Synopsys®

Synthesis Methodology Guide





UNIX® Environments

Actel Corporation, Sunnyvale, CA 94086

© 1998 Actel Corporation. All rights reserved.

Printed in the United States of America

Part Number: 5579009-1

Release: July 1998

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Actel Corporation.

Actel makes no warranties with respect to this documentation and disclaims any implied warranties of merchantability or fitness for a particular purpose. Information in this document is subject to change without notice. Actel assumes no responsibility for any errors that may appear in this document.

This document contains confidential proprietary information that is not to be disclosed to any unauthorized person without prior written consent from Actel Corporation.

Trademarks Actel and the Actel logotype are registered trademarks of Actel Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems, Inc.

Cadence is a registered trademark of Cadence Design Systems, Inc.

Mentor Graphics is registered trademark of Mentor Graphics, Inc.

Synopsys, Design Compiler, VHDL Compiler, HDL Compiler, and Library Compiler are trademarks or registered trademarks of Synopsys, Inc.

UNIX is a registered trademark of X/Open Company Limited.

Verilog is a registered trademark of Open Verilog International.

Viewlogic is a registered trademark and MOTIVE is a trademark of Viewlogic Systems, Inc.

All other products or brand names mentioned are trademarks or registered trademarks of their respective holders.

Table of Contents

	Introduction	ci
	Document Organization	ci
	Document Assumptions	ii
	Document Conventions	ii
	HDL Keywords and Naming Conventions	ii
	Designer Series Manuals	v
	On-Line Help	ii
1	Setup	1
	Software Requirements	1
	System Setup	1
	User Setup	2
2	Actel-Synopsys Design Flow	7
	Design Flow Illustrated	7
	Design Flow Overview	8
3	Actel-Synopsys Coding Considerations 1	1
	Multiplexer Encoding	1
	Finite State Machine Design	4
	DesignWare Module Coding	1
4	Synthesis Constraints	1
	Operating Conditions	1
	Design Constraints	1
	Design Hierarchy	5
	Internal Tri-State	9
	Inferring Buffers	0
	Reducing the Maximum Fanout Value 6	1
	Register Type Preferences	2
	Avoid Using Certain Cells	2
	Register Balancing.	2
	Removing Attributes	4
		-

Table of Contents

	Using (Q)CLKINT	64
	Wide Decode Cells in 3200DX and 42MX	65
5	Actel-Synopsys Design Considerations.	67
	Compiling Designs with DesignWare Components	67
	Translating Designs from Other Technologies	68
	Translating Timing Constraints into Designer	69
	Assigning Pins in Synopsys	69
	Using ACTmap to Optimize I/O Placement	70
	Bus Array Syntax	70
	Script Mode Place and Route	71
	Control Flow Commands.	71
	Complex Act 3 I/O Mapping	72
	Instantiating ACTgen Macros	80
	Generating an EDIF Netlist.	84
	Generating a Structural HDL Netlist.	85
	Designing for Radiation Environments	86
	Maintaining Technology Independence	86
Α	Synthesis Library Information	87
	Timing Parameters.	87
	Attributes	87
	Max Fanout	91
	ACT 3 Specific Information.	92
	54SX Specific Information	92
	Additional Information.	93
	Synthesis Library Operating Conditions	94
В	DesignWare Library Information	97
	DesignWare Library Description	97
	DesignWare Library Adders	98
	DesignWare Library Subtractors	99
	DesignWare Library Comparators.	100

Table of Contents

	DesignWare Library Counters
	DesignWare Library Incrementer
	DesignWare Library Decrementer
	Improving Compilation Time
	Module Count and Performance
С	Common Problems
	Library Errors
	Inferring DesignWare
	Internal Tri-State
	Multiplexer Inferencing
D	Product Support
	Actel U.S. Toll-Free Line
	Customer Service
	Customer Applications Center
	Guru Automated Technical Support
	Action Facts
	Web Site
	HLD Web Site
	FTP Site
	Electronic Mail
	Worldwide Sales Offices
	Index

List of Figures

DesignWare Libraries Directory Structure
Synthesis Libraries Directory Structure
Actel-Synopsys Design Flow
Multiplexer Diagram
FSM Diagram
Compile-Characterize-Recompile Methodology Diagram 57
Schematic Before Register Balancing
Schematic after Register Balancing
ACT 3 I/O Macros Directory Structure
IOPCLBUF and IOCLKBUF driven sequential cells
Compilation Results
Script Execution Results
"CLK" Pad Driving Sequential I/O Cells and Other Logic 76
Corrected Design After Script Implementation
Sequential I/O Cell to ACT 3 I/O Cell Link
ACTgen Generated 32 x 32 bit Dual Port RAM
ACTgen Generated 32 x 32 bit FIFO
DesignWare Adder Symbol
DesignWare Subtractor Symbol
DesignWare Comparator Symbol
DesignWare Counter Symbol
DesignWare Incrementer Symbol
DesignWare Decrementer Symbol
Adder Module Count
Adder Logic Level
54SX Adder Module Count
54SX Adder Logic Level
Subtractor Module Count
Subtractor Logic Level
54SX Subtractor Module Count
54SX Subtractor Logic Level
Comparator Module Count

