-14. Warp Option Dialog Box.

Clicking on Cancel returns you to the Warp VHDL Files dialog box, discarding any changes you have made to *Warp* options.

3.4. The Warp VHDL Files Dialog Box

3.4.3. Specifying Warp Options

3.4.3.1. Selecting the Target Device

Warp can synthesize designs from VHDL descriptions for Cypress PLDs, CPLDs, or FPGAs. The Devices window in the Warp Option dialog box lets you specify the target device.

The Devices window (Figure 3-15) in the Warp Option dialog box lists all the possible target devices for synthesis. To select a target device for synthesis, use the scroll bar to bring the name of the desired device into view, then click on the device's name. Selecting the "default" device from the Devices window tells Warp to use a device named in the VHDL source file, e.g., in an architecture that specifies pin binding, or in an "attribute part_name" statement.

The Package window allows you to choose a package for the device you have selected. The default package for each device is usually the highest-speed package available for that device. To select a package, click with the right mouse button on the down-arrow below the word "Package:". Then, scroll through the available selections until the package you want appears, and click on it.

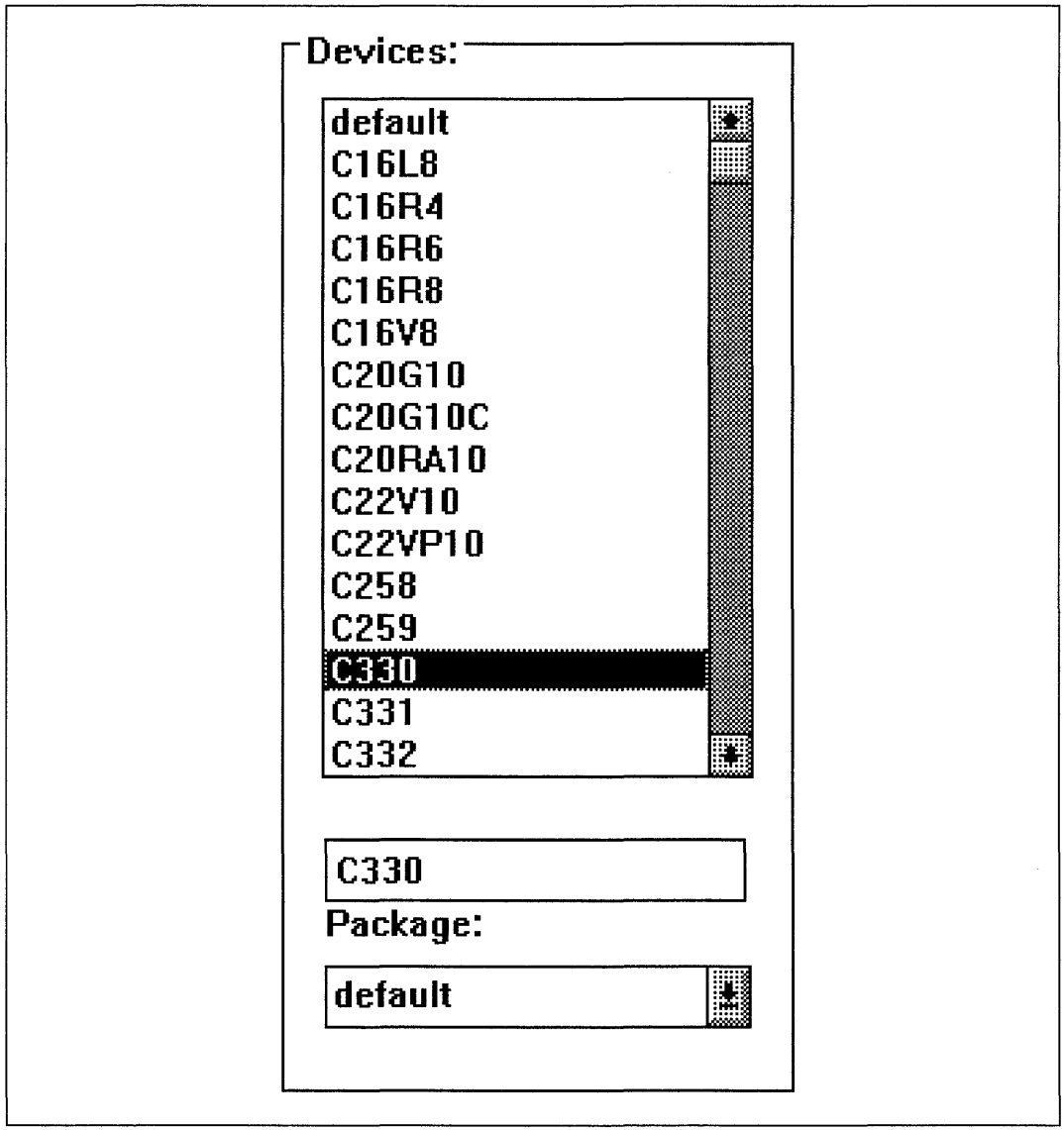


Figure 3-15. Devices Window in Warp Option Dialog Box.

3.4. The Warp VHDL Files Dialog Box

3.4.3. Specifying Warp Options

3.4.3.2. Selecting Optimization Level

Warp can employ different levels of optimization, which can help make more efficient use of chip resources.

When Warp optimizes a design, it simplifies the Boolean equations contained in that design to use fewer chip resources and make the design easier to fit onto a target device.

The Optimize window (Figure 3-16) in the Warp Option dialog box controls the level of optimization that *Warp* performs:

- None performs no optimization;
- Quick (the default) performs a standard, device-specific optimization;
- Large performs the default optimization, plus some pre-processing optimizations. In general, you don't need the Large optimization setting for PLD designs unless you are having trouble fitting a very large design. For pASIC designs, "Large" optimization is highly recommended.

The XOR button tells *Warp* to preserve XOR structure, which gives better results for certain kinds of architectures (e.g., arithmetic circuits) and devices with XOR gates in their macrocells (e.g., C335's, C34X's, etc.).

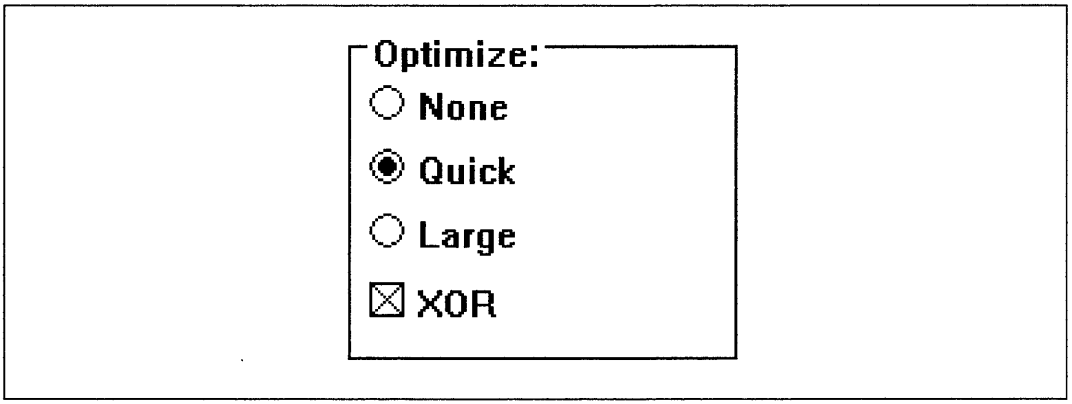


Figure 3-16. Optimize Window in Warp Option Dialog Box.

3.4. The Warp VHDL Files Dialog Box

3.4.3. Specifying Warp Options

3.4.3.3. Selecting Synthesis Output

Warp can produce four kinds of primary output: JEDEC files,.QDF files, and VHDL models.

The Output window of the Warp Options dialog box specifies which files a given *Warp* run actually produces:

- **Create JEDEC file** produces a file that can be input directly into a device programmer;
- **Create Hex file** allows output of JEDEC files encoded in Hex format to conserve disk space. The **Create Hex file** option is enabled only for the 34X and the 37X families.
- **Create QDIF file**, which appears when targeting a pASIC device (C38x), produces a file that can be input to the SpDE Place and Route tool;
- **Create ViewSim Model** applies to non-pASIC designs, and when enabled, converts the JEDEC file produced by the fitter into a VHDL file. This file is placed under the sub-directory 'vhd', and has the same name as the original top level VHDL file. It is then analyzed to allow a ViewSim simulation. The analyzed results are then placed in the current directory.

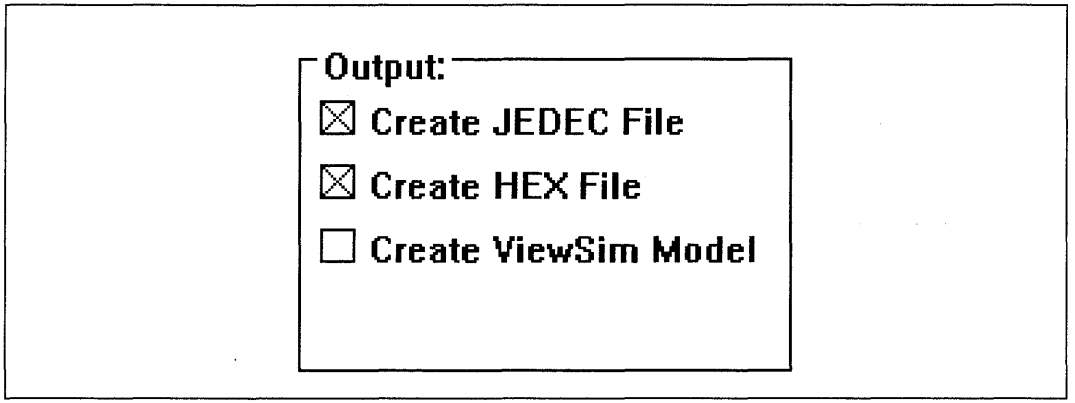


Figure 3-17. Output Window.

3.4. The Warp VHDL Files Dialog Box

3.4.3. Specifying Warp Options

3.4.3.4. Selecting Fitter Options

The Fitter window in the Warp Options dialog box lets you specify whether *Warp* should use D- or T-type flip-flops; whether *Warp* should determine the optimal signal polarity to fit the design into the target device; and whether *Warp* should assign signals to pins.

The Fitter window (Figure 3-18) in the Warp Options dialog box contains four check boxes labeled "Force Flip-Flop Types," "Force Polarity Optimization," "Allow Fitter to Change Pin Assignments," and "Force Logic Factoring."

Force Flip-Flop Types determines whether *Warp* should force flip-flops used in synthesis to be D- or T-type. If checked, this box enables the three boxes below it:

- **Use 'D'-type Form** forces the use of D-type flip-flops;
- **Use 'T'-type Form** forces the use of T-type flip-flops;
- **Use optimal 'D' or 'T' Form** tells *Warp* to determine which type of flip-flop best utilizes chip resources and use that one.

Force Polarity Optimization tells *Warp* to choose the optimal signal polarity for internal signals to fit the design into the target device.

Allow Fitter to Change Pin Assignments tells *Warp* to ignore any pin assignments in the VHDL source file(s), and use its own algorithms to map the external signals of the design to pins on the target device.

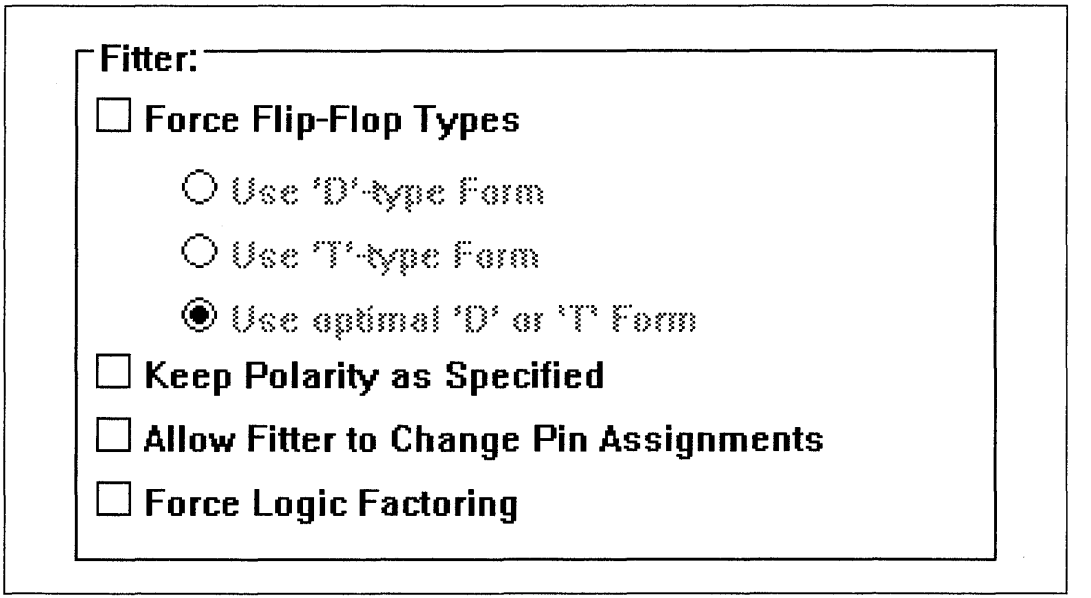


Figure 3-18. Fitter Window.

Force Logic Factoring tells *Warp* to perform three-level logic factoring instead of just two-level (sum of products) factoring. This is recommended for the 7c34x device family. For pASIC architectures, it allows common sub-expressions between different signals to be shared, thereby reducing fanout. This option is recommended for VHDL designs targeting pASIC devices, and is only applicable for non-frag based signals. (See also the descriptions for the synthesis_off and dont_touch attributes, in Chapter 5 of this manual.)

3.4. The Warp VHDL Files Dialog Box

3.4.3. Specifying Warp Options

3.4.3.5. Choosing Run Options

The Run Options window in the Warp Options dialog box specifies whether or not Warp runs in "quiet" mode.

Quiet Mode suppresses the equation-by-equation display of progress messages in the text area during compilation and synthesis.

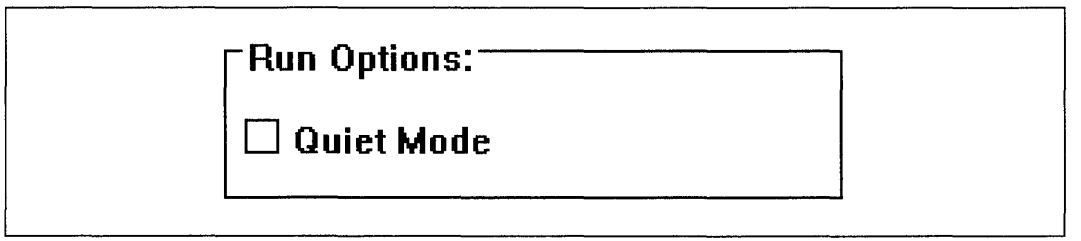


Figure 3-19. Run Options Window.

3.5. Running *Warp*

To run *Warp*, click on OK from the Warp VHDL Files dialog box.

Clicking on OK in the Warp VHDL Files dialog box runs *Warp* to compile and synthesize the selected files.

Warp is not a single program, but actually consists of a series of programs. As *Warp* runs, various messages appear in the text area of the Galaxy dialog box, informing you of *Warp*'s progress.

If an error is found in the source file, a message appears telling you the error number and informing you of the nature of the error. Whenever possible, the name of the source file and the line and column number where the error was found also appear. Be warned, however, that often an error in one position may give rise to other errors, later in the file. Be ready to look around for the location of the "root" error, if it isn't obvious.

Warp reports "Done" at the completion of its run, even if errors in the VHDL source file(s) cause no output to be created.

Chapter

4

Basic VHDL Elements

Basic VHDL Elements

4.1. Introduction

This section discusses some of the fundamental elements of VHDL implemented in *Warp*.

Topics include:

- identifiers
- data objects (constants, variables, signals)
- data types, including pre-defined types, user-definable types, subtypes, and composite types
- operators, including logical, relational, adding, multiplying, miscellaneous, assignment, and association operators
- entities
- architectures, for behavioral, data flow, and structural descriptions
- packages and libraries

Designs in VHDL are created in what are called entity/ architecture pairs. Entities and architectures are discussed in Sections 4-6 and 4-7. Sections leading up to this discussion cover VHDL language basics such as identifiers, data objects, data types, operators, and syntax.

4.2. Identifiers

An identifier in VHDL is composed of a sequence of one or more alphabetic, numeric, or underscore characters.

Legal characters for identifiers in VHDL include uppercase letters (A...Z), lowercase letters (a...z), digits (0...9) and the underscore character (_).

The first character in an identifier must be a letter.

The last character in an identifier cannot be an underscore character. In addition, two underscore characters cannot appear consecutively.

Lowercase and uppercase letters are considered identical when used in an identifier; thus, SignalA, signala, and SIGNALA all refer to the same identifier.

Comments in a VHDL description begin with two consecutive hyphens (--), and extend to the end of the line. Comments can appear anywhere within a VHDL description.

VHDL defines a set of reserved words, called keywords, that cannot be used as identifiers.

Examples

```
-- this is a comment.

-- this is the first line of
-- a three-line comment. Note the repetition
-- of the double hyphens for each line.

entity mydesign is -- comment at the end of a line
.
.
.
```

The following are legal identifiers in VHDL:

```
SignalA
Hen3ry
Output_Enable
C3PO
THX_1138
```

The following are not legal identifiers in VHDL:

```
3POC              -- identifier can't start with a digit
_Output_Enable    -- or an underscore character
My__Design        -- contains two consecutive underscores
My_Entity_        -- can't end with an underscore, either
Sig%              -- percent sign is an illegal character
Signal            -- reserved word
```

4.3. Data Objects

A data object holds a value of some specified type. In VHDL, all data objects belong to one of three classes: constants, variables, or signals.

Syntax (constant declaration):

```
constant identifier[,identifier...]:type:=value;
```

Syntax (variable declaration):

```
variable identifier[,identifier...]:type[:=value];
```

Syntax (signal declaration):

```
signal identifier[,identifier...]:type [:=value];
```

An object of class constant can hold a single value of a given type. A constant must be assigned a value upon declaration. This value cannot be changed within the design description.

An object of class variable can also hold a single value of a given type at any point in the design description. A variable, however, can take on many different values within the design description. Values are assigned to a variable by means of a variable assignment statement.

An object of class signal is similar to an object of class variable in *Warp*, with one important difference: signals can hold or pass logic values, while variables cannot. Signals can therefore be synthesized to memory elements or wires.

Variables have no such hardware analogies. Instead, variables are simply used as indexes or value holders to perform computations incidental to modeling components.

Most data objects in VHDL, whether constants, variables, or signals, must be declared before they can be used. Objects can be given a value at declaration time by means of the ":=" operator.

Exceptions to the "always-declare-before-using" rule include:

1. the ports of an entity. All ports are implicitly declared as signals.
2. the generics of an entity. These are implicitly declared as constants.
3. the formal parameters of procedures and functions. Function parameters must be constants or signals, and are implicitly declared by the function declaration. Procedure parameters can be constants, variables, or signals, and are implicitly declared by the procedure declaration.
4. the indices of a loop or generate statement. These objects are implicitly declared when the loop or generate statement begins, and disappear when it ends.

Examples

```
constant bus_width:integer := 8;
```

This example defines an integer constant called bus_width and gives it a value of 8.

```
variable ctrl_bits:bit_vector(7 downto 0);
```

This example defines an eight-element bit vector called ctrl_bits.

```
signal sig1, sig2, sig3:bit;
```

This example defines three signals of type bit, named sig1, sig2, and sig3.

4.4. Data Types

A data type is a name that specifies the set of values that a data object can hold, and the operations that are permissible on those values.

Warp supports the following pre-defined VHDL types:

- integer;
- boolean;
- bit;
- character;
- string;
- bit_vector;
- x01z;
- x01z_vector

Warp also gives you the capability to define subtypes and composite types by modifying these basic types, and to define your own types by combining elements of different types.

Warp's pre-defined types, and *Warp*'s facilities for defining subtypes, composite types, and user-defined types, are all discussed in the following pages.

Note: VHDL is a strongly typed language. Data objects of one type cannot be assigned to a data objects of another, and operations are not allowed on data objects of differing types. *Warp* provides functions for converting bit_vectors to integers and integers to bit_vectors and functions for allowing certain operations on differing data types.

4.4. Data Types

4.4.1. Pre-Defined Types

Warp supports the following pre-defined VHDL types: integer, boolean, bit, character, string, and bit_vector.

Integer

VHDL allows each implementation to specify the range of the integer type differently. However, the range must extend from at least -(2**31-1) to +(2**31-1), or -2147483648 to +2147483647. *Warp* allows data objects of type integer to take on any value in this range.

Boolean

Data objects of this type can take on the values 'true' or 'false'.

Bit

Data objects of this type can take on the values '0' or '1'.

Character

Data objects of type character can take on values consisting of any of the 128 standard ASCII characters. The non-printable ASCII characters are represented by three-character identifiers, as follows: NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CC, S0, S1, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FSP, GSP, RSP, and USP.

String

A string is an array of characters. Here's an example:

```
variable greeting:string(1 to 13):="Hello, world!";
```

Bit_Vector

A bit vector is an array of bits in ascending or descending order and provides an easy means to manipulate buses. Bit_vectors can be declared as follows:

```
signal a, b:bit_vector(0 to 7);
signal c, d:bit_vector(7 downto 0);
signal e:bit_vector(0 to 5);
```

NOTE: bit vector constants are specified with double quote marks ("), whereas single bit constants are specified with single quote marks (').

If these signals are subsequently assigned the following values,

```
a <= "00110101";
c <= "00110101";
b <= x"7A";
d <= x"7A";
e <= O"25";
```

then we can compare the individual bits of "a"and "c" to discover that a(7) is '0', a(6) is '0', a(5) is '1', a(4) is '0',..., a(0) is '1', whereas c(7) is '1', c(6) is '0', c(5) is '1', c(4) is '0',... c(0) is '0'. This is because the bits of signal "a" are in ascending order, and the bits of signal "b" are in descending order.

A prefix of "X"or "x" denotes a hexadecimal value; a prefix of "O"or "o" denotes an octal value; a prefix of "B" or "b" denotes a binary value. If no prefix is included, a value of "b" is assumed. Underscore characters may be freely mixed in with the bit_vector value for clarity. Hexadecimal and octal designators should only be used if the hexadecimal or octal value can be directly mapped to the size of the bit_vector. For example, if "x" is a bit_vector(0 to 5), then the assignment 'a <= x"B";' cannot be made because the hexadecimal number "B" uses four bits and does not match the size of the bit_vector it is being assigned to.

String Literals

A value that represents a (one-dimensional) string of characters is called a string literal. String literals are written by enclosing the characters of the string within double quotes ("..."). String literals can be assigned either to objects of type string or to objects of type bit_vector, as long as both objects have been declared with enough elements to contain all the characters of the string:

```
variable err_msg:string(1 to 18);
err_msg := "Fatal error found!";


signal bus_a:bit_vector(7 downto 0);
bus_a<= "10011110";
```

x01z

Data objects of type x01z can have values of 'x', '0', '1', or 'z'.

x01z_vector

An x01z_vector is an array of x01z's in ascending or descending order, and can be defined in the same manner as a bit_vector.

4.4. Data Types

4.4.2. Enumerated Types

An enumerated type is a type with a user-defined set of possible values.

Syntax (enumerated type declaration)

```
type name is (value[,value...]);
```

The order in which the values are listed in an enumeration type's declaration defines the lexical ordering for that type. That is, when relational operators are used to compare two objects of an enumerated type, a given value is always less than another value that appears to its right in the type declaration. The position number of the leftmost value is 0; the position number of other values is one more than that of the value to its left in the type declaration.

Examples

```
type arith_op is (add,sub,mul,div);
```

This example defines an enumerated type named arith_op whose possible values are add, sub, mul, and div.

```
type states is (state0, state1, state2, state3)
```

This example defines an enumerated type named states, with four possible values: state0, state1, state2, and state3.

4.4. Data Types

4.4.3. Subtypes

A subtype is a subset of a larger type.

Syntax (subtype declaration):

```
subtype type is
   base_type range value {to|downto} value;
```

Subtypes are useful for range checking or for enforcing constraints upon objects of larger types.

Examples

```
subtype byte is bit_vector(7 downto 0);
subtype single_digit is integer range 0 to 9;
```

These examples define subtypes called byte and single_digit. Signals or variables that are declared as byte are bit_vectors of eight bits in descending order. Signals or variables that are declared as single_digit are integers with possible values consisting of the integers 0 through 9, inclusive.

```
subtype byte is bit_vector(7 downto 0);

type arith_op is (add,sub,mul,div);
subtype add_op is arith_op range add to sub;
subtype mul_op is arith_op range mul to div;
```

This example first defines an enumerated type called arith_op, with possible values add, sub, mul, and div. It then defines two subtypes: add_op, with possible values add and sub, and mul_op, with possible values mul and div.

4.4. Data Types

4.4.4. Composite Types

A composite type is a type made up of several elements from another type. There are two kinds of composite types: arrays and records.

Syntax (array type declaration)

```
type name is array ({low to high}|
                    {high downto low}) of base_type;
```

Syntax (record type declaration)

```
record type is record
   element:element_type
   [;element:element_type...];
end record;
```

An array is a data object consisting of a collection of elements of the same type. Arrays can have one or more dimensions. Individual elements of arrays can be referenced by specifying an index value into the array (see examples). Multiple elements of arrays can be referenced using aggregates.

A record is a data object consisting of a collection of elements of different types. Records in VHDL are analogous to records in Pascal and struct declarations in C. Individual fields of a record can be referenced by using selected names (see examples). Multiple elements of records can be referenced using aggregates.

Examples

The following are examples of array type declarations:

```
type big_word is array (0 to 63) of bit;
type matrix_type is array (0 to 15, 0 to 31) of bit;
type values_type is array (0 to 127) of integer;
```

Possible object declarations using these types include:

```
signal word1,word2:big_word;
signal device_matrix:matrix_type;
variable current_values:values_type;
```

Some possible ways of assigning values to elements of these objects include:

```
word1(0)<='1'; -- assigns value to 0th element in word1
word1(5)<='0'; -- assigns value to 5th element in word1
word2 <= word1; -- makes word2 identical to word1
word2(63) <= device_matrix(15,31); -- transfers value
-- of element from device_matrix to element of word2
current_values(0) := 0;
current_values(127) := 1000;
```

The following includes an example of a record type declaration:

```
type opcode is (add,sub,mul,div);
type instruction is record
   operator:opcode;
   op1:integer;
   op2:integer;
   end record;
```

Here are two object declarations using this record type declaration:

```
variable inst1, inst2:instruction;
```

Some possible ways of assigning values to elements of these objects include:

```
inst1.opcode := add; -- assigns value to opcode of inst1
inst2.opcode := sub; -- assigns value to opcode of inst2
inst1.op1 := inst2.op2; -- copies op2 of inst2
                        -- to op1 of inst2
inst2 := inst1; -- makes inst2 identical to inst1
```

4.5. Operators

VHDL provides a number of operators used to construct expressions to compute values. VHDL also uses assignment and association operators.

VHDL's expression operators are divided into five groups. They are (in increasing order of precedence): logical, relational, adding, multiplying, and miscellaneous.

In addition, there are assignment operators that transfer values from one data object to another, and association operators that associate one data object with another.

Table 4-1 lists the VHDL operators that *Warp* supports.

Table 4-1: VHDL Operators

Logical Operators					
and	or	nand	nor	xor	not
Relational Operators					
=	/=	<	<=	>	>=
Adding Operators					
+	-	&			
Multiplying Operators					
*	/	mod	rem		
Miscellaneous Operators					
abs	**				
Assignment Operators					
<=	:=				
Association Operator					
=>					

4.5. Operators

4.5.1. Logical Operators

The logical operators AND, OR, NAND, NOR, XOR, and NOT are defined for predefined types BIT and BOOLEAN.

AND, OR, NAND, and NOR are "short-circuit" operations. The right operand is evaluated only if the value of the left operand is not sufficient to determine the result of the operation. For operations AND and NAND, the right operand is evaluated only if the value of the left operand is TRUE. For operations OR and NOR, the right operand is evaluated only if the value of the left operand is FALSE.

Note that there is no differentiation of precedence among the binary boolean operators. Thus, successive boolean operators in an expression must be delimited by parentheses to guarantee error-free parsing and evaluation, e.g.,

```
a <= b AND c OR d
```

is not legal;

```
a <= (b AND c) OR d
```

is.

4.5. Operators

4.5.2. Relational Operators

Relational operators include tests for equality, inequality, and ordering of operands.

The operands of each relational operator must be of the same type. The result of each relational operation is of type BOOLEAN.

The equality operator (=) returns TRUE if the two operands are equal, FALSE otherwise. The inequality operator (/=) returns FALSE if the two operands are equal, TRUE otherwise.

Two scalar values of the same type are equal if and only if their values are the same. Two composite values of the same type (e.g., bit vectors) are equal if and only if for each element of the left operand there is a matching element of the right operand, and the values of matching elements are equal.

The ordering operators are defined for any scalar type and for array types (e.g., bit vectors). For scalar types, ordering is defined in terms of relative values (e.g., '0' is always less than '1'). For array types, the relation < (less than) is defined such that the left operand is less than the right operand if and only if:

- the left operand is a null array and the right operand is a non-null array; otherwise
- both operands are non-null arrays, and one of the following conditions is satisfied:
- the leftmost element of the left operand is less than that of the right; or
- the leftmost element of the left operand is equal to that of the right, and the tail of the left operand is less than that

of the right. The tail consists of the remaining elements to the right of the leftmost element and can be null.

The relation <= (less than or equal to) for array types is defined to be the inclusive disjunction of the results of the < and = operators for the same two operands (i.e., it's true if either the < or = relations are true). The relations > (greater than) and >= (greater than or equal to) are defined to be the complements of <= and <, respectively, for the same two operands.

4.5. Operators

4.5.3. Adding Operators

In VHDL, the '+' and '-' operators perform addition and subtraction, respectively. The '&' operator concatenates characters, strings, bits or bit vectors. All three of these operators have the same precedence, and so are grouped under the category "Adding Operators."

