DesignWare® Building Block IP

User Guide

DWBB_201806.0
O-2018.06
June 2018

Copyright Notice and Proprietary Information

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com

Contents

Preface

1.1 About This Manual

This manual explains the use of the DesignWare Building Block (DWBB) IP, and is intended for users of Synopsys synthesis tools. DesignWare Building Block IP are part of the overall DesignWare IP Library.

These building blocks are technology-independent, micro architecture-level components that are tightly integrated into the Synopsys synthesis environment.

1.1.1 Web Resources

- Fast access to DWBB component instantiation code snippets, function inference snippets (where appropriate), and links to downloads and documentation:

  http://www.synopsys.com/dw/buildingblock.php

- Product documentation for all DesignWare IP Library products:

  http://www.synopsys.com/dw/dwlibdocs.php

- Up-to-date information about the latest DesignWare Library IP and verification IP:

  http://www.designware.com

- General Synopsys Licensing (SCL) information:

  http://www.synopsys.com/Support/LI/Pages/default.aspx

1.1.2 Manual Overview

This manual contains the following chapters and appendixes:

Preface (this document)	Describes the manual and lists the typographical conventions and symbols used in it; tells how to get technical assistance.
"Introduction"	Introduces the DesignWare Library IP and briefly describes the DWBB IP.
"Using DesignWare Building Block IP: Basic"	Provides a basic look at how to use the DWBB IP.
"Using DesignWare Building Block IP: Advanced"	Provides an advanced look at how to use the DWBB IP.

"Using Licensed Implementations"	Describes how to use Licensed implementation of DesignWare Building Block IP.
"Standard Synthetic Operators"	Provides a list of HDL operators that are mapped to synthetic operators in the Synopsys standard synthetic library standard.sldb.
"Downloading the Latest Building Block IP"	Describes how you can download the latest DesignWare Building Block IP electronic software transfer (EST) release.

1.2 Customer Support

- First, prepare the following debug information, if applicable:
  - The coreConsultant-dwfc tool can create a tar file with information that is useful to debug problems. First, select the component, then use the **File > Build Debug Tar-file** menu item.
  - In addition, it is useful to provide the Support Center with the following information:
    - Create a waveforms file (such as VPD or VCD)
    - Identify the hierarchy path to the DWBB instance
    - Identify the timestamp of any signals or locations in the waveforms that are not understood
- Then, contact Support Center, with a description of your question and supplying the above information, using one of the following methods:
  - *For fastest response*, use the SolvNet website. If you fill in your information as explained below, your issue is automatically routed to a support engineer who is experienced with your product. The **Sub Product 1** entry is critical for correct routing.

    Go to http://solvnet.synopsys.com/EnterACall and click on the link to enter a call. Provide the requested information, including:
    - **Product:** *DesignWare Building Blocks*
    - **Product Version:** *DWBB_201806.0*
    - **Problem Type:**
    - **Issue Severity:**
    - **Problem:** Provide a short title for the issue you have encountered.

      Describe the details to clarify your problem; include names and configurations of DWBB components you are using and any warning or error messages. For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood.

    After creating the case, attach any debug files you created in the previous step.
  - Or, send an e-mail message to support_center@synopsys.com (your e-mail will be queued and then, on a first-come, first-served basis, manually routed to the correct support engineer):
    - Include the information as identified above so your email can be routed correctly.
    - For simulation issues, include the timestamp of any signals or locations in waveforms
    - Attach any debug files you created in the previous step.
  - Or, telephone your local support center:
    - http://www.synopsys.com/Support/GlobalSupportCenters

1 Introduction

The DesignWare Building Block IP (formally called Foundation Library) is a collection of reusable intellectual property blocks that are tightly integrated into the Synopsys synthesis environment. Using DesignWare Building Block IP allows transparent, high-level optimization of performance during synthesis. With the large number of parts available, design reuse is enabled and significant productivity gains are possible.

This library contains high-performance implementations of Basic Library IP plus many IP that implement more advanced arithmetic and sequential logic functions. The DesignWare Building Block IP consists of:

- Basic Library: A set of IP bundled with HDL Compiler that implements several common arithmetic and logic functions.
- Logic: Combinational and Sequential IP.
- Math: Arithmetic and Trigonometric IP.
- Digital Signal Processing (DSP) IP: FIR and IIR filters.
- Memory: Registers, FIFOs, and FIFO Controllers, Synchronous and Asynchronous RAMs, and Stack IP.
- Interface: Clock Domain Crossing (CDC) IP.
- Application Specific: Data Integrity, Interface, JTAG IP, and others.

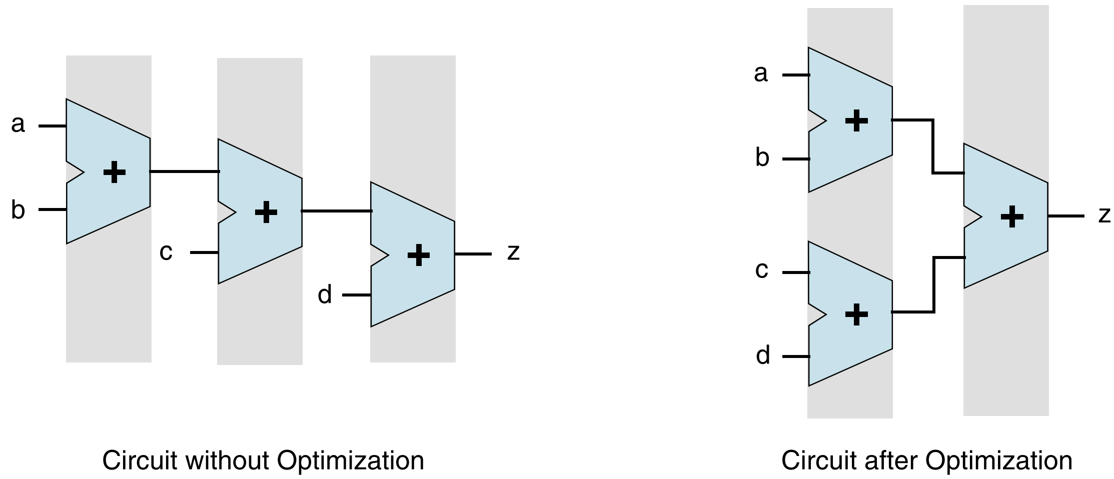
1.1 A Library Strategy for High-Level Design

The DesignWare Building Block IP enables a high degree of automation in design reuse, making it possible for the Synopsys synthesis tools to transparently perform various high-level optimizations.

For example, when you use the HDL addition operator "+" in a design description, HDL Compiler (Verilog or VHDL) *infers* the need for an adder resource, and puts an abstract representation of the addition operation into your circuit netlist. That representation — called a *synthetic operator* — is manipulated by the high-level optimization algorithms HDL Compiler applies to your design. These optimizations include arithmetic optimization, resource sharing and pin permutation.

Arithmetic optimization uses the rules of algebra to improve circuit area and performance by rearranging operations. For example, the expression `a + b + c + d` describes three levels of cascaded addition operations (Figure 1-1). Arithmetic optimization can rearrange this expression to be `(a + b) + (c + d)`, which may result in faster logic (having only two levels of cascaded additions).

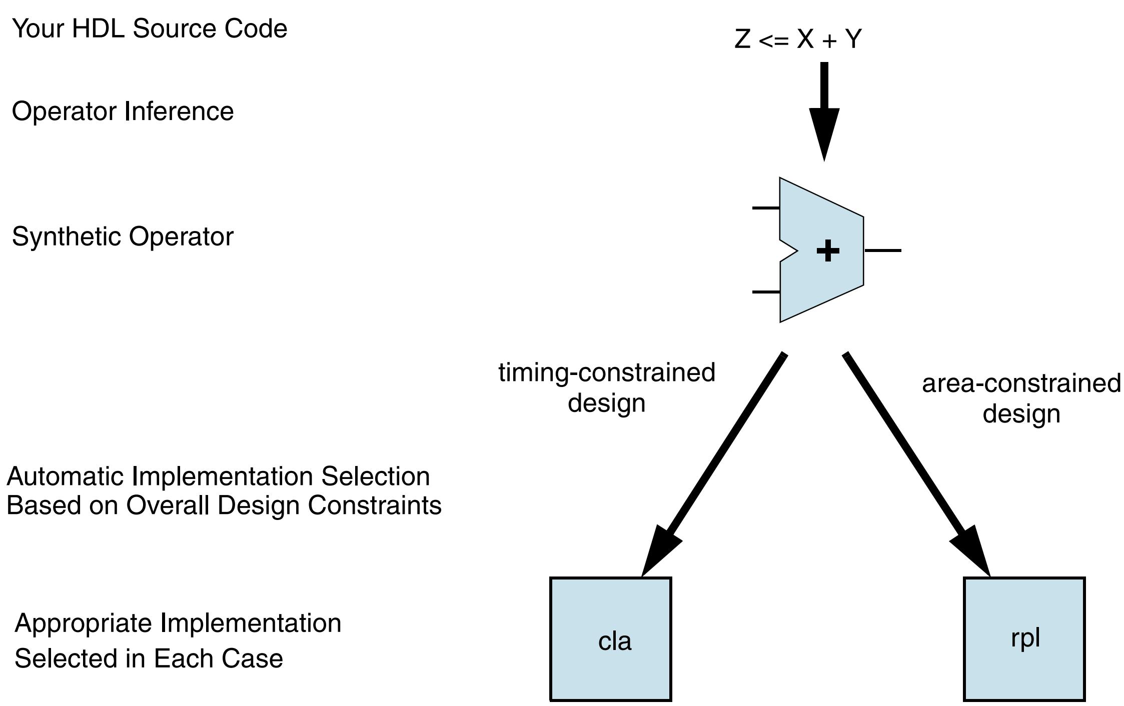
Figure 1-1 Arithmetic Optimization



Resource sharing allows similar operations that do not overlap in time to be carried out by the same physical hardware. *Pin permutation* takes advantage of the fact that some operations (like addition and multiplication) are not affected by switching their input ports. For details, refer to the *VHDL Compiler Reference Manual* or the *HDL Compiler for Verilog Reference Manual*.

With DesignWare Building Block IP, a given operation can be implemented in many different ways, with the choice of implementation in a particular circuit context left up to the synthesis tools. For example, unsigned addition can be implemented with either a ripple or a carry look-ahead architecture. You can let the synthesis tools choose which architecture to use, based on the optimization constraints you have set on your overall design (Figure 1-2).

Figure 1-2 Implementation Selection

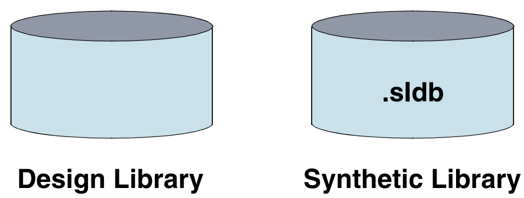


1.1.1 The Structure of the Supporting Libraries

The DesignWare Building Block IP has two parts, a *design library*, and a *synthetic library* (Figure 1-3):

- A design library is a UNIX directory that contains circuit descriptions for the various IP architectures. These are usually parameterizable.
- A synthetic library is a binary file (with a `.sldb` filename extension) that links the circuits in a design library to the Synopsys synthesis tools.