List of Figures

Comparator Logic Levels .										.110
Counter Module Count										. 111
Counter Logic Levels										.111
Incrementer Module Count										. 112
Incrementer Logic Levels .										. 112
Decrementer Module Count										. 113
Decrementer Logic Levels .										. 113

List of Tables

FSM Table
Sequential Input Cells Available for Synthesis
Sequential Output Cells Available for Synthesis
ACT 1/40MX "dont_touch" and "dont_use" Macros
ACT 2/1200XL "dont_touch" and "dont_use" Macros
3200DX/42MX "dont_touch" and "dont_use" Macros
ACT 3 "dont_touch" and "dont_use" Macros 90
54SX "dont_touch" and "dont_use" Macros
Macros that Cannot be Connected to HCLKBUF
Default Operating Conditions
Synthesis Library Operating Conditions
Supported Modules
Adder Pin Description
Subtractor Pin Description
Comparator Pin Description
Counter Pin Description
Counter Operation Truth Table
Incrementer Pin Description
Decrementer Pin Description

Introduction

The *Synopsys Synthesis Methodology Guide* contains information about using Synopsys UNIX synthesis tools with the Actel Designer Series FPGA development software to create designs for Actel devices. Refer to the *Designing with Actel* manual for additional information about using the Designer series software and the Synopsys documentation for additional information about using Synopsys software.

Document Organization

The *Synopsys Synthesis Methodology Guide* is divided into the following chapters:

Chapter 1 - Setup contains information and procedures about setting up Synopsys software for use in creating Actel designs.

Chapter 2 - Actel-Synopsys Design Flow illustrates and describes the design flow for creating Actel designs using Synopsys and Designer Series software.

Chapter 3 - Actel-Synopsys Coding Considerations describes Actel-Synopsys specific HDL coding techniques.

Chapter 4 - Synthesis Constraints contains descriptions, examples, and procedures for using design constraints on Actel designs.

Chapter 5 - Actel-Synopsys Design Considerations contains information and procedures to assist you in creating Actel designs with Synopsys and Designer Series software.

Appendix A - Synthesis Library Information describes the features of the Actel synthesis libraries available for use in design synthesis.

Appendix B - DesignWare Library Information describes the features of the Actel DesignWare libraries available for use in design synthesis.

Appendix C - Common Problems describes problems that may occur during synthesis and the solution to the problem.

Appendix D - Product Support provides information about contacting Actel for customer and technical support.

Document Assumptions

The information in this manual is based on the following assumptions:

- 1. You have installed the Designer Series software.
- 2. You have installed the Synopsys software.
- 3. You are familiar with UNIX workstations and operating systems.
- 4. You are familiar with FPGA architecture and FPGA design software.

Document Conventions

The following conventions are used throughout this manual.

Information that is meant to be input by the user is formatted as follows:

keyboard input

The contents of a file is formatted as follows:

file contents

HDL code appear as follows, with HDL keyword in bold:

```
entity actel is
port (
    a: in bit;
    y: out bit);
end actel;
```

Messages that are displayed on the screen appear as follows:

Screen Message

Introduction

The <act_fam> variable represents Actel device family library directories and files. To reference an actual family library directory or file, substitute the actual name of the family when you see this variable. Available families are act1, act2 (for ACT 2 and 1200XL devices), act3, 3200dx, 40mx, 42mx, and 54sx. For example, to include the ACT 3 setup script in your ".synopsys_cd.setup" file, add the following lines:

script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/act3/actsetup.scr

HDL Keywords and Naming Conventions

There are naming conventions you must follow when writing Verilog or VHDL code. Additionally, Verilog and VHDL have reserved words that cannot be used for signal or entity names. This section lists the naming conventions and reserved keywords for each.

VHDL

The following naming conventions apply to VHDL designs:

- VHDL is not case sensitive.
- Two dashes "--" are used to begin comment lines.
- Names can use alphanumeric characters and the underscore "_" character.
- Names must begin with an alphabetic letter.
- You may not use two underscores in a row, or use an underscore as the last character in the name.
- Spaces are not allowed within names.
- Object names must be unique. For example, you cannot have a signal named A and a bus named A(7 downto 0).

Introduction

The following is a list of the VHDL reserved keywords that cannot be used in your design:

abs	downto	library	postponed	subtype		
access	else	linkage	procedure	then		
after	elsif	literal	process	to		
alias	end	loop	pure	transport		
all	entity	map	range	type		
and	exit	mod	record	unaffected		
architecture	file	nand	register	units		
array	for	new	reject	until		
assert	function	next	rem	use		
attribute	generate	nor	report	variable		
begin	generic	not	return	wait		
block	group	null	rol	when		
body	guarded	of	ror	while		
buffer	if	on	select	with		
bus	impure	open	severity	xnor		
case	in	or	shared	xor		
component	inertial	others	signal			
configura- tion	inout	out	sla			
constant	is	package	sra			
disconnect	label	port	srl			

Verilog

The following naming conventions apply to Verilog HDL designs:

- Verilog is case sensitive.
- Two slashes "//" are used to begin single line comments. A slash and asterisk "/*" are used to begin a multiple line comment and an asterisk and slash "*/" are used to end a multiple line comment.
- Names can use alphanumeric characters, the underscore "_" character, and the dollar "\$" character.
- Names must begin with an alphabetic letter or the underscore.
- Spaces are not allowed within names.