The adding operators + and - are defined for integers, and have their conventional meaning.

These operations are also supported for bit vectors, through the use of the BV_MATH package. (See "Bit Vector Operations" later in this chapter for more information.)

In *Warp*, concatenation is defined for bits and arrays of bits (bit vectors). The concatenation operator in *Warp* is "&".

If both operands are bit vectors, the result of the concatenation is a one-dimensional array whose length is the sum of the lengths of the operands, and whose elements consist of the elements of the left operand (in left-to-right order) followed by the elements of the right operand (in left-to-right order). The left bound of this result is the left bound of the left operand, unless the left operand is a null array, in which case the result of the operation is the right operand. The direction of the result is the direction of the left operand, unless the left operand is a null array, in which case the direction of the result is that of the right operand.

If one operand is a bit vector and the other is a bit, or if both are bits, the bit operand is replaced by an implicit one-element bit vector having the bit operand as its only element. The left bound of the implicit bit vector is 0, and its direction is ascending.

It is always safest to use ascending ranges in bit vector declarations used in concatenation. Given the example

```
signal a,b:bit_vector(0 to 2);
signal c:bit_vector;
c <= a & b; --concatenation
```

the result has a(0) in c(0) and b(0) in c(3), the concatenation of a and b occupying the range from 0 to 5 in c.

Given the example

```
signal a,b:bit_vector(2 downto 0);
signal c:bit_vector;
c <= a & b; --concatenation
```

the result has a(2) in c(2), a(1) in c(1), a(0) in c(0), b(2) in c(-1), b(1) in c(-2), and b(0) in c(-3). The concatenation occupies the range from 2 down to -3 in c.

4.5. Operators

4.5.4. Multiplying Operators

In VHDL, the '*' and '/' operators perform multiplication and division, respectively. Two other operands of the same precedence include the mod and rem operators. Both operators return the remainder when one operand is divided by another.

All the multiplication operators are defined for both operands being of the same integer or bit_vector type. The result is also of the same type as the operands.

The rem operation is defined as

A rem B = A-(A/B)*B

where "/" in the above example indicates an integer division. The result has the sign of A and an absolute value less than the absolute value of B.

The mod operation is similar, except that the result has the sign of B. In addition, , for some integer N, the result satisfies the relation:

A mod B = A-B*N

4.5. Operators

4.5.5. Miscellaneous Operators

The two "miscellaneous" expression operators in VHDL are abs and **.

The abs operator, defined for integers, returns the absolute value of its operand.

The ** operator raises a number to a power of two. It is defined for an integer first operand and a power-of-two second operand. Its result is the same as shifting the bits in the first operand left or right by the amount specified by the second operand.

4.5. Operators

4.5.6. Assignment Operations

VHDL has two assignment operators: "<=" and ":=". The first is used for signal assignments, the second for variable assignments.

Syntax (variable assignment)

variable_name := *expression*;

Syntax (signal assignment)

signal_name <= *expression*;

Variable assignments can only occur inside a process. Signal assignments can occur anywhere inside an architecture.

Assignments to objects of composite types can be assigned values using aggregates, which is simply a way of specifying more than one value to be assigned to elements of an object with a single assignment statement. Examples of the use of aggregates are shown below.

Examples

```
type opcode is (add,sub,mul,div);
type instruction is record
   operator:opcode;
   op1:integer;
   op2:integer;
variable inst1,inst2:instruction;
signal vec1, vec2 : bit_vector(0 to 3):

vec1 <= ('1','0','1','0'); -- aggregate assignment
vec2 <= vec1; -- another aggregate assignment
inst1 := (add,5,10); -- aggregate assignment to record
vec1 <= (0=>'0',others=>'1'); -- assign 0 to 0th bit,
                              -- set others to 1
```

4.5. Operators

4.5.7. Association Operations

Whenever you instantiate a component in a *Warp* description, you must specify the connection path(s) between the ports of the component being instantiated and the interface signals of the entity/architecture pair you are defining. You do this by means of an association list within a port map or a generic map.

Warp supports both *named* and *positional* association.

In *named* association, you use the "=>" association operator to associate a formal (the name of the port in the component being instantiated) with an actual (the name of the signal in the entity you're defining). If you think of the association operator as an "arrow" indicating direction, it's easy to remember which way to make the arrow point: it always points to the actual. For example, in the following instantiation of a predefined D flip-flop,

```
st0: DSRFF port map(
        d    => dat,
        s    => set,
        r    => rst,
        clk  => clk,
        q    => duh);
```

the arrow always points toward the ports of the defined component, the DSRFF in this case. Named association allows you to associate the signals in any order you want. In the previous example, you could have listed the "q => duh" before "d => dat".

In positional association, you don't use the association operator. Instead, you list the actuals (signals names) in the port map in the same order as the formals of the component being instantiated, without including the formal names at all.

For example, the jkff component for the C22V10 is declared as follows:

```
component jkff port(
   j   : in bit;
   k   : in bit;
   clk: in bit;
   q   : out bit);
end component;
```

An association list for an instantiation of this component could use either named association, like this:

```
jk1:jkff port map(j_in=>j,k_in=>k,clk=>clk,q_out=>q);
```

or positional association, like this:

```
jk1:jkff port map(j_in, k_in, clk, q_out);
```

Either form maps signals j_in, k_in, clk, and q_out in the entity being defined to ports j, k, clk, and q, respectively, on the instantiated component.

4.5. Operators

4.5.8. Bit-Vector Operations

Addition, subtraction, incrementing, decrementing, inverting, and relational operators for bit vectors are defined in the INT_MATH and BV_MATH packages.

With the appropriate package, INT_MATH or BV_MATH, the addition (+) and subtraction (-) operators allow you to add or subtract two bit-vectors, or a bit-vector and an integer constant, or a bit vector and an integer variable whose value is constant In all of the examples in this section, count is a bit_vector(0 to 5). The following example requires the use of the INT_MATH package by including the clause "use work.int_math.all;" immediately before the architecture.

```
count <= count + 1; -- vector + integer
count <= count - 5; -- vector - integer
```

The increment function, named inc_bv, adds one to the value of a bit vector. Similarly, the decrementing function, named dec_bv, subtracts one from the value of a bit vector. The following examples require the use of the BV_MATH package by including the clause "use work.bv_math.all;" immediately before the architecture

```
count <= inc_bv(count);-- incrementing
count <= dec_bv(count);-- decrementing
```

The inversion function, named inv, performs a bit-by-bit inversion of each element of a bit vector:

```
count <= inv(count);-- inverting
```

The inv function also operates on individual bits. This allows the function to be used in port maps for signal polarity conversion.

The relational operators =, /=, <, <=, >, >= allow you to compare a bit vector to an integer constant or an integer variable whose value is constant:

```
signal count:bit_vector(0 to 5);
  .
  .
  .
IF (count < 10) THEN
  .
  .
  .
END IF;
```

4.6. Entities

Entities describe the design I/O.

VHDL designs consist of entity/architecture pairs in which the entity describes the design I/O, or interface, and the architecture describes the content of the design.Together, entity/architecture pairs can be used as complete design descriptions or as components in a hierarchical design or both.

The syntax for an entity declaration is as follows:

```
ENTITY entity IS PORT(
   [signal][sig-name,...]:[direction] type
   [;signal[sig-nam,...]:[direction] type]
   .
   .
   .
   );
END entity-name;
```

The entity declaration specifies a name by which the entity can be referenced in a design architecture. In addition, the entity declaration specifies ports. Ports are a class of signals that define the entity interface. Each port has an associated signal name, mode, and type.

Choices for mode are IN (default), OUT, INOUT and BUFFER. Mode IN is used to describe ports that are inputs only; OUT is used to describe ports that are outputs only, with no feedback internal to the associated architecture; INOUT is used to describe bi-directional ports; BUFFER is used to describe ports that are outputs of the entity but are also fed back internally.

Two sample entity declarations appear in Figures 4-1 and 4-2.

Figure 4-1 shows the proper declaration for a bidirectional signal (which, in this case, is also a bit vector), along with several input

```
entity cnt3bit is port(
   q:inout X01Z_vector(0 to 2);
   inc,grst,rst,clk:in bit;
   carry:out bit);
   end cnt3bit;
```

Figure 4-1. Sample Entity Declaration.

```
ENTITY Bus_Arbiter IS PORT(
   Clk,      -- Clock
   DRAM_Refresh_Request,-- Refresh Request
   VIC_Wants_Bus,-- VIC Bus Request
   Sparc_Wants_Bus: IN BIT;-- Sparc Bus Request
   Refresh_Control,-- DRAM Refresh Control
   VIC_Has_Bus,-- VIC Has Bus
   Sparc_Has_Bus: OUT BIT);-- Sparc Has Bus
END Bus_Arbiter;
```

Figure 4-2. Sample Entity Declaration. Note the use of comments to document the purpose of each signal.

signals and an output signal.

Figure 4-2 shows how comments can be included within an entity declaration to document each signal's use within the entity.

4.7. Architectures

An architecture describes the behavior or structure of an associated entity.

Architectures describe the behavior or structure of associated entities and can be either or a combination of

1. Behavioral descriptions.

These descriptions provide a means to define the "behavior" of a circuit in abstract, "high level" algorithms, or in terms of "low level" boolean equations.

2. Structural descriptions.

These descriptions define the "structure" of the circuit in terms of components, and resemble a net-list that could describe a schematic equivalent of the design. Structural descriptions contain hierarchy in which components are defined at different levels.

The architecture syntax follows:

```
ARCHITECTURE aname OF entity IS
   [type-declarations]
   [signal-declarations]
   [constant-declarations]
   BEGIN
   [architecture definition]
   END aname;
```

Each architecture has a name and specifies the entity which it defines. Types, signals, and constant must all be declared before the beginning of the architecture definition. The architecture defines the concurrent signal assignments, component instantiations, and processes.

Examples

```
use work.int_math.all;
architecture archcounter of counter is
begin
proc1:  process (clk)
        begin
                if (clk'event and clk = '1') then
                        count <= count + 1;
                end if;
        end process proc1;
        x <= '1' when count = "1001" else '0';
end archcounter;
```

Archcounter is an example of a behavioral architecture description of a counter and a signal x that is asserted when count is a particular value. This design is considered behavioral because of the algorithmic way in which it is described. The details of such descriptions will be covered later.

```
use work.rtlpkg.all;
architecture archcapture of capture is
        signal c: bit;
begin
        c <= a AND b;
d1: dff port map(c, clk, x);
end archcapture;
```

Archcapture is the name of an architectural description that is both structural and behavioral in nature. It is considered structural because of the component instantiation, and it is considered behavioral because of the boolean equation. VHDL provides the flexibility to combine behavioral and structural architecture descriptions.

4.5. Operators

4.7.1. Behavioral Descriptions

Behavioral descriptions consist of concurrent signal assignments and/or processes that enable both low-level and high-level, abstract design descriptions.

Behavioral design descriptions consist of two types of statements:

1. Concurrent statements which define concurrent signal assignments by way of association operators.
2. Sequential statements within a PROCESS which enable an algorithmic way of describing a circuit's behavior. Sequential statements enable signal assignments to be based on relational and conditional logic.

These types of statements, as well as structural descriptions, may be combined in any architecture description.

Concurrent Statements

Concurrent statements are found outside of processes and are used to implement boolean equations, when... else constructs, signal assignments, or generate schemes. Here are some examples:

```
u <= a;
v <= u;
w <= a XOR b;
x <= (a AND s) or (b AND NOT(s));
y <= ('1' when (a='0' and b = '1') else '0';
z <= A when (count = "0010") else b;
```

Signal u is assigned the value of signal a and is its equivalent.

Likewise, v is equivalent to both signals u and a. The order of these signal assignments does not matter because they are outside of a process and are concurrent. The next two statements implement boolean equations, while the last two statements implement "when... else" constructs. You could read the assignment for signal y as "y gets (is assigned) '1' when a is zero and b is one, otherwise y gets '0'." Likewise, "z gets a when count is "0010," otherwise z gets b."

Sequential Statements

Sequential statements must be within a PROCESS and enable you to describe signal assignments in an algorithmic fashion. All statements in a process are evaluated sequentially, and therefore the order of the statements is important. For example, in the process

```
proc1: process (x)
   begin
      a <= '0';
      if x = '1011' then
         a <= '1';
      end if;
   end process proc1;
```

signal "a" is first assigned '0.' Later in the process, if x is found to be equivalent to "1011" then signal "a" is assigned the value '1.'

Final signal assignments occur at the end of the process. In other words, the VHDL compiler evaluates the code sequentially before determining the equations to be synthesized, whereas the compiler synthesizes equations for concurrent statements upon encountering them. A process taken as a whole is a concurrent statement.

The Process

In most cases, a process has a sensitivity list: a list of signals in

parentheses immediately following the key word "process". Signals assigned within a process can only change value if one of the signals in the sensitivity list transitions. If the sensitivity list is omitted, then the compiler infers that signal assignments are sensitive to changes in any signal.

You may find it helpful to think of processes in terms of simulation (VHDL is also used for simulation) in which a process is either active or inactive. A process becomes active only when a signal in the sensitivity list transitions. In the process

```
proc1: process (rst, clk)
   begin
      if rst = '1' then
         q <= '0';
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
```

only transitions in rst and clk cause the process to become active. If either clk or rst transition, then the process becomes active, and the first condition is checked (if rst = '1'). In the case that rst ='1,' q will be assigned '0,' otherwise the second condition is checked (if clk'event and clk ='1'). This condition looks for the rising edge of a clock. All signals within this portion of the process are sensitive to this rising edge clock, and the compiler infers a register for these signals. This process creates a D flip-flop with d as its input, q as its output, clk as the clock, and rst as an asynchronous reset.

4.5. Operators

4.7.2. Structural Descriptions

Structural descriptions of architectures are net-lists that enable you to instantiate components in hierarchical designs.

Structural descriptions are net-lists that enable you to instantiate components in hierarchical designs. A port map is part of every instantiation and indicates how the ports of the component are connected. Structural descriptions can be combined with behavioral descriptions, as in the following example:

```
architecture archmixed of mixed is
begin
--instantiations
cntl1: motor port map(clk, ld, en, c1, chg1, start1, stop1);
cntl2: motor port map(clk, ld, en, c2, chg2, start2, stop2);
safety: mot_check port map(status, c1, c2);
--concurrent statement
en <= '1' when (status='1' and status = '1') else '0';
-- concurrent process with sequential statements
ok:process (clk)
   begin
      if (clk'event and clk='1') then
         status <= update;
      end if;
   end process ok;
end archmixed;
```

This example shows that two motor components and one mot_check component are instantiated. The port maps are associated with inputs and outputs of the motor and mot_check components by way of positional association. Signal en is assigned by a concurrent statement, and signal status is assigned by a process that registers a signal using the common clock clk.

4.5. Operators

4.7.3. Design Methodologies

VHDL provides the flexibility to describe designs in multiple ways, depending on the designers coding preferences.

Designers can choose from multiple methods of describing designs in VHDL, depending on coding preferences. In this section, we'll discuss how to implement combinatorial logic, registered logic, counters, and state machines. The discussion of state machines will cover multiple implementations and the design and synthesis trade-offs for those implementations. Section 4.10, "Additional Design Examples" contains further design examples. Most of the design examples in this section can be found in the directory \warp\examples.

Combinatorial Logic

Following are examples of a four-bit comparator implemented in four different ways, all yielding the same result. In all examples, the entity is the same:

```
entity compare is port(
        a, b:   in bit_vector(0 to 3);
        aeqb:   out bit);
end compare;
```

The entity declaration specifies that the design has three ports: two input ports (a, b), and one output port (aeqb). The input ports are of type BIT_VECTOR and the output port is of type BIT.

Using a process, the comparator can be implemented like this:

```
architecture archcompare of compare is
begin
comp:   process (a, b)
        begin
```

```
                if a = b then
                        aeqb <= '1';
                else
                        aeqb <= '0';
                end if;
        end process comp;
end archcompare;
```

The design behavior is given in the architecture section. The architecture description consists of the process "comp". The process includes the sensitivity list (a,b) so that the process becomes active each time there is a change in one of these signals. The process permits the use of an algorithm to assert aeqb when a equals b.

With one concurrent statement, making use of the case...when construct, the same comparator can be described like this:

```
architecture archcompare of compare is
begin
        aeqb <= '1' when (a = b) else '0';
end;
```

In this example, the process in the previous example has been replaced by a concurrent signal assignment for aeqb.

Using boolean equations, the comparator looks like this:

```
architecture archcompare of compare is
begin
        aeqb <= NOT(
                (a(0) XOR b(0)) OR
                (a(1) XOR b(1)) OR
                (a(2) XOR b(2)) OR
                (a(3) XOR b(3)));
end;
```

In this example, a boolean equation replaces the when... else construct.

Finally, a structural design which implements a net list of XOR gates and a 4-input NOR gate, looks like this:

```
use work.gatespkg.all;
use work.rtlpkg.all;
architecture archcompare of compare is
        signal c: bit_vector(0 to 3);
begin
        x0: xor2 port map(a(0), b(0), c(0));
        x1: xor2 port map(a(1), b(1), c(1));
        x2: xor2 port map(a(2), b(2), c(2));
        x3: xor2 port map(a(3), b(3), c(3));

        a1: nor4 port map(c(0), c(1), c(2), c(3), aeqb);

end;
```

In this example, the compare architecture is described by instantiating gates much the same as one would by placing gates in a schematic diagram. The XOR and NOR gates used in this architecture are the same as those available in the *Warp* schematic library under "gates". The port map lists are associated with the inputs and outputs of the gates through positional association.

Many other functions or components can be implemented in multiple ways. We leave you with one last combinatorial example: a four-bit wide four-to-one multiplexer. In all versions, the entity is the same:

```
entity mux is port(
        a, b, c, d:     in bit_vector(3 downto 0);
        s:              in bit_vector(1 downto 0);
        x:              out bit_vector(3 downto 0));
end mux;
```

Using a process, the architecture looks like this:

```
architecture archmux of mux is
```

```
begin
mux4_1: process (a, b, c, d)
        begin
                if s = "00" then
                        x <= a;
                elsif s = "01" then
                        x <= b;
                elsif s = "10" then
                        x <= c;
                else
                        x <= d;
                end if;
        end process mux4_1;
end archmux;
```

Using a concurrent statement with a case... when construct, the architecture can be written as

```
architecture archmux of mux is
begin
        x <= a when (s = "00") else
             b when (s = "01") else
             c when (s = "10") else
             d;
end archmux;
```

If you prefer to write boolean equations, then you can write the architecture as follows.

```
architecture archmux of mux is
begin
    x(3) <=     (a(3) and not(s(1)) and not(s(0)))
            OR  (b(3) and not(s(1)) and s(0))
            OR  (c(3) and s(1) and not(s(0)))
            OR  (d(3) and s(1) and s(0));

    x(2) <=     (a(2) and not(s(1)) and not(s(0)))
            OR  (b(2) and not(s(1)) and s(0))
            OR  (c(2) and s(1) and not(s(0)))
            OR  (d(2) and s(1) and s(0));
```

```
    x(1) <=     (a(1) and not(s(1)) and not(s(0)))
            OR  (b(1) and not(s(1)) and s(0))
            OR  (c(1) and s(1) and not(s(0)))
            OR  (d(1) and s(1) and s(0));

    x(0) <=     (a(0) and not(s(1)) and not(s(0)))
            OR  (b(0) and not(s(1)) and s(0))
            OR  (c(0) and s(1) and not(s(0)))
            OR  (d(0) and s(1) and s(0));

end archmux;
```

A structural approach that also makes use of a generate scheme can be written like this:

```
use work.muxpkg.all;
architecture archmux of mux is
begin
mux_array: for i in 3 downto 0 generate
        mux_i: mux1of4 port map(one, s(0), s(1), a(i), b(i),
                                c(i), d(i), x(i));
           end generate;
end archmux;
```

This design makes use of the multiplexers in the *Warp* library. Of course, you could build up your own multiplexers and instantiate them instead. The generate scheme implemented here consists of a loop in variable i which is used to instantiate a mux1of4 for each i in 3 downto 0. In each instantiation, i is used to reference the individual bit of the bit_vector that is under consideration.

Registered Logic

There are two methods for implementing registered logic: instantiating a register (or other component with registers), or using a process that is sensitive to a clock edge. For example, if you wanted to use a D register and a 4-bit counter, you could simply instantiate these components after including the appropriate packages:

```
use work.rtlpkg.all;
use work.counterpkg.all;
...
d1: dsrff port map(d, s, r, clk, q);
c1: cntr4 port map(one, open, one, one, d3, d2, d1, d0,
                   clk, rst, cnt3, cnt2, cnt1, cnt0);
```

Another method of using registered elements is to include a process that is sensitive to a clock edge or that waits for a clock edge. In processes that are sensitive to clock edges or that wait for clock edges, the compiler infers a register for the signals defined within that process. Four basic templates are supported:

```
process_label: process
   begin
      wait until clk = '1';
      . ..
   end process;
```

This process does not have a sensitivity list. Instead it begins with a WAIT statement. The process will become active when clk transitions to a one (clk--or whatever identifier you give to your clock--can also wait for zero for devices that support such clocking schemes). All signal assignments within such a process will be registered, as these signals only change values on clock edges and retain their values between clock edges.

```
my_proc: process (clk)
   begin
      if (clk'event and clk ='1') then
         ...
      end if;
end process;
```

This process is sensitive only to changes in the clock, as the sensitivity list indicates. The first statement within the process looks for a transition from zero to one in signal clk. All signals that are assigned within this process are also registered because the assignments only occur on rising clock edges, and the signals

retain their values between rising clock edges.

```
your_proc: process (rst, clk)
   begin
      if rst = '1' then
         ...
      elsif (clk'event and clk='1') then
         ...
      end if;
   end process;
```

This process is sensitive to changes in the clock and signal rst, as the sensitivity list indicates. This process is intended to support signals that must be registered and have an asynchronous set and/or reset. The first statement within the process checks to see if rst has been asserted. Signals that are assigned in this portion of the template are assumed to be registered with rst assigned as either the asynchronous reset or set of the register, as appropriate. If rst has not been asserted, then the remainder of this process works as does the previously described process.

```
proc1: process (rst, pst, clk)
   begin
      if rst = '1' then
         ...
      elsif pst = '1' then
         ...
      elsif (clk'event and clk='1') then
         ...
      end if;
   end process;
```

This process is sensitive to changes in the clock and signals rst and pst, as the sensitivity list indicates. This process is intended to support signals that must be registered and have an asynchronous set and reset. The first statement within the process checks to see if rst has been asserted. Signals that are assigned in this portion of the template are assumed to be registered with rst used as either the asynchronous reset or set

of the register, as appropriate. The second condition assigns pst as the asynchronous reset or set of the register, as appropriate. If rst and pst have not been asserted, then the remainder of this process works as does the previous process.

To register 32-bits with an asynchronous reset, you could simply write the code

```
regs32: process (r, clk2)
   begin
      if (r = '1') then
         q <= x"ABC123DE";
      elsif (clk2'event and clk2='1') then
         q <= d;
      end if;
   end process;
```

Assuming that q and d are declared as 32-bit signals or ports, then this code example implements 32 registers with d(i) as the input, q(i) as the output, clk2 as the clock, and r as the asynchronous reset for some of the registers and r as the asynchronous preset for the others. This is because resetting the q to the value x"ABC123DE" will cause some registers to go high and other registers to go low when r is asserted.

Counters and state machines designed with processes are described in more detail in the following discussions.

Counters

This is a 4-bit loadable counter:

```
entity counter is port(
   clk, load:  in bit;
   data:       in bit_vector(3 downto 0);
   count:      buffer bit_vector(3 downto 0));
end counter;

use work.int_math.all;
architecture archcounter of counter is
```

```
begin
upcount: process (clk)
   begin
      if (clk'event and clk= '1') then
         if load = '1' then
            count <= data;
         else
            count <= count + 1;
         end if;
      end if;
   end process upcount;
end archcounter;
```

The "use work.int_math.all;" statement is included to make the integer math package visible to this design. The integer math package provides an addition function for adding integers to a bit vector. The native VHDL addition operator applies only to integers. The architecture description is behavioral. In this design the counter counts up or synchronously loads depending on the LOAD control input. The counter is described by the process "upcount". The statement "if (clk'event AND clk = '1') then..." implies that operation of the counter takes place on the rising edge of the signal clk. The following IF statement describes the loading and counting operation.

In this description, the "if (clk'event AND clk = '1') then..." statement (and its associated "end if") could have been replaced by the statement "wait until clk = '1';".

Following is a 4-bit loadable counter with synchronous reset:

```
entity counter is port(
   clk, reset, load:    in bit;
   data:                in bit_vector(3 downto 0);
   count:               buffer bit_vector(3 downto 0));
end counter;

use work.int_math.all;
architecture archcounter of counter is
begin
```

```
upcount: process (clk)
   begin
      if (clk'event and clk= '1') then
         if reset = '1' then
            count <= "0000";
         elsif load = '1' then
            count <= data;
         else
            count <= count + 1;
         end if;
      end if;
   end process upcount;
end archcounter;
```

In this design the counter counts up, synchronously resets depending on the RESET input, or synchronously loads depending on the LOAD control input. The counter is described by the process "upcount." The statement "if (clk'event AND clk = '1') then..." appearing first implies that all operations of the counter take place on the rising edge of the signal, clk. The following IF statement describes the synchronous reset operation; the counter is synchronously reset on the rising edge of clk. The remaining operations (load and count) are described in elsif or else clauses in this same if statement, therefore the reset takes precedence over loading or counting. If reset is not '1', then the operation of the counter depends upon the load signal. This operation is then identical to the counter in the previous example.

Following is a 4-bit loadable, enablable counter with asynchronous reset:

```
entity counter is port(
   clk, reset, load,counten:  in bit;
   data:                in bit_vector(3 downto 0);
   count:               buffer bit_vector(3 downto 0));
end counter;

use work.int_math.all;
```

```
architecture archcounter of counter is
begin
upcount: process (clk, reset)
   begin
      if reset = '1' then
         count <= "0000";
      elsif (clk'event and clk= '1') then
         if load = '1' then
            count <= data;
         elsif counten = '1' then
            count <= count + 1;
         end if;
      end if;
   end process upcount;
end archcounter;
```

In this design the counter counts up, resets depending on the RESET input, or synchronously loads depending on the LOAD control input. This counter is similar to the one in the previous example except that the reset is asynchronous. The sensitivity list for the process contains both clk and reset. This causes the process to be executed at any change in these two signals.

The first IF statement, "if reset = '1' then...," states that this counter will assume a value of "0000" whenever reset is '1'. This will occur when the process is activated by a change in the signal reset. The elsif clause that is part of this IF statement, "elsif (clk'event AND clk = '1') then...," implies that the subsequent statements within the IF are performed synchronously (clk'event) on the rising edge (clk = '1') of the signal clk (providing that the previous If / ELSIF clauses were not satisfied). The synchronous operation of this process is similar to the previous example, with the exception of the counten signal enabling the counter. If counten is not asserted, then count retains its previous value.

Following is a 4-bit loadable, enablable counter with asynchronous reset and preset.

```
entity counter is port(
   clk, rst, pst, load,counten:     in bit;
   data:                in bit_vector(3 downto 0);
   count:               buffer bit_vector(3 downto 0));
end counter;

use work.int_math.all;
architecture archcounter of counter is
begin
upcount: process (clk, rst, pst)
   begin
      if rst = '1' then
         count <= "0000";
      elsif pst = '1' then
         count <= "1111";
      elsif (clk'event and clk= '1') then
         if load = '1' then
            count <= data;
         elsif counten = '1' then
            count <= count + 1;
         end if;
      end if;
   end process upcount;
end archcounter;
```

In this design the counter counts up, resets depending on the RESET input, presets depending upon the pst signal, or synchronously loads depending on the LOAD control input. This counter is similar to the previous example except that a preset control has been added (pst). The sensitivity list for this process contains clk, pst, and rst. This causes the process to be executed at any change in these three signals.

The first IF statement "if rst = '1' then.." implies that this counter will assume a value of "0000" whenever rst is '1'. This will occur when the process is activated by a change in the signal rst. The first elsif clause that is part of this IF statement, "elsif pst = '1' then...," implies that this counter will assume a value of "1111" whenever pst is '1' and rst is '0'. This will occur when the process is activated by a change in the signal pst and rst is not '1'.

The second elsif clause that is part of this IF statement, "elsif (clk'event AND clk = '1') then...," implies that the following statements within the IF are performed synchronously (clk'event) and on the rising edge (clk = '1') of the signal clk providing that the previous If / ELSIF clauses were not satisfied. In this regard the operation is identical to the counter in the previous example.

The following is an 8-bit loadable counter. The data is loaded by disabling the three-state output, and using the same i/o pins to load.

```
entity ldcnt is port (
   clk, ld, oe:    in bit;
   count_out:      inout x01z_vector(7 downto 0));
end ldcnt;

use work.rtlpkg.all;
use work.int_math.all;

architecture archldcnt of ldcnt is
   signal count, data:bit_vector(7 downto 0);
begin
counter: process (clk)
   begin
      if (clk'event and clk='1') then
         if ld = '1' then
            count <= data;
         else
            count <= count + 1;
         end if;
      end if;
   end process counter;
bidirs:for i in 7 downto 0 generate
   bi_x: bufoe port map (count(i), oe, count_out(i),
data(i));
   end generate;
end archldcnt;
```

This design performs a synchronous counter that can be loaded. The load occurs by disabling the output pins. This allows a signal to be driven from off chip to load the counter. The three-state for I/O pins is accomplished with the use of a bufoe component which is defined as follows:

```
COMPONENT bufoe -- Three-state with feedback for I/O.
        PORT (
            x   : IN BIT; -- Logic input to Buffer.
            oe  : IN BIT; -- Output Enable input.
            y   : inout x01z; -- bit output pin.
            yfb : OUT BIT -- Output feed back as bit.
        );
    END COMPONENT;
```

The generation scheme instantiates a bufoe for each i in 7 downto 0. For each bufoe, a counter bit is the input to the buffer, oe is the enable, count_out the output, and data the feed back. Count_out and data are essentially the same node, but the bufoe component which implements the three state requires two separate signal names and provides the type conversion between x01z and bit.