Figure 1-3 Supporting Libraries



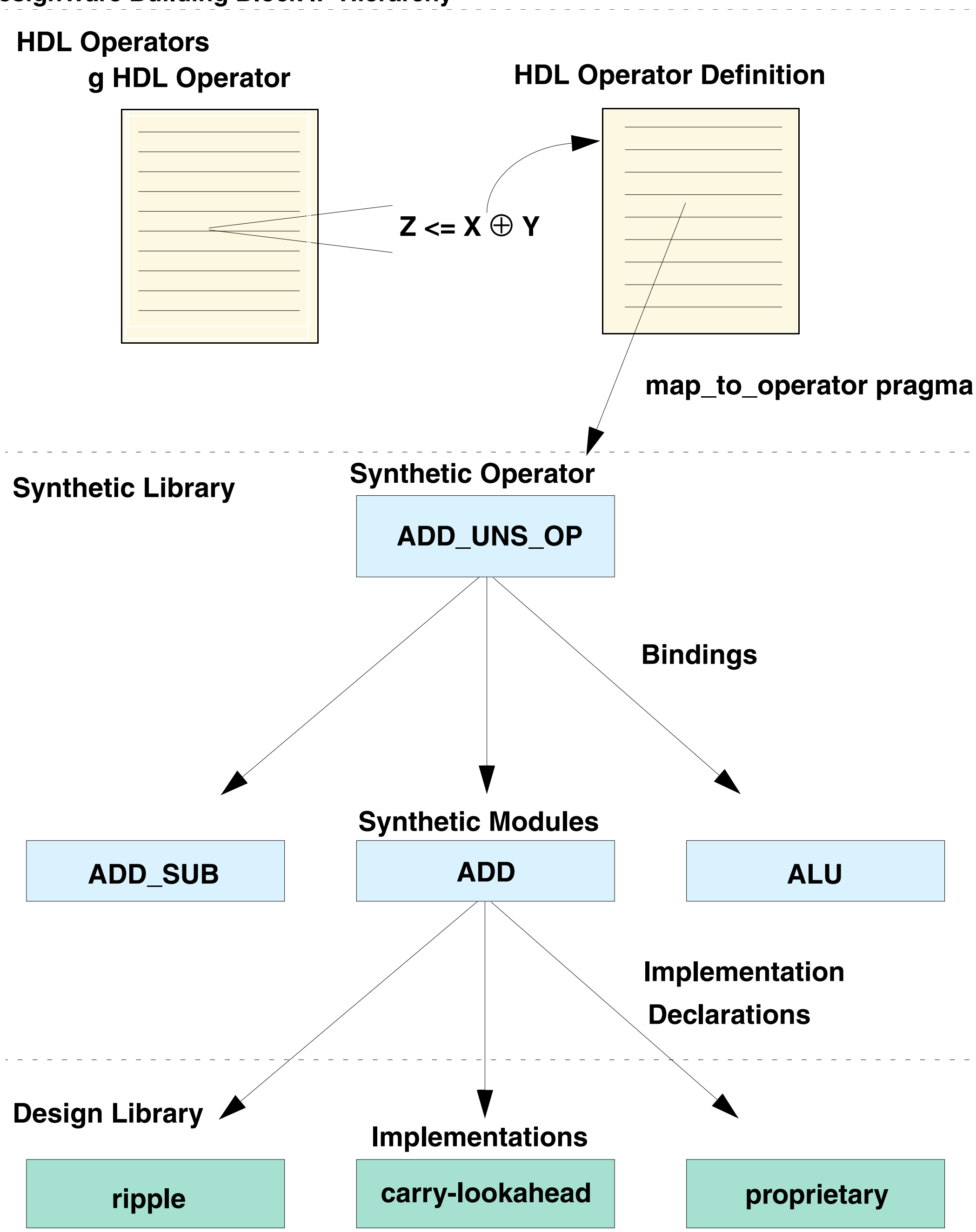
The circuit descriptions in a design library are stored in binary formats immediately usable by the Synopsys tools. The circuits can vary from a technology-specific netlist or hard macro that will not be altered by synthesis, up to a full hierarchical description of a parameterizable, optimizable design.

The synthetic library contains the information that enables the synthesis tools to perform high-level optimizations, including implementation selection.

Connections between your source code, synthetic libraries, and design libraries are established by means of a hierarchy of abstractions (Figure 1-4). *HDL operators* are associated with *synthetic operators*, which are in turn bound to *synthetic modules*. Each synthetic module can have multiple architectural realizations called *implementations*.

Another class of DesignWare Building Block IP are *subblocks*, which have only one implementation and are only instantiated. Subblocks are useful for large parts, such as the error checking and correction IP, that do not have multiple implementations. As a result, subblocks do not use the same hierarchy of abstractions as described in Figure 1-4.

Figure 1-4 DesignWare Building Block IP Hierarchy



1.1.1.1 HDL Operators

An *HDL operator* is a VHDL or Verilog language construct that manipulates input values to produce output values. Some operators are built into the language, like `+`, `-`, and `*`; user-defined subprograms (functions and procedures) are also considered HDL operators.

DesignWare Building Block IP implement many of the built-in HDL operators. These operators include `+`, `-`, `*`, `<`, `>`, `<=`, `>=`, `/`, and the operations defined by `if` and `case` statements. Each operator has a definition written in HDL. Each definition contains a simulatable specification for the operator behavior, and, optionally, a `map_to_operator` pragma that links the HDL operator to an equivalent synthetic operator. The "`/`" operator is required for the DesignWare license.

Many HDL operators, including most of the built-in infix operators, are mapped by default to synthetic operators in the Synopsys standard synthetic library, `standard.sldb`.

1.1.1.2 Synthetic Libraries

A synthetic library contains definitions for synthetic operators, synthetic modules, and bindings. It also contains declarations that associate synthetic modules with their implementations. The implementations themselves reside in the corresponding design library.

- *Synthetic operator* – Represents the operation called for by the HDL operator. The synthesis tools perform high-level optimizations like arithmetic optimization and resource sharing by manipulating synthetic operators.
- *Synthetic module* – Defines a common interface for a family of implementations. All implementations of a given module have the same ports and the same input-output behavior. (This term is not to be confused with the term "module" in Verilog.)
- *Bindings* – Associate synthetic operators with synthetic modules. For example, a binding associates the synthetic operator for addition with the adder module (you can also say that the synthetic addition operator is *bound* to the adder module). More than one synthetic operator can be bound to a given synthetic module, and each operator can be bound to more than one module.
- *Implementation declarations* – Link synthetic modules to implementations in a design library. Implementation declarations thus connect the synthetic library with the design library.

1.1.1.3 Design Library

The design library contains the actual circuit implementations that perform the functions you call for when you include DesignWare Building Block IP in your design.

The DesignWare Building Block IP concepts of *synthetic module* and *implementation* closely correspond to the VHDL concepts of *entity* and *architecture*. An implementation can be viewed as an architectural realization of a synthetic module. An implementation can be anything from a technology-specific netlist to a synthesizable RTL-level design description.

1.2 How Do I Use DesignWare Building Block IP?

You determine which IP libraries are available to the tools by setting certain `dc_shell-t` variables. These setup variables are discussed in "Using DesignWare Building Block IP: Basic" on page 17.

You include IP in a design either through *operator inference* or *component instantiation*. In operator inference, synthetic operators are automatically inferred from the presence of particular operators in your HDL code. In component instantiation, your HDL code explicitly instantiates a synthetic module. Detailed procedures for inference and instantiation are given in "Using DesignWare Building Block IP: Basic" on page 17.

1.2.1 Inference

Operator inference occurs when the synthesis tool encounters an HDL operator whose definition ties it to a synthetic operator. The tool finds the required synthetic operator, inserts it into your design, and performs high-level optimizations on the resulting netlist.

The tool then performs implementation selection. It looks in the available synthetic libraries to determine which modules have bindings that link them to the synthetic operator. Implementations of all these modules are candidates for selection.

To characterize the implementations for comparison, the synthesis tool creates a pre-optimized model for each one. The timing and area characteristics of the models serve as the basis for implementation selection. To avoid duplicate work, the models are stored in a UNIX directory called the *synthetic library cache*. "Using DesignWare Building Block IP: Advanced" on page 33 includes a discussion of cache-management issues.

The implementation that best meets the optimization goals you have set for the entire circuit is selected. The chosen implementation is inserted, by default, as a level of hierarchy in your design, which is then mapped to the target technology and optimized.

Note It is possible for you to steer or even override the automated implementation-selection process. "Using DesignWare Building Block IP: Advanced" on page 33 details this advanced feature.

1.2.2 Instantiation

Synthetic modules also are explicitly instantiatable. This is sometimes necessary because inferring using the HDL Compiler family of tools is available only for operations that can be realized with combinational logic. (Note, however, that with Behavioral Compiler, sequential parts can be inferred.) Instantiation is possible for any synthetic module.

The following example shows one way to instantiate the parameterized synthetic module `DW01_add` in VHDL. The lines of code that refer to the adder module are shown in bold.

```
library IEEE, DWARE, DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_foundation_comp.all;
entity DW01_add_inst is
generic ( inst_width : NATURAL := 8 );
  port ( inst_A   : in std_logic_vector(inst_width-1 downto 0);
         inst_B   : in std_logic_vector(inst_width-1 downto 0);
         inst_CI  : in std_logic;
         SUM_inst : out std_logic_vector(inst_width-1 downto 0);
```

```
          CO_inst  : out std_logic );
end DW01_add_inst;

architecture inst of DW01_add_inst is
begin

  -- Instance of DW01_add
  U1 : DW01_add
  generic map ( width => inst_width )
  port map ( A => inst_A, B => inst_B, CI => inst_CI,
             SUM => SUM_inst, CO => CO_inst );
end inst;
```

The following example shows how to instantiate the same synthetic module in Verilog:

```
module DW01_add_inst( inst_A, inst_B, inst_CI, SUM_inst, CO_inst );

  parameter width = 8;

  input [width-1 : 0] inst_A;
  input [width-1 : 0] inst_B;
  input inst_CI;
  output [width-1 : 0] SUM_inst;
  output CO_inst;

  // Instance of DW01_add
  DW01_add #(width)
    U1 (.A(inst_A), .B(inst_B), .CI(inst_CI), .SUM(SUM_inst), .CO(CO_inst) );
endmodule
```

When the synthesis tool encounters the reference to `DW01_add`, it attempts to resolve the reference by looking for a module of that name in the available synthetic libraries.

When the synthesis tool finds the appropriate synthetic module, it looks for all the implementations that are associated with that module, and performs implementation selection.

2
Using DesignWare Building Block IP: Basic

You include DesignWare Building Block IP in your designs by inference or by instantiation. The use of DesignWare Building Block IP enables you to take full advantage of high-level optimization, automatic implementation selection, and design re-use.

The primary tasks involved in using this IP include the following:

- "Setting Up Your Environment" on page 17
- "Displaying Information" on page 19
- "Inferring IP with HDL Operators" on page 22
- "Instantiating IP" on page 25

2.1 Setting Up Your Environment

To make DesignWare Building Block IP accessible to the Synopsys tools, you specify paths to the necessary synthetic library files and design library directories.

Synthetic operators and modules for many basic arithmetic and logic functions are defined in the Synopsys standard synthetic library, `standard.sldb`. These include IP that support built-in HDL functions. This library, and the associated design libraries, are set up for you as part of the Synopsys software installation.

2.1.1 Accessing Synthetic Libraries

To access modules in synthetic libraries other than `standard.sldb`, you must set two `dc_shell-t` variables: `synthetic_library` and `link_library`.

The `synthetic_library` variable is analogous to the `target_library` variable in technology libraries. Set `synthetic_library` as a list of `.sldb` files that you want to use in the `compile` or `replace_synthetic` commands. When synthetic operators and modules are processed during a compile command, the operators, bindings, modules, and implementations of the designated library or libraries are used. Synthetic libraries are searched in the order that they are listed. If two modules in different libraries have the same name, the module in the first library listed is used.

As with target technology libraries, you must include synthetic libraries in your `link_library` variable.

For example, to use the synthetic library `my_synlib.sldb` and the technology library `my_techlib.db`, set the following `dc_shell-t` variables:

```
dc_shell-t> set target_library [list my_techlib.db]
dc_shell-t> set synthetic_library [list my_synlib.sldb]
dc_shell-t> set link_library [list my_techlib.db my_synlib.sldb]
```

Note You do not need to set your `synthetic_library` variable or your `link_library` variable to include `standard.sldb`. This library is included by default.

You cannot disable the standard synthetic library, but you can disable individual modules and implementations by using the `set_dont_use` command. To replace a particular implementation, disable it with `set_dont_use` and provide a substitute from another synthetic library (the `set_dont_use command` is explained in "Disabling Selected Synthetic Modules and Implementations" on page 39).

2.1.2 Accessing Design Libraries

To use a synthetic library, you must specify the location of the design library that contains the netlists and HDL files for the corresponding implementations.

Note Paths to Synopsys-supplied design libraries are established as part of the software installation procedure. You need to define design library paths explicitly only if you are working with design libraries that do not come from Synopsys.

You can define a design library in either of two ways: you can use the `define_design_lib` command during a `dc_shell-t` session, or you can set up a mapping from the library name to a UNIX path name in your design library file. The design library file, by default, is `.synopsys_sim.setup`.

2.1.2.1 Using the define_design_lib Command

The `dc_shell-t` command `define_design_lib` maps a library name to the design library directory. The syntax is

```
define_design_lib library_name -path directory
```

`library_name` is the name of the design library, and `-path` *`directory`* is the name of a UNIX directory.

For example, the synthetic library `dw_foundation.sldb` refers Design Compiler to implementations in the design library. The following command defines the path where these implementations are located:

```
dc_shell-t> define_design_lib dw_foundation
 -path [format "%s%s" $synopsys_root "/dw/dw_foundation/lib"]
```

`synopsys_root` is the Synopsys root directory. This command maps the name dw_foundation to the path `$SYNOPSYS/dw/dw_foundation/lib`.

You can implement `define_design_lib` commands in your `.synopsys_dc.setup` file.

2.1.2.2 Using the Design Library File

The other method of mapping design library names to UNIX directory paths is to use your design library file. This file contains mapping information in the following format:

```
library_name : directory_name
```

For example, the path for the `my_synthlib` design library is defined by

```
my_synthlib : $SYNOPSYS/dw/my_synthlib/lib
```

By default, the design library file is named `.synopsys_sim.setup`. See the *System Installation and Configuration Guide* for a description of this file.

You can specify a different file to be your design library file by setting the `design_library_file` variable in your `.synopsys_dc.setup` file.

2.1.3 Other Set-up Options

There are various other DesignWare-related set-up options you can use to configure your environment. Options are available for

- *Releasing HDL Compiler Licenses* – Implementation selection requires a Synopsys HDL Compiler license (either HDL Compiler for Verilog or VHDL Compiler). In previous releases, the HDL license was kept checked out through the whole process of compilation. A new variable, `hdl_keep_license`, was added in Version 3.1 to allow you to release the HDL Compiler license after the implementation-selection phase.