The following is a list of the Verilog reserved keywords that cannot be used in your design:

always	endfunction	macromodule	realtime	tran		
and	endmodule	medium	reg	tranif0		
assign endprimitive		module	release	tranif1		
attribute	endspecify	nand	repeat	tri		
begin	endtable	negedge	rnmos	tri0		
buf	endtask	nmos	rpmos	tri1		
bufif0	event	nor	rtran	triand		
bufif1	for	not	rtranif0	trior		
case	force	notif0	rtranif1	trireg		
casex	forever	notif1	scalared	unsigned		
casez	fork	or	signed	vectored		
cmos	function	output	small	wait		
const	highz0	parameter	specify	wand		
deassign	highz1	pmos	specparam	weak0		
default	if	posedge	strength	weak1		
defparam	ifnone	primitive	strong0	while		
disable	initial	pull0	strong1	wire		
edge	inout	pull1	supply0	wor		
else	input	pulldown	supply1	xnor		
end	integer	pullup	table	xor		
endattribute	join	remos	task			
endcase	large	real	time			

Designer Series Manuals

The Designer Series software includes printed and on-line manuals. The on-line manuals are in PDF format on the CD-ROM in the "/doc" directory. These manuals are also installed onto your system when you install the Designer software. To view the on-line manuals, you must have Adobe® Acrobat Reader® installed. Actel provides Reader on the Designer Series CD-ROM.

Introduction

The Designer Series includes the following manuals, which provide additional information on designing Actel FPGAs:

Designing with Actel. This manual describes the design flow and user interface for the Actel Designer Series software, including information about using the ACTgen Macro Builder and ACTmap VHDL Synthesis software.

Actel HDL Coding Style Guide. This guide provides preferred coding styles for the Actel architecture and information about optimizing your HDL code for Actel devices.

ACTmap VHDL Synthesis Methodology Guide. This guide contains information, optimization techniques, and procedures to assist designers in the design of Actel devices using ACTmap VHDL.

Silicon Expert User's Guide. This guide contains information and procedures to assist designers in the use of Actel's Silicon Expert tool.

Cadence Interface Guide. This guide contains information and procedures to assist designers in the design of Actel devices using Cadence CAE software and the Designer Series software.

Mentor Graphics Interface Guide. This guide contains information and procedures to assist designers in the design of Actel devices using Mentor Graphics CAE software and the Designer Series software.

 $MOTIVE_{TM}$ Static Timing Analysis Interface Guide. This guide contains information and procedures to assist designers in the use of the MOTIVE software to perform static timing analysis on Actel designs.

Synopsys Synthesis Methodology Guide. This guide contains preferred HDL coding styles and information and procedures to assist designers in the design of Actel devices using Synopsys CAE software and the Designer Series software.

Viewlogic Powerview Interface Guide. This guide contains information and procedures to assist designers in the design of Actel devices using Powerview CAE software and the Designer Series software.

Viewlogic Workview Office Interface Guide. This guide contains information and procedures to assist designers in the design of Actel devices using Workview Office CAE software and the Designer Series software.

VHDL Vital Simulation Guide. This guide contains information and procedures to assist designers in simulating Actel designs using a Vital compliant VHDL simulator.

Verilog ® *Simulation Guid*e. This guide contains information and procedures to assist designers in simulating Actel designs using a Verilog simulator.

Activator and APS Programming System Installation and User's Guide. This guide contains information about how to program and debug Actel devices, including information about using the Silicon Explorer diagnostic tool for system verification.

Silicon Sculptor User's Guide. This guide contains information about how to program Actel devices using the Silicon Sculptor software and device programmer.

Silicon Explorer Quick Start. This guide contains information about connecting the Silicon Explorer diagnostic tool and using it to perform system verification.

Designer Series Development System Conversion Guide UNIX Environments. This guide describes how to convert designs created in Designer Series versions 3.0 and 3.1 for UNIX to be compatible with later versions of Designer Series.

Designer Series Development System Conversion Guide Windows® *Environments.* This guide describes how to convert designs created in Designer Series versions 3.0 and 3.1 for Windows to be compatible with later versions of Designer Series.

FPGA Data Book and Design Guide. This guide contains detailed specifications on Actel device families. Information such as propagation delays, device package pinout, derating factors, and power calculations are found in this guide.

Macro Library Guide. This guide provides descriptions of Actel library elements for Actel device families. Symbols, truth tables, and module count are included for all macros.

Introduction

On-Line Help

The Designer Series software comes with on-line help. On-line help specific to each software tool is available in Designer, ACTgen, ACTmap, Silicon Expert, and APSW.