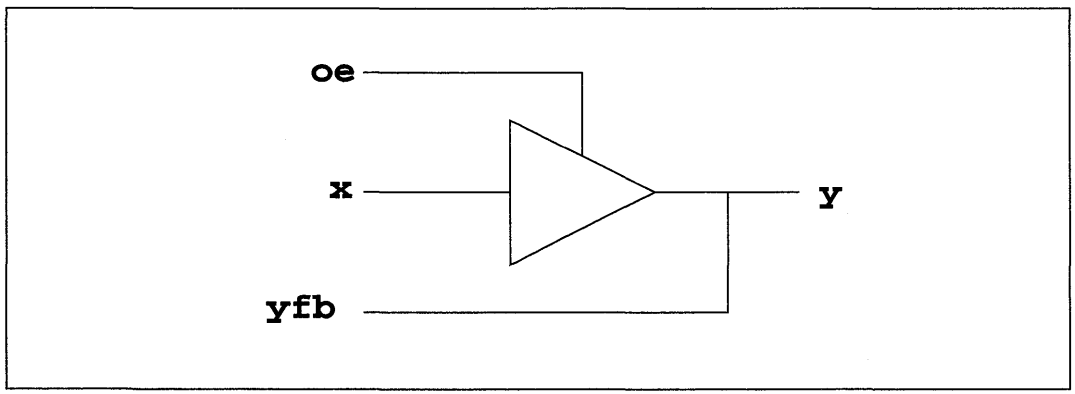


Figure 4-3. Bufoe Component.

State Machines

VHDL provides constructs that are well suited for coding state machines. VHDL also provides multiple ways to describe state machines. In this section, we'll take a look at some coding implementations and how the implementation affects synthesis (the way in which the design description is realized in terms of logic and the architectural resources of the target device).

The implementation that you choose when coding may depend on the issues that are important to you: fast time-to-market or squeezing all the capacity and performance you can out of a device. Often times, choosing one coding style over another will not result in much difference and will allow you to meet your performance and capacity requirements while achieving fast time-to-market.

We will consider Moore and Mealy state machines, discussing Moore machines first. Moore machines are characterized by the outputs changing only with a change in state. Moore machines can be implemented in multiple ways:

1. Outputs are decoded from state bits combinatorially.
2. Outputs are decoded in parallel output registers.
3. Outputs are encoded within the state bits. A state encoding is chosen such that a set of the state bits are the required outputs for the given states.
4. One-hot encoded. One register is asserted "hot" per state. This encoding scheme often reduces the amount of logic required to transition to the next state at the expense of more registers. This implementation is particularly well suited to FPGA, register-intensive devices.
5. Truth Tables. A truth table maps the current state and inputs to a next state and outputs.

We will take the same state machine and implement it five different ways as a Moore machine, discussing the design and synthesis issues. Figure 4-4 shows the state diagram.

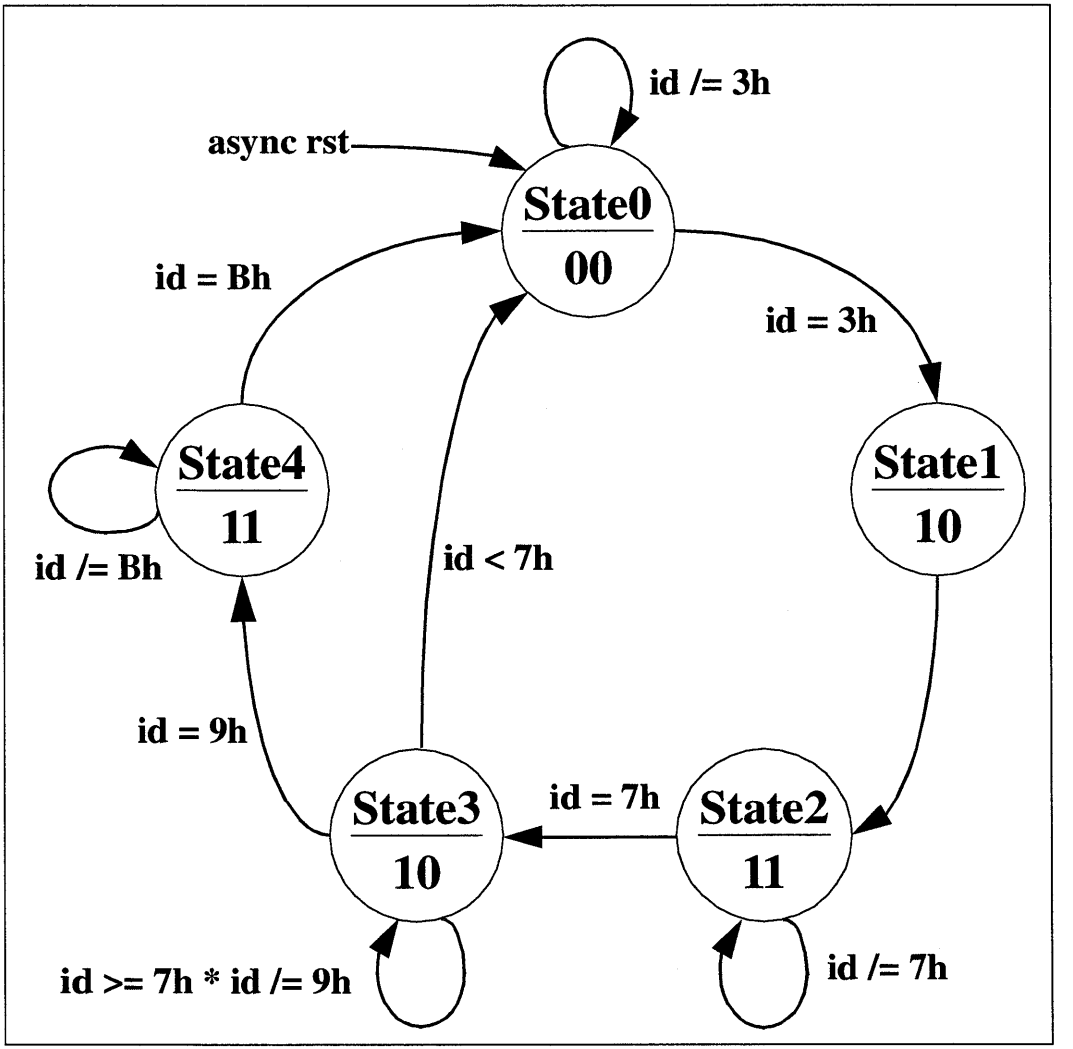


Figure 4-4. State Diagram of Moore State Machine

Outputs decoded combinatorially

Figure 4-5 shows a block diagram of an implementation in which the state machine outputs are decoded combinatorially. The code follows:

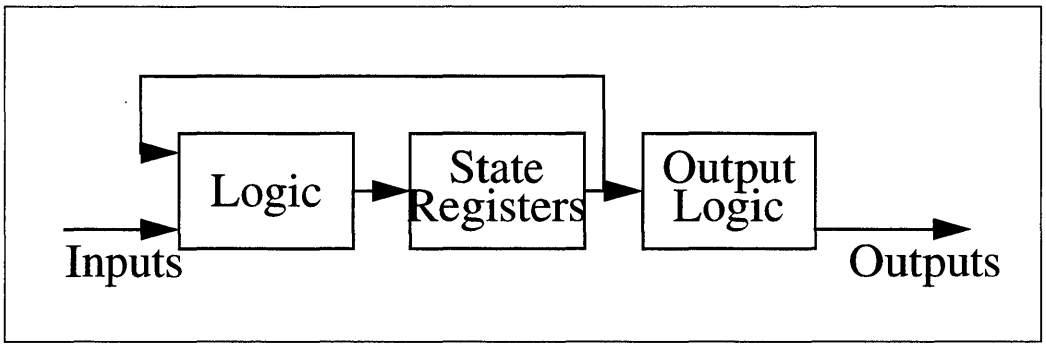


Figure 4-5. Outputs Decoded Combinatorially

```
entity moore1 is port(
   clk, rst:in bit;
   id:    in bit_vector(3 downto 0);
   y:     out bit_vector(1 downto 0));
end moore1;

architecture archmoore1 of moore1 is
   type states is (state0, state1, state2, state3, state4);
   signal state: states;
begin
moore: process (clk, rst)
   begin
      if rst='1' then
         state <= state0;
      elsif (clk'event and clk='1') then
         case state is
            when state0 =>
               if id = x"3" then
                  state <= state1;
```

```
                else
                    state <= state0;
                end if;
            when state1 =>
                state <= state2;
            when state2 =>
                if id = x"7" then
                    state <= state3;
                else
                    state <= state2;
                end if;
            when state3 =>
                if id < x"7" then
                    state <= state0;
                elsif id = x"9" then
                    state <= state4;
                else
                    state <= state3;
                end if;
            when state4 =>
                if id = x"b" then
                    state <= state0;
                else
                    state <= state4;
                end if;
        end case;
    end if;
end process;

--assign state outputs;
y <= "00" when (state=state0) else
     "10" when (state=state1 or state=state3) else
     "11";
end archmoore1;
```

The architecture description begins with a type declaration, called an enumerated type, for "states" which defines five states labeled state0 through state4. A signal, state, is then declared to be of type states. This means that the signal called state can take on values of state0, state1, state2, state3, or state4.

The state machine itself is described within a process. The first condition of this process defines the asynchronous reset condition which puts the state machine in state0 whenever the signal rst is a '1'. If the rst signal is not a '1' and the clock transitions to a '1'--"elsif (clk'event and clk='1')"--then the state machine algorithm is sequenced. The design can be rising edge triggered as it is in this example, or falling edge triggered by specifying clk='0'.

On a rising edge of the clock, the CASE statement (which contains all of the state transitions for the Moore machine) is evaluated. The "when" statements define the state transitions which are based on the input id. For example, in the "case when" the current state is state0, the state machine will transition to state1 if id=x"3", otherwise the state machine will remain in state0. In a concurrent statement outside of the process, the output vector y is assigned a value based on the current state.

This implementation demonstrates the algorithmic and intuitive fashion in which VHDL permits in the description of state machines. Simple case... when statements enable you to succinctly define the states and their transitions. There are two design and synthesis issues with this implementation which some designers may wish to consider: clock-to-out times for the combinatorially decoded state machine outputs, and an alternative state encoding to use minimal product terms.

As Figure 4-5 shows, the clock-to-out times for the state machine outputs are determined by the time it takes for the state bits to be combinatorially decoded. For designs that require minimal clock-to-out times, a similar implementation as the one above can be used with a design modification: a second process could register the outputs after combinatorial decode. This would introduce a one clock-cycle latency, however. If this latency is not acceptable, then you will need to choose from the second implementation (outputs decoded in parallel registers) or the third implementation (outputs encoded within state bits).

For designs in which the number product terms must be minimized, you can implement your design similar as above, with one exception: rather than using the enumerated encoding, you will want to implement your own encoding scheme. The third implementation shows how to do this.

Outputs decoded in parallel output registers

Figure 4-6 shows a block diagram of an implementation in which the state machine outputs are determined at the same time the next state is, by using output registers. The code follows:

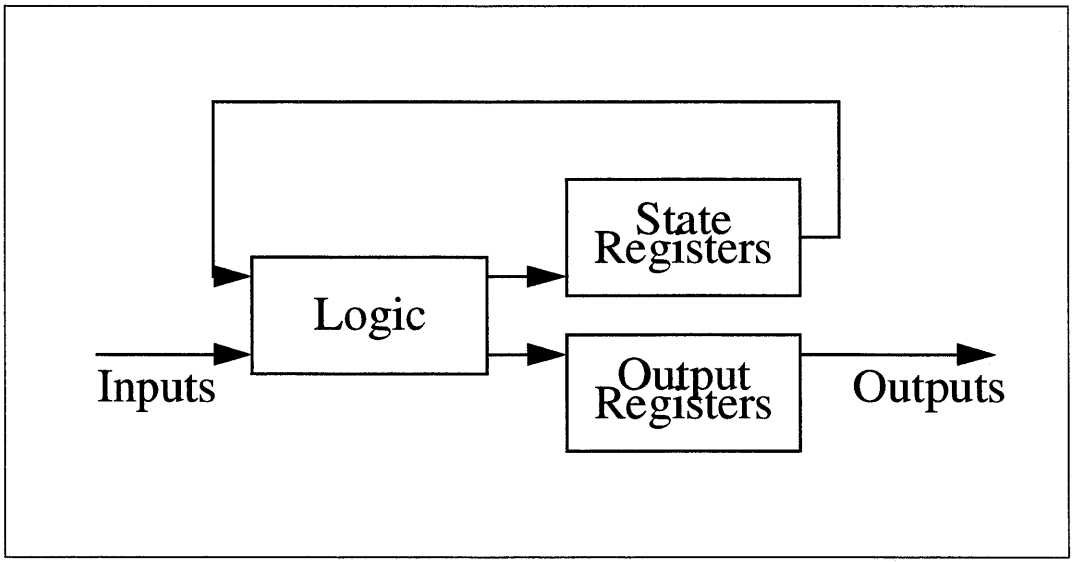


Figure 4-6. Outputs decoded in parallel

```
entity moore2 is port(
   clk, rst:in bit;
   id:    in bit_vector(3 downto 0);
   y:     out bit_vector(1 downto 0));
end moore2;
```

```
architecture archmoore2 of moore2 is
   type states is (state0, state1, state2, state3, state4);
   signal state: states;
begin
moore: process (clk, rst)
   begin
      if rst='1' then
         state <= state0;
         y <= "00";
      elsif (clk'event and clk='1') then
         case state is
            when state0 =>
               if id = x"3" then
                  state <= state1;
                  y <= "10";
               else
                  state <= state0;
                  y <= "00";
               end if;
            when state1 =>
               state <= state2;
               y <= "11";
            when state2 =>
               if id = x"7" then
                  state <= state3;
                  y <= "10";
               else
                  state <= state2;
                  y <= "11";
               end if;
            when state3 =>
               if id < x"7" then
                  state <= state0;
                  y <= "00";
               elsif id = x"9" then
                  state <= state4;
                  y <= "11";
               else
                  state <= state3;
                  y <= "10";
               end if;
```

```
            when state4 =>
               if id = x"b" then
                  state <= state0;
                  y <= "00";
               else
                  state <= state4;
                  y <= "11";
               end if;
         end case;
      end if;
   end process;
end archmoore2;
```

This implementation requires that you specify--in addition to the state transitions--the state machine outputs for every state and every input condition because the outputs must be determined in parallel with the next state. Assigning the state machine outputs in the synchronous portion of the process causes the compiler to infer registers for the output bits. Having output registers rather than decoding the outputs combinatorially results in a smaller clock-to-out time. This implementation has one design/synthesis issue which some may wish to consider: while this implementation achieves a better clock-to-out time for the state machine outputs (as compared to the first implementation), it uses more registers (and possibly more product terms) than the first implementation. The next implementation (outputs encoded within state bits) achieves the fastest possible clock-to-out times while at the same time using the fewest total number of macrocells in a PLD/CPLD.

Outputs encoded within state bits

Figures 4-7 and 4-8 show the state encoding table and a block diagram of an implementation in which the outputs are encoded within the state registers--the two least significant state bits are the outputs. Therefore, no decoding is required for the outputs, and the output signals can be directed from the state registers to output pins. The code follows:

State	Output	State Encoding
s0	00	000
s1	10	010
s2	11	011
s3	10	110
s4	11	111

Figure 4-7. State encoding table

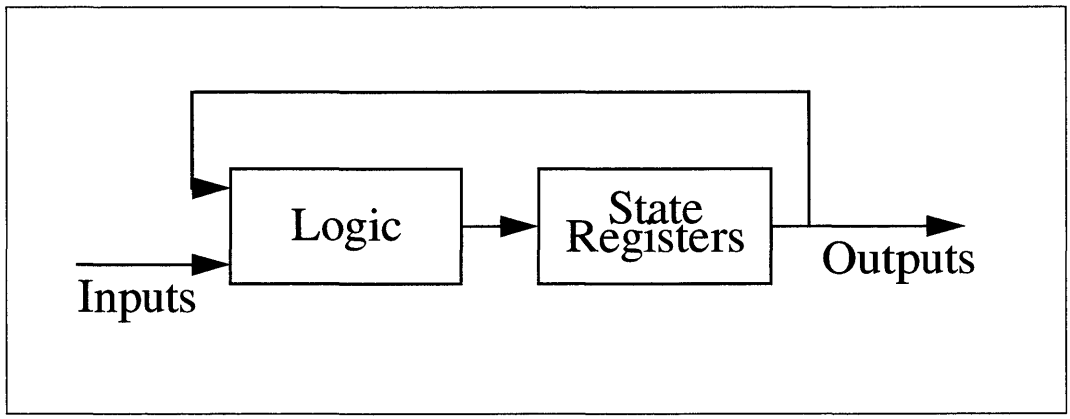


Figure 4-8. Outputs Encoded Within State Bits

```
entity moore1 is port(
   clk, rst:in bit;
   id:    in bit_vector(3 downto 0);
   y:     out bit_vector(1 downto 0));
end moore1;
```

```
architecture archmoore1 of moore1 is
   signal state: bit_vector(2 downto 0);
-- State assignment is such that 2 LSBs are outputs
constant state0: bit_vector(2 downto 0) := "000";
constant state1: bit_vector(2 downto 0) := "010";
constant state2: bit_vector(2 downto 0) := "011";
constant state3: bit_vector(2 downto 0) := "110";
constant state4: bit_vector(2 downto 0) := "111";
begin
moore: process (clk, rst)
   begin
      if rst='1' then
         state <= state0;
      elsif (clk'event and clk='1') then
         case state is
            when state0 =>
               if id = x"3" then
                  state <= state1;
               else
                  state <= state0;
               end if;
            when state1 =>
               state <= state2;
            when state2 =>
               if id = x"7" then
                  state <= state3;
               else
                  state <= state2;
               end if;
            when state3 =>
               if id < x"7" then
                  state <= state0;
               elsif id = x"9" then
                  state <= state4;
               else
                  state <= state3;
               end if;
            when state4 =>
               if id = x"b" then
                  state <= state0;
               else
```

```
                state <= state4;
            end if;
          when others =>
            state <= state0;
        end case;
      end if;
    end process;

--assign state outputs (equal to state bits)
y <= state(1 downto 0);
end archmoore1;
```

A state encoding was chosen for this design such that the last two bits were equivalent to the state machine outputs for that state. By using constants, the state machine could be encoded, and the transitions specified as in the first implementation. The output was specified in a concurrent statement. This statement shows that the outputs are a set of the state bits. One synthesis issue is highlighted in this example: the use of "when others =>."

"When others" is used when not all possible combinations of a bit sequence have been specified in other "when" clauses. In this, the states "001," "100," and "101" are not defined, and no transitions are specified for these states. If "when others" is not used, then next state logic must be synthesized assuming that if the machine gets in one of these states then it will remain in that state. This has the effect of utilizing more logic (product terms in the case of a PLD/CPLD). Supplying a simple "when others" is a quick solution to this design issue.

One-hot-one state machines

In a one-hot-one state machine, there is one register for each state. Only one register is asserted, or "hot," at a time, corresponding to one distinct state. Figure 4-9 shows three states of a state machine and how one of the state bits would be implemented. From this implementation, you can see that the

next state logic is quite simple. The trade-off is in the number of registers that is required. For example, a state machine with eight states could be coded in three registers. The equivalent one-hot coded state machine would require eight registers. The trade-off is that the next-state logic is more simple, often times enabling faster performance in FPGA architectures which are register intensive but that would require multiple levels of logic to decode a complex state transition. Following is the code:

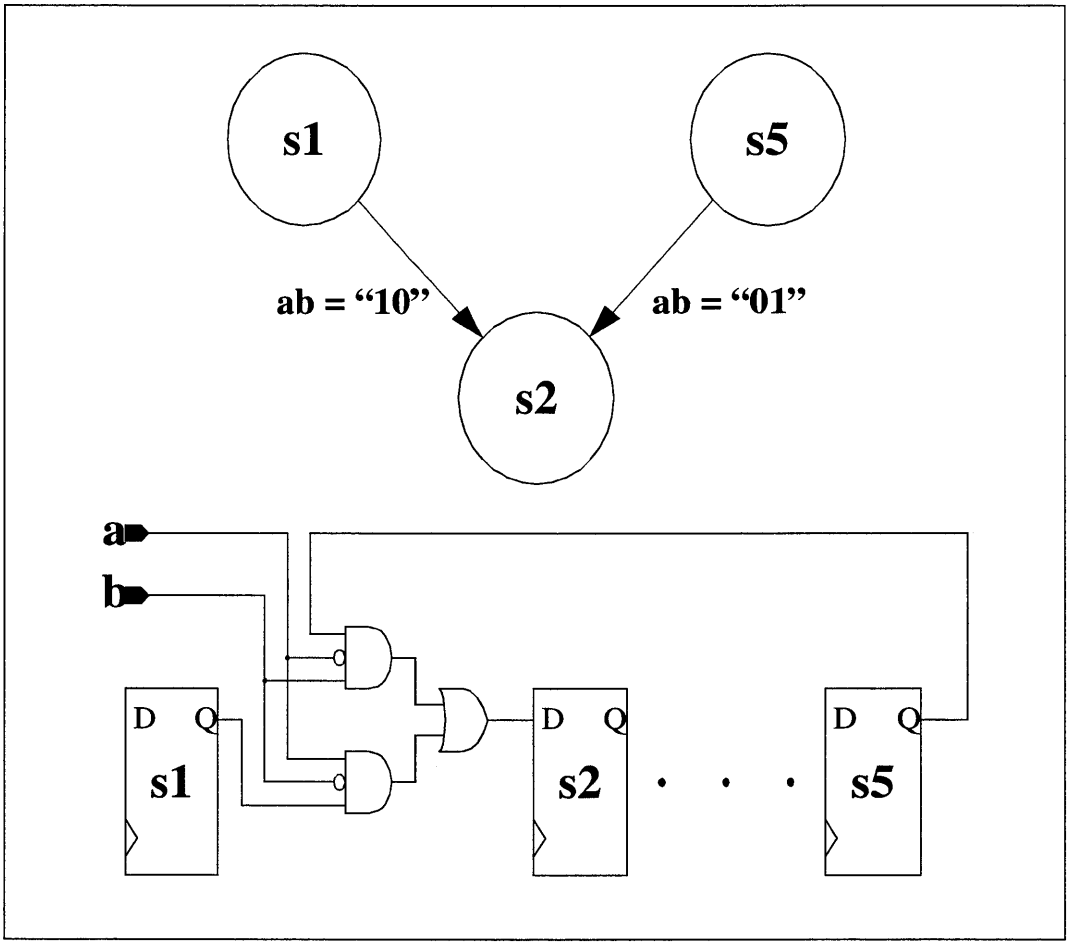


Figure 4-9. Implementation of one-hot state machine bits

```
entity one_hot is port(
   clk, rst:in bit;
   id:    in bit_vector(3 downto 0);
   y:     out bit_vector(1 downto 0));
end one_hot;

architecture archone_hot of one_hot is
```

```
   type states is (state0, state1, state2, state3, state4);
   attribute state_encoding of states:type is one_hot_one;
   signal state: states;
begin
machine: process (clk, rst)
   begin
      if rst='1' then
         state <= state0;
      elsif (clk'event and clk='1') then
         case state is
            when state0 =>
               if id = x"3" then
                  state <= state1;
               else
                  state <= state0;
               end if;
            when state1 =>
               state <= state2;
            when state2 =>
               if id = x"7" then
                  state <= state3;
               else
                  state <= state2;
               end if;
            when state3 =>
               if id < x"7" then
                  state <= state0;
               elsif id = x"9" then
                  state <= state4;
               else
                  state <= state3;
               end if;
            when state4 =>
               if id = x"b" then
                  state <= state0;
               else
                  state <= state4;
               end if;
         end case;
      end if;
   end process;
```

```
--assign state outputs;
y <= "00" when (state=state0) else
     "10" when (state=state1 or state=state3) else
     "11";
end archone_hot;
```

This implementation is almost the same as the first implementation, the only difference being the additional attribute which causes the state encoding to use one register for each state.

State Transition Tables

The final Moore implementation of this state machine uses a truth table. The state transition table can be found in the VHDL code.

The code follows:

```
entity ttf_fsm is port(
   clk, rst:in bit;
   id:    in bit_vector(0 to 3);
   y:     out bit_vector(0 to 1));
end ttf_fsm;

use work.table_bv.all;
architecture archttf_fsm of ttf_fsm is
   signal table_out: bit_vector(0 to 4);
   signal state: bit_vector(0 to 2);
constant state0:x01_vector(0 to 2) := "000";
constant state1: x01_vector(0 to 2) := "001";
constant state2: x01_vector(0 to 2) := "010";
constant state3: x01_vector(0 to 2) := "011";
constant state4: x01_vector(0 to 2) := "100";

constant table: x01_table(0 to 21, 0 to 11) := (
-- present state  inputs  nextstate  output
-- -------------  ------  ---------  ------
       state0  &  "xx0x" &  state0 &  "00",
       state0  &  "xxx0" &  state0 &  "00",
       state0  &  "0011" &  state1 &  "10",
```

```
        state1  &  "xxxx" &  state2 &  "11",
        state2  &  "1xxx" &  state2 &  "11",
        state2  &  "x0xx" &  state2 &  "11",
        state2  &  "xx0x" &  state2 &  "11",
        state2  &  "xxx0" &  state2 &  "11",
        state2  &  "0111" &  state3 &  "10",
        state3  &  "0111" &  state3 &  "10",
        state3  &  "1000" &  state3 &  "10",
        state3  &  "11xx" &  state3 &  "10",
        state3  &  "101x" &  state3 &  "10",
        state3  &  "0110" &  state0 &  "00",
        state3  &  "010x" &  state0 &  "00",
        state3  &  "00xx" &  state0 &  "00",
        state3  &  "1001" &  state4 &  "11",
        state4  &  "0xxx" &  state3 &  "10",
        state4  &  "100x" &  state3 &  "10",
        state4  &  "11xx" &  state4 &  "11",
        state4  &  "1010" &  state4 &  "11",
        state4  &  "1011" &  state0 &  "00");

begin
machine: process (clk, rst)
   begin
      if rst ='1' then
         table_out <= "00000";
      elsif (clk'event and clk='1') then
         table_out <= ttf(table,state & id);
      end if;
   end process;
state <= table_out(0 to 2);

--assign state outputs;
y <= table_out(3 to 4);
end archttf_fsm;
```

This implementation uses the ttf function, truth table function, which enables you to create a state transition table that lists the inputs, the current state, the next state, and the associated outputs. Within the architecture statement, a few signals and constants are defined. The signal called table_out is the vector which will contain the output from the state table. The signal

called state is the state variable itself. Six constants are defined which contain the state encoding - state0, state1, state2, state3, and state4 - and table which contains the entire state transition table. The table itself is created as an array with a certain number of rows designating the number of transitions, and a certain number of columns designating the number of input bits, present state bits, next state bits, and output bits.

Since the ttf function is not a standard part of VHDL, it has been defined in a separate package and provided to you as part of the *Warp* software. This package is located in the work library and is called table_bv. To allow your design to have access to the ttf function, you must add the statement "use work.table_bv.all;" to your VHDL description immediately above your architecture definition.

Most of the work is in creating the truth table, and the process becomes fairly simple. The first portion of the process defines the asynchronous reset. Next, the synchronous portion of the process (elsif clk'event and clk='1') is defined in which the signal table_out is assigned the returned value of the ttf function. The function is called with two parameters: the name of the state transition table, and the set of bits which contain the inputs and the present state information. The value that is returned is the remainder of the columns in the table (total number of columns - second parameter). These bits will contain the next state value and the associated outputs. The only task remaining is to split the state information from the output information and assign them to the appropriate signal names. Both of these assignments must occur outside of the process, otherwise another level of registers will be created, as this portion of the process defines synchronous assignments.

This design, as implemented, uses more registers than required, but it could easily be modified. Registers must be created for both the state registers and the output registers, as in the second implementation (outputs decoded in parallel). The truth table

can be modified such that the outputs are encoded in the state bits, as in the third example. Thus, rather than specifying both next state values and outputs, you can simply specify next state values in which the outputs are encoded.

Mealy state machines are characterized by the outputs which can change depending on the current inputs. We will implement the state machine shown in Figure 4-10 which has Moore outputs and one Mealy output. Figure 4-11 shows a block diagram of a Mealy machine.