  When `hdl_keep_license` is `FALSE`, the HDL Compiler license is kept only during implementation selection. For compatibility, the default value of `hdl_keep_license` is `TRUE`. To economize license use, set `hdl_keep_license` to `FALSE` in your `.synopsys_dc.setup` file.

- *Setting Up the Cache* – The synthesis tools create component models during implementation selection. To avoid duplicating work, they cache these models in a UNIX directory called the synthetic library cache. For a description of cache setup options, see "Maintaining the Synthetic Library Cache" on page 43.

2.2 Displaying Information

The commands `report_synlib` and `report_design_lib` show you the contents of the synthetic libraries and design libraries that are currently accessible. The command `get_attribute` shows which subdesigns in a design hierarchy came from synthetic libraries.

2.2.1 Reporting the Contents of Synthetic Libraries

To determine the contents of a synthetic library, use the `report_synlib` command. The resulting report lists all of the operators and their pins; followed by all the modules with their pins, parameters, attributes, implementations and bindings.

The syntax of `report_synlib` is

```
report_synlib library_name [list_of_modules]
```

The arguments and command-line options are

`library_name`

Name of the library to report on.

`list_of_modules`

Restricts the report to the modules in the list.

Implementations are annotated to show the attributes specified for them, including legality and priority formulas and priority set IDs.

The library must either be loaded into `dc_shell-t`, or accessible through the `search_path`. The command `list -libraries` displays the names of libraries that are currently loaded.

See the `report_synlib` man page for further details.

The following example illustrates the use of the `report_synlib` command on the synthetic library `standard.sldb`.

```
dc_shell-t> report_synlib standard.sldb
****************************************
Report : library
Library: standard.sldb
Version: 1997.08
Date : Fri Oct 17 10:52:04 1997
****************************************
Library Type : Synthetic
Tool Created : 1997.08
Date Created : Dec 16, 1993
Library Version : 3.1
Operators:
Operator Ports Dir
----------------------------------------------------------
ADD_TC_CI_OP A in
    B    in
    CI   in
    Z    out
ADD_TC_OP A in
    B    in
    Z    out
```

2.2.2 Identifying the Contents of Design Libraries

To determine the packages, entities, and architectures defined in your design libraries, use the `report_design_lib` command. The syntax of this command is

```
report_design_lib [-libraries][-designs][-architectures]
                    [-packages][library_names]
```

The command-line options are:

- `libraries`

  Lists the library names and their respective UNIX directories, but does not list their contents.

- `designs`

  Lists the designs in libraries (Verilog modules, VHDL entities, or configurations).

- `architectures`

  Lists the designs in libraries and their architectures (which correspond to implementations in Synthetic Libraries).

- `packages`

  Lists the VHDL packages contained in libraries.

- *library_names*

Is the list of libraries whose contents are to be displayed. If `library_names` is not used, all libraries are displayed by default.

The design library report also lists

- parameterized designs (designs that can perform functions on data of varying sizes)
- the most recently analyzed architecture of each design
- files without a source
- outdated files

If you execute the `report_design_lib` command without options, the command lists all contents of all available design libraries.

The following example shows the use of the `report_design_lib` command to list the mapping information and contents of one user's current design libraries, which are `WORK, SYNOPSYS, IEEE, DW01`, and `DW02`.

```
dc_shell-t> report_design_lib
****************************************
Report : hdl libraries
Version: 1997.08
Date   : Fri Oct 17 10:52:04 1997
****************************************
Contents of current design libraries
 DEFAULT (/usr/remote/designs/WORK)
 WORK (/usr/remote/designs/WORK)
 entity : p DWSL_addov
 architecture : DWSL_addov(cla)
 architecture :  m DWSL_addov(proprietary)
 architecture : DWSL_addov(rpl)
 DW01 (/synopsys/sparc/dw/dw01/lib)
 entity : DW01_ADD_AB
 architecture :  m DW01_ADD_AB(str)
 entity : DW01_ADD_AB1
 architecture :  m DW01_ADD_AB1(str)
 ...
 DW02 (/synopsys/sparc/dw/dw02/lib)
 entity : p DW02_booth
 architecture : DW02_booth(sim)
 architecture :  m DW02_booth(str)
 package : DW02_components
 ...
 IEEE (/synopsys/sparc/packages/IEEE/lib)
 package : GS_Types
 package : STD_LOGIC_SIGNED
 package : STD_LOGIC_UNSIGNED
 package : std_logic_1164
 ...
 SYNOPSYS (/synopsys/sparc/packages/synopsys/lib)
 package : ATTRIBUTES
 package : BV_ARITHMETIC
p --- This design has parameters.
m --- This architecture is the most recently analyzed.
```

2.2.3 Identifying Synthetic Objects in a Design

It can be useful to detect the presence of synthetic objects in designs. For example, you might want to understand the mapping that occurred from your HDL operator to a synthetic operator. Another example would be to see which implementation of a module Design Compiler chose, then determine whether to set the implementation manually.

The `get_attribute` command in `dc_shell-t` can report the presence of certain synthetic objects. Table 2-1 shows which objects `dc_shell-t` can detect and related attributes to arch for in a design.

Table 2-1 Synthetic Objects Detectable in Designs

Synthetic Object	Attribute to Search For
synthetic operators	is_synlib_operator
synthetic modules	is_synlib_module

For details of the attributes and command syntax, see the `get_attribute` man page.

The following example shows a query of the cell `U1` and reference `DW01_add_width5` for the attribute `is_synlib_module`.

```
dc_shell-t> get_attribute [list U1 DW01_add_width5] is_synlib_module
Performing get_attribute on cell 'U1'.
Performing get_attribute on reference 'DW01_add_width5'.
{"true", "true"}
```

2.3 Inferring IP with HDL Operators

One method of incorporating DesignWare Building Block IP into your design is to use an HDL operator whose definition maps it to a synthetic operator. Inferred synthetic operators enter into the high-level optimizations performed by HDL Compiler. (See "Introduction" on page 9.)

This is the procedure:

1. Use the HDL operator in your design description.
2. Analyze and elaborate your HDL file.
3. Set constraints and compile.

During compilation, Design Compiler automatically selects the appropriate synthetic module and implementation to perform the operation.

Note To infer arithmetic operations in VHDL on non-numeric types, you need to declare the `std_logic_arith` package and use the Synopsys-defined data types `SIGNED` and `UNSIGNED` from that package.

Suppose you want to infer an adder in your design by using the HDL operator for addition. You can perform the following tasks to accomplish this.

2.3.1 Task 1. Inferring an Adder in Your Description

The following example is a VHDL design description that uses an addition operator (+) to infer an adder module.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity  DW01_add_oper is
  generic(wordlength: integer := 8);
  port(in1,in2 : in STD_LOGIC_VECTOR(wordlength-1 downto 0);
    sum : out STD_LOGIC_VECTOR(wordlength-1 downto 0));
end DW01_add_oper;

architecture oper of DW01_add_oper is
  signal in1_signed,in2_signed,
         sum_signed: SIGNED(wordlength-1 downto 0);
begin
  in1_signed <= SIGNED(in1);
  in2_signed <= SIGNED(in2);
  -- infer the "+" addition operator
  sum_signed <= in1_signed + in2_signed;
  sum <= STD_LOGIC_VECTOR(sum_signed);

end oper;
```

Design Compiler selects the appropriate module and implementation to use.

> **Note** You have the option of specifying in advance which module and implementation will be selected. Instructions for manual control of implementation selection are given in "Using DesignWare Building Block IP: Advanced" on page 33.

The following example is a Verilog design description that uses an addition operator (+) to infer a synthetic module (an adder).

```
module DW01_add_oper(in1,in2,sum);
  parameter wordlength = 8;

  input [wordlength-1:0] in1,in2;
  output [wordlength-1:0] sum;

  assign  sum = in1 + in2;

endmodule
```

2.3.2 Task 2. Analyzing and Elaborating Your File

You use the `analyze` command to translate your HDL code into a form that the Synopsys tools can use. Each HDL design unit gives rise to one or more files in the target design library. You then use the `elaborate` command to link the analyzed design units into a single design database.

In the following example, the VHDL file `addition.vhdl` is analyzed and stored in the design library `LIB`. (To analyze the Verilog file `addition.v` you would use the `-format verilog` option instead of `-format vhdl`.) The `elaborate` command then builds the design.

The `report_cell` command, which lists the synthetic library cells used in a design, shows that the HDL addition operator (+) has been replaced by the appropriate synthetic operator.

```
dc_shell-t> analyze addition.vhdl -format vhdl -library LIB
/usr/remote/designs/addition.vhdl:
1
dc_shell-t> elaborate addition -lib LIB
Current design is now 'addition'
1
dc_shell-t> report_cell
****************************************
Report : cell
Design : addition
Version: 1997.08
Date   : Fri Oct 17 10:52:04 1997
****************************************
Attributes:
b - black box (unknown)
h - hierarchical
n - noncombinational
r - removable
s - synthetic operator
u - contains unmapped logic
Cell     Reference     Library     Area     Attributes
----------------------------------------------------------
plus     *ADD_TC_OP_8_8_8          0.00     s, u
----------------------------------------------------------
Total 1 cells                      0.00
1
```

Attention When working with unmapped synthetic operators (prior to compilation), do not use the `extract` command, or try to write your design out in VHDL or Verilog format. Design Compiler does not recognize synthetic operator names as valid HDL identifiers. You may not be able to simulate these HDL files or read them back into Design Compiler.

2.3.3 Task 3. Setting Constraints and Compiling

In the following example, constraints are set and compilation is initiated.

```
dc_shell-t> set_output_delay 4 [all_outputs]
Performing set_output_delay on port 'sum[7]'.
...
1
dc_shell-t> compile
...
1
```

The next example shows reports generated after compilation.

The `report_cell` command shows that the synthetic operator has been replaced by the implementation of a synthetic module. `addition` is implemented with a single hierarchical cell with an area of 69 gate-equivalents.

The `report_resources` command displays a resource sharing report and an implementation report for `addition`. The `Current Implementation` section shows that Design Compiler selected the `cla` implementation of the synthetic module `DW01_add` to build this cell.

```
dc_shell-t> report_cell
****************************************
Report : cell
Design : addition
Version: 1997.08
Date   : Fri Oct 17 10:52:04 1997
****************************************
Attributes:
  b - black box (unknown)
  BO - reference allows boundary optimization
  h - hierarchical
  n - noncombinational
  r - removable
  u - contains unmapped logic
Cell     Reference     Library        Area        Attributes
----------------------------------------------------------
add_18/plus addition_DW01_add_8_0   69.00         BO, h
----------------------------------------------------------
Total 1 cells                       69.00
1
dc_shell-t> report_resources
****************************************
Report : resources
Design : addition
Version: 1997.08
Date   : Fri Oct 17 10:52:04 1997
****************************************
Resource Sharing Report for design addition in file
/usr/remote/designs/addition.vhdl
==========================================================
|          |        |            | Contained | Contained |
| Resource | Module | Parameters | Resources | Operations|
==========================================================
| r29      |DW01_add| width=8    |           |add_18/plus|
==========================================================
Implementation Report
==========================================================
|            |         |Current       | Set             |
|Cell        |Module   |Implementation| Implementation  |
==========================================================
|add_18/plus |DW01_add |cla           |                 |
==========================================================
```

2.4 Instantiating IP

Another way to incorporate a DesignWare Building Block IP in your design is to instantiate a synthetic module explicitly. This is the procedure:

1. Include a reference to the synthetic module in your description.
2. Analyze and elaborate your file.
3. Set constraints and compile.

During compilation, Design Compiler automatically selects the appropriate implementation for the module based on the constraints you set.

Note You have the option of specifying in advance which implementation will be selected. Instructions for manual control of implementation selection are given in "Using DesignWare Building Block IP: Advanced" on page 33.

Suppose you want to instantiate an adder in your design. You can perform the following tasks to accomplish this.

2.4.1 Task 1. Including a Reference to an Adder in Your Description

The following example shows how to instantiate the parameterized synthetic module `DW_fp_mac` in VHDL. The lines of code that refer to the adder module are shown in bold.

```
library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_Foundation_comp_arith.all;

entity DW_fp_mac_inst is
     generic (
      inst_sig_width : POSITIVE := 23;
      inst_exp_width : POSITIVE := 8;
      inst_ieee_compliance : INTEGER := 0
      );
     port (
      inst_a : in std_logic_vector(inst_sig_width+inst_exp_width downto 0);
      inst_b : in std_logic_vector(inst_sig_width+inst_exp_width downto 0);
      inst_c : in std_logic_vector(inst_sig_width+inst_exp_width downto 0);
      inst_rnd : in std_logic_vector(2 downto 0);
      z_inst : out std_logic_vector(inst_sig_width+inst_exp_width downto 0);
      status_inst : out std_logic_vector(7 downto 0)
      );
    end DW_fp_mac_inst;

architecture inst of DW_fp_mac_inst is
begin

    -- Instance of DW_fp_mac
    U1 : DW_fp_mac
    generic map (
       sig_width => inst_sig_width,
       exp_width => inst_exp_width,
       ieee_compliance => inst_ieee_compliance
       )
    port map (
       a => inst_a,
       b => inst_b,
       c => inst_c,
       rnd => inst_rnd,
       z => z_inst,
       status => status_inst
       );

end inst;
```

The following example shows how to instantiate the parameterized synthetic module DW_fp_mac in Verilog. The lines of code that refer to the adder module are shown in bold.