Setup

This chapter contains information about setting up UNIX Synopsys tools to create Actel designs. This includes setting environment variables and information about setting up a system to access the Actel macro and synthesis libraries. Refer to the Synopsys documentation for additional information about setting up Synopsys tools.

Software Requirements

The information in this guide applies to the Actel Designer Series software release R2-1998 or later and Synopsys DC Compiler and FPGA Compiler. For specific information about which versions are supported with this release, go to the Guru automated technical support system on the Actel Web site (http://www.actel.com/guru) and type the following in the Keyword box:

third party

System Setup

After installing Synopsys, make sure the proper environment variables are set in your UNIX shell script. The following are C shell variables. If you are using another shell, adjust the syntax accordingly.

```
setenv SYNOPSYS <synopsys_install_directory>
source $SYNOPSYS/admin/install/sim/bin/environ.csh
setenv ALSDIR <actel_install_directory>
setenv ACT_SYNOPDIR $ALSDIR/lib/synop
set path=($ALSDIR/bin $path)
set path=($SYNOPSYS/<operating_system>/syn/bin $path)
```

Replace the <operating_system> variable in the "set path" line with "sparc" if you use SunOS, "sparcOS5" if you use Solaris, or "hp700" if you use HP-UX.

If you use SunOS or Solaris, the following variable must also be set:

setenv LD_LIBRARY_PATH \$ALSDIR/lib

If you use HP-UX, the following variable must also be set:

setenv SHLIB_PATH \$ALSDIR/lib

Refer to the *Designing with Actel* manual and the Synopsys documentation for additional information about setting environment variables.

User Setup

If you use Actel macros or synthesis libraries when creating designs in Synopsys, you must setup your system to access them. This section describes how to access Actel DesignWare and synthesis libraries.

Reanalyzing DesignWare Libraries

Before creating a design in Synopsys, you must reanalyze the encrypted DesignWare and simulation libraries to achieve compatibility with your version of Synopsys. During reanalysis, the existing DesignWare libraries are overwritten. If you wish to retain the old libraries for use with earlier Synopsys versions, copy the "\$ALSDIR/lib/ synop" tree to a new location before you reanalyze the libraries.

To reanalyze all installed Actel DesignWare libraries:

- 1. Acquire write permission.
- **2. Go to the "scripts" directory.** Type the following command at the prompt:

cd \$ACT_SYNOPDIR/scripts

3. Reanalyze the DesignWare libraries. Type:

update_all_dw

To reanalyze a specific Actel family DesignWare library:

- 1. Acquire write permission.
- **2. Go to the "scripts/<act_fam>" directory.** Type the following command at the prompt:

cd \$ACT_SYNOPDIR/scripts/<act_fam>

3. Reanalyze the DesignWare library. Type the following command at the prompt:

update_dwact

Accessing DesignWare Libraries

To access the DesignWare libraries, set the search path in the ".synopsys_dc.setup" file to include the "actsetup.scr" file of the Actel device family you want to access and include the "DWACT" library and component package in your VHDL description each time you infer or instantiate a synthetic component from the DesignWare libraries. Add the following lines to the ".synopsys_dc.setup" file to access the "actsetup.scr" file:

```
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr.
```

Add the following lines to your VHDL description to include the "DWACT" library and component package each time you infer or instantiate a synthetic component from the DesignWare libraries:

```
library dwact.act;
use dwact.dwact_components.all;
```





Figure 1-1. DesignWare Libraries Directory Structure

The "dwact.sldb" file is a compiled description of the Actel synthetic libraries and "DWACT_COMPONENTS.syn" is a "compile_package" file. An ASCII version of the package file can also be found in the same directory. Refer to "DesignWare Module Coding" on page 31 for information about using the DesignWare library modules.

^{1.} DesignWare libraries are not available for ACT 1 and 40MX devices.

Accessing Synthesis Libraries

To access the synthesis libraries, set the search path in the ".synopsys_dc.setup" file to include the "actsetup.scr" file of the Actel device family you want to access. Add the following lines to the ".synopsys_dc.setup" file to access the "actsetup.scr" file:

```
actlib = get_unix_variable ("ACT_SYNOPDIR")
include actlib + /scripts/<act_fam>/actsetup.scr.
```

Note: To target the 1200XL family, synthesize using the ACT 2 library and use the "XL" operating conditions for timing. Refer to "Synthesis Library Operating Conditions" on page 94 for additional information.

Figure 1-2 shows the directory structure for the Synthesis libraries.



Figure 1-2. Synthesis Libraries Directory Structure

To verify the library version:

Type the following command at the prompt:

\$ACT_SYNOPDIR/scripts/version

Actel-Synopsys Design Flow

This chapter illustrates and describes the design flow for creating Actel designs using Synopsys tools and the Designer Series software.

Design Flow Illustrated

Figure 2-1 illustrates the design flow for creating an Actel device using Synopsys Designer Series software¹.