The code follows:

```
entity mealy1 is port(
   clk, rst:in bit;
   id:    in bit_vector(3 downto 0);
   w:     out bit;
   y:     out bit_vector(1 downto 0));
end mealy1;

architecture archmealy1 of mealy1 is
   type states is (state0, state1, state2, state3, state4);
   signal state: states;
begin
moore: process (clk, rst)
   begin
      if rst='1' then
         state <= state0;
      elsif (clk'event and clk='1') then
         case state is
            when state0 =>
               if id = x"3" then
                  state <= state1;
               else
                  state <= state0;
               end if;
            when state1 =>
               state <= state2;
            when state2 =>
               if id = x"7" then
                  state <= state3;
```

```
                else
                    state <= state2;
                end if;
            when state3 =>
                if id < x"7" then
                    state <= state0;
                elsif id = x"9" then
                    state <= state4;
                else
                    state <= state3;
                end if;
            when state4 =>
                if id = x"b" then
                    state <= state0;
                else
                    state <= state4;
                end if;
        end case;
    end if;
end process;

--assign moore state outputs;
y <= "00" when (state=state0) else
     "10" when (state=state1 or state=state3) else
     "11";
--assign mealy output;
w <= '0' when (state=state3 and id < x"7") else
     '1';
end archmealy1;
```

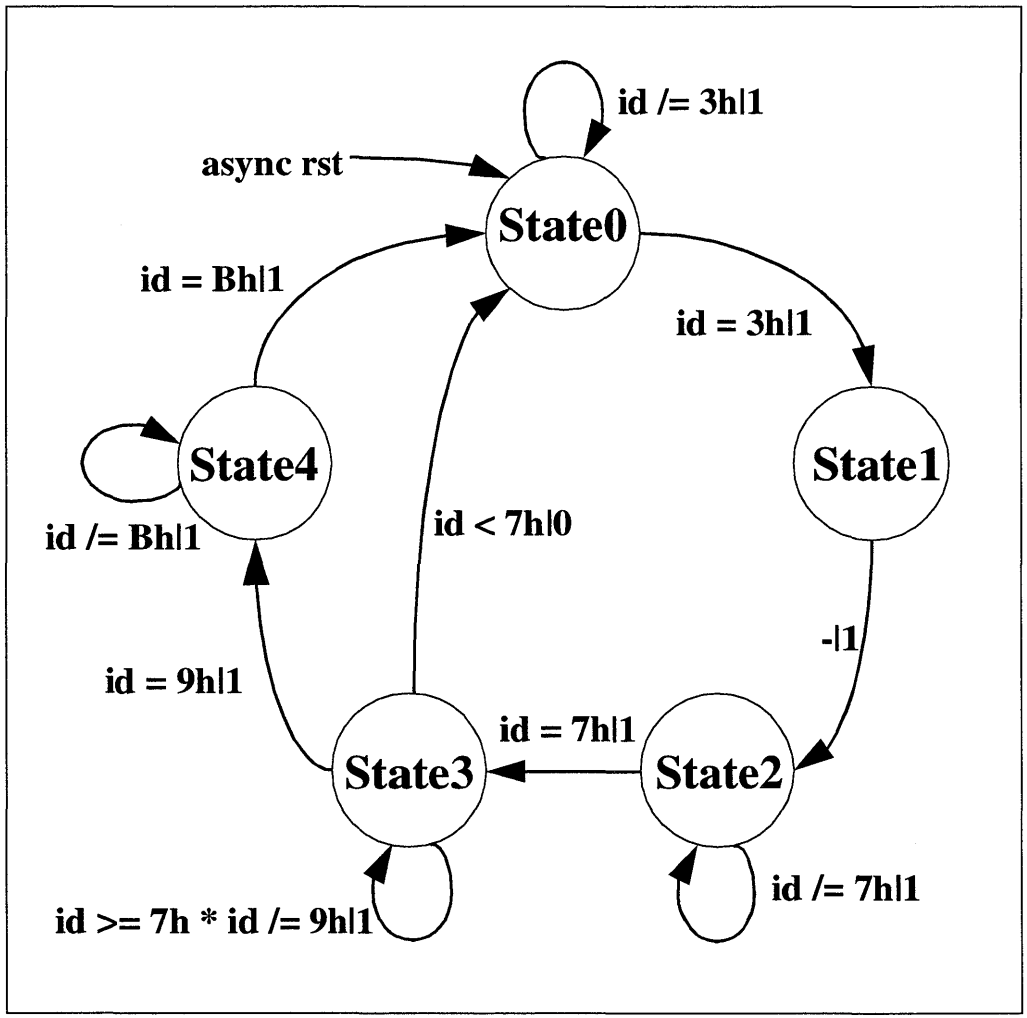


Figure 4-10. State Diagram for Combination Moore-Mealy State Machine

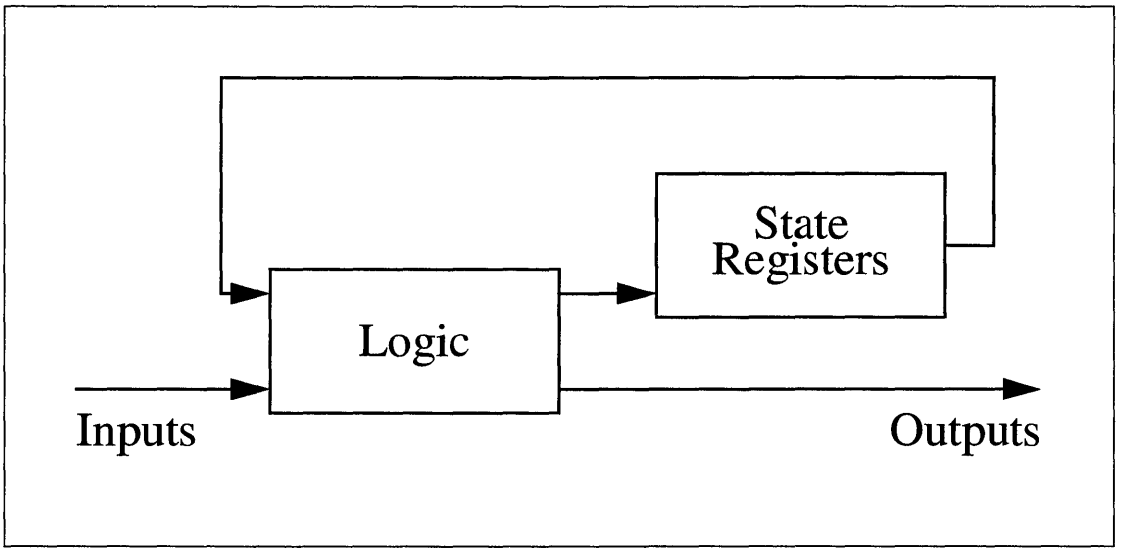


Figure 4-11. Block Diagram of Mealy State Machine

This implementation is almost identical to the first Moore implementation. The only difference is the additional Mealy output defined at the end of the architecture. We'll now take a look at a Mealy state machine, with all Mealy outputs.

Figure 4-12 is the state diagram. Two implementations follow.

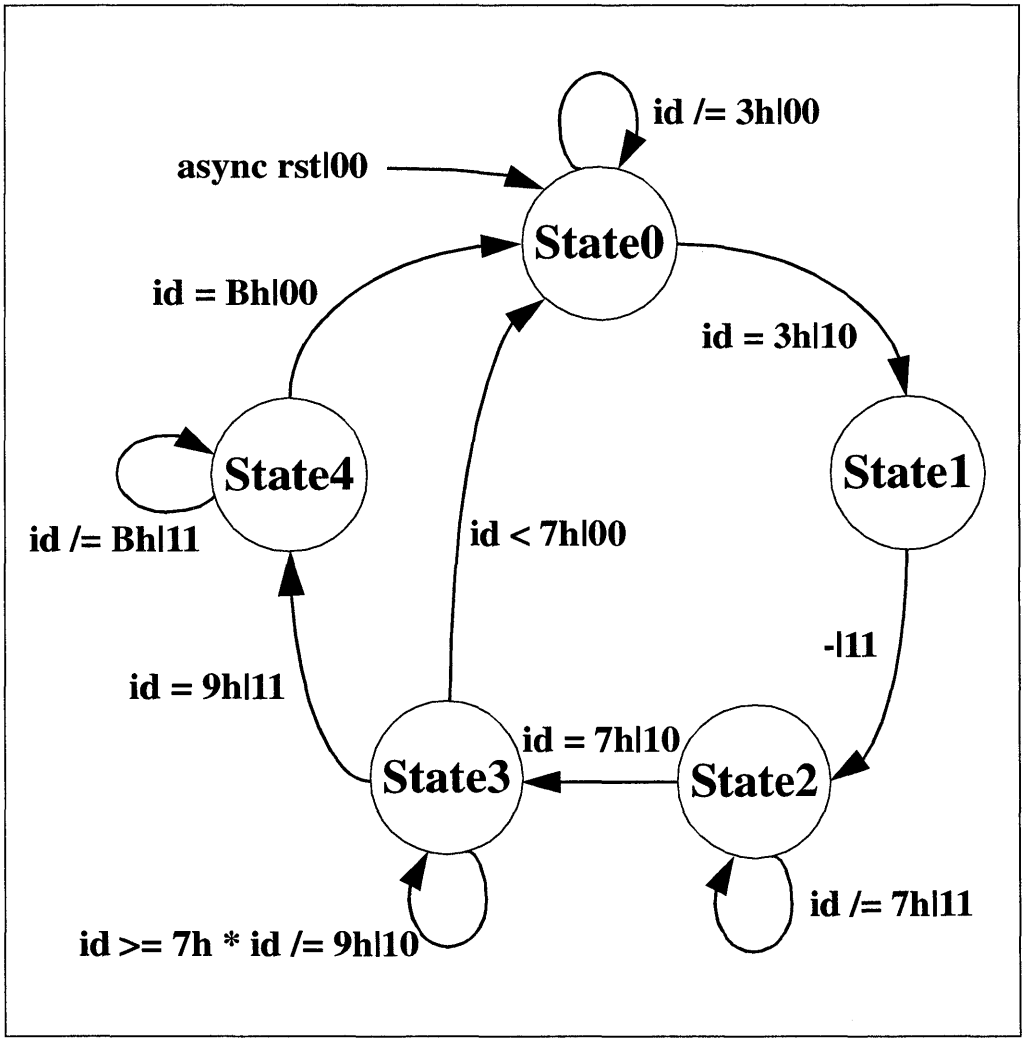


Figure 4-12. State Diagram for Second Mealy Machine

The following implementation specifies the state transitions in a

synchronous process, and the mealy outputs with a concurrent statement.

```
entity mealy1 is port(
        clk, rst:       in bit;
        id:             in bit_vector(3 downto 0);
        y:              out bit_vector(1 downto 0));
end mealy1;

architecture archmealy1 of mealy1 is
        type states is (state0, state1, state2, state3,
state4);
        signal state: states;
begin
machine: process (clk, rst)
        begin
                if rst='1' then
                        state <= state0;
                elsif (clk'event and clk='1') then
                        case state is
                                when state0 =>
                                        if id = x"3" then
                                            state <= state1;
                                        else
                                            state <= state0;
                                        end if;
                                when state1 =>
                                        state <= state2;
                                when state2 =>
                                        if id = x"7" then
                                            state <= state3;
                                        else
                                            state <= state2;
                                        end if;
                                when state3 =>
                                        if id < x"7" then
                                            state <= state0;
                                        elsif id = x"9" then
                                            state <= state4;
                                        else
                                            state <= state3;
                                        end if;
```

```
                                    when state4 =>
                                            if id = x"b" then
                                                state <= state0;
                                            else
                                                state <= state4;
                                            end if;
                            end case;
                    end if;
            end process;

--assign mealy output;
y <= "00" when ((state=state0 and id /= x"3") or
                (state=state3 and id < x"7") or
                (state=state4 and id = x"B")) else
     "10" when ((state=state0 and id = x"3") or
                (state=state2 and id = x"7") or
                (state=state3 and (id >= x"7") and
                  (id /= x"9"))) else
     "11";
end archmealy1;
```

This implementation of the Mealy state machine uses a synchronous process in much the same way as all of the other examples. An enumerated type is used to define the states. As in all but the one_hot coding implementation, you can choose your own state assignment, as in the third Moore implementation. The Mealy outputs in this implementation are define in a concurrent "when... else" construct. Thus, the output y is a function of the current state and the present inputs.

A second implementation of the same state machine follows. This implementation uses one synchronous process (in which the next state is captured by the state registers) and one combinatorial process in which the state transitions and Mealy outputs are defined.

```
entity mealy1 is port(
        clk, rst:       in bit;
        id:             in bit_vector(3 downto 0);
        y:              out bit_vector(1 downto 0));
```

```
end mealy1;

architecture archmealy1 of mealy1 is
        type states is (state0, state1, state2, state3,
state4);
        signal state, next_state: states;
begin
st_regs: process (clk, rst)
        begin
                if rst='1' then
                        state <= state0;
                elsif (clk'event and clk='1') then
                        state <= next_state;
                end if;
        end process;

mealy: process (id)
        begin
                case state is
                        when state0 =>
                                if id = x"3" then
                                        next_state <= state1;
                                         y <= "10";
                                else
                                        next_state <= state0;
                                         y <= "00";
                                end if;
                        when state1 =>
                                next_state <= state2;
                                y <= "11";
                        when state2 =>
                                if id = x"7" then
                                        next_state <= state3;
                                         y <= "10";
                                else
                                        next_state <= state2;
                                         y <= "11";
                                end if;
                        when state3 =>
                                if id < x"7" then
                                        next_state <= state0;
                                         y <= "00";
```

```
                                elsif id = x"9" then
                                        next_state <= state4;
                                         y <= "11";
                                else
                                        next_state <= state3;
                                         y <= "10";
                                end if;
                        when state4 =>
                                if id = x"b" then
                                        next_state <= state0;
                                         y <= "00";
                                else
                                        next_state <= state4;
                                         y <= "11";
                                end if;
                end case;
        end process;
end archmealy1;
```

In this implementation, the first process, "st_regs," captures the next state value. The second process, "mealy," defines the state transitions and the Mealy outputs. This second process is not synchronous and is activated each time the signal id transitions. Because the second process is not synchronous, the outputs can change even if the state doesn't, as would be expected in a Mealy state machine.

This concludes our discussion of state machines. Additional state machine, counter, and logic examples are documented in Section 4.10. We will move on to discuss hierarchical design, but first we'll discuss the concept of packages.

4.8. Packages

A package is a collection of declarations that make component, type, constant, and function declarations visible to more than one design.

A package can declare components (which are entity/architecture pairs), types, constants, or functions as a way to make these items visible in other designs.

The form of a package declaration is

```
PACKAGE package_name IS
   declarations
   END package_name;
```

Package declarations are typically used in *Warp* to declare types, constants, and components to be used by other VHDL descriptions. Most commonly, you put a package declaration (containing component declarations) at the beginning of a design file (before the entity/architecture pair definitions) in order to use the components in a subsequent or hierarchical design.

Packages which contain only components do not need a package body. However, if you intend to write VHDL functions to be used in multiple designs, then these functions must be declared in the package declaration as well as defined in a package body:

```
PACKAGE BODY package_name IS
   declarations
   END package_name;
```

A package body always has the same name as its corresponding package declaration, and is preceded by the reserved words PACKAGE BODY. A package body contains the function bodies whose declarations occur in the package declaration, as well as declarations that are not intended to be used by other VHDL descriptions.

The following example shows a package that declares a component named "demo," whose design (entity/architecture pair) follows the package declaration:

```
package demo_package is
   component demo
   port(x:out bit; clk, y, z:in bit);
   end component;
end package;

entity demo is
   port(x:out bit; clk, y, z:in bit);
end demo;

architecture fsm of demo is
.
.
.
end fsm;
```

If this description were in the file "demofile.vhd," you could analyze the package and add it to the current WORK library with the command

```
warp -a demofile
```

Items declared inside a package declaration are not automatically visible to another VHDL description. A use clause within a VHDL description makes items analyzed as part of a separate package visible within that VHDL design unit.

Use clauses may take one of three forms:

- USE library_name.package_name;
- USE package_name.object;
- USE library_name.package_name.object;

The portion of the USE clause argument preceding the final period is called the prefix; that after the final period is called the suffix.

Some examples of use clauses are:

```
LIBRARY project_lib;
USE project_lib.special_pkg;
USE project_lib.special_pkg.comp1;
```

The LIBRARY statement makes the library "project_lib" visible within the current VHDL design unit. The first USE clause makes a package called "special_pkg" contained within library "project_lib" visible within the current VHDL design unit. The second USE clause makes a component called "comp1," contained within "special_pkg," visible within the current VHDL design unit.

The suffix of the name in the use clause may also be the reserved word ALL. The use of this reserved word means that all packages within a specified library, or all declarations within a specified package, are to be visible within the current VHDL design unit. Some examples are:

```
USE project_lib.all;
```

This example makes all packages contained within library "project_lib" visible within the current VHDL design unit.

```
USE project_lib.special_pkg.all;
```

This example makes all declarations contained within package "special_pkg," itself contained within library "project_lib," visible within the current VHDL design unit.

Note the important difference between

```
USE project_lib.special_pkg;
```

and

```
USE project_lib.special_pkg.all;
```

The first USE clause just makes the package named special_pkg

within library project_lib visible within the current VHDL design unit. However, while the package name may be visible, ITS CONTENTS ARE NOT. The second USE clause makes all contents of package special_pkg visible to the current VHDL design unit.

Example

The following code defines a four bit counter:

```
package counter_pkg is
        subtype nibble is bit_vector(3 downto 0);
        component upcnt port(
                clk:    in bit;
                count:  buffer nibble);
        end component;
end counter_pkg;

use work.counter_pkg.all;
use work.int_math.all;

entity upcnt is port(
        clk: in bit;
        count: buffer nibble);
end upcnt;

architecture archupcnt of upcnt is
begin
counter:process (clk)
        begin
                if (clk'event and clk='1') then
                        count <= count + 1;
                end if;
        end process counter;
end archupcnt;
```

The package declaration will allow you to use the upcnt component and the type nibble in other designs. For example, suppose you needed five of these counters, but you did not want to write five separate process. You might prefer to simply

instantiate the upcnt counter defined above in a new design, creating a level of hierarchy. The code follows:

```
use work.counter_pkg.all;

entity counters is port(
        clk1, clk2:              in bit;
        acnt, bcnt, ccnt:        buffer nibble;
         deqe:                  out bit);
end counters;

architecture archcounters of counters is
signal dcnt, ecnt: nibble;
begin
counter1: upcnt port map (clk1, acnt);
counter2: upcnt port map (clk2, bcnt);
counter3: upcnt port map (clk => clk1, count => ccnt);
counter4: upcnt port map (clk2, dcnt);
counter5: upcnt port map (count => ecnt, clk => clk2);
deqe <= '1' when (dcnt = ecnt) else '0';
end archcounters;
```

The initial use clause makes the counter_pkg available to this design. Counter_pkg is required for the nibble definition used in the entity, and the upcnt component used in the architecture. Five counters are then instantiated by using the port map to associate the component I/O with the appropriate entity ports or architecture signals. Three of the instantiations use positional association in which the position of the signals in the port map determines to what I/O of the component the signal is associated. In counter3, and counter5, named association is used to explicitly define the signal to component I/O connections. In named association, the order of the signal assignment is not important.

When using Galaxy to compile and synthesize the counters design, the design file that contains upcnt must be compiled first, before counters can be compiled and synthesized. This is because the contents of the counter_pkg must first be added to the WORK library. Therefore, when selecting (in Galaxy) the Warp input

files to be compiled and synthesized, select the upcnt design as the first file and the counters design as the second. This will ensure that upcnt is compiled first. Once you have added the upcnt design to the current WORK library, you do not need to recompile it when synthesizing your top-level design unless you make changes to it or target your design to a different device.

4.8. Packages

4.8.1. Predefined Packages

Special purpose packages are provided with the Warp compiler to simplify description and synthesis of frequently used and useful structures and operations not native to VHDL.

The following packages are supplied standard with the *warp* compiler. These packages are found in \warp\lib\common.

Package Name	Purpose
bv_math	math operations with bit vector data types
int_math	math operations which mix integer and bit vector data types
table_bv	provides a state transition table format for description of state machines.
rtlpkg	provides a set of simple logic functions useful for creation hierarchical structural VHDL design files
mth34x8_pkg	defines 8-bit arithmetic components for use in designs targeted to CY7C340 family CPLDs.Implemented using CY7C340 architecture primitive elements.The mth34x12, mth34x16, and mth34x24 packages provide similar arithmetic components of widths 12, 16, and 24 respectively.

The USE statement is required in a design file to make the package "visible" to the design file. The USE statement should immediately precede the architecture and appear as follows:

```
USE work.package_name.all;
```

where package_name refers to one of the package names listed above.

Package Contents and Usage of bv_math

This package contains functions which allow arithmetic operations on bit vectors. VHDL is a strongly typed language which does not recognize the bit data type as a type compatible with arithmetic manipulation. This package contains functions which when invoked in the design file allow arithmetic operations on bit vectors.

Several of the functions in this package are implemented by "overloading" of the native VHDL operators for arithmetic operations on integers. Overloading is a scheme by which one or more functions can be called by use of the same conventional arithmetic operator symbol. The compiler will call the correct function by determining the data types of the arguments of the function call. Multiple functions can be represented by the same symbol as long as no two functions accepts the same combination of argument data types. By this means the "+" sign can be used, for example, to call a bit vector addition routine since the data type of the arguments (both of type bit vector) will signal that "+" should call the bit vector addition function.

The operators for which functions are provided are:

inc_bv(a) increment bit vector a. If function is assigned to a signal within a clocked process, the synthesized result will be an up counter. Equivalent to a <= a + 1;

Usage: a <= inc_bv(a);

dec_bv(a) decrement a bit vector a. If function is assigned to a signal within a clocked process, the synthesized result will be a down counter. Equivalent to a <= a - 1;.

Usage: a <= dec_bv(a);

+(a; b) regular addition function for two bit vectors a and b. The "+" operator overloads the existing "+" operator for definition for arithmetic operations on integers. The output vector is the same length as the input vector so there is no carry output. If a carry out is required the user should increase the length of the input vectors and use the MSB as the carry out.

Usage for two vectors of length 8 with carry out:

```
signal a: bit_vector(0 to 8);
signal b: bit_vector(0 to 8);
signal q: bit_vector(0 to 8);
q <= a + b;
```

+(a; b) regular addition function for adding to bit vector a the object b of type bit. This is the equivalent of a conditional incrementing of bit vector a. The "+" operator overloads the existing "+" operator for definition for arithmetic operations on integers. The output vector is the same length as the input vector so there is no carry output. If a carry out is required the user should increase the length of the input vector and use the MSB as the carry out.

Usage for 16 bit vector with no carry out:

```
signal a: bit_vector(0 to 15);
signal b: bit;
signal q: bit_vector(0 to 15);
 q <= a + b;
```

-(a; b) regular subtraction function for two bit vectors. The "-" operator overloads the existing "-" operator definition for arithmetic operations on integers.

Usage: signal a: bit_vector(0 to 7);

```
signal b: bit_vector(0 to 7);
```

```
signal q: bit_vector(0 to 7);
q <= a - b;
```

-(a; b) regular subtraction function for subtracting from bit vector a the object b of type bit. This is equivalent to the conditional decrementing of bit vector a. The "-" operator overloads the existing "-" operator definition for arithmetic operations on integers.

Usage: signal a: bit_vector(0 to 7);

```
signal b: bit;
signal q: bit_vector(0 to 7);
q <= a - b;
```

inv(b) unary invert function for object b of type bit. For use in port maps and sequential assignments.

Usage: signal b: bit;

```
signal z: bit;
z <= inv(b);
```

inv(a) invert function which inverts each bit of bit vector a and returns resulting bit vector.

Usage: signal a: bit_vector(0 to 15);

```
signal q: bit_vector(0 to 15);
q <= inv(a);
```

Package Contents and Usage of int_math

This package contains functions which allow mixed arithmetic operations on bit vectors and integers.

VHDL is a strongly typed language which does not recognize the bit data type as a type compatible with arithmetic manipulation. This package contains functions which when invoked in the

design file allow arithmetic operations which mix integers and bit vectors.

Several of the functions in this package are implemented by "overloading" of the native VHDL operators for arithmetic operations on integers. Overloading is a scheme by which one or more functions can be called by use of the same conventional arithmetic operator symbol. The compiler will call the correct function by noting the data types of the arguments of the function call. Multiple functions can be represented by the same symbol as long as no two functions accepts the same combination of argument data types. By this means the "+" sign can be used, for example, to call a routine for adding mixed data types (bit vector and integer) since the data type of the arguments will signal that "+" should call the package function for mixed addition rather than the native function for integer addition or the bit vector addition function of the above package "bv_math".

The operators for which functions are provided are:

bv2i(a) converts bit vector a to an integer.

Usage: variable z: integer range 0 to 15;

```
signal a: bit_vector(0 to 3);
z := bv2i(a);
```

i2bv(i; w) converts integer i to binary equivalent and expresses as a bit vector of length w.

Usage: variable i: integer range 0 to 31;

```
signal a: bit_vector(0 to 4);
a <= i2bv(i, 5);
```

i2bvd(i; w) converts integer i to a binary coded decimal bit vector of length w.

Usage: variable i: integer range 0 to 31;

```
signal a: bit_vector(0 to 7);
a <= i2bv(i, 8);
```

=(a; b) converts bit vector a to integer and checks for equality with integer b. Returns boolean value TRUE if equal, FALSE if not equal. Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 15);

```
variable b: range 0 to 64;
variable z: boolean;
z := (a = b);
```

/=(a; b) converts bit vector a to integer and checks for equality with integer b. Returns boolean value TRUE if not equal, FALSE if equal.Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 15);

```
variable b: range 0 to 128;
variable z: boolean;
z := (a /= b);
```

>(a; b) converts bit vector a to integer and compares with integer b. If a is > b, returns boolean value TRUE, otherwise returns FALSE. Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 15);

```
variable b: range 0 to 128;
variable z: boolean;
z := (a > b);
```

<(a; b) converts bit vector a to integer and compares with integer b. If a is < b, returns boolean value TRUE, otherwise returns FALSE. Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 15);

```
variable b: range 0 to 128;
variable z: boolean;
z:= (a < b);
```

>=(a; b) converts bit vector a to integer and compares with integer b. If a is >= b, returns boolean value TRUE, otherwise returns FALSE. Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 15);

```
variable b: range 0 to 128;
variable z: boolean;
z:= (a >= b);
```

<=(a; b) converts bit vector a to integer and compares with integer b. If a is <= b, returns boolean value TRUE, otherwise returns FALSE. Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 3);

```
variable b: range 0 to 128;
variable z: boolean;
z:= (a <= b);
```

+(a; b) increments bit vector a the number of times indicated by the value of integer b and returns bit vector result. Implemented by conversion of integer b to bit vector and adding to bit vector a. Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 15);

```
variable b: range 0 to 128;
signal z: bit_vector(0 to 15);
z <= a + b;
```

-(a; b) decrements bit vector a the number of times indicated by the value of integer b and returns bit vector result. Implemented by conversion of integer b to bit vector and subtracting from bit vector a. Overloads native operator for integer arithmetic.

Usage:signal a: bit_vector(0 to 15);

```
variable b: range 0 to 128;
signal z: bit_vector(0 to 15);
z <= a - b;
```

Package Contents and Usage of table_bv

The table_bv package describes a truth table function, ttf, which can be used to implement state transiting tables or other truth tables. A description and example of the ttf function can be found in Section 4.7.3, "Design Methodologies."

Package Contents and Usage of rtlpkg

The package rtlpkg contains VHDL component declarations for basic VHDL components which can be used to construct structural design files of more complex logic circuits. These components are useful for controlling implementation of the design by the WARP compiler to insure that specific performance or architecture choices are preserved in the final synthesized design. These components are generic components which can be used to describe retargetable designs which can be synthesized and fit to any desired Cypress device. The compiler makes the appropriate synthesis choices based on the target device's architectural resources to achieve the best possible utilization of the device and by preserving the specified interconnection of the declared components maintains the specific circuit implementation intended by the designer.

Components Contained In the Package rtlpkg are:

Name	Function
bufoe	bidirectional I/O with three state output driver y with type bit feedback to logic array (yfb)
dlatch	transparent latch with active low latch enable (e) (transparent high)
dff	positive edge triggered D-Type flip flop
xdff	positive edge triggered D-Type flip flop with XOR of two inputs (x1 & x2) feeding D input
jkff	positive edge triggered jk flip flop
buf	signal buffer to represent a signal not to be removed during synthesis to enable visibility during simulation.
srl	set/reset latch with reset dominant, set and reset active high
srff	positive edge triggered set/reset flip flop, reset dominant, set and reset active high
dsrff	positive edge triggered D-Type flip flop without asynchronous set and reset, reset dominant, set and reset active high
tff	toggle flip flop
xbuf	two input exclusive OR gate
triout	three state buffer with active high output enable input

Package Contents and Usage of mth34x8_pkg

The packages mth34x8_pkg, mth34x12, mth34x16, and mth34x24 contain VHDL component declarations for optimal

arithmetic components to be implemented in the CY7C34X family of devices. These components are useful for controlling implementation of arithmetic functions by the WARP compiler to insure that specific performance or architecture choices are preserved in the final synthesized design. There are four packages containing components that are functionally the same, but that operate on signals of different widths. The usage for these components is the same. The components included in the mth34x8pkg are as follows.

Name	Function
add_8	eight-bit adder
sub_8	eight-bit subtracter
gt_8	eight-bit greater than comparator
lt_8	eight-bit less than comparator
ge_8	eight-bit greater than or equal to comparator
le_8	eight-bit less than or equal to comparator

4.9. Libraries

A library is a collection of previously analyzed design elements (packages, components, entities, architectures) that can be referenced by other VHDL descriptions.

If all information about a design description had to appear in one file, many VHDL files would be huge and cumbersome, and information re-use would be impossible. However, VHDL allows you to share information between files by means of two constructs: libraries and packages.

In VHDL, a library is a collection of previously analyzed[1] design elements (packages, components, entities, architectures) that can be referenced by other VHDL descriptions. In *Warp*, a library is implemented as a directory, containing one or more VHDL files and an index to the design elements they contain.

To make the contents of a library accessible by a VHDL description, use a library clause. A library clause takes the form:

```
LIBRARY library_name [, library name...];
```

For example, the statement

```
LIBRARY gates, my_lib;
```

makes the contents of two libraries called gates and my_lib accessible in the VHDL description in which the LIBRARY clause is contained.

You seldom need to use a library clause in *Warp* VHDL

1. In VHDL, analysis is the examination of a VHDL description to guarantee compliance with VHDL syntax, and the extraction of design elements (packages, components, entities, architectures) from that description.
Synthesis is the production of a file (to be written onto a physical chip) that embodies the design elements extracted from the VHDL descriptions by analysis.

descriptions. This is because all VHDL implementations include a special library, named WORK. WORK is the symbolic name given to the current working library during analysis. The results of analyzing a VHDL description are placed by default into the WORK library for use by other analyses. (In other words, you don't need a LIBRARY clause to make the WORK library accessible.

4.10. Additional Design Examples

This section provides additional design examples not described in earlier sections of this chapter.