```
module DW_fp_mac_inst( inst_a, inst_b, inst_c, inst_rnd, z_inst, status_inst );

parameter inst_sig_width = 23;
parameter inst_exp_width = 8;
parameter inst_ieee_compliance = 0;

input [inst_sig_width+inst_exp_width : 0] inst_a;
input [inst_sig_width+inst_exp_width : 0] inst_b;
input [inst_sig_width+inst_exp_width : 0] inst_c;
input [2 : 0] inst_rnd;
output [inst_sig_width+inst_exp_width : 0] z_inst;
output [7 : 0] status_inst;

    // Instance of DW_fp_mac
    DW_fp_mac #(inst_sig_width, inst_exp_width, inst_ieee_compliance) U1 (
          .a(inst_a),
          .b(inst_b),
          .c(inst_c),
          .rnd(inst_rnd),
          .z(z_inst),
          .status(status_inst) );

endmodule
```

Since a specific implementation is not instantiated in this file, implementation selection is performed automatically by Design Compiler.

2.4.2 Task 2. Analyzing and Elaborating Your File

In the following example, the VHDL file `adder.vhdl` is analyzed and stored in the design library `LIB`. The `elaborate` command then builds the adder. The `report_cell` command shows that the synthetic module `DW01_add` has been instantiated as a black box (see the cell attributes).

```
dc_shell-t> analyze adder.vhdl -format vhdl -library LIB
/usr/remote/designs/adder.vhdl:
1
dc_shell-t> elaborate adder -lib LIB
Current design is now 'adder'
1
dc_shell-t> report_cell

****************************************
Report : cell
Design : adder
Version: 1997.08
Date   : Fri Oct 17 10:52:04 1997
****************************************
Attributes:
  b - black box (unknown)
  h - hierarchical
  n - noncombinational
  r - removable
  S - synthetic module
  u - contains unmapped logic
Cell    Reference         Library    Area    Attributes
----------------------------------------------------------
U1      DW01_add_width8              0.00    b, S
----------------------------------------------------------
Total 1 cells                        0.00
1
```

2.4.3 Task 3. Setting Constraints and Compiling

In the following example, constraints are set (using the `set_output_delay` command) and compilation is initiated.

```
dc_shell-t> set_output_delay 4 [all_outputs]
Performing set_output_delay on port 'sum[7]'.
...
Performing set_output_delay on port 'sum[0]'.
Performing set_output_delay on port 'carry_out'.
1
dc_shell-t> compile
...
1
```

The following example shows reports generated after compilation. The `report_cell` command shows that the synthetic module `DW01_add` has been implemented with a single hierarchical cell with area 80.

The `report_resources` command displays a resource sharing report and an implementation report for `adder`. The `Current Implementation` section shows that Design Compiler selected the `cla` implementation of the synthetic module `DW01_add` to build this cell.

```
dc_shell-t> report_cell
****************************************
Report : cell
Design : adder
Version: 1997.08
Date  : Fri Oct 17 10:52:04 1997
****************************************
Attributes:
```

```
  b - black box (unknown)
  BO - reference allows boundary optimization
  h - hierarchical
  n - noncombinational
  r - removable
  u - contains unmapped logicCell

         Reference     Library    Area    Attributes
----------------------------------------------------------
U1       adder_DW01_add_8_0       80.00   BO, h
----------------------------------------------------------
Total 1 cells                     80.00
1

dc_shell-t> report_resources
****************************************
Report : resources
Design : adder
Version: 1997.08
Date   : Fri Oct 17 10:52:04 1997
****************************************
Resource Sharing Report for design adder
==========================================================
|          |        |              | Contained   | Contained |
| Resource | Module | Parameters   | Resources   | Operations|
==========================================================
| r28      |DW01_add| width=8      |             | U1        |
==========================================================
Implementation Report
==========================================================
|            |         |Current         | Set             |
|Cell        |Module   |Implementation  | Implementation  |
==========================================================
|U1          |DW01_add |cla             |                 |
==========================================================
1
```

2.4.4 Pre-Compiling Subblocks

Compiling large subblocks can be a lengthy process. Pre-elaborating and pre-compiling subblocks speeds up the compilation.

The first step is to elaborate a subblock, then save the results. The following example shows the command you use to save the results of elaborating the `DW_ecc` IP.

```
elaborate DW_ecc -arch str -lib DW04 -p "8,5,0"
write -hier -o DW_ecc_8_5_0.db
```

The next time you link a design that contains the DW_ecc, Design Compiler automatically reads in and links the file `DW_ecc_8_5_0.db` without re-elaborating it.

You can also save the results of a compilation. The following example shows the commands to save the compilation of the `DW_ecc`.

```
elaborate DW_ecc -arch str -lib DW04 -p "8,5,0"
/* set design constraints  */
compile
dont_touch   DW_ecc_8_5_0
```

```
write -hier -o DW_ecc_8_5_0.db
```

On subsequent compilations, the `DW_ecc` will not be re-optimized. If you want to re-optimize the design for new constraints, you can do either of the following:

- remove the `DW_ecc_8_5_0.db` file and start over, or
- re-compile the subblock again

The following example shows how to re-compile the subblock again.

```
remove_attribute DW_ecc_8_5_0 dont_touch
compile
dont_touch   DW_ecc_8_5_0
write -hier -o DW_ecc_8_5_0.db
```

2.4.5 Optional: Using the VHDL Components Package

In VHDL, before a synthetic module can be instantiated in the architecture body, it must be declared as a component in the design's architecture. You can avoid the necessity of providing individual declarations for every synthetic module by declaring a library component package instead.

Every synthetic library from Synopsys includes a package that declares all designs in that library. The name of this package is `libraryname_components`, where `libraryname` is the name of the design library. If you declare this package with the `library` and `use` statements at the beginning of your VHDL description, you do not need to explicitly declare synthetic modules before you instantiate them.

For example, to declare all designs in the `DW01` library, include the following statements at the beginning of your code:

```
library IEEE,DWARE,DWARE;
use DWARE.DWpackages.all;
use DWARE.DW_foundation_comp.all;
```

The following example demonstrates the use of this package (the required statements are shown in bold).

```
library IEEE,DWARE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_foundation_comp.all;

entity DW01_add_inst is
  generic(wordlength: integer := 8);
  port(in1, in2: in STD_LOGIC_VECTOR(wordlength-1 downto 0);
    ci: in STD_LOGIC;
    sum: out STD_LOGIC_VECTOR(wordlength-1 downto 0);
    cout: out STD_LOGIC);
end DW01_add_inst;

architecture inst of DW01_add_inst is
begin
  -- instantiate DW01_add
  U1: DW01_add
    generic map(width => wordlength)
    port map(A => in1, B => in2, CI => ci, SUM => sum, CO => cout);
end inst;
```

2.4.6 Viewing DesignWare Building Block IP in Design Analyzer

When you display a schematic in Design Analyzer, DesignWare Building Block IP are drawn on a separate layer called `designware_layer`. You can use the `set_layer` command in Design Compiler or the `View -> Style` menu in Design Analyzer to change the appearance of specific layers. You can modify the color, line width, and plot line width, and specify whether the layer will be visible or invisible.

2.5 Summary of Procedure for Using DesignWare Building Block IP

1. To access synthetic libraries other than `standard.sldb`, you must include those libraries in your `synthetic_library` and `link_library` variables.
2. To access design libraries (other than those associated with standard.sldb), you must specify paths to the library directories using either the `dc_shell-t` command `define_design_lib`, or the mapping mechanism in the design library file `.synopsys_sim.setup`.
3. To list the operators, modules, and implementations available in a synthetic library, use the `report_synlib` command.
4. To list the packages, entities, and architectures available in a design library, use the `report_design_lib` command.
5. To infer DesignWare Building Block IP:
   a. include the appropriate HDL operator in your design description,
   b. analyze and elaborate your design description (analyze and elaborate commands), and
   c. set constraints and compile (compile command).
6. To instantiate a synthetic module:
   a. include a reference to the module in your design description,
   b. analyze and elaborate, and
   c. set constraints and compile.
7. When using VHDL, you can replace multiple component declarations with `library` and `use` statements that refer to the library components package. For example, to make all IP from the DW library available without having to declare each one, include the following statements in your VHDL design description:

```
library IEEE,DWARE,DWARE;
use DWARE.DWpackages.all;
use DWARE.DW_foundation_comp.all;
```

3
Using DesignWare Building Block IP: Advanced

The Synopsys synthesis tools match HDL operators with synthetic operators and select implementations based on your constraints. During the process of implementation selection, the tools create timing models of implementations and cache them in a UNIX directory called the *synthetic library cache*. By default, all these processes are handled automatically.

To meet your specific requirements, you can use manual controls to guide or to override many of the automated processes associated with DesignWare Building Block IP.

Advanced use of DesignWare Building Block IP includes:

- "Controlling Module and Implementation Selection" on page 33
- "Controlling Hierarchy" on page 42
- "Maintaining the Synthetic Library Cache" on page 43

3.1 Controlling Module and Implementation Selection

During compilation, the synthesis tools automatically select the best implementation for each IP that occurs in your design. When you include DesignWare Building Block IP by operator inferencing, the tools can choose among all the implementations of all the synthetic modules bound to the operator. When you instantiate a synthetic module, the tools can choose among the implementations of that module. This process of automatic *implementation selection* occurs (by default) each time you compile your design.

You have the capability, however, to control this process in various ways. You can do the following:

- Override the automatic selection and explicitly determine which synthetic module and implementation are used. See "Manually Selecting Modules and Implementations" on page 34.
- Replace unmapped synthetic operators in your design with generic logic. See "Replacing Unmapped Synthetic Operators" on page 39.
- Prevent Design Compiler from selecting specified implementations. See "Disabling Selected Synthetic Modules and Implementations" on page 39.
- Prioritize the implementations of a given module. See "Prioritizing Implementations" on page 39.

- Control when implementation selection takes place. See "Controlling Incremental Implementation Selection" on page 40.

There are several reasons to override the automatic selection of modules and implementations as follows:

- You already have the specific implementation of a module in mind.
- You want a specific layout implementation for a hard macro (on the basis of area, aspect ratio, pinout, and so on).
- You want to select the lowest-power implementation of a module to keep overall system power consumption as low as possible.

Note You can also manually control resource sharing. For information on manual resource sharing, refer to Chapter 7 of the *HDL Compiler for Verilog Reference Manual* or Chapter 9 of the *VHDL Compiler Reference Manual*.

3.1.1 Manually Selecting Modules and Implementations

You can manually select synthetic modules and implementations by including the appropriate directives in your HDL design description, or by issuing the appropriate commands after reading your design into `dc_shell-t`. The following sections show you the procedure to follow, first in the case of operator inferencing, then in the case of component instantiation.

3.1.1.1 Manual Selection with Operator Inferencing

To force an HDL operator to infer a specific synthetic module and implementation, you add the appropriate code to your HDL descriptions. The syntax differs for VHDL and Verilog and are described separately.

3.1.1.1.1 Example in VHDL

Suppose you want to use the VHDL addition operator to imply the ripple-carry implementation of a synthetic adder module in your design. The VHDL description in the following example shows how this is done.

The VHDL addition operator (+) is manually bound to a specific resource (adder module `DW01_add`) and implementation (ripple-carry implementation `rpl`). The lines of code shown in bold define the operator's binding to the specific module and implementation.