Figure 2-1. Actel-Synopsys Design Flow

1. Actel-specific utilities/tools are denoted by the grey boxes in Figure 2-1.

Design Flow Overview

The Actel-Synopsys design flow has four main steps; design creation/ verification, design implementation, programming, and system verification. These steps are described in the following sections.

Design Creation/ Verification

During design creation/verification, a design is captured in an RTLlevel (behavioral) HDL source file. After capturing the design, a behavioral simulation of the HDL file can be performed to verify that the HDL code is correct. The code is then synthesized into an Actel gate-level (structural) HDL netlist. After synthesis, a structural simulation of the design can be performed. Finally, an EDIF netlist is generated for use in Designer and an HDL structural netlist is generated for structural and timing simulation.

HDL Design Source Entry

Enter your HDL design source using a text editor or a context-sensitive HDL editor. Your HDL design source can contain RTL-level constructs, as well as instantiations of structural elements, such as ACTgen macros.

Behavioral Simulation

Perform a behavioral simulation of your design before synthesis. Behavioral simulation verifies the functionality of your HDL code. Typically, unit delays are used and a standard HDL test bench can be used to drive simulation. Refer to the *VHDL VITAL Simulation Guide* or *Verilog Simulation Guide* for information about performing behavioral simulation.

Synthesis

After you have created your behavioral HDL design source, you must synthesize it before placing and routing it in Designer. Synthesis transforms the behavioral HDL file into a gate-level netlist and optimizes the design for a target technology. Refer to the Synopsys documentation for information about performing design synthesis.

EDIF Netlist Generation

After you have created, synthesized, and verified your design, you must generate an EDIF netlist for place and route in Designer. This EDIF netlist is also used to generate a structural HDL netlist. Refer to "Generating an EDIF Netlist" on page 84 for information about generating an EDIF netlist.

Structural HDL Netlist Generation

Generate a structural HDL netlist from your EDIF netlist for use in structural and timing simulation by either exporting it from Designer or by using the Actel "edn2vlog" or "edn2vhdl" program. Refer to "Generating a Structural HDL Netlist" on page 85 for information about generating a structural netlist.

Structural Simulation

Perform a structural simulation of your design before placing and routing it. Structural simulation verifies the functionality of your post-synthesis structural HDL netlist. Default unit delays included in the Actel libraries are used for every gate. Refer to the *VHDL VITAL Simulation Guide* or *Verilog Simulation Guide* for information about performing structural simulation.

Design Implementation

During design implementation, a design is placed and routed using Designer. Additionally, static timing analysis can be performed on a design in Designer with the DT Analyze tool. After place and route, post-layout (timing) simulation is performed.

Place and Route

Use Designer to place and route your design. Make sure to specify GENERIC as the EDIF Flavor and Verilog or VHDL as the Naming Style when importing the EDIF netlist into Designer. Refer to the *Designing with Actel* manual for information about using Designer.

Static Timing Analysis

Use the DT Analyze tool in Designer to perform static timing analysis on your design. Refer to the *Designing with Actel* manual for information on using DT Analyze. Chapter 2: Actel-Synopsys Design Flow

Timing Simulation

Perform a timing simulation of your design after placing and routing it. Timing simulation requires timing information exported from Designer, which overrides default unit delays in the Actel libraries. Refer to the *Designing with Actel* manual for information about exporting timing information from Designer. Refer to the *VHDL VITAL Simulation Guide* or *Verilog Simulation Guide* for information about performing structural simulation.

Programming

Program a device with programming software and hardware from Actel or a supported 3rd party programming system. Refer to the *Designing with Actel* manual and the *Activator and APS Programming System Installation and User's Guide* or *Silicon Sculptor User's Guide* for information about programming an Actel device.

System Verification

You can perform system verification on a programmed device using the Actel Silicon Explorer diagnostic tool. Refer to the *Activator and APS Programming System Installation and User's Guide* or *Silicon Explorer Quick Start* for information about using the Silicon Explorer.

Actel-Synopsys Coding Considerations

This chapter describes preferred coding styles for the Actel architecture when using Synopsys synthesis and simulation tools. Examples of HDL code are also given. Included in this chapter is information about multiplexer encoding, finite state machine design, and DesignWare module coding. Refer to the *Actel HDL Coding Style Guide* for additional information about HDL coding for Actel devices.

Multiplexer Encoding

The multiplexer based Actel architecture provides area and speed efficient implementations if multiplexers are inferred using case statements. Multiplexer inference using case statements is more efficient than inference of priority encoders using the if-then-else statements. This coding style provides the synthesis tool a good starting point because case statements imply that all conditions are mutually exclusive.

You can synthesize a multiplexer using a case statement in your HDL code. However, current synthesis technology is based on "reconstruction of logic" where the logic is broken into boolean terms, optimized, and mapped to gates. Often, it is difficult to reconstruct a multiplexer when it is broken down.

Use the Synopsys directive, "full_case parallel_case" to force multiplexer inference. You should also embed an attribute, "infer_mux," in the HDL code to instruct (V)HDL Compiler that certain case statements should be inferred as generic multiplexer cells (MUX_OPs). (V)HDL Compiler maps these MUX_OPs in the design to multiplexers in the technology.