Many examples demonstrating design methodologies can be found in section 4.7 of this chapter. Most of these examples can be found in the /warp/examples directory. This section provides a discussion for additional design examples found in the \warp\examples directory but not discussed earlier in this chapter. These designs include:

Logic

- DEC24: a two-to-four bit decoder.
- PINS: shows how to use the part_name and pin_numbers attributes to map signals to pins.
- NAND2_TS: a two-input NAND gate with three-state output.

Counters

- CNT4_EXP: Four bit counter with synchronous reset. The counter uses expressions for clocks and resets.
- CNT4_REC: Four bit counter with load on the bidirectional pins. Demonstrates use of a record.

State Machines

- DRINK: a behavioral description of a mythical drink machine (the drinks only cost 30 cents!).
- TRAFFIC: a traffic-light controller.
- SECURITY: a simple security system.

4.10. Additional Design Examples

4.10.1. DEC24

This example demonstrates a two-to-four decoder.

Source Code:

```
-- two to four demultiplexer/decoder

ENTITY demux2_4 IS
   PORT(in0, in1: IN BIT;
    d0, d1, d2, d3: OUT BIT);
END demux2_4;

ARCHITECTURE behavior OF demux2_4 IS
BEGIN
   d0 <= (NOT(in1) AND NOT(in0));
   d1 <= (NOT(in1) AND in0);
   d2 <= (in1 AND NOT(in0));
   d3 <= (in1 AND in0);
END behavior;
```

Discussion:

The entity declaration specifies two input ports, in0 and in1, and four output ports, d0, d1, d2, and d3, all of type BIT.

The architecture specifies the various ways that the two inputs are combined to determine the outputs. This is one of several ways that a two-to-four decoder can be implemented.

4.10. Additional Design Examples

4.10.2. PINS

This example shows how to use the part_name and pin_numbers attributes to map signals to pins.

Source Code:

```
--Signals that are not assigned to pins can be automatically
--assigned pins by Warp. This design uses the C22V10-25DMB.
ENTITY and5Gate IS
   PORT (a: IN BIT_VECTOR(0 TO 4);
       f: OUT BIT);

ATTRIBUTE part_name of and5Gate:ENTITY IS "C22V10";
ATTRIBUTE pin_numbers of and5Gate:ENTITY IS
    "a(0):2 a(1):3 " --The spaces after 3 and 5 are necessary
    & "a(2):4 a(3):5 " --for concatenation (& operator)
    & "f:23";  --signal a(4) will be assigned a pin by warp
END and5Gate;

ARCHITECTURE see OF and5Gate IS
BEGIN
   f <= a(0) AND a(1) AND a(2) AND a(3) AND a(4);
END see;
```

Discussion:

Of particular importance in this example is the space just before the closing right-quote of each portion of the attribute value to be concatenated. As shown, this value resolves to;

```
a(0):2 a(1):3 a(2):4 a(3):5 f:23
```

Had the spaces not been included, this value would have been

```
a(0):2 a(1):3a(0):4 a(1):5f:23
```

which is an unrecognizable string.

4.10. Additional Design Examples

4.10.3. NAND2_TS

This example is a two-input NAND gate with three-state output.

Source Code:

```
--Two input NAND gate with three-state output
--This design is DEVICE DEPENDENT.

USE work.rtlpkg.all;--needed for triout

ENTITY threeStateNand2 IS
   PORT (a, b, outen: IN BIT;
      c: INOUT x01z);
END threeStateNand2;

ARCHITECTURE show OF threeStateNand2 IS
SIGNAL temp: BIT;

BEGIN
   temp <= a NAND b;
   tri1: triout PORT MAP (temp, outen, c);
END show;
```

Discussion:

This design is implemented by instantiating one triout component from rtlpkg. Temp is a signal created to be the input to the tristate buffer. Outen is the output enable, and c is the output (the NAND of signals a and b).

4.10. Additional Design Examples

4.10.4. CNT4_EXP

This example is a counter that uses expressions for clocks and resets.

Source Code:

```
-- Fits to a c344

USE work.bv_math.all;

ENTITY testExpressions IS
   PORT (clk1, clk2, res1, res2, in1, in2: IN BIT;
       count: BUFFER BIT_VECTOR(0 TO 3));
END testExpressions;

ARCHITECTURE cool OF testExpressions IS
SIGNAL clk, reset: BIT;

BEGIN
   clk <= clk1 AND clk2;--both clocks must be asserted;
   reset <= res1 OR res2; --either reset

proc1:PROCESS
   BEGIN
   WAIT UNTIL clk = '1';
      IF reset = '1' THEN
         count <= x"0";
      ELSE
         count <= inc_bv(count);
      END IF;
   END PROCESS;
END cool;
```

Discussion:

The entity declaration specifies two clock signals and two reset signals as external interfaces, as well as two input data ports and a four-bit bit vector for output.

The architecture declares two new signals, clk and reset, which are later defined to be the AND of clk1 and clk2 and the OR of reset1 and reset2, respectively. Both clocks must be asserted to detect a clock pulse and trigger the execution of the process. If either reset is asserted when a clock pulse is detected, the counter resets itself, else it increments by one and waits for the next clock pulse.

4.10. Additional Design Examples

4.10.5. CNT4_REC

This example is a four bit counter with load on the bidirectional pins, and demonstrates the use of a record.

Source Code:

```
-- loads on the i/o pins
-- temp is a RECORD used to simplify instantiating bufoe

USE work.bv_math.all;-- necessary for inc_bv();
USE work.rtlpkg.all;

ENTITY counter IS
   PORT (clk, reset, load, outen: IN BIT;
      count: INOUT x01z_VECTOR(0 TO 3));
END counter;

ARCHITECTURE behavior OF counter IS
TYPE bufRec IS -- record for bufoe
   RECORD          -- inputs and feedback
      cnt: BIT_VECTOR(0 TO 3);
      dat: BIT_VECTOR(0 TO 3);
END RECORD;
SIGNAL temp: bufRec;

CONSTANT counterSize: INTEGER:= 3;
BEGIN
g1:FOR i IN 0 TO counterSize GENERATE
      bx: bufoe PORT MAP(temp.cnt(i), outen, count(i),
temp.dat(i));
   END GENERATE;

proc1:PROCESS
   BEGIN
   WAIT UNTIL (clk = '1');
      IF reset = '1' THEN
         temp.cnt <= "0000";
      ELSIF load = '1' THEN
```

```
            temp.cnt <= temp.dat;
         ELSE
            temp.cnt <= inc_bv(temp.cnt); -- increment vector
         END IF;
      END process;
END behavior;
```

Discussion:

The entity declaration specifies that the design has four input bits (clk, reset, load, and outen) and a four-bit bit vector for output.

The architecture implements a counter with synchronous reset and load, and also demonstrates the use of RECORD types and the GENERATE statement.

4.10. Additional Design Examples

4.10.6. DRINK

This example a behavioral description of a mythical drink dispensing machine (the drinks only cost 30 cents!)

Source Code:

```
--In keeping with the fact that this is a mythical drink
--machine, the cost of the drink is 30 cents!

entity drink is port (
   nickel,dime,quarter,clock : in bit;
   returnDime,returnNickel,giveDrink: out bit);
end drink;

architecture fsm of drink is
   type drinkState is (zero,five,ten,fifteen,twenty,twenty-
five,owedime);
   signal drinkStatus: drinkState;
begin
   process begin
      wait until clock = '1';
      -- set up default values
      giveDrink <= '0';
      returnDime <= '0';
      returnNickel <= '0';
      case drinkStatus is
         when zero =>
            IF (nickel = '1') then
               drinkStatus <= Five;
            elsif (dime = '1') then
               drinkStatus <= Ten;
            elsif (quarter = '1') then
               drinkStatus <= TwentyFive;
            end if;
         when Five =>
            IF (nickel = '1') then
               drinkStatus <= Ten;
            elsif (dime = '1') then
```

```
            drinkStatus <= Fifteen;
         elsif (quarter = '1') then
            giveDrink <= '1';
            drinkStatus <= zero;
         end if;
      when Ten =>
         IF (nickel = '1') then
            drinkStatus <= Fifteen;
         elsif (dime = '1') then
            drinkStatus <= Twenty;
         elsif (quarter = '1') then
            giveDrink <= '1';
            returnNickel <= '1';
            drinkStatus <= zero;
         end if;
      when Fifteen =>
         IF (nickel = '1') then
            drinkStatus <= Twenty;
         elsif (dime = '1') then
            drinkStatus <= TwentyFive;
         elsif (quarter = '1') then
            giveDrink <= '1';
            returnDime <= '1';
            drinkStatus <= zero;
         end if;
      when Twenty =>
         IF (nickel = '1') then
            drinkStatus <= TwentyFive;
         elsif (dime = '1') then
            giveDrink <= '1';
            drinkStatus <= zero;
         elsif (quarter = '1') then
            giveDrink <= '1';
            returnNickel <= '1';
            returnDime <= '1';
            drinkStatus <= zero;
         end if;
      when TwentyFive =>
         IF (nickel = '1') then
            giveDrink <= '1';
            drinkStatus <= zero;
         elsif (dime = '1') then
```

```
                returnNickel <= '1';
                giveDrink <= '1';
                drinkStatus <= zero;
             elsif (quarter = '1') then
                giveDrink <= '1';
                returnDime <= '1';
                drinkStatus <= oweDime;
             end if;
          when oweDime =>
             returnDime <= '1';
             drinkStatus <= zero;
-- The following WHEN makes sure that the state machine
-- resets itself if it somehow gets into an undefined state.
          when others =>
             drinkStatus <= zero;
          end case;
       end process;
end fsm;
```

Discussion:

The entity declaration specifies that the design has four inputs: nickel, dime, quarter, and clock. The outputs are giveDrink, returnNickel, and returnDime. The last two outputs tell the design when to give change after the 30-cent price of the drink has been satisfied.

The architecture then defines an enumerated type with one value for each possible state of the machine, i.e., each possible amount of money deposited. Thus, the initial state of the machine is zero, while other states include five, ten, fifteen, etc.

After some initialization statements, the major part of the architecture consists of a large CASE statement, containing a WHEN clause for each possible state of the machine. Each WHEN clause contains an IF...THEN...ELSIF statement to handle each possible input and change of state.

4.10. Additional Design Examples

4.10.7. TRAFFIC

This example is a traffic-light controller.

Source Code:

```
-- This state machine implements a simple traffic light.
-- The N - S light is usually green, and remains green
-- for a minimum of five clocks after being red. If a
-- car is travelling E-W, the E-W light turns green for
-- only one clock.

ENTITY traffic_light IS
   PORT(clk, car: IN BIT;--car is an E-W travelling car
    lights: BUFFER BIT_VECTOR(0 TO 5));
END traffic_light;

ARCHITECTURE moore1 OF traffic_light IS
-- The lights (outputs) are encoded in the following states.
-- For example, the
-- state green_red indicates the N-S light is green and the
-- E-W light is red.
-- "001" indicates green light, "010" yellow, "100" red;
-- "&" concatenates
   CONSTANT green_red : BIT_VECTOR(0 TO 5) := "001" & "100";
   CONSTANT yellow_red : BIT_VECTOR(0 TO 5) := "010" & "100";
   CONSTANT red_green : BIT_VECTOR(0 TO 5) := "100" & "001";
   CONSTANT red_yellow : BIT_VECTOR(0 TO 5) := "100" & "010";

-- nscount to verify five consecutive N-S greens
   SIGNAL nscount: INTEGER RANGE 0 TO 5;

BEGIN
   PROCESS
   BEGIN
      WAIT UNTIL clk = '1';
      CASE lights IS
         WHEN green_red =>
```

```
            IF nscount < 5 THEN
               lights <= green_red;
               nscount <= nscount + 1;
            ELSIF car = '1' THEN
               lights <= yellow_red;
               nscount <= 0;
            ELSE
               lights <= green_red;
            END IF;
         WHEN yellow_red =>
            lights <= red_green;
         WHEN red_green =>
            lights <= red_yellow;
         WHEN red_yellow =>
            lights <= green_red;
         WHEN others =>
            lights <= green_red;
      END CASE;
   END PROCESS;
END moore1;
```

Discussion:

The states in this example are defined such that the outputs are encoded in the state, using red/yellow/green triplets for each of the north-south and east-west light. For example, if the north-south light is red and the east-west light is green, then the state encoding is "100001".

In this design, the north-south light remains green for a minimum of five clock cycles, while the east-west light only remains green for one clock cycle. Note the use of signal nscount to keep track of the number of clock cycles the north-south light has remained green. This is less confusing than creating five extra states that do basically nothing.

4.10. Additional Design Examples

4.10.8. SECURITY

This example is a simple security system.

Source Code:

```
ENTITY securitySystem IS
   PORT (set, intruder, clk: IN BIT;
      horn: OUT BIT);
END securitySystem;

ARCHITECTURE behavior OF securitySystem IS
TYPE states IS (securityOff, securityOn, securityBreach);
SIGNAL state, nextState: states;

BEGIN
PROC1:PROCESS (set, intruder)
   BEGIN
      CASE state IS
      WHEN securityOff =>
         IF set = '1' THEN
            nextState <= securityOn;
         END IF;
      WHEN securityOn =>
         IF intruder = '1' THEN
            horn <= '1';  --Mealy output
            nextState <= securityBreach;
         ELSIF set = '0' THEN
            horn <= '0';
            nextState <= securityOff;
         END IF;
      WHEN securityBreach =>
         IF set = '0' THEN
            horn <= '0';
            nextState <= securityOff;
         END IF;

      WHEN others =>
```

```
            nextState <= securityOff;
         END CASE;
      END PROCESS;
   proc2:PROCESS
      BEGIN
         WAIT UNTIL clk ='1';
         state <= nextState;
      END PROCESS;

   END behavior;
```

Discussion:

The entity declaration specifies that the design has three inputs (set, intruder, and clk) and one output (horn), all of type bit.

The architecture declares an enumerated type with three possible values: securityOn, securityOff, and securityBreach. It also declares two state variables, named state and nxtState.

The rest of the architecture defines two concurrent processes that interact via the nextState signal. The first process is activated whenever a change occurs in the set or intruder signals, and defines what the new state of the machine will be as of the next clock signal. The second is activated with each rising clock pulse.

Chapter 5

Warp VHDL Reference

Warp VHDL Reference

5.1. Introduction

This chapter provides an encyclopedic reference to each VHDL language element that *Warp* supports.

This chapter shows the syntax of each language element, explains the purpose of the language element, and gives an example of its use.

5.2. ALIAS

ALIAS lets you define an alternate name by which to reference a VHDL object. Use ALIAS to create a shorter reference to a long object name, or to provide a mnemonic reference to a name that may be difficult to remember otherwise.

Syntax

```
alias identifier[:subtype_indication] is name;
```

Discussion

Identifier is the alias for *name* in the alias declaration. An alias of a signal denotes a signal; an alias of a variable denotes a variable; an alias of a constant denotes a constant.

An alias of an object can be updated if and only if the object itself can be updated. Thus, an alias for a constant or for a port of mode **in** cannot be updated.

An alias may be constrained to a sub-type of the object specified in *name*, but *identifier* and *name* must have the same base type.

Example

```
signal Instrctn:Bit_vector(15 downto 0);
alias Opcode:Bit_vector(3 downto 0) is
Instrctn(15 downto 12);
alias Op1:Bit_vector(5 downto 0) is Instrctn(11 downto 6);
alias Op2:Bit_vector(5 downto 0) is Instrctn(5 downto 0);
alias Sign1:Bit is Op1(5);
alias Sign2:Bit is Op2(5);
```

The first line of this example declares a signal called Instrctn, containing 16 bits. Succeeding lines define several aliases from sub-elements of this bit vector: two six-bit operands (Op1 and Op2) and two sign bits (Sign1 and Sign2). The alias declarations for Sign1 and Sign2 make use of previously declared aliases.

5.3. ARCHITECTURE

An architecture (or, more formally, an "architecture body") describes the internal view of an entity, i.e., it specifies the functionality or the structure of the entity.

Syntax

```
architecture name of entity is
   architecture_declarations;
   begin
      concurrent_statements;
   end [name];

architecture_declaration ::=
   subtype_declaration
   | constant_declaration
   | signal_declaration
   | component_declaration
   | attribute_specification

concurrent_statement ::=
   process_statement
   | concurrent_signal_assignment_statement
   | component_instantiation_statement
   | generate_statement
```

Discussion

Architectures describe the behavior, data flow, or structure of an accompanying entity. (See Section 5.8, "ENTITY"for more information about entities.)

Architectures start with the keyword `architecture`, followed by a name for the architecture being declared, the keyword `of`, the name of the entity to which the architecture is being bound, and the keyword `is`.

A list of architecture declarations follows. This list declares components, signals, types, constants, and attributes to be used in the architecture. If a USE clause appears before the architecture, any elements referenced by the USE clause need not be re-declared.

The architecture body follows, consisting of component instantiation statements, generate statements, processes, and/or concurrent signal assignment statements.

In practice, architectures in *Warp* perform one of the following functions:

1. they describe the behavior of an entity; or
2. they describe the data flow of an entity; or
3. they describe the structure of an entity.

Examples of each of these uses of an architecture are given in section 4.5, Operators and section 4.10, Additional Design Examples.

5.4. ATTRIBUTE

An attribute is a property that can be associated with an entity, architecture, label, or signal in a VHDL description. This property, once associated with the entity, architecture, label, or signal, can be assigned a value, which can then be used in expressions.

Syntax (Attribute Declaration)

```
attribute attribute-name:type;
```

Syntax (Attribute Specification)

```
attribute attribute-name
   of name-list:name-class is expression;
```

Syntax (Attribute Reference)

```
item-name'attribute-name
```

Discussion

Attributes are constants associated with names. When working with attributes, it is helpful to remember the following order of operations: declare—specify—reference:

1. declare the attribute with an attribute declaration statement;
2. associate the attribute with a name and give the attribute a value, with an attribute specification statement;
3. reference the value of the attribute in an expression.

VHDL contains pre-defined and user-defined attributes.

Pre-defined attributes are part of the definition of the language. *Warp* supports a subset of these attributes that relate to synthesis operations. This subset is discussed in section 5.4.1.

User-defined attributes are additional attributes that annotate VHDL models with information specific to the user's application. Several user-defined attributes are supplied with *Warp* to support synthesis operations. These attributes are discussed in sections 5.4.2 through 5.4.12.

Declaring New Attributes

To declare a new attribute, use an attribute declaration:

```
attribute smart is boolean;
attribute charm is range 1 to 10;
```

This example declares two attributes. The first is called smart, of type boolean. The second is called charm, and has as possible values the integers 1 through 10, inclusive.

Associating Attributes With Names

To associate an attribute with a name and assign the attribute a value, use an attribute specification:

```
attribute smart of sig1:signal is true;
attribute charm of ent1:entity is 5;
```

This example associates the attribute smart with signal sig1, and assigns smart a value of TRUE, then associates the attribute charm with entity ent1 and assigns charm a value of 5.

Referencing Attribute Values

To use the value of an attribute in an expression, use an attribute reference:

```
if (sig1'smart = TRUE) then a <= 1 else a<= o;
```

This example tests the value of the attribute "smart" for signal sig1, then assigns a value to signal a depending on the result of the test.

5.4. ATTRIBUTE

5.4.1. Pre-defined Attributes

Warp supports a large set of pre-defined attributes, including value, function, type, and range attributes.

Table 5-1 lists the pre-defined attributes that *Warp* supports:

- Value attributes operate on items of scalar type or subtype.
- Function attributes operate on types, objects, or signals.
- Type attributes operate on types.
- Range attributes operate on constrained (i.e., bounded) array types.

Value Attributes

All scalar types or subtypes have the following value attributes:

- 'LEFT: returns the leftmost value in the type declaration.
- 'RIGHT: returns the rightmost value in the type declaration.
- 'HIGH: returns the highest value in the type declaration. For enumerated types, this is the rightmost value. For integer sub-range types, it's the value of the highest integer in the range. For other sub-range types, it's the rightmost value if the type declaration uses the keyword "to"; it's the leftmost value if the type declaration uses the keyword "downto".

Table 5-1: Pre-defined Attributes Supported by *Warp*.

Value Attributes		
'Left	'Right	'High
'Low	'Length	
Function Attributes (types)		
'Pos	'Val	'Succ
'Pred	'Leftof	'Rightof
Function Attributes (objects)		
'Left	'Right	'High
'Low	'Length	
Function Attributes (signals)		
'Event		
Type Attributes		
'Base		
Range Attributes		
'Range	'Reverse_range	

- 'LOW: returns the lowest value in the type declaration. For enumerated types, this is the leftmost value. For integer sub-range types, it's the value of the lowest integer in the range. For other sub-range types, it's the leftmost value if the type declaration uses the keyword "to"; it's the rightmost value if the type declaration uses the keyword "downto".

Constrained array types have the following value attribute:

- 'LENGTH(N): returns the number of elements in the N'th dimension of the array.

Constrained array objects also use these same attributes. For objects, the attributes are implemented in VHDL as functions instead of value attributes.

Examples:

For the following type declarations:

```
type countup is range 0 to 10;
type countdown is range 10 downto 0;
type months is (JAN,FEB,MAR,APR,MAY,JUN,
                JUL,AUG,SEP,OCT,NOV,DEC);
type Q1 is months range MAR downto JAN;
```

the value attributes are:

```
countup'left = 0                countdown'left = 10
countup'right = 10              countdown'right = 0
countup'low = 0                 countdown'low = 0
countup'high = 10               countdown'high = 10
countup'length = 11             countdown'length = 11

months'left = JAN               Q1'left = MAR
months'right = DEC              Q1'right = JAN
months'low = JAN                Q1'low = JAN
months'high = DEC               Q1'high = MAR
months'length = 12              Q1'length = 3
```

Function Attributes (Types)

All discrete (i.e., "ordered") types and their subtypes have the following function attributes:

- 'POS(V): returns the position number of the value V in the list of values in the declaration of the type.
- 'VAL(P): returns the value that corresponds to position P in the list of values in the declaration of the type.
- 'SUCC(V): returns the value whose position is one larger than that of value V in the list of values in the declaration of the type.
- 'PRED(V): returns the value whose position is one smaller than that of value V in the list of values in the declaration of the type.
- 'LEFTOF(V): returns the value whose position is immediately to the left of that of value V in the list of values in the declaration of the type. For integer and enumerated types, this is the same as 'PRED(V). For sub-range types, this is the same as 'PRED(V) if the type was declared using the keyword "to"; it is the same as 'SUCC(V) if the type was declared using the keyword "downto".
- 'RIGHTOF(V): returns the value whose position is immediately to the right of that of value V in the list of values in the declaration of the type. For integer and enumerated types, this is the same as 'SUCC(V). For sub-range types, this is the same as 'SUCC(V) if the type was declared using the keyword "to"; it is the same as 'PRED(V) if the type was declared using the keyword "downto".

Examples:

For the following type declarations (the same as those used in the previous example set):

```
type countup is range 0 to 10;
type countdown is range 10 downto 0;
type months is (JAN,FEB,MAR,APR,MAY,JUN,
                JUL,AUG,SEP,OCT,NOV,DEC);
type Q1 is months range MAR downto JAN;
```

the function attributes are:

```
countup'POS(0) = 0
countup'POS(10) = 10
countup'VAL(1) = 1
countup'VAL(9) = 9
countup'SUCC(4) = 5
countup'PRED(4) = 3
countup'LEFTOF(4) = 3
countup'RIGHTOF(4) = 5

months'POS(JAN) = 1
months'POS(DEC) = 12
months'VAL(1) = JAN
months'VAL(12) = DEC
months'SUCC(FEB) = MAR
months'PRED(FEB) = JAN
months'LEFTOF(FEB) = JAN
months'RIGHTOF(FEB) = MAR
```

```
countdown'POS(10) = 0
countdown'POS(0) = 10
countdown'VAL(1) = 9
countdown'VAL(9) = 1
countdown'SUCC(4) = 3
countdown'PRED(4) = 5
countdown'LEFTOF(4) = 5
countdown'RIGHTOF(4) = 3

Q1'POS(JAN) = 1
Q1'POS(MAR) = 3
Q1'VAL(1) = MAR
Q1'VAL(12) = error
Q1'SUCC(FEB) = MAR
Q1'PRED(FEB) = JAN
Q1'LEFTOF(FEB) = MAR
Q1'RIGHTOF(FEB) = JAN
```

Function Attributes (Objects)

All constrained (i.e., bounded) array objects have the following function attributes:

- 'LEFT(N): returns the left bound of the Nth dimension of the array object.

- 'RIGHT(N): returns the right bound of the Nth dimension of the array object.
- 'LOW(N): returns the lower bound of the Nth dimension of the array object. This is the same as 'LEFT(N) for ascending ranges, 'RIGHT(N) for descending ranges.
- 'HIGH(N): returns the upper bound of the Nth dimension of the array object. This is the same as 'RIGHT(N) for ascending ranges, 'LEFT(N) for ascending ranges.

In the discussion above, the value of N defaults to 1, which is also the lower bound for the number of dimensions in an array.

Examples:

For the following type and variable declarations:

```
type two_d_array is array (8 downto 0, 0 to 4);
variable my_array:two_d_array;
```

the function attributes are:

```
my_array'left(1)= 8          my_array'left(2) = 0
my_array'right(1) = 0        my_array'right(2) = 4
my_array'low(1) = 0          my_array'low(2) = 0
my_array'high(1) = 8         my_array'high(2) = 4
```

Function Attributes (Signals)

Warp supports a single function attribute for signals, namely the 'EVENT attribute. 'EVENT is a boolean function that returns TRUE if an event (i.e., change of value) has just occurred on the signal.

Warp supports the 'EVENT attribute only for clock signals such as in the following example.

Example:

```
PROCESS BEGIN
   WAIT UNTIL (clk'EVENT AND clk='1');
   .
   .
   .
END PROCESS;
```

This example shows a process whose statements are executed when an event occurs on signal clk and signal clk goes to '1'.

Type Attributes

All types and subtypes have the following attribute:

- 'BASE: returns the base type of the original type or subtype.

At first glance, this attribute doesn't appear very useful in expressions, since it returns a type. But it can be used in conjunction with other attributes, as in the following examples.

Examples:

For the following type declarations:

```
type day_of_week is (SUN,MON,TUE,WED,THU,FRI,SAT);
subtype work_day is day_of_week range MON to FRI;
```

the following value attributes are true:

```
work_day'left = MON        work_day'BASE'left = SUN
work_day'right = FRI       work_day'BASE'right = SAT
work_day'low = MON         work_day'BASE'low = SUN
work_day'high = FRI        work_day'BASE'high = SAT
work_day'length = 5        work_day'BASE'length = 7
```

Range Attributes

Constrained array objects have the following attributes:

- 'RANGE(N): returns the range of the Nth index of the array. If N is not specified, it defaults to 1.
- 'REVERSE_RANGE(N): returns the reversed range of the Nth index of the array. If N is not specified, it defaults to 1.

The range attributes give you a way to parameterize the limits of FOR loops and FOR-GENERATE statements, as in the following example.

Example:

Consider a variable declared as:

```
variable my_bus:bit_vector(0 to 7);
```

Then, the value of the 'RANGE and 'REVERSE_RANGE attributes for my_bus are:

```
my_bus'RANGE = 0 to 7
my_bus'REVERSE_RANGE = 7 downto 0
```

You could use this attribute in a FOR loop, like this:

```
for index in my_bus'REVERSE_RANGE loop
.
.
.
end loop;
```

5.4. ATTRIBUTE

5.4.2. dont_touch

The dont_touch attribute is used when targeting pASICs to specify that a component is to pass through synthesis and optimization untouched. Using this attribute allows you to "freeze" the structural implementation of an optimized component, such as a hand-tuned carry-select adder.

NOTE

The dont_touch attribute has no effect when the target device is not a pASIC.

Syntax

```
attribute dont_touch of label-name:label is true;

OR

attribute dont_touch of entity-name:entity is true;
```

The dont_touch attribute takes the value TRUE or FALSE. The default is FALSE.

When the dont_touch attribute is set to TRUE for an entity or a component instance, the structural implementation of that entity/component is not modified by subsequent synthesis, or during Level 1 optimization within SpDE. Setting the dont_touch attribute to TRUE is similar to using Level 0 optimization on a component or entity, in that very little optimization or packing is done. This allows hand-optimized portions of the design to stay untouched within SpDE while the rest of the design is optimized and packed with SpDE's Level 1 optimization.

When using the dont_touch attribute, structural or schematic designs must resolve to pASIC primitives (not equations). These primitives are PAfrag_a, PAfrag_f, PAfrag_m, PAfrag_q, PAlcell and logico, which constitute portions or the pASIC logic cell.

Note, however, that the dont_touch attribute does not apply to packing. For example, suppose you have a schematic or a structural implementation that uses a PAfrag_a, PAfrag_m, PAfrag_f, and PAfrag_q, and the dont_touch attribute is set to TRUE on the entire schematic or on all the individual instances. Even if components are ideally connected to each other so that they can be packed together, it is not guaranteed that these four frags will pack into a single logic cell (although it is highly likely that they would). To gain control over the packing of such schematics, higher-level elements like PAlcell and logico should be used. PAlcell represents the whole logic cell. Logico represents the whole logic cell except the flip-flop portion, and has only one output.

Another important use of the dont_touch attribute is for buffering high fanout nets, or for special buffering situations. Sometimes, the logic optimizer inadvertently removes gates that the user intended for buffering. Placing the dont_touch attribute forces such gates to be preserved. Buffering is also best done using the pASIC primitives.

Higher level library elements available from the math, counter, shifter, pasic, mux, and memory packages, and certain large gates (like AND14I7, NAND13I6, OR13I6,NOR14I7, SOP14I7) have already been hand-optimized and use the highest level pASIC primitives possible. However, these elements do not have the dont_touch attribute set within them. In most instances, you will get better performance and density if the dont_touch attribute is set to TRUE on these components. However, when very small combinational equations feed the inputs of these components, it might be better not to set the dont_touch attribute to TRUE, to give the optimizer/packer in SpDE more

freedom to insert the input logic into the logic cells of the component. Also, please note that most simple gates are not implemented at a frag level, to allow better packing and optimization by SpDE.