```
library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use synopsys.attributes.all;

entity DW01_add_oper_rpl is
  generic (wordlength : INTEGER := 8);

  port (in1, in2 : in STD_LOGIC_VECTOR(wordlength-1 downto 0);
        sum      : out STD_LOGIC_VECTOR(wordlength-1 downto 0) );
end DW01_add_oper_rpl;

architecture oper of DW01_add_oper_rpl is
begin
  process (in1, in2)
    constant r0 : resource := 0;
```

```
      attribute map_to_module of r0: constant is "DW01_add";
      attribute implementation of r0: constant is "rpl";
      attribute ops of r0: constant is "a1";
      variable in1_signed, in2_signed : SIGNED(wordlength-1 downto 0);
      variable sum_signed : SIGNED(wordlength-1 downto 0);
    begin
      in1_signed := SIGNED(in1);
      in2_signed := SIGNED(in2);
      sum_signed := in1_signed + in2_signed; -- pragma label a1
      sum <= STD_LOGIC_VECTOR(sum_signed);
    end process;
  end oper;
```

The addition operation is labeled `a1`. The `a1` operation is bound to the resource `r0` with the `ops` attribute. This resource is declared in the beginning of the process and is defined by the `map_to_module` and `implementation` attributes as a `DW01_add` synthetic module with an `rpl` implementation.

Note The manual specification of the "rpl" implementation is only available in DC-Expert. DC-Ultra uses C-based generators exclusively performing automatic selection.

3.1.1.1.2 General Procedure in VHDL

These are the steps you must take to infer a specific synthetic module and implementation from a particular occurrence of a VHDL operator:

1. Begin by creating a process or a function in the architecture body of your VHDL description. A resource can be declared in only a process or a function.
2. Declare an identifier for your resource. Use the syntax

   ```
   constant resource_name : resource := 0;
   ```

   `resource_name` becomes the netlist cell name (unless the resource is merged with another resource).
3. Use the `map_to_module` attribute to indicate the module selection of your choice. The syntax for this attribute is

   ```
   attribute map_to_module of resource_name: constant is "module_name";
   ```

   `resource_name` is the identifier for your resource and `module_name` is the name of the module you want bound to it.
4. To set a specific implementation for your module, use the `implementation` attribute. The syntax of this attribute is

   ```
   attribute implementation of resource_name: constant is "impl_name";
   ```

   `resource_name` is the identifier for your resource and `impl_name` is the name of the implementation you want bound to it.

Note You can skip step 4 if you want to specify only the synthetic module and leave the choice of implementation to the tools.

5. Put a label on the operation:

```
Z <= A + B;   -- pragma label label_name
```

6. Use the `ops` attribute to bind the labeled operation to the specific resource. The syntax of this attribute is

```
attribute ops of resource_name: constant is "label_name";
```

`resource_name` is the identifier for your resource and `label_name` identifies the operation for which you want to use the resource.

The `ops` attribute of a given resource can have more than one label associated with it. The labelled operations will then share the resource.

Note

The attributes `map_to_module, implementation`, and `ops` and the subtype `resource` are declared in the `attributes` package of the `synopsys` library. If you include the `library` and `use` statements, you do not need to declare these attributes and subtypes in your design. For example:

```
library synopsys;
use synopsys.attributes.all;
...
```

3.1.1.1.3 Example in Verilog

Suppose you want to use the Verilog addition operator to imply the ripple-carry implementation of a synthetic adder module in your design. The Verilog description in the following example shows how this is done.

The Verilog addition operator (+) is manually bound to a specific resource (adder module `DW01_add`) and implementation (ripple-carry implementation `rpl`). The lines of code shown in bold define the operator's binding to the specific module and implementation.

```
module DW01_add_oper_rpl(in1,in2,sum);
  parameter wordlength = 8;

  input [wordlength-1:0] in1,in2;
  output [wordlength-1:0] sum;
  reg [wordlength-1:0] sum;

  always @(in1 or in2) begin :b1
  /* synopsys resource r0:
     map_to_module = "DW01_add",
     implementation = "rpl",
     ops = "a1"; */

     sum <= in1 + in2; //synopsys label a1
  end

endmodule
```

The addition operation is labeled `a1`. The `a1` operation is bound to the resource `r0` with the `ops` directive. This resource is declared in the beginning of the block and is defined by the `map_to_module` and `implementation` directives as a `DW01_add` synthetic module with an `rpl` implementation.

Note The manual specification of the "rpl" implementation is only available in DC-Expert. DC-Ultra uses C-based generators exclusively performing automatic selection.

3.1.1.1.4 General Procedure in Verilog

These are the steps you must take to infer a specific synthetic module and implementation from a particular occurrence of a Verilog operator:

1. Begin by creating a block or a function in your Verilog description. A resource can be declared only in an `always` block or a function.
2. Declare an identifier for your resource. Use the syntax:

   ```
   /* synopsys resource resource_name */
   ```

   `resource_name` becomes the netlist cell name (unless the resource is merged with another resource).
3. Place a label on the operation.

   ```
   Z = A + B;   // synopsys label label_name
   ```

4. Use the `map_to_module` directive to indicate the module selection of your choice. This directive must follow the resource declaration.

   ```
   /* synopsys resource resource_name:
      map_to_module = "module_name"; */
   ```

   `resource_name` is the identifier for your resource and `module_name` is the name of the module you want bound to it.
5. To select a specific implementation for your design, use the `implementation` attribute. This attribute must follow the `map_to_module` directive.

   ```
   /* synopsys resource resource_name:
      map_to_module = "module_name",
      implementation = "impl_name";*/
   ```

   `resource_name` is the identifier for your resource and `impl_name` is the name of the implementation you want bound to it.
6. Use the `ops` directive to bind the labeled operation to the specific module and implementation.

   ```
   /* synopsys resource resource_name:
      map_to_module = "module_name",
      implementation = "impl_name",
      ops = "label_name"; */
   ```

   `resource_name` is the identifier for your resource and `label_name` identifies the operation.

3.1.1.2 Manual Selection with Component Instantiation

To force Design Compiler to select a specific implementation for a synthetic module that is instantiated in your design, you add the appropriate code to your HDL descriptions. The syntax differs for VHDL and Verilog, and are described separately.

3.1.1.2.1 Example in VHDL

Suppose you want to instantiate the carry-lookahead implementation of a synthetic adder module in your design. The VHDL description in the following example shows how this is done.

The line of code shown in bold binds the carry-lookahead implementation to the label `U1` (an instance of the adder module `DW01_add`).

```
library IEEE, DWARE, DWARE;
```

```
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_foundation_comp.all;

entity DW01_add_inst_cla is
  generic(wordlength: integer := 8);
  port(in1, in2: in STD_LOGIC_VECTOR(wordlength-1 downto 0);
    ci: in STD_LOGIC;
    sum: out STD_LOGIC_VECTOR(wordlength-1 downto 0);
    cout: out STD_LOGIC);
end DW01_add_inst_cla;

architecture inst of DW01_add_inst_cla is
  attribute implementation: STRING;
  attribute implementation of U1 : label is "cla";

begin
  U1: DW01_add
    generic map(width => wordlength)
    port map(A => in1, B => in2, CI => ci, SUM => sum, CO => cout);

end inst;
```

3.1.1.2.2 General Procedure in VHDL

Declare the `implementation` attribute in your architecture. The syntax of this attribute is

```
attribute implementation of instance_name: label is "impl_name";
```

`instance_name` is the name of the instance in your component instantiation statement, and `impl_name` is the name of the implementation you want bound to it.

3.1.1.2.3 Example in Verilog

Suppose you want to instantiate the carry-lookahead implementation of a synthetic adder module in your design. The Verilog description in the following example accomplishes this.

The lines of code shown in bold bind the carry-lookahead implementation to the instance U1 of the synthetic adder module `DW01_add`.

```
module DW01_add_inst_cla(in1, in2, ci, sum, cout);
  parameter wordlength = 8;
  input [wordlength-1:0] in1, in2;
  input ci;
  output [wordlength-1:0] sum;
  output cout;

    // synopsys dc_script_begin
    // set_implementation cla U1
    // synopsys dc_script_end
    // instantiate DW01_add
  DW01_add #(wordlength)
    U1(.A(in1), .B(in2), .CI(ci), .SUM(sum), .CO(cout));
endmodule
```

3.1.1.2.4 General Procedure in Verilog

Use the `set_implementation` command to select a specific implementation for your synthetic module. The syntax of this command is

```
set_implementation implementation_name cell_list
```

`implementation_name` is the name of the implementation to use and `cell_list` is a list of the synthetic modules you want implemented in this manner.

You must insert this command as a directive in a `dc_script`.

3.1.1.3 Specifying an Implementation in dc_shell-t

You can also read your design into `dc_shell-t`, then use the `set_implementation` command to specify an implementation. For example, to select the `rpl` implementation for an instance `U1` of the `DW01_add` module, enter the following command:

```
dc_shell-t> set_implementation rpl U1
```

> **Note** `set_implementation` does not work on synthetic operators; it works only on synthetic modules.

3.1.2 Replacing Unmapped Synthetic Operators

You can replace all the synthetic operators in your design by using the `replace_synthetic` command. `replace_synthetic` performs simple area-based implementation selection and resource sharing on your design. All synthetic operators are mapped to generic logic representations of selected modules. The syntax of the command is

```
replace_synthetic [-ungroup]
```

The `-ungroup` option ungroups all implementations into their containing designs.

> **Note** Several high-level optimizations during compilation (including timing-driven resource sharing) depend on synthetic operators. Using `replace_synthetic` before compile disables these optimizations.

3.1.3 Disabling Selected Synthetic Modules and Implementations

You can disable individual synthetic modules and implementations with the `dont_use` command. For example, to disable the module `DW01_addsub` in `standard.sldb`, use the command:

```
set_dont_use standard.sldb/DW01_addsub
```

To disable a particular implementation of a module (`rpl,` in the example below), use an extra field:

```
set_dont_use standard.sldb/DW01_addsub/rpl
```

The `report_synlib` command lists the modules and implementations in a library, and shows which ones have a `dont_use` attribute.

3.1.4 Prioritizing Implementations

You can use *priorities* to express your preferences among the available implementations of a given module. A priority is an integer between 0 and 10. An implementation with no priority assigned to it has priority of 5 by default.

For a given module, only the highest-priority implementation (or implementations, in case of a tie) is considered for implementation selection.

Note
Priorities distinguish only among implementations of the same module. The priorities of implementations of different modules are unrelated and hence cannot be used to establish preferences.

Implementation priorities can be coded (as functions of module parameters) into the library definition of a part. The `dc_shell-t` command `set_impl_priority` lets you override the priorities specified in a synthetic library without modifying the library itself.

The syntax of `set_impl_priority` is:

```
set_impl_priority [-priority priority] [-set_id id] implementation_list
```

The arguments and command-line options are:

`-priority` *`priority`*

Specifies the implementation priority. `priority` is a quoted integer between 0 and 10 (inclusive).

`-set_id` *`id`*

Non-negative integer that puts the implementation into a given set. Priority comparisons are made only between implementations in the same set. The default value is 0.

`implementation_`*`list`*

Names the implementation(s) whose priority you want to set.

For example, the following command specifies the priority for `my_impl` is "3" and that it belongs to set `2`.

```
dc_shell-t> set_impl_priority my_lib/my_mod/my_impl "3" -set_id 2
```

Note You cannot use the `set_impl_priority` command in shell scripts embedded in HDL descriptions.

You can disable priority testing by setting the `dc_shell-t` variable `hlo_ignore_priorities` to "`true`".

When you begin to use a new library, or when you have questions about how implementation priorities are affecting your synthesis results, follow this procedure:

1. Issue the `report_synlib` command to generate a report that shows implementation priorities.
2. Use the `set_impl_priority` command, if necessary, to change priorities.

3.2 Controlling Incremental Implementation Selection

Implementation selection of synthetic modules occurs on every compile. However, Resource-sharing decisions remain fixed after the first compile.

Note Implementation selection will NOT be performed if you choose "`map_effort low`", unless you control the process manually by using the `set_implementation` command.

Implementation selection is controlled by the `dc_shell-t` variable, `compile_implementation_selection`, which defaults to "`true`". When the variable is set to "`false`", implementation selection is disabled and implementation decisions will not be changed from the current choice, unless you select an implementation manually by using the `set_implementation` command.

3.2.1 Effect on set_implementation

The `set_implementation` command interacts with incremental implementation selection in two ways:

- The values you set remain effective for all compiles, not just the first.

  ```
  set_implementation impl_name cell_list
  ```

- The command syntax stays the same, but "." is now a valid `impl_name`:

  If you specify "." as the `impl_name` for a synthetic module that was mapped during a previous compile, the implementation for that module will not change in subsequent compiles. For example, the following command keeps the implementations of cells `U1` and `U2` fixed during subsequent compiles:

  ```
  set_implementation . { U1 U2 }
  ```

  To allow the implementation to be changed again, use the `remove_attribute` command:

  ```
  remove_attribute cell_list "implementation"
  ```

  For the example given above, the appropriate command is

  ```
  remove_attribute { U1 U2 } "implementation"
  ```

3.2.2 Effect on report_resources

The `report_resources` command generates two reports. The first report shows the results of resource sharing, but not implementation selection. The second report shows the implementations that are still in your design (ungrouped implementations are not listed). It tells you the current implementation for each instance, as well as any implementation selected by `set_implementation`. The resource sharing report does not change after the first compile; the implementation report reflects the results of incremental implementation selection.

The following example shows both reports.