Multiplexer Inferencing Variables

Three "hdlin" attributes determine how and when MUX_OPs are inferred by (V)HDL compiler. These variables can be set in the ".synopsys_dc.setup" file or in a compile script during compilation. In the following examples, the attributes have been set in a compile script. After setting the "hdlin" and compile variables, constraints are set on the design. Typical constraints are delay constraints (such as "max_delay") and fanout constraints (such as "max_fanout"). Failure to set the "max_fanout" constraint can result in a design that has excessive fanout on nets (>24) resulting in errors during compilation in Designer. The three "hdlin" variables are described below with recommended settings when targeting the Actel architecture. Chapter 3: Actel-Synopsys Coding Considerations

hdlin_infer_mux

The "hdlin_infer_mux" variable controls MUX_OP inferencing for the current design and all subsequent designs unless the variable is changed. This variable can be set to three values: "default" (MUX_OPs are inferred for case statements that have the "infer_mux" attribute or directive attached), "none" (no MUX_OPs are inferred) and "all" (MUX_OPs are inferred for every case statement in the design). For best results when targeting the Actel architecture, set this variable to "all."

hdlin_dont_infer_mux_for_resoruce_sharing

The "hdlin_don_infer_mux_for_resoruce_sharing" variable determines whether MUX_OPs are inferred when two or more synthetic operators drive the inputs of the MUX_OP. The default setting of "true" prevents the inference of MUX_OPs when synthetic operators drive the inputs of the MUX_OP. Setting the variable to "false" allows a MUX_OP to be inferred. For best results when targeting the Actel architecture, set this variable to "false."

hdlin_mux_size_limit

The "hdlin_mux_size_limit" variable sets the maximum size of the MUX_OP to be inferred. The default value is 32. This variable should be set to a larger value when multiplexers with more than 32 inputs are inferred. However, this results in longer compilation time.

Compilation Variables

In addition to the "hdlin" variables, two compile variables are used to control multiplexer logic. These are described below with recommended settings for best results when targeting the Actel architecture:

compile_create_mux_op_hierarchy

When the "compile_create_mux_op_hierarchy" variable is set to "true" (default), (V)HDL Compiler creates MUX_OPs with their own level of hierarchy. When "false," (V)HDL Compiler removes this level of hierarchy. For best results when targeting the Actel architecture, set this variable to "true."

Multiplexer Encoding

compile_mux_no_boundary_optimization

When the "compile_mux_no_boundary_optimization" variable is set to "false" (default), (V)HDL Compiler performs boundary optimization on all MUX_OP implementations. When "true," no boundary optimization is performed. Boundary optimization can result in sub-optimal implementations when inferred multiplexers have inputs connected to constant values. For best results when targeting the Actel architecture, set this variable to "true."

Multiplexer Inferencing

MUX_OPs are only inferred for case statements contained in processes (VHDL) or always blocks (Verilog). MUX_OPs are not inferred for ifthen-else statements or case statements contained within if-then-else statements. Consequently, case statements should be always be used when describing multiplexers in Verilog or VHDL. The following examples describe the behavioral syntax for inferring a 4 to 1 multiplexer using a case statement. Figure 3-1 illustrates the multiplexer.



Figure 3-1. Multiplexer Diagram

Verilog

```
module mux4_1 (c, d, e, f, s, mux_out);
input c, d, e, f;
input [1:0] s;
output mux_out;
reg mux_out;
always @(c or d or e or f or s)
begin
    case (s)
    2'b00 : mux_out = c;
    2'b01 : mux_out = d;
    2'b10 : mux_out = e;
    default : mux_out = f;
    endcase
end
endmodule
```

Chapter 3: Actel-Synopsys Coding Considerations

VHDL

```
library ieee;
library synopsys;
use ieee.std_logic_1164.all;
use synopsys.attributes.all;
entity mux4_1 is
port (c, d, e, f : in std_logic;
       S
                  : in std_logic_vector(1 downto 0);
                  : out std_logic );
      mux_out
end mux4_1;
architecture behave of mux4_1 is
begin
 mux1: process (s, c, d, e, f)
   begin
      case s is
       when "00" => mux_out <= c;</pre>
       when "01" => mux_out <= d;</pre>
       when "10" => mux_out <= e;</pre>
       when others => mux_out <= f;
      end case;
end process mux1;
end behave;
```