5.4. ATTRIBUTE

5.4.3. enum_encoding

The enum_encoding attribute lets you specify the internal encoding to be used for each value of a user-defined enumerated type. The internal encoding is reflected in the gate-level design when targeting a device.

Syntax

```
attribute enum_encoding of type-name:type is "string";
```

The enum_encoding attribute takes a single argument, consisting of a string of 0's and 1's separated by white space (spaces or tabs). Each contiguous string of 0's and 1's represents the encoding for a single value of the enumerated type. The number of contiguous strings in the enum_encoding argument must equal the number of values in the enumerated type.

When the enum_encoding attribute is included in a *Warp* description, it overrides the value of a state_encoding attribute appearing in the same description.

Example:

```
type state is (s0,s1,s2,s3);
attribute enum_encoding of
 state:type is "00 01 10 11";
```

The first statement in this example declares an enumerated type, called "state," with four possible values. The possible state values of type state can therefore be represented in two bits. The second statement specifies the internal representation of each value of type "state". Value s0's internal representation is "00". Value s1's internal representation is "01". Value s2's internal representation is "10". Value s3's internal representation is "11".

5.4. ATTRIBUTE

5.4.4. fixed_ff

The fixed_ff attribute is used when targeting pASICs to assign a signal to a specific internal register. This fixed placement overrides the default placement assigned by the SpDE Placer.

Syntax

```
attribute fixed_ff of signal-name:signal is "register-name";
```

The fixed_ff attribute is similar to the pin_numbers attribute, in that it locks a signal to a specific fixed placement. The difference is that fixed_ff is for fixed internal placement, while the pin_numbers attribute is for fixed external placement.

A given signal could have both a pin number and a fixed internal flip-flop placement. For instance, you may want to fix the output of a register to internal cell A1, and also have that output signal fixed to the output pad attached to pin 59 of the chip.

The fixed_ff attribute only applies to the Q output signal from a register. If the fixed_ff attribute is attached to any other signal besides the Q output of a register, it is ignored.

5.4. ATTRIBUTE

5.4.5. ff_type

The ff_type attribute specifies the flip-flop type used to synthesize individual signals.

Syntax

```
attribute ff_type of signal-name:signal is value;
```

Legal values for the ff_type attribute are ff_d, ff_t, ff_opt, and ff_default.

- A value of ff_d tells *Warp* to synthesize the signal as a D-type flip-flop.
- A value of ff_t tells *Warp* to synthesize the signal as a T-type flip-flop.
- A value of ff_opt tells *Warp* to synthesize the signal to the "optimum" flip-flop type (i.e., the one that uses the fewest resources on the target device).
- A value of ff_default tells *Warp* to synthesize the signal based on the default flip-flop type selection strategy, determined by the command line switches or dialog box settings used in invoking *Warp*.

Example:

```
attribute ff_type of abc:signal is ff_opt;
```

The command above tells *Warp* to optimize the flip-flop type used to synthesize a signal named "abc".

5.4. ATTRIBUTE

5.4.6. node_num

The node_num attribute tells *Warp* to map an individual signal to the pin on the target device that it determines would fit best.

Syntax

```
attribute node_num of signal-name:signal is integer;
```

The node_num attribute has a single legal value, nd_auto. Assigning this attribute to a signal tells *Warp* to map the signal to the pin on the target device it determines would fit best.

When you map a signal using this method, you need not include the signal's name in the port map of a binding architecture file. The node_num attribute of a signal overrides the port map's pin specification.

Example:

```
attribute node_num of hoohah:signal is nd_auto;
```

The command above maps a signal named hoohah to a *Warp*-determined pin on the target device.

```
attribute node_num of hoohah:signal is 23;
```

The command above maps a signal named hoohah to a specific node within the device being targeted. This value is both device- and package-specific, and may not be portable to other packages or devices.

5.4. ATTRIBUTE

5.4.7. order_code

The order_code attribute tells *Warp* which device package to use when synthesizing a design for a target device.

Syntax

```
attribute order_code of entity-name:entity is "order-code";
```

The order_code attribute specifies the package for a chip. The package name tells *Warp* the pin names and pin ordering for the device and package that you are targeting.

Legal order codes can be found in the "Ordering Code" column of the ordering information table for each device in the *Cypress Semiconductor Data Book*, or in Appendix B of this manual.

Example:

```
attribute order_code of mydesign:entity is "PALC22V10-25HC";
```

This example specifies a package type of "PALC22V10-25HC" for the entity named "mydesign."

5.4. ATTRIBUTE

5.4.8. part_name

The part_name attribute specifies the device to target for synthesis.

Syntax

```
attribute part_name of entity-name:entity is "part-name";
```

The part_name attribute tells *Warp* what part you're targeting for synthesis. If this attribute is present, it overrides the target device specified by a binding architecture, a command line switch, or a Galaxy dialog box setting.

Example:

```
attribute part_name of my_design:entity is "C335";
```

This examples specifies the C335 as the target device for synthesis.

5.4. ATTRIBUTE

5.4.9. pin_numbers

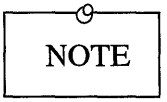
The pin_numbers attribute maps the external signals of an entity to pins on the target device.

Syntax

```
attribute pin_numbers of entity-name:entity is string;
```

The pin_numbers attribute maps the external signals (ports) of an entity with pin numbers on a target device.

The string used in the Attribute statement consists of one or more pairs of the form `signal-name:number`. Pairs must be separated from each other by white space (spaces or tabs). This string can consist of several smaller, concatenated strings.

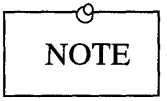


If the string contains an embedded line break (carriage return or line feed), a VHDL syntax error may result. Thus, for target devices with lots of pins, it may be more convenient to express the signal-to-pin mapping as a series of concatenated strings, making sure to leave a space between successive concatenated sub-strings.

Example

```
attribute pin_numbers of my_design:entity is
        "x:1 y:2 clk:3 a(0):4";
```

This example maps four signals from an entity called "mydesign" onto the pins of a target device. Signal x is mapped to pin 1, signal y to pin 2, signal clk to pin 3, and signal a(0) to pin 4.



When targeting pASICs: you can use the pin_numbers attribute to assign an input signal to more than one high-drive pad in order to give the signal a higher drive strength. Just separate the pin numbers by commas within the attribute string, e.g., to assign a signal named in1 to pins 2 and 3 of a pASIC, you would write

```
attribute pin_numbers of my_design:entity is
    "in1:2,3"
```

You can use this feature to assign a signal to any desired combination of input and input/clock pins. Be sure that the pin numbers specified in the attribute string match the input and clock pins of the actual device.

```
attribute pin_numbers of my_design:entity is
          "sig1:1 " &
          "sig2:2 " &
          "sig3:3 " &
          "sig4:4 " &
          "sig5:5 " &
          "sig6:6 " &
          "sig7:7 " &
          "sig8:8";
```

This example maps eight signals from entity "mydesign" onto the pins of a target device. Note the space character before the end-quote on the specifications for signals 1 through 7. This guarantees that the string for the pin_numbers attribute is syntactically correct.

5.4. ATTRIBUTE

5.4.10. polarity

The polarity attribute specifies polarity selection for individual signals.

Syntax

```
attribute polarity of signal-name:signal is value;
```

Legal values for the polarity attribute are pl_keep, pl_opt, and pl_default:

- A value of pl_keep tells *Warp* to keep the polarity of the signal as currently specified.
- A value of pl_opt tells *Warp* to optimize the polarity of the signal to use the fewest resources on the target device.
- A value of pl_default tells *Warp* to synthesize the signal based on the default polarity selection strategy. This default is determined by the command line switches or Galaxy dialog settings, if any, used in invoking *Warp*.

Example:

```
attribute polarity of abc:signal is pl_opt;
```

This example tells *Warp* to optimize polarity for signal "abc".

```
attribute polarity of abc:signal is pl_keep;
```

This example tells *Warp* to keep the polarity of signal "abc" as currently specified.

5.4. ATTRIBUTE

5.4.11. state_encoding

The state_encoding attribute specifies the internal encoding scheme for values of an enumerated type.

Syntax

```
attribute state_encoding of type-name:type is value;
```

The legal values of the state_encoding attribute are sequential, one_hot_zero, one_hot_one, and gray.

When the state_encoding attribute is set to "sequential", the internal encoding of each value of the enumerated type is set to a sequential binary representation. The first value in the type declaration gets an encoding of 00, the second gets 01, the third gets 10, the fourth gets 11, and so on. Sufficient bits are allocated to the representation to encode the number of enumerated type values included in the type declaration.

When the state_encoding attribute is set to "one_hot_zero", the internal encoding of the first value in the type definition is set to 0. Each succeeding value in the type definition has its own bit position in the encoding. That bit position is set to 1 when the state variable has that value. Thus, a one_hot_zero encoding of an enumerated type with N possible values requires N-1 bits. For example, if an enumerated type had four possible values, three bits would be used in its one_hot_zero encoding. The first value in the type definition would have an encoding of "000". The second would have an encoding of "001". The third would have an encoding of "010". The fourth would have an encoding of "100".

"One_hot_one" state encoding works similarly to one_hot_zero, except that no zero encoding is used; every value in the enumerated type has a bit position, which is set to one when the

state variable has that value. Thus, a one_hot_one encoding of an enumerated type with N possible values requires N bits. For example, if an enumerated type had four possible values, four bits would be used in its one_hot_one encoding. The first value in the type definition would have an encoding of "0001". The second would have an encoding of "0010". The third would have an encoding of "0100". The fourth would have an encoding of "1000".

When the state_encoding attribute is set to "gray", the internal encoding of successive values of the enumerated type follow a Gray code pattern, where each value differs from the preceding one in only one bit.

Examples:

```
type state is (s0,s1,s2,s3);
attribute state_encoding of state:type
   is one_hot_zero;
```

The first statement in this example declares an enumerated type, called "state," with four possible values. The second statement specifies that values of type state are to be encoded internally using a one_hot_zero encoding scheme.

```
type s is (s0,s1,s2,s3);
attribute state_encoding of s:type is gray;
```

The first line of this example declares an enumerated type, called "s," with four possible values. The second line specifies that values of type s are to be encoded internally using a Gray code encoding scheme.

5.4. ATTRIBUTE

5.4.12. synthesis_off

The synthesis_off attribute controls the flattening and factoring of expressions feeding signals for which the attribute is set to TRUE. This attribute causes a signal to be made into a factoring point for logic equations, which keeps the signal from being substituted out during optimization.

Syntax

```
attribute synthesis_off of signal_name:signal is true;
```

The synthesis_off attribute can only be applied to signals. The default value of the synthesis_off attribute for a given signal is FALSE. The attribute gives the user control over which equations or sub-expressions need to be factored into a node (i.e., assigned to a physical routing path).

For PLD's and CPLD's:

- When set to TRUE for a given signal, synthesis_off causes that signal to be made into a "node" (i.e., a factoring point for logic equations) for the target technology. This keeps the signal from being substituted out during the optimization process. This can be helpful in cases where performing the substitution causes the optimization phase to take an unacceptably long time (due to exponentially increasing CPU and memory requirements) or uses too many resources.
- Making equations into nodes forces signals to take an extra pass through the array, thereby decreasing performance, but may allow designs to fit better.

- The synthesis_ off attribute should only be used on combinational equations. Registered equations and pins are already natural factoring points; the use of synthesis_off on such equations results in unnecessary factoring.

For pASIC's:

- The synthesis_off attribute prevents logic factoring on any signal for which the attribute is set to TRUE. The gates that form such a signal's equation are not shared with other signals.
- This attribute should be used on signals whose performance is critical, and where sub-expression factoring is not desired. In some rare cases, factoring also causes more utilization of resources; this attribute will aid in those situations also. (See also the description of the -fL command line option in Section 2.2, "*Warp* Command Line Options.")
- In general, for designs targeting pASIC architectures, it is a good practice to enable the logic factoring option in Galaxy, and set the synthesis_off attribute on certain critical signals only.

Example

```
attribute synthesis_off of sig1:signal is true;
```

This example sets the synthesis_off attribute to true for a signal named "sig1".

PLD/CPLD Example:

Suppose you have the following equations:

```
x <= a OR b OR c ;
y <= NOT x OR d ;
```

where x is an internal signal, a, b and c are inputs to the device, and y is an output pin. By default (i.e., with the synthesis_off attribute for y set to FALSE), the fitter implements the y function as

```
y <= NOT (a OR b OR c) OR d ;
```

thereby substituting x in y. This is desirable in most cases. However, there can be cases where x is a large equation, and is used in other large equations, possibly with inversions. This situation might cause the logic optimizer to take unacceptably long (read "forever") to complete, due to constantly expanding CPU and memory requirements.

A situation might also occur where signal x is used by multiple other large outputs or registered nodes, which might cause the design to not fit, use too many resources, or fail to optimize. Setting the synthesis_off attribute for signal x to TRUE assigns x to a macrocell instead of substituting it into other equations.

pASIC Example

If you have the following equations:

```
x <= a OR b OR c ;
y <= a OR b OR d ;
```

In the default case (i.e., synthesis_off set to FALSE for both x and y), the logic factoring algorithm implements these equations as:

```
x_tmp <= a OR b ;
x <= x_tmp OR c ;
y <= x_tmp OR d ;
```

[Note: the equations in the example above are too small to have a significant effect on resource utilization or design performance. But they do serve to illustrate the concept of logic factoring and the use of the synthesis_off attribute. "Real-life" equations are likely to have many more product terms.]

For large equations, the sharing of subexpression x_tmp reduces resource utilization and signal loading for the inputs. However, there can be cases where sharing subexpressions results in higher resource utilization or worsened performance. For these situations, setting the synthesis_off attribute on signal x or y to TRUE prevents subexpression x_tmp from being shared, and yields the following implementation:

```
x <= a OR b OR c ;
y <= a OR b OR d ;
```

which causes signals a and b to support a higher loading, but makes both x and y faster.

5.5. CASE

The CASE statement selects one or more statements to be executed within a process, based on the value of an expression.

Syntax:

```
case expression is
   when case1 [| case2...] =>
      sequence_of_statements;
   when case3 [| case4...] =>
      sequence_of_statements;
         .
         .
         .]
   [when others =>
      sequence_of_statements;]
   end case;
```

Discussion:

In *Warp*, the expression that determines the branching path of the CASE statement must evaluate to a bit vector or to a discrete type (i.e., a type with a finite number of possible values, such as an enumerated type or an integer type).

The vertical bar ('|') operator may be used to indicate multiple cases to be checked in a single WHEN clause. This may only be used if the sequence of statements following the WHEN clause is the same for both cases.

The keyword OTHERS may be used to specify a sequence of statements to be executed if no other case statement alternative applies. Because CASE statements execute sequentially, the test for OTHERS should be the last test in the WHEN list."

When *Warp* synthesizes a CASE statement, it synthesizes a memory element for the condition being tested (in order to

maintain any outputs at their previous values) unless:

1. all outputs within the body of the CASE statement are previously assigned a default value within the process; or

2. the CASE statement completely specifies the design's behavior following any possible result of the conditional test. The best way to ensure complete specification of design behavior is to include an OTHERS clause within the CASE statement.

When a CASE statement does not specify a branch for all possible results, *Warp* synthesizes a memory element for the conditional test. This could use up more PLD/FPGA resources than would otherwise be required.

Therefore, to use the fewest possible resources during synthesis, either assign default values to outputs in a process, or make sure all CASE statements include an OTHERS clause.

Example:

In the following example, signal s is declared as

```
s :in bit_vector(0 to 2);
```

In addition, i and o are declared as eight-element bit vectors:

```
i : in bit_vector(0 to 7);
o : out bit_vector(0 to 7);
```

The architecture follows:

```
architecture demo of Barrel_shifter is
   begin process (s, i)
      begin
      case s is
         WHEN "000"=>
            o<=i;
```

```
            WHEN "001"=>
               o<=(i(1),i(2),i(3),i(4),i(5),i(6),i(7),i(0));
            WHEN "010"=>
               o<=(i(2),i(3),i(4),i(5),i(6),i(7),i(0),i(1));
            WHEN "011"=>
               o<=(i(3),i(4),i(5),i(6),i(7),i(0),i(1),i(2));
            WHEN "100"=>
               o<=(i(4),i(5),i(6),i(7),i(0),i(1),i(2),i(3));
            WHEN "101"=>
               o<=(i(5),i(6),i(7),i(0),i(1),i(2),i(3),i(4));
            WHEN "110"=>
               o<=(i(6),i(7),i(0),i(1),i(2),i(3),i(4),i(5));
            WHEN "111"=>
               o<=(i(7),i(0),i(1),i(2),i(3),i(4),i(5),i(6));
            end case;
         end process;
end demo;
```

In this example, signal s evaluates to a 3- element bit-string literal. The appropriate statement is executed, depending on the value of s, and output bit vector o gets the value given by the specified ordering of elements in input bit vector i. Note that the CASE statement completely specifies the results of the conditional test; all possible values of s are covered by a WHEN clause. Hence, no OTHERS clause is needed.

5.6. COMPONENT

A component declaration specifies a component to be synthesized, and lists the local signal names of the component. The component declaration serves the same purpose in VHDL as a function declaration or prototype serves in the C programming language.

Syntax (Component Declaration):

```
component identifier
   [generic (generic_list);]
   [port (port_list);]
   end component;
```

Example:

```
component barrel_shifter port(
   clk : IN BIT;
   s :in bit_vector(0 to 2);
   insig :in bit_vector(0 to 7);
   outsig :out bit_vector(0 to 7));
end component;
```

This example declares a component called barrel_shifter with a 3-bit input signal, an 8-bit input signal, and an 8-bit output signal.

Syntax (Component Instantiation):

```
instantiation_label:component_name
   [generic generic_mapping]
   [port port_mapping];
```

A component instantiation creates an instance of a component that was previously declared with a component declaration statement. Think of component instantiation as "placing" a

previously declared component into an architecture, then "wiring" the newly placed component into the design by means of the generic map or port map.

Example:

```
a1:barrel_shifter
   port map(
      clk=>pin1,
      s(0)=>pin2,
      s(1)=>pin3,
      s(2)=>pin4,
      insig(0)=>pin5,
      insig(1)=>pin6,
      insig(2)=>pin7,
      insig(3)=>pin8,
      insig(4)=>pin9,
      insig(5)=>pin10,
      insig(6)=>pin11,
      insig(7)=>pin12,
      fbx(outsig(0))=>pin14,
      fbx(outsig(1))=>pin15,
      fbx(outsig(2))=>pin16,
      fbx(outsig(3))=>pin17,
      fbx(outsig(4))=>pin18,
      fbx(outsig(5))=>pin19,
      fbx(outsig(6))=>pin20,
      fbx(outsig(7))=>pin21);
```

The line "a1:barrel_shifter" in the example above instantiates a component named a1 of type barrel-shifter. The port map statement that follows maps each signal from this instance of barrel-shifter to a pin on a physical part.

Note the use of the fbx() function to map the output signals (of type bit) from the component onto the output pins (of type X01Z) on the PLD.

Note also the direction of the "arrow" in each mapping: from the "formal" (the signal name on the component) to the "actual" (the

name of the pin to which the signal is being mapped).

5.7. CONSTANT

A constant is an object whose value may not be changed.

Syntax:

```
constant identifier_list:type[:=expression];
```

Example:

```
TYPE stvar is bit_vector(0 to 1);
constant s0:stvar := "00";
constant s1:stvar := "01";
constant s2:stvar := "10";
constant s3:stvar := "11";
```

This example declares a bit vector subtype with length 2, called "stvar". It then defines four constants of type stvar, and gives them values of 00, 01, 10, and 11, respectively.

```
subtype bit8 is bit_vector(0 to 7);
type v8_table is array(0 to 7) of bit8;
constant xtbL1:v8_table := (
"00000001",
"00000010",
"00000100",
"00001000",
"00010000",
"00100000",
"01000000",
"10000000");
```

This example declares a bit vector subtype with length 8 called "bit8," a 1-dimensional array type of bit8 called "v8_table" with eight elements, and a constant of type v8_table called "xtbL1."

Values are assigned to the constant by concatenating a sequence of string literals (e.g., "00000001") into bit-vector form. Only characters '0' and '1' are allowed in these string literals, but the

values could have been written in hex format (e.g., x"01" is the same as "00000001" for this purpose). The result is a table of constants such that xtbL1(0) is "00000001" and xtbL1(7) is "10000000".

5.8. ENTITY

An entity declaration names a design entity and lists its ports (i.e., external signals). The mode and data type of each port are also declared.

Syntax:

```
entity identifier is port(
   port_name: mode type [;
   port_name: mode type...])
   end [identifier];
```

Choices for mode are IN, OUT, BUFFER, and INOUT.

Example:

```
entity Barrel_Shifter is port(
   clk : IN BIT;
   s :in bit_vector(0 to 2);
   insig :in bit_vector(0 to 7);
   outsig :out bit_vector(0 to 7));
end Barrel_Shifter;
```

This example declares an entity called barrel_shifter with a 3-bit and an 8-bit input signal as well as an 8-bit output signal.

5.9. EXIT

The EXIT statement causes a loop to be exited. Execution resumes with the first statement after the loop. The conditional form of the statement causes the loop to be exited when a specified condition is met.

Syntax:

```
exit [loop_label] [when condition];
```

Example:

```
i <= 0;
loop
   outsig(i)<=barrel_mux8(i,s,insig);
   i <= i+1;
   exit when i>7;
   end loop;
```

The EXIT statement in the example above causes the loop to exit when variable i becomes greater than 7. The example thus calls the function barrel_mux8 eight times.

5.10. GENERATE

Generate statements specify a repetitive or conditional execution of the set of concurrent statements they contain. Generate statements are especially useful for instantiating an array of components.

Syntax:

```
label:generation_scheme generate
   {concurrent_statement}
      end generate [label];

   generation_scheme ::=
      for generate_parameter_specification
   | if condition
```

A "generation scheme" in the syntax above refers to either a FOR-loop specification or an IF-condition specification, as shown in the example below.

Example:

```
architecture test of serreg is
begin
   m1: for i in 0 to size-1 generate
      m2: if i=0 generate
         x1:dsrff port map (si, zero, mreset,    clk, q(0));
         end generate;
      m3: If i>0 generate
         x2:dsrff port map(q(I-1), zero, mreset,clk, q(I));
         end generate;
      end generate;
end test;
```

The example above instantiates a single component labeled x1, and size-1 components labeled x2. For size=3, for instance, the code shown above is the equivalent of

```
architecture test of serreg is
begin
   x1:dsrff port map(si, zero, mreset, clk, q(0));
   x2:dsrff port map(q(0), zero, mreset, clk, q(1));
   x3:dsrff port map(q(1), zero, mreset, clk, q(2));
end test;
```

5.11. GENERIC

Generics are the means by which instantiating (parent) components pass information to instantiated (child) components in VHDL. Typical uses are to specify the size of array objects or the number of subcomponents to be instantiated.

Syntax

```
generic(identifier:type[:=value]);
```

Example

```
component serreg
   generic (size:integer:=8);
   port (si,
         clk,
         mreset : in bit;
         q : inout bit_vector(0 to size-1));
   end component;
```

This example declares a component with a bidirectional array of 8 bits, among other signals. The number of bits is given by the value of the size parameter in the generic statement.

5.12. IF-THEN-ELSE

The IF statement selects one or more statements to be executed within a process, based on the value of a condition.

Syntax

```
IF condition THEN sequence_of_statements
   [ELSIF condition THEN
      sequence_of_statements...]
   [ELSE sequence_of_statements]
      END IF;
```

Discussion

A condition is a boolean expression, i.e., an expression that resolves to a boolean value. If the condition evaluates to true, the sequence of statements following the THEN keyword is executed. If the condition evaluates to false, the sequence of statements following the ELSIF or ELSE keyword(s), if present, are executed.

When *Warp* synthesizes an IF-THEN-ELSE statement, it synthesizes a memory element for the condition being tested (in order to maintain any outputs at their "previous" values) unless:

1. all outputs within the body of the IF-THEN-ELSE statement are previously assigned a "default" value within the process; or

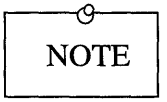
2. the IF-THEN-ELSE statement completely specifies the design's behavior following any possible result of the conditional test. The best way to ensure complete specification of design behavior is to include an ELSE clause within the IF statement. (See example following.)

When an IF-THEN-ELSE statement does not specify a branch for all possible results, *Warp* synthesizes a memory element for the conditional test. This could use up more PLD resources than would otherwise be required.

In short, to use the fewest possible PLD resources during synthesis, either assign default values to outputs in a process, or make sure all IF-THEN-ELSE statements include ELSE clauses.

Example

```
if (not tmshort=one)then stvar <= ewgo;
elsif (tmlong=one)then stvar <= ewwait;
elsif (not ew=one AND ns=one)then stvar<=ewgo;
elsif (ew=one AND ns=one)then stvar <= ewwait;
elsif (not ew=one)then stvar <= ewgo;
else stvar <= ewwait;
end if;
```



In the case above, the 'else' statement is to be left out, as opposed to suggestions made on the previous page.

The example above sets a state variable called stvar. The value that stvar receives depends on the value of variables tmshort, tmlong, ew, and ns.

Asynchronous Sets, Resets

Use an IF-THEN-ELSE statement to synthesize the operation of a synchronous circuit containing asynchronous sets or resets.

To do so, use a sensitivity list in the PROCESS statement, naming the sets, resets, and clock signals that will trigger the execution of the process. Then, use a sequence of IF..ELSIF clauses to specify the behavior of the circuit.

The structure of the process should be something like this:

```
process (set, reset, clk) begin
  if (reset = '0')then
    --assign signals to their "reset" values;
  elsif (set = '0')then
    --assign signals to their "set" values;
  elsif (clk'EVENT AND clk='1')then
    --perform synchronous operations;
  end if;
end process;
```

The example above shows the VHDL description for active-low set and reset signals. It could just as easily have been coded for active-high sets and resets by using the conditions set='1' and reset='1'.

The assignments made in the statements that follow the set or reset signal conditions must be "simple" assignments (i.e., of the form *name=constant)*, to a signal of type bit, bit vector, or enumerated type (for state variables).

5.13. LIBRARY

In *Warp*, a library is a storage facility for previously analyzed design units.

Syntax

```
library library-name [, library-name];
```

A library clause declares logical names to make libraries visible within a design unit. A design unit is an entity declaration, a package declaration, an architecture body, or a package body.

Example

```
library mylib;
```

The above example makes a library named mylib visible within the design unit containing the library clause.

5.14. Loops

Loops execute a sequence of statements repeatedly.

Syntax

```
[loop_label:] [iteration_scheme] loop
   sequence_of_statements
   end loop [loop_label];

iteration_scheme ::=
   while condition
   | for loop_parameter in
      {lower_limit to upper_limit
      |upper_limit downto lower_limit}
```

Discussion

There are three kinds of loops in VHDL:

1. Simple loops are bounded by a loop/end loop statement pair. These kinds of loops require an exit statement, otherwise they execute forever;
2. FOR loops execute a specified number of times; and
3. WHILE loops execute while a specified condition remains true.

Example

Simple loop:

```
i := 0;
loop
   outsig(i)<=barrel_mux8(i,s,insig);
   i := i+1;
   if (i>7) then
      exit;
      END IF;
   end loop;
```

FOR loop:

```
for i in 0 to 7 loop
   outsig(i)<=barrel_mux8(i,s,insig);
   end loop;
```

WHILE loop:

```
i := 0;
while (i<=7) loop
   outsig(i)<=barrel_mux8(i,s,insig);
   i := i+1;
   end loop;
```

The examples above show three ways of implementing the same loop. All of these loops call the function barrel_mux8 eight times. (In all three, i must be defined as a variable.)

5.15. NEXT

NEXT advances control to the next iteration of a loop.

Syntax

```
next [loop_label] [when condition];
```

Discussion

Example

```
for i in 0 to 7 loop
   if ((i =0) or (i=2) or (i=4) or (i=6)) then
      outsig(i)<=barrel_mux8(i,s,insig)
   else
      next i;
   end if;
end loop;
```

The example above performs an operation on the even-numbered bits of an 8-element bit vector. It uses a NEXT statement to advance to the next iteration of the loop for the odd-numbered bits.

5.16. PACKAGE

A VHDL package is a collection of declarations that can be used by other VHDL descriptions. A VHDL package consists of two parts: the package declaration and the package body.

Syntax (package declaration)

```
package identifier is
   function_declaration
   | type_declaration
   | subtype_declaration
   | constant_declaration
   | signal_declaration
   | component_declaration
   | attribute_declaration
   | attribute_specification
   | use_clause
   [;{function_declaration
   | type_declaration
   | subtype_declaration
   | constant_declaration
   | signal_declaration
   | component_declaration
   | attribute_declaration
   | attribute_specification
   | use_clause}...]
   end [identifier];
```

Syntax (package body)

```
package body identifier is
   {function_declaration
   | function_body
   | type_declaration
   | subtype_declaration
   | constant_declaration
   | use_clause}
   [;{function_declaration
```

```
    | function_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | use_clause}...]
  end [identifier];
```

The package declaration declares parts of the package that can be used by other VHDL descriptions, i.e., by other designs that use the package.

The package body provides definitions and additional declarations, as necessary, for functions whose interfaces are declared in the package declaration.

Example (package declaration)

```
package bv_tbl is
subtype bit8 is bit_vector(0 to 7);
type v8_table is array(0 to 7) of bit8;

-- defining vectors for i2bv8 function
constant xtbl1:v8_table := (
"00000001",
"00000010",
"00000100",
"00001000",
"00010000",
"00100000",
"01000000",
"10000000");

-- function declaration
function i2bv8(ia:integer) return bit8;
subtype bit3 is bit_vector(0 to 2);
type v3_table is array(0 to 7) of bit3;

-- defining vectors for i2bv3 function
constant xtbl2:v3_table := (
"000",
"001",
```

```
"010",
"011",
"100",
"101",
"110",
"111"
);

--function declaration
function i2bv3(ia:integer) return bit3;

end bv_tbl;
```

The example above declares several types, subtypes, constants, and functions. These items become available for use by any VHDL description that uses package bv_tbl.