```
Resource Sharing Report for design test in file /usr/remote/designs/
report.vhd
===========================================================
|         |          |             | Contained  | Contained  |
|Resource|   Module  | Parameters  | Resources  | Operators  |
===========================================================
| r38     | DW01_add | width=8     |            |add_1 add_2|
| r40     | DW01_add | width=8     |            |U1          |
===========================================================
```

```
Implementation Report
===========================================================
|         |          | Current            | Set                |
| Cell    |  Module  | Implementation     | Implementation     |
===========================================================
| U1      | DW01_add | cla                | cla                |
| r38     | DW01_add | rpl                |                    |
===========================================================
```

Note that the Set Implementation field may differ from the Current Implementation field. This happens when you have issued the set_implementation command, but have not yet compiled. Before the first compile, the report shows only Set Implementation information, since nothing else has been decided.

3.3 Controlling Hierarchy

After the compile command is completed, implementations are, by default, instantiated as a level of hierarchy in your design. For example, each adder is represented by an instance of an adder design.

In some cases, you can ungroup implementations, collapsing their contents into the containing design. Small arithmetic parts and all multiplexers supplied by Synopsys are automatically ungrouped so that they can be optimized with surrounding logic. Additionally, you can force modules to be ungrouped by using the set_ungroup command before you run compile.

3.4 Removing Unconnected Ports

Implementations of DesignWare Building Block IP often have unused ports. For example, a particular implementation may not use all of the features of the IP, or the implementation may not use the entire bit-width of the IP's parameterized input or output buses. An unused output port is an output that is not connected on the outside of the design. An unused input port is an input port that is not connected on the outside or is connected to a constant.

When a design is compiled with boundary optimization enabled, the logic that feeds an unused output port (or reads an unused input port) is optimized out of the design, leaving the port unconnected on the inside of the design. Unconnected ports, in turn, can cause problems for downstream tools (for example, layout tools or simulation tools).

To remove unconnected ports from the design, use the remove_unconnected_ports command. The syntax is:

```
remove_unconnected_ports [-blast_buses] cell_list
```

The remove_unconnected_ports command deletes the unconnected ports from the cells in cell_list. The command also deletes unconnected ports from the cell's reference and subdesigns. Because different cells with the same reference may be connected differently, you must uniquify the design before you execute remove_unconnected_ports.

By default, for bused ports, remove_unconnected_ports only deletes a bus when all members of the bus are unconnected. Otherwise, all members of the bus (connected and unconnected) are left intact.

If you specify the -blast_buses option, remove_unconnected_ports performs the following steps when it finds an unconnected port that is part of a bus:

1. Deletes the bus.
2. Deletes the unconnected ports.

3. Creates individual buses for the remaining connected ports to replace the deleted bus.

The following example removes all unconnected ports in the current design:

```
dc_shell-t> remove_unconnected_ports [find -hierarchy cell "*"]
```

The following example deletes buses that contain unconnected ports in the current design, deletes the unconnected ports, then creates individual buses for the remaining connected ports:

```
dc_shell-t> remove_unconnected_ports -blast_buses [find -hierarchy cell "*"]
```

The `remove_unconnected_ports` command does not affect cells that have the `dont_touch attribute` set.

3.5 Maintaining the Synthetic Library Cache

During implementation selection, Design Compiler compares the timing and area characteristics of different implementations. To perform the comparison, Design Compiler creates an optimized timing model for every implementation it considers.

To avoid repeating work, Design Compiler stores the models it creates in a UNIX directory called the *synthetic library cache*. When Design Compiler considers an implementation whose timing model is already in the cache, it does not have to create that model again.

This section describes the structure of the cache, tells you about the commands and variables available to you for controlling the cache mechanism, and provides tips for using the cache effectively.

3.5.1 Structure of the Cache

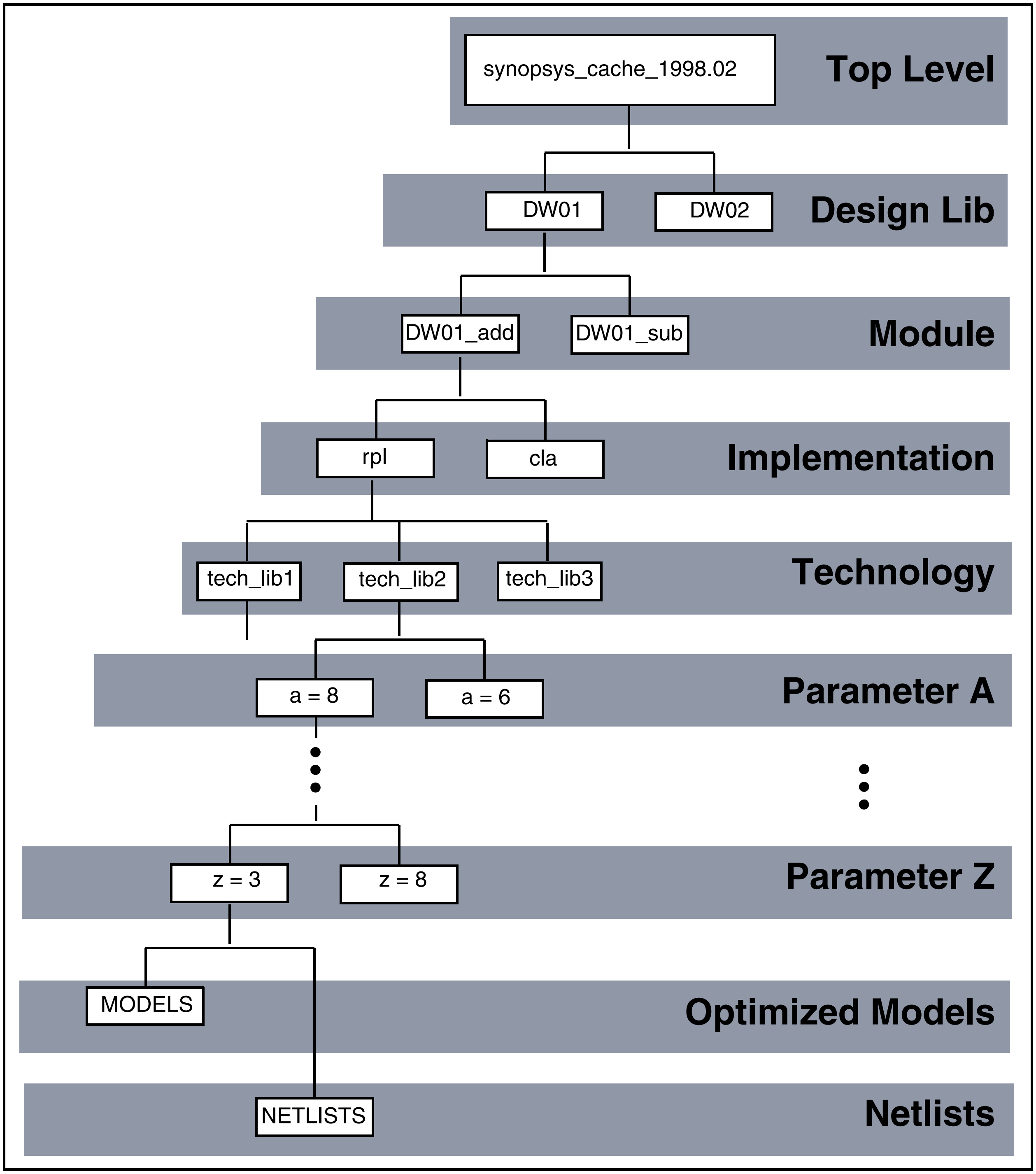
By default, the cache is created under your home directory. (For alternatives, see "Cache Variables" on page 49.) The cache is arranged as a hierarchical UNIX directory structure. `synopsys_cache_vX.X` is the top of the cache directory structure, where `vX.X` is the version number of Design Compiler.

Cache entries are indexed by design library name (DW01, for example), module, implementation, technology library, and parameters (from parameter 1 to parameter n). Figure 3-5 illustrates the cache directory hierarchy.

There are two types of elements stored in the cache: unoptimized netlists and optimized models. The cache directory structure separates netlists from models.

- Hierarchy for netlists – `design library, module, implementation, technology library, parameters, NETLIST, element.db.`
- Hierarchy for models – `design library, module, implementation, technology library, parameters, MODELS, wire load, operating conditions, element.db`

Figure 3-5 Directory Hierarchy of the Synthetic Library Cache



At the bottom of the directory tree structure, the models for the optimized implementations are stored as `.db` files.

3.5.2 Controlling the Cache

Three `dc_shell-t` commands give you control over the synthetic library cache:

- `create_cache` – Creates customized models and puts them in the cache, so you do not have long delays on your first compile.
- `report_cache` – Reports on the contents of the cache.
- `remove_cache` – Removes unwanted files from the cache.

These commands, and several variables related to cache control, are discussed in the following sections.

3.5.2.1 Creating the Cache Prior to Compilation

The `create_cache` command creates a model for each implementation that matches the conditions you specify on the command line. It uses the technology library specified by the `target_library` variable. The models are put in the directory specified by the `cache_write` variable. (See “Cache Variables” on page 49.)

With `create_cache`, you can create customized instances of synthetic parts in the cache without inferring or instantiating them through HDL descriptions. In this way, a design team leader can create a set of customized synthetic parts required for the current project, and place them in a (common) cache directory.

The syntax of `create_cache` is:

```
create_cache [-implementation list]
[-parameters parameter_list]
[-operating_condition string]
[-wire_load simple_list] [-report] -module list
```

The command-line options are:

```
-implementation list
```

Tells `create_cache` which implementations to use for each module. If you do not use this option, `create_cache` creates models for all available implementations. list is a space-separated list of strings. These strings can contain the wildcard character (*).

```
-parameters parameter_list
```

Specifies the parameter values `create_cache` will use when creating cache entries.

`parameter_list` is a space-separated list of quoted parameter specifications; a parameter specification is a comma-separated list of parameter settings. For example, {`"N=8,M=6" "N=3"`} is a valid `parameter_list` consisting of two parameter specifications.

Parameter specifications can include ranges of parameter values. For example, {`"N = [8;17)"`} means that N is greater than or equal to 8 and less than 17. See the `create_cache` man page for full details. The number and the name of the parameters you provide must match those of the given module(s); otherwise, you get an error message. The error message tells you the parameters you have to specify for each module.

```
-operating_condition string
```

Tells `create_cache` what operating conditions to assume. The operating condition you choose must be defined in the technology library specified by the `target_library` variable. If you do not use the `-operating_condition` option, `create_cache` assumes the default operating condition for that library.

```
-wire_load list
```

Tells `create_cache` what wire-load model to use. The wire-load model you choose must be defined in the technology library specified by the `target_library` variable. If you do not use the `-wire_load` option, `create_cache` uses the appropriate default wire-load model specified by that library.

`-report`

Generates and displays a very brief timing report for each created model.

`-module list`

Names the modules you want to put in the cache.

Some example command lines are

```
dc_shell-t> create_cache -mod DW01_add -p [list {width =4} -rep -o WCCOM
dc_shell-t> create_cache -mod *_mult -implementation [list wall csa] -par[list
{A_width =16, B_width =16} {A_width = (4;8], B_width = 6"}]
-oper WCCOM -wir [list "10x10" "90x90"]
```

Note Using wildcards in the `-module` and `-implementation` options can sometimes cause too many parts to be created; this can increase compile time. The parts are created, however, only if all the listed modules have the same set of parameters.

3.6 Reporting Cache Contents

The `report_cache` command reports on the cache directories listed in the `cache_read` and `cache_write` variables. The report lists the cache entries that match the conditions you specify on the command line.

The syntax of `report_cache` is:

```
report_cache [-design_lib list] [-module list]
[-implementation list] [-parameters parameter_list]
[-tech_lib list] [-wire_load list]
[-operating_conditions list] [-directory dir_list]
[-smaller size | -larger size]
[-accessed_since days | -accessed_beyond days]
[-netlist_only | -model_only]
[-sort_largest | -sort_oldest | -sort_cache_key]
```

The command-line options are:

`-design_lib list`

Restricts the report to the design libraries in the list. The default value for `design_lib` is the list of all currently defined design libraries. list is a space-separated list of strings. Strings in a list can contain the wildcard character (`*`).

`-module list`

Restricts the report to the modules in the list.

`-implementation list`

Tells `report_cache` which implementations to report on for each module. If you do not use this option, `report_cache` lists all available implementations.

`-parameters parameter_list`

Limits the report to implementations whose parameter values match the specified ranges.

`parameter_list` is a space-separated list of quoted parameter specifications; a parameter specification is a comma-separated list of parameter settings. For example, {`"N=8,M=6" "N=3"`} is a valid parameter_list consisting of two parameter specifications.

Parameter specifications can include ranges of parameter values. For example, {`"N = [8;17)"`} means that N must be greater than or equal to 8 and less than 17. See the `report_cache` man page for full details.

> **Note** The braces { } are important: `-parameters "n=8, m=6"` is parsed as `-parameters { "n=8" "m=6"}`, not as `-parameters {"n=8,m=6"}`. To match a cache entry with two parameters, `n` and `m`, you need the braces.

The special parameter setting {`"no_parameters"`} matches entries with no parameters. For example, the specification `{"N=8, no_parameters"}` matches entries with no parameters or with `N` equal to 8.

`-tech_lib list`

Restricts the report to the technology libraries in the list. The default value for `tech_lib` is the value of the `target_library` variable.

`-wire_load list`

Restricts the report to the wire loads in the list. The wire-load model you choose must be defined in the technology library specified by the `target_library` variable. If you do not use the `-wire_load` option, `create_cache` assumes the default wire-load model for that library.

`-operating_conditions list`

Restricts the report to the operating conditions in the list.

`-directory dir_list`

Tells `report_cache` to search the directories on your list, instead of those specified in the `read_cache` and `write_cache` variables. `dir_list` is a space-separated list of strings. Wildcards are not allowed.

`-smaller (-larger) size`

Reports only files smaller (larger) than the specified value. `size` is an integer that gives the threshold size in bytes.

`-accessed_since (-accessed_beyond) days`

Reports only files whose last access occurred less than (more than) the specified number of days ago. `days` is a floating-point number.

`-netlist_only (-model_only)`

Reports only netlists (models).

`-sort_largest (-sort_oldest, -sort_cache_key)`

The default output format lists cache entries clustered by directory and design library. The options `-sort_largest, -sort_oldest`, and `-sort_cache_key` cause the report to be sorted by size, last access time, and cache-key value, respectively.

For a cache entry to be reported, it must be accepted by every input option. An entry is accepted by an input option if it matches one of the items in the input option's list. For example, the following command lists the ripple adder and the Wallace-tree multiplier.