Synopsys Script

To force (V)HDL Compiler to infer multiplexers, the compile variables and "hdlin" variables must be set properly. This can be done in the ".synopys_dc.setup" file or in a compile script. The following is an example compile script.

```
/* Script file for mux4_1 design */
/* read Actel Synopsys setup script and set Mux inferencing
Switches*/
/* this could also be done in the .synopsys_dc.setup file */
actlib = get_unix_variable ("ACT_SYNOPDIR")
include actlib + /scripts/<act_fam>/actsetup.scr.
/* Set Mux inferencing switches */
hdlin_infer_mux = all;
compile_mux_no_boundary_optimization = true
hdlin_dont_infer_mux_for_resource_sharing = false
/* read design file - use this for Verilog design*/
read -f verilog mux4_1.v
```

```
/* read design file - use this for VHDL design */
read -f VHDL mux4_1.vhd
current_design = mux4_1
/* set max_fanout constraint*/
max_fanout 12
/* set delay constraint on mux4_1 from S[1:0] to output */
set_max_delay 20 -to { "mux_out" } -from { "s<1>" }
set_max_delay 20 -to { "mux_out" } -from { "s<1>" }
set_operating_conditions COMWCSTD
compile -map_effort medium
set_port_is_pad all_inputs()
set_port_is_pad all_outputs()
insert_pads
/* write out EDIF netlist */
write -f edif -h -o mux4_1.edn
```

Wide Multiplexers

When the number of multiplexer inputs is larger than four, effective use of the Synopsys multiplexer inference attributes, variables, and delay constraints are required to produce an optimal implementation in the Actel architecture. When the number of multiplexer inputs is not a power of four (such as 11) the implementation created by (V)HDL Compiler may use multiplexers with all the inputs tied to logic 1 or logic 0. These multiplexers are removed by the Actel Designer software during design compilation to produce a more area efficient implementation. The following examples describe the behavioral syntax for inferring an 11 to 1 multiplexer.

Verilog

```
module mux11 (s, a, b, c, d, e, f, g, h, i, j, k, mux_out);
input [3:0] s;
input a, b, c, d, e, f, g, h, i, j, k;
output mux_out
reg mux_out;
// create an 11:1 mux using a case statement
always @ ({s[3:0]} or a or b or c or d or e or f or g or h or
i or j or k)
begin: mux_blk
case ({s[3:0]}) // synopsys full_case parallel_case
```

Chapter 3: Actel-Synopsys Coding Considerations

```
4'b0000 :
                 mux_out = a;
    4'b0001 :
                 mux_out = b;
    4'b0010 :
                 mux_out = c;
   4'b0011 :
                 mux_out = d;
   4'b0100 :
                 mux_out = e;
   4'b0101 :
                 mux_out = f;
    4'b0110 :
                 mux_out = g;
   4'b0111 :
                 mux_out = h;
   4'b1000 :
                 mux_out = i;
    4'b1001 :
                mux_out = j;
    4'b1010 :
                 mux_out = k;
   default:
                mux_out = 1'b1;
 endcase
end
endmodule
```

VHDL

```
library ieee;
library synopsys;
use ieee.std_logic_1164.all;
use synopsys.attributes.all;
entity mux11 is
port (a, b, c, d, e, f, g, h, i, j, k: in std_logic;
       s : in std_logic_vector(3 downto 0);
       mux_out : out std_logic );
end mux11;
architecture behave of mux11 is
begin
 mux1: process (s, a, b, c, d, e, f, g, h, i, j, k)
    begin
      case s is
        when "0000" => mux_out <= a;</pre>
        when "0001" => mux_out <= b;</pre>
        when "0010" => mux_out <= c;</pre>
        when "0011" => mux_out <= d;</pre>
        when "0100" => mux_out <= e;</pre>
        when "0101" => mux_out <= f;</pre>
        when "0110" => mux out <= q;
        when "0111" => mux_out <= h;</pre>
        when "1000" => mux_out <= i;</pre>
        when "1001" => mux_out <= j;</pre>
        when "1010" => mux_out <= k;</pre>
        when others => mux_out <= `1';</pre>
    end case;
end process mux1;
end behave;
```

Synopsys Script

The following example compile script sets the compile and "hdlin" variables. A "max_delay" constraint has been set from the select lines S[3:0] to "mux_out." A tight constraint was placed on S[3:2] while a looser constraint was placed on S[1:0] to ensure that the low order bits are driven by S[1:0]. This causes (V)HDL compiler to build a 16 to 1 mux with five inputs connected to logic 1.

```
/* Script file for 11:1 multiplexer design */
actlib = get_unix_variable ("ACT_SYNOPDIR")
include actlib + /scripts/<act_fam>/actsetup.scr.
/* Set Mux inferencing switches
                                                         */
hdlin_infer_mux = all
compile mux no boundary optimization = true
hdlin_dont_infer_mux_for_resource_sharing = false
/* read design files - use the following for VHDL*/
read -f verilog mux11.vhd
/* read design files - use the following for Verilog */
read -f verilog mux11.v
current_design = mux11
max fanout = 12
/* force short delay on SEL[3:2] so designer will remove
unneeded mux */
set_max_delay 5 -to { "mux_out" } -from { "S<3>" }
set_max_delay 5 -to { "mux_out" } -from { "S<2>" }
/* set longer delay from SEL[1:0] */
set_max_delay 20 -to { "mux_out" } -from { "S<1>" }
set_max_delay 20 -to { "mux_out" } -from { "S<0>" }
set_operating_conditions COMWCSTD
compile -map_effort medium
report_area > mux11.rpt
set_port_is_pad all_inputs()
set_port_is_pad all_outputs()
insert_pads
write -f edif -h -o mux11.edn
```

Chapter 3: Actel-Synopsys Coding Considerations

Registered Multiplexers

In a datapath application, multiplexers drive registers. These structures can quickly be inferred using the MUX_OP attributes. As in the previous examples, a combination of compile variables and constraints is required to get the optimum implementation.