Example (package body)

```
package body bv_tbl is

function i2bv8(ia:integer) return bit8 is
-- translates an integer between 1 and 8
-- to an 8-bit vector
begin
return xtbl1(ia);
end i2bv8;

function i2bv3(ia:integer) return bit3 is
-- translates an integer between 1 and 8
-- to a three-bit vector
begin
return xtbl2(ia);
end i2bv3;

end bv_tbl;
```

The example above defines two functions whose declarations appeared in the package declaration example, shown previously.

5.17. PORT MAP

A port map statement associates the ports of a component with the pins of a physical part.

Syntax

```
port map ([formal_name =>] actual_name
        [,[formal_name =>] actual_name]);
```

The port map statement associates ports declared in a component declaration, known as *formals*, with the signals (known as *actuals*) being passed to those ports.

If the signals are presented in the same order as the formals are declared, then the formals need not be included in the port map statement.

Port map statements are used within component instantiation statements. (See Section 5.6, "COMPONENT", for more information about component instantiation statements.)

Example

```
and_1: AND2
   port map(A => empty_1,
            B => empty_2,
            Q => refill_bin);
```

The example above instantiates a two-input AND gate. The port map statement associates three signals (empty_1, empty_2, and refill_bin, respectively) with ports A, B, and Q of the AND gate.

If the three ports appear in the order A, B, and Q in the AND2 component declaration, the following (shorter) component instantiation would have the same effect:

```
and_1: AND2
   port map(empty_1,empty_2,refill_bin);
```

5.18. PROCESS

A process statement is a concurrent statement that defines a behavior to be executed when that process becomes active. The behavior is specified by a series of sequential statements executed within the process.

Syntax

```
[label:] process [(sensitivity_list)]
   [process_declarative_part]
   begin
      sequential_statement
      [;sequential_statement...];
   end process [label];

process_declarative_part ::=
   function_declaration
   | function_body
   | type declaration
   | subtype declaration
   | constant declaration
   | variable declaration
   | attribute specification

sequential_statement ::=
   wait_statement
   | signal_assignment_statement
   | variable_assignment_statement
   | if_statement
   | case_statement
   | loop_statement
   | next_statement
   | exit_statement
   | return_statement
   | null_statement
```

A process which is executing is said to be active; otherwise, the

process is said to be suspended. Every process in the VHDL description may be active at any time. All active processes are executed concurrently with respect to simulation time.

Processes can be activated in either of two ways: by means of a wait statement, or by means of a sensitivity list (a list of signals enclosed in parentheses appearing after the process keyword).

When a process includes a WAIT statement, the process becomes active when the value of the clock signal goes to the appropriate value ('0' or '1').

When a process includes a sensitivity list, the process becomes active when one or more of the signals in the list changes value.

Example

```
process (s, insig) begin
   for i in 0 to 7 loop
      outsig(i)<=barrel_mux8(i,s,insig);
      end loop;
   end process;
```

The example above shows a process that executes whenever activity is sensed on either of two signals, s or insig.

5.19. SIGNAL

A signal is a pathway along which information passes from one component in a VHDL description to another. Signals are also used within components to pass values to other signals, or to hold values.

Syntax (signal declaration)

```
signal name [, name ...]:type;
```

Syntax (signal assignment)

```
signal_name <= expression
   [when condition [ else expression]];
```

Signals must be declared before they can be used. Declaring a signal gives it a name and a type. Signal declarations often appear as part of port or component declarations.

To assign a value to a signal, you simply replace its current value with the value of some expression of the same type as the signal, using the signal name and the "<=" operator.

You may also specify a condition under which the replacement is to be made, as well as an alternative value for the signal if the condition is not met. This form of the signal assignment statement uses the "when" and "else" keywords.

Example

(Signal Declaration examples)

```
signal c0,c1,cin1,cin2:bit;
```

This example declares four signals (c0, c1, cin1, and cin2), each of type bit.

```
type State is (s1, s2, s3, s4, s5);
signal StVar : State;
```

This example declares an enumerated type named State, with five possible values. It then declares a signal named StVar of type State. Thus, StVar can have values s1, s2, s3, s4, or s5.

(Unconditional Signal Assignment examples)

```
c_in <= '1';
```

This example sets a variable of type bit named "c_in" to '1'.

```
StVar <= s1;
```

This example assigns the value s1 to a signal named StVar. Presumably, StVar is a signal of some enumerated type, having "s1" as one of its possible values.

(Conditional Signal Assignment example)

```
c_in2 <= '1' when (stvar=s1) OR (stvar=s2) else '0';
```

The above example illustrates the use of conditional signal assignment. Signal c_in2 is assigned one value if the specified conditions are met; otherwise, it is assigned a different value.

5.20. Subprograms

Subprograms are sequences of declarations and statements that can be invoked repeatedly from different parts of a VHDL description. VHDL includes two kinds of subprograms: procedures and functions.

Syntax (procedure declaration)

```
procedure designator [(formal-parameter-list)];
```

Syntax (procedure body)

```
procedure designator [(formal-parameter-list)] is
    [declarations]
  begin
    {sequential-statement;}
  end [designator];
```

Syntax (function declaration)

```
function designator [(formal-parameter-list)]
  return type_mark;

type_mark ::= type_name | subtype_name
```

Syntax (function body)

```
function designator [(formal-parameter-list)]
  return type_mark is
    [declarations]
  begin
    {sequential-statement;}
  end [designator];
```

A subprogram is a set of declarations and statements that you can use repeatedly from many points within a VHDL description.

There are two kinds of subprograms in VHDL: procedures and functions. Procedures may return zero or more values. Functions always return a single value. In practice, procedures are most often used to sub-divide a large behavioral description into smaller, more modular sections. Functions are most often used to convert objects from one data type to another or to perform frequently needed computations.

VHDL allows you to declare a subprogram in one part of a VHDL description while defining it in another. Subprogram declarations contain only interface information: name of the subprogram, interface signals, and return type (for functions). The subprogram body contains local declarations and statements, in addition to the interface information.

Function calls are expressions; the result of a function call is always assigned to a signal or variable, or otherwise used in a larger statement (e.g., as a parameter to be passed to a procedure call). Procedure invocations, by contrast, are entire statements in themselves.

In both procedures and functions, actual and formal parameters may use positional association or named association.

To use positional association, you list the parameters to be passed to the subprogram in the same order that the parameters were declared, without naming the parameters.

To use named association, you give the formal parameter (the name shown in the subprogram declaration) and the actual parameter (the name you're passing to the subprogram) within the procedure invocation or function call, linking the formal and actual parameters with the '=>' operator. When you use named association, you can list parameters in any order.

Example

Consider a procedure, whose declaration is shown below, that takes three signals of type bit as input parameters:

```
procedure crunch(signal sig1,sig2,sig3:in bit);
```

Then, the invocation

```
crunch(trex,raptor,spitter);
```

uses positional association to map signal trex to sig1, signal raptor to sig2, and signal spitter to sig3. You could use named association in the following procedure invocation, however, and get the same result:

```
crunch(sig2=>raptor,sig3=>spitter,sig1=>trex);
```

More information about procedures and functions is included on the following pages.

5.20. Subprograms

5.20.1. Procedures

Procedures describe sequential algorithms that may return zero or more values. They are most frequently used to decompose large behavioral descriptions into smaller, more modular sections, which can be used by many processes.

Procedure parameters may be constants, variables, or signals, and their modes may be in, out, or inout. Unless otherwise specified, a parameter is by default a constant if it is of mode in, and a variable if it is of mode out or inout.

In general, procedures can be used both concurrently (outside of any process) and sequentially (inside a process). However, if any of the procedure parameters are variables, the procedure can only be used sequentially (since variables can only be defined inside a process). Any variables declared inside a procedure cease to exist when the procedure terminates.

A procedure body can contain a wait statement, while a function body cannot. However, a process that calls a procedure with a wait statement in it cannot have a sensitivity list. (Processes can't be sensitive to signals and made to wait simultaneously.)

5.20. Subprograms

5.20.2. Functions

Functions describe sequential algorithms that return a single value. Their most frequent uses are: (1) to convert objects from one data type to another; and (2) as shortcuts to perform frequently used computations.

Function parameters must be of mode in and must be signals or constants. If no mode is specified for a function parameter, the parameter is interpreted as having mode in.

A function body cannot contain a wait statement. (Functions are only used to compute values that are available instantaneously.)

Any variables declared inside a function cease to exist when the function terminates (i.e., returns its value).

One common use of functions in VHDL is to convert objects from one data type to another. *Warp* provides two pre-defined type conversion functions in file CYPRESS.VHD: the functions fxb() ("from-X01Z-to-bit") and fbx() ("from-bit-to-X01Z").

The fxb() and fbx() functions are frequently used to bind signals to pins in an architecture. Because VHDL is a strongly typed language, it is important that signals are mapped to pins of the same type. Signals are usually of type bit, with possible values '0' and '1'. Many pins, however, are of type X01Z, with possible values '0', '1', 'X' (unknown) and 'Z' (high-impedance). To map a signal of type bit to a pin of type X01Z, you would use the fbx() function, as demonstrated in the first example.

Examples

```
architecture demo of c22v10 is
   begin
      xx1:SeqDetect
         port map(
            x0 => pin3,
            x1 => pin5,
            clk => pin1,
            fbx(Outsig) => pin16);
   end demo;
```

In this example, the value of signal Outsig, which is of type bit, is converted into a value of type X01Z by the fbx() function before being mapped to pin 16 on the C22V10.

The fxb() function, which converts a value of type x01z to a value of type bit, works in a complementary fashion to the fbx() function. It is used with I/O ports that are being used as inputs.

```
function count_ones(vec1:bit_vector)
   return integer is
-- returns the number of '1' bits in a bit vector
   variable temp:integer:=0;
   begin
      for i in vec1'low to vec1'high loop
         if vec1(i) = '1' then
            temp := temp+1;
            end if;
         end loop;
      return temp;
   end count_ones;
```

This function counts the number of '1's in a bit vector.

5.21. TYPE

In VHDL, objects are anything that can hold a value. Signals, constants, or variables are common objects. All VHDL objects have a type, which specifies the kind of values that the object can hold.

Syntax (enumerated type declaration)

```
type name is (value [,value...]);
```

Syntax (subtype declaration)

```
subtype name is base_type
   [range {lower_limit to upper_limit
          | upper_limit downto lower_limit}];
```

Syntax (bit vector declaration)

```
subtype name is bit_vector
   (lower_limit to upper_limit
   | upper_limit downto lower_limit);
```

Syntax (record declaration)

```
type name is record
   name:type;
   [name : type;...]
   end record;
```

Warp has the following pre-defined types:

- integer: VHDL allows each implementation to specify the range of the integer type differently, but the range must extend from at least -(2**31-1) to +(2**31-1), or -2147483648 to +2147483647. Only variables (not signals) can have type integer.

- boolean: an enumerated type, consisting of the values "true" and "false";
- bit: an enumerated type, consisting of the values '0' and '1';
- character: an enumerated type, consisting of one literal for each of the 128 ASCII characters. The non-printable characters are represented by three-character identifiers, as follows: NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN,EM, SUB, ESC, FSP, GSP, RSP, and USP.

VHDL objects can take other, user-defined, types as well. Possibilities include:

- enumerated types: This type has values specified by the user. A common example is a state variable type, where the state variable can have values labeled state1, state2, state3, etc.
- sub-range types: This type limits the range of a larger base type (such as integers) to a smaller set of values. Examples would be positive integers, or non-negative integers from 0 to 100, or printable ASCII characters.
- arrays (especially bit vectors): This type specifies a collection of elements of a single base type. A commonly used example is the bit_vector type, declared in the standard library, which denotes an array of bits.
- records: This type specifies a collection of elements of possibly differing types.

Examples

(Enumerated Type Declaration example)

```
type sigstates is (nsgo, nswait, nswait2,
   nsdelay, ewgo, ewwait, ewwait2, ewdelay);
```

This example declares an enumerated type and lists eight possible values for it.

(Sub-range Type Declaration example)

```
type column is range (1 to 80);type row is
   range (1 to 24);
```

The above examples declare two new sub-range integer types, column and row. Legal values for objects of type column are integers from 1 to 80, inclusive. Legal values for objects of type row are integers from 1 to 24, inclusive.

(Bit Vector Type Declaration example)

```
subtype bit8 is bit_vector(0 to 7);
signal insig, outsig : bit8;
.
.
insig <= "00000010";
.
.
outsig<=insig(2 to 7) & insig(0 to 1);
```

The above example declares a bit vector type called bit8. It then declares two signals of type bit8, insig and outsig. The signal assignment statement that concludes the example left-shifts insig by two places. (Outsig then contains the value "00001000".)

(Record Type Declaration example)

```
subtype Bit5 is bit_vector(1 to 5);
-- define a record type containing a 5-bit vector
-- and a bit
type arec is record
   abc:Bit5;
   def:bit;
   end record;
-- define a type containing
-- an array of five records
type twodim is array (1 to 5) of arec;
-- now define a couple of signals
signal v:twodim;
signal vrec:arec;
.
.
-- in record 1 of v, set array field to all 0's
v(1).abc <= "00000";

-- in record 2 of v, set first three bits to 0's,
-- others to 1's
v(2).abc <= ('0', '0', '0', others => '1');

-- set fifth bit in array within record #5 to 0.
v(5).abc(5) <= '0';

-- set bit field within record to 1
vrec.def <= '1';

-- set 3rd bit within vrec's array to 1
vrec.abc(3) <= '1';
```

The example above defines a record type and a type consisting of an array of records. The example then demonstrates how to assign values to various elements of these arrays and records.

5.22. USE

The USE statement makes items declared in a package visible inside the current design unit. A design unit is an entity declaration, a package declaration, an architecture body, or a package body.

Syntax

use *name* {, *name*};

Example

```
use work.cypress.all;
use work.rtlpkg.all;
use work.brl4pkg.all;
```

The example above makes the items declared in the specified packages available for use in the design unit in which the use statements are contained.

NOTE

The scope of a USE statement extends only to the design unit (entity, architecture, or package) that it precedes.

5.23. VARIABLE

A variable is a VHDL object (similar to a signal) that can hold a value.

Syntax (variable declaration)

```
variable name [, name ...]:type [:=expression];
```

Syntax (variable assignment)

```
signal_name <= expression;
```

Variables differ from signals in that variables have no direct analogue in hardware. Instead, variables are simply used as indices or value-holders to perform the computations incidental to higher-level modeling of components.

Example

```
function admod(i,j,max:integer) return integer is
   variable c:integer;
   begin
   c:=(i+j) mod (max+1);
   return c;
   end admod;
```

The function in the example above declares a variable c as a value-holder for a computation, then uses it and returns its value in the body of the function.

5.24. WAIT

The WAIT statement suspends execution of a process until the specified condition becomes true.

Syntax

```
wait until [condition];
```

Discussion

A WAIT statement, if used, must appear as the first sequential statement inside a process.

Example

```
wait until clk='1';
```

The example above suspends execution of the process in which it is contained until the value of signal clk goes to '1'.

Chapter 6 Synthesis

Synthesis

6.1. Introduction

This chapter describes how the following VHDL constructs are synthesized in *Warp*: architectures, processes, components, signals and variables, clock signals, global signals, and the CASE OTHERS construct.

Warp's operation can be divided into two phases: the *analysis* phase and the *synthesis* phase.

In analysis, *Warp* examines the VHDL description to guarantee that it complies with VHDL syntax rules. *Warp* then determines what design elements (packages, components, entities, architectures) make up the description.

Synthesis, as defined in the context of *Warp*, is the realization of design descriptions into logic circuits. In other words, synthesis is the process by which logic circuits are realized/created from design descriptions. These logic circuits are then fitted (PLDs) or placed and routed (FPGAs) to produce a programming file for the target device. The structural netlist description is a text file that lists the fundamental sub-components in the design, their connections, and the equations that transfer values from one sub-component to another. "Fundamental sub-components" include the RTL components declared for the target PLD: memory elements, I/O components, buffers, and macrocell components.

Note that fundamental sub-components do not include the combinatorial gates (AND, OR, XOR, NOT) that are common in a standard netlist. This is because combinatorial logic is not represented in the form of gates in *Warp* synthesis, but rather in the form of transfer equations.

6.2. Architectures

Architectures consist of collections of statements and processes that, taken together, describe the function of an entity.

Each architecture statement is either a component instantiation, a process, or a statement that is converted into a process. Note that an architecture is a concurrent description; conceptually, all assignments and processes within an architecture operate concurrently. This is radically different from a computer program written in a language such as C or Pascal, in which statements operate sequentially. Only within a process do statements evaluate sequentially.

Architectures are synthesized into RTL (Register Transfer Level) descriptions consisting of fundamental sub-components and transfer equations, which are technology-mapped to the target PLD.

6.3. Processes

In VHDL, a process is a sequence of statements that runs sequentially.

In behavioral descriptions, processes usually consist of collections of IF-THEN-ELSE statements, CASE statements, and assignment statements.

Warp synthesizes such processes differently, depending on whether or not the process begins with a WAIT statement.

With WAIT Statement (Clocked)

Processes that begin with a WAIT statement are assumed to be synchronized to a clock signal. Such processes are synthesized as follows:

1. *Warp* converts the IF-THEN-ELSE and CASE statements in the process to an equivalent set of fundamental sub-components and transfer equations;

2. For each assignment statement to a signal, *Warp* uses a memory element (e.g., a D flip-flop) to store the value of the signal and connects the memory element's clock input to the conditional expression named in the WAIT statement.

Thus, each signal assigned to in a process containing a WAIT statement uses up a memory element on the target PLD. Similarly, the more complex the IF-THEN-ELSE and CASE statements in a process (as, for example, if you are describing a really large state machine), the more PLD resources are used.

With Sensitivity List (No WAIT Statement)

Processes that have no WAIT statement are assumed to be "unclocked," i.e., asynchronous with respect to system clocks.

For VHDL compliance, such processes require a "sensitivity list" (a list of signals) to accompany the PROCESS keyword. A change in the value of any signal in the sensitivity list causes the process to become active. Note, however, that *Warp* does not currently object to missing sensitivity lists.

If all signals contained in the process are assigned a value before they are used, *Warp* synthesizes unclocked processes as combinatorial logic. No explicit memory element is used on the target PLD, nor is any combinatorial feedback generated.

If one or more signals are used before their values are initialized, however, *Warp* creates "implicit" memory in the form of combinatorial feedback. This can lead to equations that consume large amounts of PLD resources, or to an inability to fit the design on a PLD due to signal placement restrictions. The solution: initialize signals in unclocked processes.

6.4. Components

Components are recursively "flattened" into a set of fundamental subcomponents and transfer equations.

6.5. Signals and Variables

A signal is a pathway along which information passes from one component in a VHDL description to another. It is analogous to a wire in hardware.

A variable is a VHDL object that can hold a value. It differs from a signal in that a variable has no direct analogue in hardware.

There is often a great deal of confusion about the difference between a signal, which is a concept unique to VHDL, and a variable, which appears in most programming languages.

VHDL was originally designed as a simulation language. It was also created to support concurrency of statements. The implementation of signals makes this possible.

A VHDL simulator scans a source file, interpreting each VHDL statement in turn. This process is repeated for as long as the user specifies. Each pass through the source is referred to as a "simulation cycle."

Whenever the code assigns a value to a variable, the simulator simply updates the current value of that variable, as you would expect in any programming language.

But when the code assigns a value to a signal, that assignment is treated differently. The signal has a current value, which is used whenever the signal appears in an expression, and a list of "next values," each of which consists of a pair of data items: a value for the signal, and the number of simulation cycles after which the signal is actually given that value. So, a signal assignment does not affect the current value of the signal but only the value for a future simulation cycle. At the end of each simulation cycle, the simulator scans through all the signals, and updates each one from its "next values" list for the next cycle that is to occur.

6.6. Clocks

VHDL has no concept of a "clock" signal. The way to incorporate a clock signal in your designs is to use a WAIT statement in processes or to use RTL components.

When you use a WAIT statement, the RTL component synthesized for each assignment has a signal assigned to the component's CLK port.

6.7. Global Signals

Two global signals are declared in *Warp*: 'zero' and 'one'.

These signals are initialized to 0 and 1, respectively. They have constant value and are otherwise synthesized just like any other signal. They are most often used to assign a constant value to an input port on an RTL component.

6.8. CASE OTHERS

When conditions or assignments occur in the OTHERS selection of a CASE statement, the synthesis process computes the complement of the other conditions and creates corresponding equations.

Appendix

A

Error Messages

A. Error Messages

This appendix lists the error messages that may be returned by the *Warp* compiler. A brief explanation is included if the error message is not self-explanatory.

Error messages not listed in this appendix are internal errors. If you encounter an internal error, notify your local Cypress sales representative. Note: %s refers to any string, %d to a decimal.

E3 :Need a '<=' or ':=' here.

An equal sign instead of an assignment operator was used in an expression.

E4 :Missing 'PORT MAP'.

E5 :Use '=', not ':=' here.

The assignment rather than the comparison operator was used in an expression.

E6 :Can't create output file '%s'.

The specified output file couldn't be created for some reason: out of disk space, file already exists and is write-protected, etc.

E7 :Can't open file.

An input file couldn't be opened for some reason (usually, because the file doesn't exist or isn't in the current path).

E8 :Syntax error: Can't use '%s' (a %s) here.

An attempt was made to use the wrong character or keyword, or a delimiter is missing.

E9 :Can't take attribute '%s' here

An attribute was used where not allowed.

E10 :Syntax error: '%s' is a reserved symbol.

You used a reserved word in an illegal fashion, e.g., as a signal or variable name.

E14 :%s (line %d, col %d):

This message is more of an informative message than an error message. It tells you where to look for an error. The message usually appears as part of another message.

E18 :Missing THEN.

A THEN is missing from an IF-THEN-ELSE statement.

E19 :Not a TYPE or SUBTYPE name: %s

A variable or signal was defined that has an unknown type (e.g., "signal x:nerp").

E20 :'%s' already used

An attempt has been made to define a symbol that exists.

E21 :%s not an enumeration literal of %s

Attempt to assign or compare a state variable to a value that is not within its defined enumeration set.

E30 :';' after last item in interface list.

A semicolon was used after the last item in an interface list (the port declaration or parameter list for a function) instead of a right parenthesis.

E31 :Missing end-quote.

E32 :Function overloading by return type not implemented.

Functions of the same name and different parameters are allowed, but functions may not have different return types.

E34 :Undeclared name: %s

Attempt to use an undeclared (undefined) variable.

E36 :Variable declaration must be in a PROCESS statement.

Attempt to declare a variable inside an architecture. Signals can be declared in architectures, but variables have to be declared within a process.

E40 :Declaration outside ARCHITECTURE or ENTITY.

An attempt was made to declare a variable outside an architecture or sub-program.

E41 :Not a valid ENTITY declarative item.

Attempt to declare a variable inside an entity. Variables must be declared within a process.

E42 :Can't open standard library '%s'.

File CYPRESS.VHD or STD.VHD must be available, but couldn't be found in the current path.

E43 :'%s' must be a RECORD

An attempt was made to use a variable or symbol as a record when it was not.

E44 :Must be a constant.

A constant is required.

E45 :Bad direction in range

You defined a bit vector in the wrong direction, e.g., (2 TO 0) or (0 DOWNTO 2).

E46 :Error limit (%d) exceeded

The default error limit is 10 errors. More than 10 errors causes an abort.

E47 :'%s' not a formal port.

The named item is not a formal port.

E48 :Warning limit (%d) exceeded

The default number of warnings has been exceeded, causing an abort.

E49 :Not a polymorphic object file.

The application stopped running before completion or has not been updated correctly. Delete all files that are not part of the design (temporary files such as *.VIF, *.PRS files) and re-run the application.

E50 :Bad polymorphic object file version.

The application stopped running before completion or has not been updated correctly. Delete all files that are not part of the design (temporary files such as *.VIF, *.PRS files), then re-load and re-run the application.

E51 :Variable '%s' already mapped.

A port is mapped to more than one pin.

E52 :Symbol '%s' declared twice

The same name has been declared twice.

E53 :Name '%s' at end of %s, but no name at start

An un-named construct (e.g., a process) was referred to by a named label at its conclusion.

E54 :Name '%s' at end of %s does not match '%s' at start

The label referenced at the end of a construct does not match the name the construct was given at its beginning.

E55 :%s used as an identifier

Attempt to use a reserved word as an identifier.

E56 :Expected %s, but got %s

Syntax error: *Warp* expected a particular character or keyword, but found something else.

E57 :Expected %s

See error E56.

E58 :Expected %s or %s

See error E56.

E59 :Expected %s or %s, but got %s

See error E56.

E60 :Extra COMMA at end of list

A comma appeared after the last item but before the closing parenthesis in a PORT statement.

E61 :'%s' mode not compatible with '%s' mode

An input port was mapped to an output pin, or vice versa.

E62 :Warning:

The beginning portion of a warning message.

E63 :Error writing '%s'.

Couldn't write to the output file once it was created. The most likely causes are a lack of disk space, a bad disk, or no write permission.

E64 :Out of memory.

The application is out of memory. Remove memory resident programs and drivers and re-run the application.

E65 :Fatal error

The beginning portion of a fatal error message.

E66 :Can't index into '%s'

An attempt was made to use the named string as an array when the string is not an array.

E67 :Can't open report file '%s'

Couldn't open or create the report file. Most likely causes: disk is write-protected or out of disk space.

E68 :Missing right parenthesis

Syntax error.

E69 :Use '<=' for signal assignments.

Used ":=" instead of "<=" as assignment operator.

E70 :'%s' type not compatible with '%s' type

A type mismatch was found.

E71 :Positional choice follows named choice

Positional (unspecified formal port) entries may not follow named entries.

E72 :Can't RETURN when in a finite-state-machine

A return statement inside a finite state machine description (a case statement on an enumerated type) is not supported.

E74 :'%s' not a field in record '%s' of type '%s'

An attempt was made to use an inappropriate string as a field in a record.

E75 :Sensitivity name not a SIGNAL

A sensitivity list on a process must consist only of signals.

E76 :Not a '%s' enumeration literal

Inappropriate use was made of the named string as an enumeration literal.

E77 :'%s' not in enumeration '%s' list

Incorrect assignment.

E78 :Positional parameter follows named parameter

A positional (unspecified formal port) parameter may not follow a named parameter.

E79 :%s has no parameter to match '%s'

The port map contains a missing parameter.

E80 :Value '%s' out of range.

A limit, such as a vector limit, is out of range. Re-declare the variable or rework the design.

E81 :Illegal char. '%c' in literal

The named character is not allowed in this literal.

E82 :'%s' conversion to VIF not supported

Synthesis of this VHDL object is not supported.

E83 :'%s' conversion to PLD not supported

Synthesis of this VHDL object is not supported.

E86 :Procedure '%s' body not found

The named procedure was declared, but the body of the procedure could not be located.

E87 :Division by 0

E88 :Operation '%s' not supported

The named arithmetic operation is not supported.

E89 :'%s' must be a CONSTANT or VARIABLE

A constant or variable is required in the named instance.

E90 :VARIABLE '%s' has no value

Only variables having known values at compile time are supported.

E91 :Not in a loop

An EXIT or a NEXT statement was found that was not inside a loop.

E92 :Not in a loop labelled '%s'

See E91.

E95 :FOR variable '%s' not a constant

The named FOR variable is not known at compile time.

E96 :Only integer range supported.

An attempt was made to assign an invalid integer range, such as an enumeration range.

E98 :Negative exponent %ld

Negative exponents are not supported.

E99 :Cannot assign this to an array or record.

Only an aggregate with a list of values or another array or record may be assigned to an array or record.

E100 :Too many values (%d) for '%s' of size %d

The list of values in the aggregate was too large for the aggregate to be assigned to an array or record.

E101 :Can't handle function call '%s' here

The named function call cannot be handled.

E102 :Variable expected.

Symbol was incorrectly declared. The symbol must be a variable, not a signal.

E103 :'%s' must be an ARRAY

The named string must be an array.

E104 :Field name or 'OTHERS' expected

Invalid case statement qualifier.

E105 :Can't delete '%s' from library '%s'

I/O error. The named string cannot be deleted from the named library.

E106 :You need to declare this as a SUBTYPE.

You declared something as a TYPE that should have been declared a SUBTYPE.

E107 :Qualified expressions not supported.

"Qualified expressions" are the VHDL equivalent of type casting (familiar to C programmers). *Warp* doesn't support it.

E108 :Function '%s' body not found

The named function was declared, but the statements associated with the function could not be located.

E109 :Expected '%s' to return a constant

A function that was expected to return a constant didn't.

E110 :Array sizes don't match for operation %s

You tried a dyadic logical operation (AND, OR, XOR) on two bit vectors of different sizes.

E112 :Positional parameter '%s' follows named parameter

A positional parameter may not follow a named parameter.

E113 :No function '%s' with these parameter types

The named function with the specified expressions was not found.

E115 :'%s' is not a visible LIBRARY or PACKAGE name

Warp cannot identify the named string as a valid name for a library or package.

E116 :'%s' not in PACKAGE '%s'

The named string is not in the named package.

E117 :Missing/open field '%s' in %s

Missing parameter or port in the named port list.

E118 :Identifier '%s' starts with a digit

The named identifier begins with a digit. Identifiers must begin with a letter.

E119 :Slice (%d TO %d) is outside array '%s' range (%d TO %d)

The named index range of the slice is outside the named index range of the specified array. The indices of a slice must be within the indices of an array.

E120 :'%s' not an array

The named string is not an array.

E121 :'%s' is not a PACKAGE.

The named string is not a package.

E122 :'%s' must be a SIGNAL or function(SIGNAL).

The named port map parameter may not be an expression, variable, or constant; it must be a signal or signal function.

E123 :Output parm. '%s' must be a SIGNAL or VARIABLE

The named output parameter must be a single entity, like a signal or variable.

E124 :'%s' is not a COMPONENT

The named string is not a component.

E125 :-s requires a path.

The -s command line option requires a path to the library.

E126 :Wrong number (%d) of indices. %d needed.

The listed number of indices is incorrect for the multi-dimensional array.