```
command report-cache -module {DW01_add DW02_mult} -implementation {wall rpl}
```

Some combinations of the input options do not exist (ripple multipliers and Wallace-tree adders), but because each option acts as an independent filter, no difficulties arise.

A cache entry can meet a `-parameter` specification in two ways:

1. The names and values of the `parameter_list` match the cache entry parameters exactly.
2. All the cache entry parameters are matched in the `parameter_list` but the `parameter_list` contains additional parameters that do not apply to the entry.

Otherwise, the entry does not meet the `-parameter` specification.

The default output format lists cache elements clustered by directory, technology library, wire load, and operating conditions. Example 3-1 shows the default output resulting from the command

```
dc_shell-t> report_cache -larger 5000
```

3.7 Removing Items from the Cache

The syntax of the `remove_cache` command is identical to that of `report_cache`:

```
remove_cache [-design_lib list] [-module list]
[-implementation list] [-parameters parameter_list]
[-tech_lib list] [-wire_load list]
[-operating_conditions list] [-directory dir_list]
[-smaller size | -larger size]
[-accessed_since days | -accessed_beyond days]
[-netlist_only | -model_only]
[-sort_largest | -sort_oldest | -sort_cache_key]
```

`remove_cache` looks through the cache directories in the same way `report_cache` does, removing those entries that match your criteria. It also removes any empty directories it encounters.

Note The `remove_cache` command removes files from both the `cache_read` and the `cache_write` directories.

Example 3-1 Cache Report

```
****************************************
Report : cache
Version: 1998.02
Date : Wed Aug 31 14:17:03 1997
****************************************

===========================================================================
| DESIGN     | MODULE         | IMPLEMENT-     | PARAMETERS | ACCESS |  SIZE |
| LIBRARY    |                | ATION          |            |   days | bytes |
===========================================================================
Cache Directory Root: '/usr/remote/'
Technology Library: 'and_or'
Wire Load: no_wire_load
Operating Conditions: no_operating_conditions
```

```
| DW01        | DW01_ADD_ABC | str |            |  0.0 |  5203 |
| DW01        | DW01_absval  | cla | width=3    |  0.0 | 10064 |
| DW01        | DW01_add     | cla | width=5    |  0.0 | 17376 |
| DW01        | DW01_inc     | cla | width=3    |  0.0 |  9136 |
| DW01        | DW01_inc     | cla | width=6    |  0.0 | 10912 |
| DW02        | DW02_mult    | csa | A_width=3  |  0.0 | 29440 |
|             |              |     | B_width=3  |      |       |

===========================================================================
Cache Directory Root: '/usr/remote/'
Technology Library: 'and_or'
Wire Load: '-default-'
Operating Conditions: '-default-'
| DW01        | DW01_ADD_ABC | str |            |  0.0 |  6162 |
| DW01        | DW01_GP_SUM  | str |            |  0.0 |  5302 |
| DW01        | DW01_MUX     | str |            |  0.0 |  5134 |
| DW01        | DW01_XOR2    | str |            |  0.0 |  5020 |
| DW01        | DW01_absval  | cla | width=3    |  0.0 |  9768 |
| DW01        | DW01_add     | cla | width=5    |  0.0 | 20982 |
| DW01        | DW01_inc     | cla | width=3    |  0.0 | 10469 |
| DW01        | DW01_inc     | cla | width=6    |  0.0 | 13654 |
| DW02        | DW02_mult    | csa | A_width=3  |  0.0 | 31452 |
|             |              |     | B_width=3  |      |       |

===========================================================================
```

Queries can be substantially more complicated, as in the following example:

```
dc_shell-t> report_cache -dir ~ -design_lib DW02 -mod *mult -accessed_since 21
-param [list {n=(3;100), m=(9;100)} {n=(6; 100), m= (6;100)}]
```

3.8 Cache Variables

Cache-related variables and their definitions are provided below.

`cache_read : space separated list`

Defines the list of cache directories that are searched for a part. If the variable is set to an empty list ({}), no models are read from the disk, which effectively turns off caching. By default, `cache_read` is set to the system cache (under `$SYNOPSYS/libraries/syn`) and to your home directory.

`cache_write : string`

Defines the cache directory where optimized parts are written (if not already present). By default, `cache_write` is set to your home directory.

`cache_read_info : Boolean`

Prints a message when a cache element is read, if the value is true. The default value is `false`.

`cache_write_info : Boolean`

Prints a message when a cache element is written out, if the value is true. The message notifies you that a new element was created. The default value is `false`.

`synlib_optimize_non_cache_elements : Boolean`

Determines whether or not to optimize new implementation models. When a required implementation model is not found in the cache, the model is created and, possibly, optimized. When `synlib_optimize_non_cache_elements` is `true`, these new models are optimized. When it is false, the new models are not optimized. The default value is `true`.

Note Using unoptimized models decreases the quality of results of timing-driven resource sharing and implementation selection.

```
cache_file_chmod_octal : string
cache_dir_chmod_octal : string
```

Sets cache file and directory permission mode bits. When cache files are created, their mode bits are set to the value of `cache_file_chmod_octal`. When cache directories are created, their mode bits are set to the value of `cache_dir_chmod_octal`. The value of both variables is a string that can be translated to an octal number.

Because separate variables are used for directories and for files, the sticky bit can be set specifically on a directory. The sticky bit is the permission mode bit that restricts your ability to delete other users' files. If the sticky bit on a directory is set and if you have write permission on the directory, you can write your own files in the directory; however, you cannot delete other users' files in that directory.

Many UNIX systems allow a sticky bit to be set on directories (setting the sticky bit on files has a very different meaning and is not allowed for cache files). The sticky bit is important because caches are often shared among different users. For more information, refer to the UNIX man page on `sticky(8)`.

The default for `cache_file_chmod_octal` is `666`. The default for `cache_dir_chmod_octal` is `777`. If you want to use a sticky bit for cache directories, set `cache_dir_chmod_octal = 1777`.

3.9 Tips for Improving Use of the Cache

To make better use of the synthetic library cache, you can:

- "Tip 1. Share the cache" on page 50
- "Tip 2. Fill the cache with commonly used IP" on page 51
- "Tip 3. Set variables to improve compile speed" on page 51

3.9.1 Tip 1. Share the cache

One way to manage the disk effectively is to share a single cache among several users. Sharing a cache economizes space and compile time by eliminating the duplication of both model storage and model creation.

The main concern when sharing caches is that users can delete other users' cache elements. You can avoid this predicament by setting the sticky bit on the cache directory (refer to the `cache_dir_chmod_octal` variable under "Cache Variables" on page 49).

Design teams using a small number of ASIC libraries, with only one version of each ASIC library and of the Synopsys software, can share a single cache. In such cases, the whole team can have read and write permission for the cache. Each team member should have lines similar to the following in the `~/.synopsys_dc.setup` file:

```
cache_file_chmod_octal = 664
cache_dir_chmod_octal = 775
```

```
cache_read = cache_read + group_area
cache_write = group_area
```

`group_area` is the UNIX file directory where all the parts will be stored.

This setup allows the Synopsys tools, when started up by any member of the team, to cache parts in `group_area`. It also allows members of any other group to read the parts that have been cached.

3.9.2 Tip 2. Fill the cache with commonly used IP

The design team leader can use `create_cache` to set up a collection of customized synthetic parts required for the current project, and place them in a (common) cache directory.

3.9.3 Tip 3. Set variables to improve compile speed

Another aspect of cache management is setting the cache-control variables to achieve fast execution of the `compile` command. If you are exploring many different design alternatives, you can set the `synlib_optimize_non_cache_elements` variable to `false`; new implementation models added to the cache will not be optimized. The advantage is faster compilation; the cost is lower quality of results from timing-driven resource sharing and implementation selection.

Note Even when `synlib_optimize_non_cache_elements` is `false`, Design Compiler can retrieve optimized cache elements when these exist in a directory listed in your `cache_read` variable.

Later in the design process, when you are repeatedly compiling the same design or variations of the same design, you can improve your timing-driven resource sharing and implementation selection results by setting `synlib_optimize_non_cache_elements` back to `true`. Only the first `compile` requires the overhead of optimizing a given cache model.

3.10 Summary of Advanced Features

1. You can manually select modules and implementations by including directives in your HDL design description, or by reading your design into `dc_shell-t` and issuing the `set_implementation` command.
2. You can disable high-level optimization of synthetic operators by using the `replace_synthetic` command.
3. You can disable specific synthetic modules and their implementations with the `dont_use` command.
4. You can establish priorities among the implementations of a given module with the `set_implementation_priority` command.
5. Implementations are instantiated by default as a level of hierarchy. You can ungroup these levels with the `set_ungroup` command.
6. You can delete unconnected ports from selected cells in your design by using the `remove_unconnected_ports` command.
7. You can improve your use of the Synthetic Library cache by:
   - sharing the cache

- filling up the cache with commonly used parts
- Improving compile speed when you are exploring many different design alternatives by setting the `synlib_optimize_non_cache_elements` variable to `false`

4 Using Licensed Implementations

You must have a license key to use a licensed DesignWare Building Block IP. Design Compiler automatically checks out the required licenses when you execute the `compile` command. Thus, most licensing operations are transparent. However, the implementations selected by Design Compiler depend on which licenses are available.

Using designs with licensed implementations involves:

- "Displaying License Requirements of Implementations" on page 53
- "Displaying the License Status of a Design" on page 54
- "Excluding Licensed Implementations" on page 56
- "Checking Out Licenses Manually" on page 57
- "Ungrouping Licensed Implementations" on page 58

4.1 Displaying License Requirements of Implementations

To find out which implementations declared in a synthetic library are licensed, and which keys you need in order to access them, use the `report_synlib` command as shown in the following example.

```
dc_shell-t> report_synlib dw_foundation.sldb
****************************************
Report : library
Library: dw_foundation.sldb
Version: Y-2006.06
Date   : Fri Nov 16 17:50:28 2007
****************************************

Library Type            : Synthetic
Tool Created            : A-2007.12
Date Created            : 11.16.07
Library Version         : A-2007.12:A-2007.12-DWBB_0712

Synthetic Modules: ....
```

```
Module Implementations:

     Attributes/Parameters:
       v - verify_only
       V - verification implementation
       u - dont_use
       r - regular_licenses
       l - limited_licenses
       d - design_library
       s - priority_set_id
       p - priority
       leg - legal


Module           Implementations      Attributes/Parameters
----------------------------------------------------------------------

DW01_ash         astr                 r = DesignWare
                                      d = DW01
                                      leg = "(SH_width>=1) && (A_width>=2)"
...
```

According to the above example, the synthetic library dw_foundation.sldb includes the synthetic module DW01_absval, with `astr` as one of the implementations.

The `astr` implementation is the synthesizable implementation that is used to generate hardware. The `astr` implementation requires a DesignWare license.

If the DesignWare license is available, you can include the `astr` implementation in your design. You can compile a design that contains the implementation and write the resulting netlist to any Design Compiler output format.

4.2 Displaying the License Status of a Design

You may want to see what licenses are required by your design. You can display this kind of license information in several ways.

4.2.1 Displaying License Information on Designs in the Hierarchy

To find out more information on the current license status of all designs in the hierarchy, use the `report_hierarchy` command. The following example uses the `report_hierarchy` command to display the license status of the current design.

```
dc_shell-t> report_hierarchy

****************************************
Report : hierarchy
Design : top
Version: Y-2006.06
Date   : Fri May 12 20:07:58 2006
****************************************

Attributes:
    r - licensed design

top
     bottom                  r
     ...
1
```

According to the attribute information, the design `top` requires no licenses but the design bottom is licensed.

Note Designs that contain unmapped IP are sometimes reported as limited designs, even though you may have a regular license for the IP in question. The purpose of this limitation is to keep the proprietary internal structure of some IP from view. By compiling such a design, however, you turn it into a regular design.

4.2.2 Displaying License Information on a Specific Design

Although the `report_hierarchy` command displays the current status of the designs, the command does not show you which licenses are required for a design and which licenses give you full access to a design. To display this information, use the `report_design` command. In the following example, the current design is changed from `top` to `bottom`, and the license status of bottom is displayed with the `report_design` command.

```
dc_shell-xg-t> current_design bottom
Current design is 'bottom'.
{bottom}

dc_shell-xg-t> report_design

****************************************
Report : design
Design : bottom
Version: Y-2006.06
Date   : Fri May 12 20:17:42 2006
****************************************

...
Required Licenses:

    DesignWare

...
1
```

The regular DesignWare license gives you full access to the design bottom.

4.3 Excluding Licensed Implementations

During compilation, Design Compiler normally checks out the available licenses needed to build your design. Design Compiler does not consider implementations that require licenses you do not have.

In some instances, you may need to exclude specific licenses from being checked out by Design Compiler. In these cases, you can use the variables `synlib_disable_limited_licenses` and `synlib_dont_get_license` to exclude unwanted licenses.

4.3.1 synlib_disable_limited_licenses Variable

When an implementation with a limited license is used to build a design, the design cannot be written out. Because of this limitation, you may not want to use limited licenses. By default, the `synlib_disable_limited_licenses` variable is set to `TRUE`, which restricts Design Compiler from checking out any limited licenses.

4.3.2 synlib_dont_get_license Variable

If you do not want to use a particular license key (for instance, because others have a more critical need for a key in short supply), you can use the variable `synlib_dont_get_license`. Implementations requiring the keys listed in this variable are not used.

Suppose, for example, that the following implementations of an adder module have been installed at your site:

Implementation	License Required
impl0	None
impl1	VERY FAST ADD
impl2	FAST ADD or VERY FAST ADD

If you do not want to use the `VERY_FAST_ADD` license, you can set the `synlib_dont_get_license` variable to

```
synlib_dont_get_license = { VERY_FAST_ADD }
```

If you then use the adder module in a design, Design Compiler will access all the implementations it can. `impl0` does not require any licenses to be checked out, so Design Compiler can freely access the implementation. `impl1` requires the `VERY_FAST_ADD` license, so Design Compiler—given the current setting of `synlib_dont_get_license`—cannot access it. Design Compiler accesses `impl2` by checking out the `FAST_ADD` license.

4.4 Wait for Design License

During a compile, Design Compiler will, by default, exit from the process if there are no `DesignWare` licenses available. That is, if your site has a valid `DesignWare` license key, but the license is not available (the key is checked out by another user), DC will issue an error message and exit from the command.

4.4.1 synlib_wait_for_design_license Variable

Some designers prefer to have DC wait until the required key is available, then resume the process. This is done by setting the `synlib_wait_for_design_license` variable.

The default value of the `synlib_wait_for_design_license` variable is an empty list. When the variable is set to an empty list, DC will behave as described above. If it is set to a list of design license names, then the DC's compile, read, elaborate commands will wait for one of the necessary design licenses to become available rather than aborting the process.

To cause Design Compiler to wait for a `DesignWare` license to become available before proceeding with the compile, set the `synlib_wait_for_design_license` variable to:

```
dc_shell-t> set synlib_wait_for_design_license [list "DesignWare"]
```

4.4.2 Excluding Unavailable Licenses

If implementations are authorized for a site, but licenses are not currently available, the compile command generates an error message and aborts:

```
dc_shell-t> compile
Error: The synthetic library part implementation
       'clf_add' should be available for use during the
       compile command, but the implementation is not
       enabled because all of the following regular
       licenses have been checked out:
       DesignWare
       The design compiler command is being aborted
       because the missing implementation may affect
       compile results. (SYNL-16)
Information: Compile terminated abnormally. (OPT-100)
Current design is 'top'.
0
```

If you want to continue, you can use the `synlib_dont_get_license` variable to exclude unavailable licenses. Note that the quality of your results can suffer because optimal implementations may not be considered.

4.5 Checking Out Licenses Manually

Although Design Compiler automatically checks out the required licenses during compilation, you may want to check out licenses manually before that.

In the report listed in the design `bottom` requires the `DesignWare` license. You may want to reserve this license (someone else may be using it) before you compile the design. To get a license, use the `get_license` command. For example, to get the `DesignWare` license, enter the following statement (the `list_licenses` command verifies that the license has been acquired):

```
dc_shell-t> get_license DesignWare
1
dc_shell-t> list_licenses
          DesignWare
1
```

Suppose a design requires either a limited license or a regular license, and you have acquired the limited license. You may want to check out the regular license instead. Use the `get_license` command to check out a regular license; the design is automatically converted from a limited design to a regular design.

4.6 Ungrouping Licensed Implementations

Ungrouping a licensed design causes the parent design to inherit license information. When this situation occurs, a warning message is displayed, notifying you that license information has changed. In the hierarchy report in the following example, the design `bottom` requires a regular license. The parent design, `middle`, requires none.

```
dc_shell-xg-t> report_hierarchy

****************************************
Report : hierarchy
Design : top
Version: Y-2006.06
Date   : Fri May 12 20:07:58 2006
****************************************

Attributes:
    r - licensed design

top
     bottom                  r
     ...
1

dc_shell-xg-t> ungroup -all

Warning: Design 'top' inherited license information from design 'middle'. (DDB-74)

1
```

After the subdesign bottom is ungrouped into the design `top`, `top` now requires the same licenses that `bottom` required.

Grouping a design similarly causes license information to be passed on from the parent design to the newly grouped design.

4.7 Summary of Use of Licensed Implementations

1. Most licensing operations are transparent.
2. You can display information on licensed implementations by using the `report_synlib` command on a synthetic library.
3. You can display license information on designs in a hierarchy by using the `report_hierarchy` command.
4. You can display license information on specific design by using the `report_design` command.
5. Some designs are read as limited because licensed parts have not yet been mapped. Once you map the parts, the designs are no longer limited.

6. You can prevent Design Compiler from checking out specific licenses by setting the following variables:
   - `synlib_disable_limited_licenses`
   - `synlib_dont_get_license`

   Site authorization also limits the licenses that Design Compiler can check out.
7. You can check out licenses manually by using the `get_license` command.
8. Grouping and ungrouping designs leads to changes in license status. License information is inherited or passed on.

5
Using DWBB IP in FPGA Synthesis: Synplify

5.1 Overview

Synopsys supports FPGA synthesis and prototyping using the Synplify FPGA synthesis tools. The Synplify Premier and Synplify Premier with Design Planner tools can implement DesignWare IP in FPGA designs from two distinct sources:

- Synopsys DWBB IP (DesignWare Foundation Library)
- DesignWare-compatible models originally developed by Synplicity, Inc.

The Synopsys DWBB IP is licensed separately from Synopsys. The DesignWare-compatible library is a standard feature of the Synplify Premier tools. Either library can be used as the source of the DesignWare IP, but components from the two libraries cannot be intermixed.

Hint	This section is only intended as a QuickStart for using DWBB IP in Synplify tools. For more information on FPGA Synthesis using Symplify, refer to the *Synopsys FPGA Synthesis User Guide.*

5.2 Using DesignWare Building Blocks with Synplify Tools

This section describes how to use the DWBB IP with the Synplify Premier tools.

Install the Synopsys DesignWare Foundation Library; a separate license is required.

1. Set the path to the DesignWare Foundation Library by either:
   - specifying the path to the library using the `dc_root installPath` Tcl command
   - selecting the Implementation Options dialog box and using either the Verilog or VHDL tab to enter the path to the library in the Design Compiler Installation Location [$SYNOPSYS=]: field
2. Enable use of the DesignWare Foundation Library by either:
   - checking the Use DesignWare Foundation Library checkbox on either the Verilog or VHDL tab
   - entering either of the following Tcl commands

   ```
   set_option -dw_foundation 1
   set_option -dw_library {dw_foundation}
   ```
3. Set the response to being unable to locate a **DesignWare** license according to your design criteria:

- checking "Stop Synthesis if no DesignWare license found" or entering the following Tcl command stops design synthesis when DWBB IP is encountered and a "DesignWare" license feature is not found:

  ```
  set_option dw_stop_on_nolic 1
  ```

- conversely, not enabling the checkbox or setting the option to 0 (the defaults) allows synthesis to continue by black boxing the building block:

  ```
  set_option dw_stop_on_nolic 0
  ```

Note The DesignWare license feature can use the multiprocessing capability of the Synplify Premier tools when multiple processor cores are available. To set the number of licenses, open the **Options->Configure Compile Point Options** dialog box and set the "Maximum number of parallel synthesis jobs" value to the desired number of licenses. The normal ratio of license use is one DesignWare license for every two synthesis licenses.

The DWBB IP content is dependent on the version of the library you use, thus supported components may vary. For a component list of the most recent Building Block IP version, see

http://www.synopsys.com/dw/buildingblock.php.

5.2.1 Inferring Verilog Functions in DesignWare Foundation Library

Note
Function inference is only supported in Verilog-based designs.

You can infer Verilog functions for a subset of the DWBB IP models. Function inferencing is only available for the DP* functions. All other component are supported as instances. To enable function inference, add the directory containing the DesignWare-compatible functions (dw_functions) to your include path, as described in the following procedure. You can either use the GUI method described in step 1 or the command line method described in step 2.

1. To set up function inferencing from the GUI, do the following:
   a. Open the Implementation Options dialog box in the Project view.
   b. Go to the Verilog tab and type the following path in the Include Path Order field:

      ```
      install_dir/lib/designware/dw_functions
      ```

2. To set up function inferencing from the command line, add the following line to your project file:

   ```
   set_option -include_path "install_dir/lib/designware/dw_functions"
   ```

3. Synthesize the design.

5.3 Using DesignWare-Compatible Models

The Synplify Premier and Synplify Premier with Design Planner tools can replace Synopsys DesignWare foundation library building blocks instantiated in your VHDL or Verilog source code with corresponding DesignWare-compatible models originally developed by Synplicity, Inc. You can enable access to the DesignWare-compatible library using the following Tcl command:

```
set_option -enable_designware 1
```

Although the default for this option is disabled, or '0', Synopsys recommends that if you are using the DWBB IP, you explicitly set this option as follows:

```
set_option -enable_designware 0
```

For a list of the available DesignWare-compatible models, see "Available DesignWare-Compatible Models" in the *Synopsys FPGA Synthesis User Guide*.

5.4 Batch File Example

Example 5-2 Example of project file (red text is DesignWare Foundation Library specific)

```
#project files
set_option -dw_library {dw_foundation}
set_option -dc_root "/<path>"

add_file -verilog "./DW_fp_add_inst.v"

#implementation: "param1"
impl -add param1 -type fpga

#implementation attributes
set_option -vlog_std v2001
set_option -enable_designware 0
set_option -project_relative_includes 1

#implementation parameter settings (specific to DW_fp_add_inst component)
set_option -hdl_param -set sig_width 23;
set_option -hdl_param -set exp_width 8;
set_option -hdl_param -set ieee_compliance 1

#device options
set_option -technology XC4000XL
set_option -part XC4002XL
set_option -package PC84
set_option -speed_grade -09
set_option -part_companion ""

#compilation/mapping options
set_option -use_fsm_explorer 0
set_option -top_module "DW_fp_add_inst"

# sequential_optimization_options
set_option -symbolic_fsm_compiler 1

# Compiler Options
set_option -compiler_compatible 1
set_option -resource_sharing 1

# mapper_options
set_option -frequency auto
set_option -write_verilog 1
set_option -write_vhdl 0
```

```
# Xilinx XC4000
set_option -run_prop_extract 1
set_option -maxfan 100
set_option -disable_io_insertion 0
set_option -forcegsr no
set_option -pipe 1
set_option -fixgatedclocks 3
set_option -fixgeneratedclocks 3
set_option -no_sequential_opt 0

# NFilter
set_option -popfeed 0
set_option -constprop 0
set_option -createhierarchy 0

#VIF options
set_option -write_vif 1

#automatic place and route (vendor) options
set_option -write_apr_constraint 1

#set result format/file last
project -result_file "./param1/DW_fp_add_inst.edf"

#design plan options
set_option -nfilter_user_path ""
impl -active "param1"
```

A
Standard Synthetic Operators

Table A-2 lists the HDL operators that are mapped to synthetic operators in the Synopsys standard synthetic library `standard.sldb`. For information about the synthetic operators—input and output pins, associated modules, and so on—issue the following `dc_shell` command:

```
dc_shell> report_synlib standard.sldb
```

Table A-2 HDL Operators Mapped to Standard Synthetic Operators

HDL Operator	Synthetic Operator(s)
+	ADD_UNS_OP, ADD_UNS_CI_OP, ADD_TC_OP, ADD_TC_CI_OP
-	SUB_UNS_OP, SUB_UNS_CI_OP, SUB_TC_OP, SUB_TC_CI_OP
*	MULT_UNS_OP, MULT_TC_OP
<	LT_UNS_OP, LT_TC_OP
>	GT_UNS_OP, GT_TC_OP
<=	LEQ_UNS_OP, LEQ_TC_OP
>=	GEQ_UNS_OP, GEQ_TC_OP
if, case	SELECT_OP

B
Downloading the Latest Building Block IP

For information on downloading the latest Building Block IP, and to access other release-related information, refer to *DesignWare Building Block IP Release Notes*.

Index