E127 :Constraint dimension (%d) doesn't match type dimension (%d)

The named constraint dimension doesn't match the named type dimension (constraint must be in the same number of indices as the array).

E129 :Illegal '&' operands: '%s' and '%s'.

Concatenation is not allowed for the named strings. Concatenation is allowed only for identically typed one-dimensional arrays.

E130 :Bad dimension (%d) for attribute '%s'

The named dimension is incorrect for the named array.

E131 :'%s' mapped twice.

The same actual parameter was mapped to two formals.

E133 :'%s' wasn't mapped

The actual parameter identified in the message was not mapped to a formal in the port map.

E134 :Error occurred within '%s' at line %d, column %d in %s.

A descriptive message to inform the user of the exact error location.

E135 :Unexpected '%s'.

Syntax error.

E136 :'%s' is not a known ENTITY

An attempt was made to use the named string as an entity name.

E137 :Can't use OTHERS for unconstrained array '%s'

Syntax error.

E138 :OTHERS must be the last case statement alternative

No case statement alternatives may follow OTHERS.

E139 :Can't use actual function with formal output '%s'

Data is not allowed with the named formal output port.

E140 :Use ':=' for variable assignments.

A string other than ":=" was used for a variable assignment.

E141 :Can't use function(formal) with formal input '%s'

You used an fbx() or fxb() function in the wrong direction.

E209 :Allocation failed for state node

Too many states; design too complex, too many resources used.

E212 :Lrst Resources used up

Design contains too many counters, using too many local resets.

E213 : Attempt to reuse node number %d

Design does not fit and must be modified.

E214 :Unassigned local reset on toggle node number: %d

A local reset is not specified.

E217 :Error writing .cyp file

File I/O error.

E218 : write_cyp - file %s will not open

File I/O error.

E222 : a36fit - no input file name (parm1)

INTERNAL

E223 : a36fit - file %s will not open

File I/O.

E224 : a36fit - error writing .fit file

File I/O.

E225 : write_fit - file %s will not open

File I/O.

E226 :cannot open fitter file: %s

File I/O.

E228 : ftda36 - file %s will not open

File I/O.

E235 : Does not fit: Node %s has more that 16 connections

The named node may have no more than 16 connections because of a hardware constraint, but the design requires more.

E300 :VHDL parser

Message indicating progress of compilation (not really an error).

E350 :error opening file: %s

File I/O.

E351 :error closing file: %s

File I/O.

E357 :Cannot open file: %s

File I/O.

E362 :Error writing to library index

File I/O.

E363 :Error copying '%s' to library

File I/O.

E364 :Can't create library index '%s'

File I/O.

E365 :Bad library index '%s'

The named index for the library is corrupted. Delete the library and load another copy.

E366 :Can't create library '%s' with path '%s'

File I/O.

E367 :Duplicate library entry for '%s'

Duplicate library entry.

E368 :Error deleting existing library index entries

File I/O.

E369 :Can't open library '%s' with path '%s'

File I/O.

E370 :Error reading library '%s'

File I/O.

E371 :Can't find '%s' in library '%s'

File I/O.

E372 :Can't open library module '%s'

File I/O.

E374 :'%s' not a PACKAGE

File I/O.

E375 :'%s' is not in '%s'

The named package is not in the named library.

E376 :User aborted.

The user aborted the run, probably by typing Ctrl-C.

W377 :'%s' from '%s' replaces that from '%s' in library '%s'

Warning message. When a module is compiled into a library, module design units with names the same as existing names overwrite the existing design units.

E378 :Don't work from within your library directory ('%s')

An attempt was made to run *Warp* with the named library directory as the current directory.

E403 :Output and inverted output are both open

Some built in components in the *Warp* library have both output and inverted output pins. One of these outputs pins must be used (both outputs may not be left open).

E404 :Can't use both output and inverted output

Similar to E403, except both outputs may not be used.

E406 :Illegal '%s'. Only a NOT is allowed here.

Some expressions allow only a NOT statement.

E407 :Bad Mealy input/output pair: %s / %s

An attempt was made to select an inappropriate named input pin for the named output.

E410 :Missing RTL port

An attempt was made to use a built-in (RTL) component without specifying a required input port.

E411 :Illegal Mealy function '%s'

Mealy output expressions can only take the form of an (optional) Mealy input signal followed by AND, OR, or XOR, followed by an OR list.

E412 :Port '%' in RTL component '%s' is missing or improper

An attempt was made to use the named RTL component without assigning the named port or assigning it inappropriately.

E413 :Variable '%s' has no node # assigned.

A signal is used in an expression, but the signal has not been assigned to any node or pin in a chip.

E414 :Expression too complex: %s

Output assignments must follow a specific format.

E416 :'%s' has no signal assigned.

The design has no signal assigned because it is too large. See the .RPT file for more information.

E418 :%d too many macrocells in design

The design contains too many macrocells and must be modified to use fewer macrocell resources.

W419 :Can't synthesize signal initialization (ignored)

Initialization is ignored when signals are declared as non-enumerated types and initialization is attempted.

E421 :Bad Mealy input/output pair: %s / %s

An attempt was made to select an inappropriate named input pin for the named output.

E422 :Target '%s' of '%s' already assigned to node %d

An attempt was made to assign the named target to the named node. The name has been used previously.

E423 :Target '%s' must be an input/output pin

An attempt was made to connect the named signal to something other than an I/O pin.

E427 :Target must be a variable.

Target may not be a constant or an aggregate.

E428 :Could not find entity '%s (%s)' for component '%s'

Warp was unable to find the named entity for the named component.

E429 :Could not find entity '%s' for component '%s'

See E428.

E430 :No entity for architecture '%s'

No entity exists for the named architecture.

E431 :'%s' has already been used as an output.

You attempted to assign two outputs to the same pin.

E432 :Conversion from AONG to '%s' not supported

Finite state machine enumerated type synthesis is not supported.

E434 :Unsupported PLD '%s'

An attempt was made to compile to an unsupported or nonexistent device.

E436 :Only simple waveform supported

Warp allows only simple wave forms with no timing information.

E438 :RTL '%s' not supported for %s

An attempt was made to use the named built-in component for an incorrect named device.

E440 :Missing RTL field '%s'

An RTL component is missing a port.

E443 :SIGNAL '%s' has more than one driver

An attempt was made to use more than one driver with the named signal.

E444 :Can't handle chip '%s'

Warp does not support the named chip.

E500 :Can't handle '%s' expression.

An unsupported operation was included in an expression.

E501 :'%s' State_node has no exit.

In a state machine, *Warp* cannot find an exit for the named state.

E503 :Initializer of '%s' must be an enum. literal

Attempt to initialize a state variable to a value not included in its enumerated list.

W507 :No entry to state '%s' of '%s'.

Warp cannot find an entry to the named state, so the state is not used and can be removed.

W508 :No exit from state '%s' of '%s'.

Warp cannot find an exit from the named state, so the state is a sink.

E509 :Array sizes don't match for operation %s

You attempted a dyadic logical operation (AND, OR, XOR) on two bit vectors of different sizes.

E510 :Operation '%s' not implemented for vectors

Implementation not supported.

E511 :BIT or array required

Warp does not synthesize integers; a bit or array is required.

E512 :Unsupported expression type '%s'.

Some operators in the expression are not supported.

E513 :'%s' not a BIT or array

Warp does not synthesize integers; a bit or array is required.

E600 :Array lengths %ld, %ld don't match for '%s'.

For the named operation, the named array lengths do not match.

E601 :Bad operand types '%s' and '%s' for operator '%s'.

Type check violation.

E602 :Bad operand type '%s' for operator '%s'.

Type check violation.

E603 :Wrong character '%c' in string

Type check violation.

E604 :Expression type '%s' does not match target type '%s'.

Type check violation.

E605 :BOOLEAN required here.

Type check violation.

E606 :Numeric expression required here.

Type check violation.

E607 :Choice type '%s' doesn't match case type '%s'.

Type check violation.

E608 :'%s' not readable. Mode is OUT.

The mode is defined as OUT, so you can't put it on the right side of an expression.

E609 :'%s' not writable. Mode is IN.

The mode is defined as IN, so you can't put it on the left side of an expression.

E610 :SEVERITY_LEVEL required here

An ASSERT statement requires a severity level.

E611 :RETURN not in a function or procedure

A return statement was found that was not inside a function or procedure.

E612 :Can't RETURN a value from a procedure

The application cannot return a value from a procedure; a function is required.

E613 :Function RETURN needs a value.

You attempted to return from a function without specifying a value.

E614 :Return type '%s' does not match function type '%s'.

Syntax error.

E615 :Can't assign to CONSTANT '%s'.

Invalid constant.

E3001 :illegal device\n

Check legal device names in manual.

W3002 :phase ignored\n

The fitter is ignoring a statement in the .PLA file. You can, too.

E3027 :Internal Error\n

Call your Cypress representative.

Appendix

B

Glossary

B. Glossary of VHDL Terminology

Listed here are the definitions of terms that you will encounter frequently in using VHDL. Note that the context for the definitions is that of synthesis (as opposed to simulation) modeling.

actual – in port maps used in binding architectures, the name of the pin to which the signal is being mapped.

analysis – the examination of a VHDL description to ensure that it complies with VHDL syntax rules. During analysis, *Warp* determines the design elements (packages, components, entities, and architectures) that make up the description and places these design elements into a VHDL library and an associated index. The library and index are then available for use in synthesis by other descriptions.

architecture – the part of a VHDL description that specifies the behavior or structure of an entity. Entities and architectures are always paired in VHDL descriptions.

attribute – a named characteristic of a VHDL item. An attribute can be a value, function, type, range, signal, or constant. It can be associated with one or more names in a VHDL description, including entity names, architecture names, labels, and signals. Once an attribute value is associated with a name, the value of the attribute for that name can be used in expressions.

binding architecture – an architecture used to map the ports of an entity to the pins of a PLD.

bit vector – a collection of bits addressed by a common name and index number (an array of bits).

component – a description of a design that can be used in another design.

component declaration – that part of a VHDL description that defines what a component is. It is usually encapsulated in a package for export via the library mechanism.

component instantiation statement – a statement in a VHDL description that creates an instance of a previously defined component.

concurrent statement – a statement in an architecture that executes or is modeled concurrently with all other statements in the architecture.

constant declaration – an element of a VHDL description that declares a named data item to be a constant value.

design architecture – an architecture paired with a previously declared entity that describes the behavior or structure of that entity.

design unit – an entity declaration, a package declaration, an architecture body, or a package body.

entity – that part of a VHDL description that lists or describes the ports (the interfaces to the outside) of the design. An entity describes the names, directions, and data types of each port.

formal – in port maps used in binding architectures, the signal name on the component.

function – a subprogram whose invocation is an expression, and therefore returns a value. See *subprogram* and *procedure*.

function body – a portion of a VHDL description that defines the implementation of a function.

function declaration – a portion of a VHDL description that defines the parameters passed to and from a function invocation, such as the function name, return type, and list of parameters.

function invocation – a reference to a function from inside a VHDL description.

generic map – a VHDL construct that enables an instantiating component to pass environment values to an instantiated component. Typically, a generic map is used to size an array or a bit vector or provide true/false environment values.

instantiation – the process of creating an instance (a copy) of a component and connecting it to other components in the design.

library – a collection of previously analyzed VHDL design units. In *Warp* a library is a directory containing an index and one or more .VHD files.

package – a collection of declarations, including component, type, subtype, and constant declarations, that are intended for use by other design units.

package body – the definition of the elements of a package. A package body typically contains the bodies of functions declared within the package.

package declaration – the declaration of the names and values of components, types, subtypes, constants, and functions contained in a package.

port – a point of connection between a component and anything that uses the component.

port map – an association between the ports of a component and the signals of an entity instantiating that component. Within the context of a binding architecture, a port map is a VHDL construct that associates signals from an entity with pins on a PLD.

procedure – a subprogram whose invocation is a statement, and therefore does not return a value. See *subprogram* and *function*.

procedure body – a portion of a VHDL description that defines the implementation of a procedure.

procedure declaration – a portion of a VHDL description that defines the parameters passed to and from a procedure invocation, such as the procedure's name and list of parameters.

procedure invocation – a reference to a procedure from inside a VHDL description.

process – a collection of sequential statements appearing in a design architecture.

sensitivity list – a list of signals that appears immediately after the process keyword and specifies when the process statements are activated. The process is executed when any signal in the sensitivity list changes value.

sequential statement – a statement appearing within a process. All statements within a process are executed sequentially.

signal – a data path from one component to another.

signal declaration – a statement of a signal name, its direction of flow, and the type of data that it carries.

subprogram – a sequence of declarations and statements that can be invoked repeatedly from different locations in a VHDL description. VHDL recognizes two kinds of subprograms: procedures and functions.

subtype – a restricted subset of the legal values of a type.

subtype declaration – a VHDL construct that declares a name for a new type, known as a subtype. A subtype declaration specifies the base type and declares the value range of the subtype.

synthesis – the production of a file to be mapped to a PLD containing the design elements extracted from VHDL descriptions during the analysis phase. The file is a technology mapped structural netlist description that is fitted to a user-specified device.

type – an attribute of a VHDL object that determines the kind of value the object can hold. Examples of types are bit and X01Z. Objects of type bit can hold values '0' or '1'. Objects of type X01Z can hold values of '0', '1', 'X' ("don't care") or 'Z' ("high-impedance").

type declaration – a VHDL construct that allows you to define new types as a sub-type or combination of existing types.

variable – a VHDL object that can hold a data value, which can be used during analysis or synthesis, but is not itself synthesized.

Appendix

C

BNF of Supported VHDL

C. BNF of Supported VHDL

This appendix presents a simplified Backus-Naur Form (BNF) of the VHDL subset supported by *Warp*.

Conventions Used In This Appendix

The form of a VHDL description is described by means of context-free syntax, together with context-dependent syntactic and semantic requirements expressed by narrative rules. The context-free syntax of the language is described using a simple variant of Backus-Naur Form, in particular:

(a) Lower case words, some containing embedded underlines, are used to denote syntactic categories, for example:

formal_port_list

(b) Boldface is used to denote reserved words and literal characters, for example:

array

(c) A vertical bar separates alternative items, unless it appears in boldface immediately after an opening brace, in which case it stands for itself:

letter_or_digit ::= letter | digit

choices ::= choice { | choice}

(d) Square brackets enclose optional items. Thus, the following two rules are equivalent

return_statement ::= return [expression];

return_statement ::= return; | return expression;

(e) Braces enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

term ::= factor {multiplying_operator factor}

term ::= factor | term multiplying_operator factor

(f) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *type*_name and *subtype*_name are both equivalent to name alone.

(g) The term simple_name is used for any occurrence of an identifier that already denotes some declared entity.

BNF

actual_designator ::=
    expression
    | *signal*_name
    | *variable*_name
    | **open**

actual_parameter_part ::=
    *parameter*_association_list

actual_part ::=
    actual_designator
    | *function*_name (actual_designator)

adding_operator ::=
    + | **-** | **&**

aggregate ::=
    (element_association {**,** element_association})

alias_declaration ::=
    alias identifier**:**subtype_indication **is** name**;**

architecture_body ::=
    architecture identifier **of** *entity*_name **is**
    architecture_declarative_part
    begin
        architecture_statement_part
    end [*architecture*_simple_name]**;**

architecture_declarative_part ::=
    {block_declarative_item}

architecture_statement_part ::=
    {concurrent_statement}

array_type_definition ::=
    unconstrained_array_definition
    | constrained_array_definition

assocation_list ::=
    assocation_element {**,** association_element}

association_element ::=
    [formal_part **=>**] actual_part

attribute_declaration ::=
    attribute identifier**:**type_mark**;**

attribute_designator ::=
    *attribute*_simple_name

attribute_name ::=
    prefix ' attribute_designator [(expression)]

attribute_specification ::=
    attribute attribute_designator **of** entity_specification
    is expression

block_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | alias_declaration
    | component_declaration
    | attribute_specification
    | use_clause

block_declarative_part ::=
    { block_declarative_item }

block_statement ::=
    *block*_label:
    block block_declarative_part
    begin
        block_statement_part
    end block [block_label]**;**

block_statement_part ::=
    { concurrent_statement }

case_statement ::=
    case expression **is** case_statement_alternative
    { case_statement_alternative }
    end case;

case_statement_alternative ::=
    when choices => sequence_of_statements

choice ::=
    simple_expression
    | discrete_range
    | *element*_simple_name
    | **others**

choices ::=
    choice [| choice]

component_declaration ::=
    component identifier
    [local_generic_clause]
    [local_port_clause]
    end component;

component_instantiation_statement ::=
    *instantiation*_label**:**
        *component*_name
            [generic_map_aspect]
            [port_map_aspect]**;**

composite_type_definition ::=
    array_type_definition
    | record_type_definition

concurrent_signal_assignment_statement ::=
    [label**:**] conditional_signal_assignment
    | [label**:**] selected_signal_assignment

concurrent_statement ::=
block_statement
| process_statement
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
| generate_statement

condition ::=
*boolean*_expression

conditional_signal_assignment ::=
target **<=** options conditional_waveforms**;**

conditional_waveforms ::=
{ waveform **when** condition **else** } waveform

condition_clause ::=
until condition

constant_declaration
constant identifier_list**:**subtype_indication [**:=**expression]**;**

constrained_array_definition ::=
array index_constraint **of** *element*_subtype_indication

constraint ::=
range_constraint
| index_constraint

declaration ::=
type_declaration
| subtype_declaration
| object_declaration
| interface_declaration
| alias_declaration
| attribute_declaration
| component_declaration
| entity_declaration
| subprogram_declaration
| package_declaration

designator ::=
    identifier | operator_symbol

direction ::=
    to | **downto**

discrete_range ::=
    *discrete*_subtype_indication
    | range

element_association ::=
    [choices **=>**] expression

element_declaration ::=
    identifier_list : element_subtype_definition**;**

element_subtype_definition ::=
    subtype_indication

entity_class ::=
    entity | **architecture** | **configuration**
    | **function** | **package** | **type**
    | **subtype** | **constant** | **signal**
    | **variable** | **component** | **label**

entity_declaration ::=
    entity identifier **is**
        entity_header
        entity_declarative_part
    [**begin**
        entity_statement_part]
    end [*entity*_simple_name]**;**

entity_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

entity_declarative_part ::=
    {entity_declarative_item}

entity_designator ::=
    simple_name | operator_symbol

entity_header ::=
    [*formal*_generic_clause]
    [*formal*_port_clause]

entity_name_list ::=
    entity_designator {, entity_designator}
    | **others**
    | **all**

entity_specification ::=
    entity_name_list **:** entity_class

enum_literal ::=
    identifier | character_literal

enum_type_definition ::=
    (enum_literal {,enum_literal})

exit_statement ::=
    exit [*loop*_label] [**when** condition];

expression ::=
    relation { **and** relation}
    | relation { **or** relation}
    | relation { **xor** relation}
    | relation { **nand** relation}
    | relation { **nor** relation}

factor ::=
    primary
    | **not** primary

formal_designator_ ::=
    generic_name
    | port_name
    | parameter_name

formal_parameter_list ::=
    *parameter*_interface_list

formal_part ::=
    formal_designator
    | function_name (formal_designator)

full_type_declaration ::=
    type identifier **is** type_definition**;**

function_call ::=
    *function*_name [(actual_parameter_part)]

generate_statement ::=
    *generate*_label:
        generation_scheme **generate**
            { concurrent_statement }
        end generate [*generate*_label]**;**

generation_scheme ::=
    for generate_parameter_specification
    | **if** condition

generic_clause ::=
    generic (generic_list)**;**

generic_list ::=
    *generic*_interface_list

identifier_list ::=
    identifier {, identifier}

if_statement ::=
    if condition **then** sequence_of_statements
    {**elsif** condition **then** sequence_of_statements}
    [**else** sequence_of_statements]
    end if;

indexed_name ::=
    prefix (expression {, expression})

index_constraint ::=
    (discrete_range {, discrete_range})

index_subtype_definition ::=
    type_mark **range <>**

integer_type_definition ::=
    range_constraint

interface_constant_declaration ::=
    [**constant**] identifier_list**:** [**in**] subtype_indication
        [**:=***static*_expression]

interface_declaration ::=
    interface_constant_declaration
    | interface_signal_declaration
    | interface_variable_declaration

interface_element ::=
    interface_declaration

interface_list ::=
    interface_element {**;** interface_element}

interface_signal_declaration ::=
    [**signal**] identifier_list**:** [mode] subtype_indication [**bus**]
        [**:=***static*_expression]

interface_variable_declaration ::=
    [**variable**] identifier_list: [mode] subtype_indication
        [:=*static*_expression]

iteration_scheme ::=
    while condition
    | **for** *loop*_parameter_specification

label ::=
    identifier

literal ::=
    numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
    | **null**

logical_operator ::=
    and | **or** | **nand** | **nor** | **xor**

loop_statement ::=
    [*loop*_label:]
        [iteration_scheme] **loop** sequence_of_statements
    end loop [*loop*_label]

miscellaneous_operator ::=
    ** | **abs** | **not**

mode ::=
    in | **out** | **inout** | **buffer** | **linkage**

multiplying_operator ::=
    * | / | **mod** | **rem**

name ::=
    simple_name
    | operator_symbol
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name

next_statement ::=
    next [*loop*_label] [**when** condition];

null_statement ::=
    null;

numeric_literal ::=
    abstract_literal
    | physical_literal

object_declaration ::=
    constant_declaration
    | signal_declaration
    | variable_declaration

operator_symbol ::=
    string_literal

options ::=
    [**guarded**] [**transport**]

package_body ::=
    package body *package*_simple_name **is**
        package_body_declarative_part
    end [*package*_simple_name];

package_body_declarative_item ::=
    function_declaration
    | function_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | alias_declaration
    | use_clause

package_body_declarative_part ::=
    {package_body_declarative_item}

package_declaration ::=
    package identifier **is**
        package_declarative_part
    end [*package*_simple_name];

package_declarative_item ::=
    function_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

package_declarative_part ::=
    {package_declarative_item}

parameter_specification ::=
    identifier **in** discrete_range

port_clause ::=
    port (port_list);

port_list ::=
    *port*_interface_list

prefix ::=
    name
    | function_call

primary ::=
    name
    | literal
    | aggregate
    | function_call
    | qualified_expression
    | type_conversion
    | allocator
    | (expression)

procedure_call_statement ::=
    *procedure*_name [(actual_parameter_part)];

process_declarative_item ::=
    function_declaration
    | function_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

process_statement ::=
    [*process*_label]
    process [(sensitivity_list)]
        process_declarative_part
    begin
        process_statement_part
    end process [process_label];

process_statement_part :=
    { sequential statement }

range ::=
    *range*_attribute_name
    | simple_expression direction simple_expression

range_constraint ::=
    range range

record_type_definition ::=
    record
        element_declaration
        {element_declaration}
    end record

relation ::=
    simple_expression [relational_operation simple_expression]

relational_operator ::=
    + | /= | < | <= | > | >=

return_statement ::=
    return [expression]**;**

scalar_type_definition ::=
    enum_type_definition
    | integer_type_definition

selected_name ::=
    prefix.suffix

selected_signal_assignment ::=
    with expression **select**
        target **<=** options selected_waveforms

selected_waveforms ::=
    {waveform **when** choices,}
    waveform **when** choices

sensitivity_clause ::=
    on sensitivity_list

sensitivity_list ::=
    *signal*_name {, *signal*_name}

sequence_of_statements ::=
    { sequential_statement }

sequential_statement ::=
    wait_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement

sign ::=
    + | **-**

signal_assignment_statement ::=
    target **<= [transport]** waveform**;**

signal_declaration ::=
    signal identifier_list**:**subtype_indication
        [signal_kind] [**:=**expression]

signal_kind ::=
    register | **bus**

simple_expression ::=
    [sign] term {adding_operator term}

simple_name ::=
    identifier

slice_name ::=
    prefix (discrete_range)

subprogram_body ::=
    subprogram_specification **is**
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [designator]**;**

subprogram_declaration ::=
    subprogram_specification;

subprogram_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

subprogram_declarative_part ::=
    {subprogram_declarative_item}

subprogram_specification ::=
    procedure designator **[**(formal_parameter_list)]
    | **function** designator [(formal_parameter_list)] **return** type_mark

subprogram_statement_part ::=
    {sequential_statement}

subtype_declaration ::=
    subtype identifier **is** subtype_indication**;**

subtype_indication ::=
    [*resolution_function*_name} type_mark [constraint]

suffix ::=
    simple_name
    | character_literal
    | operator_symbol
    | **all**

target ::=
    name
    | aggregate

term ::=
    factor {multiplying_operator factor}

timeout_clause ::=
    for *time*_expression

type_declaration ::=
    full_type_declaration

type_definition ::=
    scalar_type_definition
    | composite_type_definition
    | access_type_definition

type_mark :=
    *type*_name
    | *subtype*_name

unconstrained_array_definition ::=
    array (index_subtype_definition{, index_subtype_definition})
        of element_subtype_indication

variable_assignment_statement ::=
    target **:=** expression**;**

variable_declaration ::=
    variable identifier_list**:**subtype_indication [**:=**expression]**;**

wait_statement ::=
    wait [sensitivity_clause] [condition_clause] [timeout_clause]**;**

waveform ::=
    waveform_element {**,** waveform_element}

waveform_element ::=
*value*_expression

Appendix

D

Bibliography

D. Bibliography

The following list of books and articles on VHDL is not intended to be comprehensive, but merely to point you to further reading that can help you become skilled in the use of VHDL simulation and synthesis tools.

Books:

1. Armstrong, J.R., *Chip-Level Modeling with VHDL*, Englewood cliffs, NJ: Prentice-Hall, 1988.
2. Barton, D., *A First Course in VHDL*, VLSI Systems Design, January 1988.
3. Bhasker, J., *A VHDL Primer*, Englewood Cliffs, NJ: Prentice Hall, 1992.
4. Carlson, S., *Introduction to HDL-Based Design Using VHDL*, Synopsys Inc., 1991.
5. Coelho, D., *The VHDL Handbook*, Boston: Kluwer Academic, 1988.
6. Harr, R. E., et al., *Applications of VHDL to Circuit Design*, Boston: Kluwer Academic, 1991.
7. *IEEE Standard VHDL Language Reference Manual, Std 1076-1987*, IEEE, NY, 1988.
8. *IEEE Standard 1076 VHDL Tutorial*, CLSI, Maryland, March 1989.
9. Leung, S., *ASIC System Design With VHDL*, Boston: Kluwer Academic, 1989.
10. Leung, S., and M. Shanblatt, *ASIC System Design With VHDL: A Paradigm*, Boston: Kluwer Academic, 1989.

11. Lipsett, R., C. Shaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Boston: Kluwer Academic, 1989.

12. Mazor, S., and P. Langstraat, *A Guide to VHDL*, Boston: Kluwer Academic, 1992.

13. Perry, D., *VHDL*, New York: McGraw-Hill, 1991.

14. *Military Standard 454*, U.S. Government Printing Office, 1988.

15. Schoen, J.M., *Performance and Fault Modeling with VHDL*, Englewood Cliffs, NJ: Prentice Hall, 1992.

Articles:

1. Armstrong, J.R., et al., "The VHDL Validation Suite," *Proc. 27th Design Automation Conference*, June 1990, pp. 2-7.

1. Barton, D., "Behavioral Descriptions in VHDL," *VLSI Systems Design*, June 1988.

2. Bhasker, J., "Process-Graph Analyzer: A Front-End Tool for VHDL Behavioral Synthesis," *Software Practice and Experience*, vol. 18, no. 5, May 1988.

3. Bhasker, J., "An Algorithm for Microcode Compaction of VHDL Behavioral Descriptions," *Proc. 20th Microprogramming Workshop*, December 1987.

4. Coelho, D., "VHDL: A Call for Standards," *Proc. 25th Design Automation Conference*, June 1988.

5. Coppola, A., and J. Lewis, "VHDL for Programmable Logic Devices," *PLDCON'93*, Santa Clara, March 29-31, 1993.

6. Coppola, A., J. Freedman, et al., "Tokenized State Machines for PLDs and FPGAs," *Proceedings of IFIP WG10.2 / WG10.5 Workshop on Control-Dominated Synthesis from a Register-Transfer-Level Description*, Grenoble, France, 3-4 September, 1992, G. Saucier and J. Trilhe, editors, Elsevier Science Publishers.

7. Coppola, A., J. Freedman, et al., "VHDL Synthesis of Concurrent State Machines to a Programmable Logic Device," *Proc. of the IEEE VHDL International User's Forum*, May 3-6, 1992, Scottsdale, Arizona.

8. Coppola, A., and M. Perkowski, "A State Machine PLD and Associated Minimization Algorithms," *Proc. of the FPGA'92 ACM / SIGDA First International Workshop on Field-Programmable Gate Arrays*, Berkeley, California, Feb. 16-18, 1992, pp. 109-114.

9. Dillinger, T.E., et al., "A Logic Synthesis System for VHDL Design Description," *IEEE ICCAD-89*, Santa Clara, California.

10. Farrow, R., and A. Stanculescu, "A VHDL Compiler Based on Attribute Grammar Methodology," *SIGPLAN 1989*.

11. Gilman, A.S., "Logic Modeling in WAVES," *IEEE Design and Test of Computers*, June 1990, pp. 49-55.

12. Hands, J.P., "What Is VHDL?", *Computer-Aided Design*, vol. 22, no. 4, May 1990.

13. Hines, J., "Where VHDL Fits Within the CAD Environment," *Proc. 24th Design Automation Conference*, 1987.

14. Kim, K., and J. Trout, "Automatic Insertion of BIST Hardware Using VHDL," *Proc. 25th Design Automation Conference*, 1988.

15. Moughzail, M., et al., "Experience with the VHDL Environment," *Proc. 25th Design Automation Conference*, June 1988.

16. Saunders, L., "The IBM VHDL Design System," *Proc. 24th Design Automation Conference*, 1987.

17. Ward, P.C., and J. Armstrong, "Behavioral Fault Simulation in VHDL," *Proc. 27th Design Automation Conference*, June 1990, pp. 587-593.

Index