Verilog

```
module reg_mux11 (rst, clk, s, a, b, c, d, e, f, g, h, i, j, k,
mux_out);
input [3:0] s;
input a, b, c, d, e, f, g, h, i, j, k;
input clk, rst;
output mux_out;
reg mux_out;
out
// synopsys infer_mux "mux_blk"
// create a registered 11:1 mux using a case statement
always @ (posedge clk or negedge rst)
begin: mux_blk
  if (~rst)
   mux_out = 1'b0;
  else
   case ({s[3:0]}) // synopsys full_case parallel_case
     4'b0000 :
                  mux_out = a;
     4'b0001 :
                  mux_out = b;
     4'b0010 :
                  mux_out = c;
     4'b0011 :
                  mux_out = d;
     4'b0100 :
                  mux_out = e;
     4'b0101 :
                  mux_out = f;
     4'b0110 :
                  mux_out = g;
     4'b0111 :
                  mux_out = h;
     4'b1000 :
                  mux_out = i;
     4'b1001 :
                 mux_out = j;
     4'b1010 : mux_out = k;
     default:
                  mux_out = 1'b1;
    endcase
end
endmodule
```
```
VHDL
 library ieee;
 library synopsys; -- for synopsys mux inferencing
 use ieee.std_logic_1164.all;
 use synopsys.attributes.all; -- for synopsys mux inferencing
 entity reg_mux11 is
 port
        (s: in std_logic_vector (3 downto 0); -- mux select
          a, b, c, d, e, f, g, h, i, j, k : in std_logic;
          clk, rst : in std_logic;
         mux_out : out std_logic); -- mux output
 end reg_mux11;
 architecture synth of reg_mux11 is
 begin
 procl: process (rst, clk)
 begin
   if (rst = '0') then
     mux_out <= '0';</pre>
   elsif (clk 'event and clk = '1') then
     case s is
       when "0000"
                       => mux_out <= a;
       when "0001"
                       => mux_out <= b;
       when "0010"
                        => mux_out <= c;
       when "0011"
                        => mux_out <= d;
       when "0100"
                        => mux_out <= e;
       when "0101"
                        => mux_out <= f;
       when "0110"
                        => mux_out <= g;
       when "0111"
                        => mux_out <= h;
       when "1000"
                        => mux_out <= i;
       when "1001"
                       => mux_out <= j;
       when "1010"
                       => mux out <= k;
       when others
                        => mux_out <= '1';
     end case;
   end if;
 end process procl;
 end synth;
```

Synopsys Script

The following example compile script sets the compile and "hdlin" variables. A "max_delay" constraint has been set from the select lines S[3:0] to the MUX_OUT register. A tight constraint was placed on S[3:2] while a looser constraint was placed on S[1:0] to ensure that the low order bits are driven by S[1:0]. This causes (V)HDL compiler to build a 16 to 1 mux with five inputs connected to logic 1. One of the multiplexers is removed during compile phase in Designer.

```
/* Script file for reg_mux11.v design*/
actlib = get unix variable ("ACT SYNOPDIR")
include actlib + /scripts/<act_fam>/actsetup.scr.
/* Set Mux inferencing switches*/
hdlin_infer_mux = all
compile_mux_no_boundary_optimization = true
hdlin_dont_infer_mux_for_resource_sharing = false
/* read design files - use the following for VHDL*/
read -f vhdl reg_mux11.vhd
/* read design files - use the following for Verilog*/
read -f verilog reg_mux11.v
current_design = reg_mux11
max fanout 12
/* force short delay on S[3:2] so designer will remove
unneeded mux*/
set_max_delay 5 -to find(cell, mux_out_reg) -from { "S<3>" }
set_max_delay 5 -to find(cell, mux_out_reg) -from { "S<2>" }
/* set longer delay from S[1:0] */
set_max_delay 20 -to find(cell, mux_out_reg) -from { "S<1>" }
set_max_delay 20 -to find(cell, mux_out_reg) -from { "S<0>" }
/* set clock constraint*/
create_clock -name "clk" -period 25 -waveform { "0" "12.5" }
{ "clk" }
set operating conditions COMWCSTD
compile -map_effort medium
set_port_is_pad all_inputs()
set_port_is_pad all_outputs()
insert_pads
write -f edif -h -o reg_mux11.edn
```

Under Utilized Case

If a case statement is not fully specified, i.e., when the number of data inputs specified for the MUX_OPs is not the same as the number of data inputs required based on the number of selected lines, (V)HDL Compiler may not create an efficient implementation. In the example below, there should be 32 independent cases because the select line is 5-bits wide. However, when the most significant bit of the select s4 is logic 1, the output is always logic 0 and only 16 cases are specified. Proper use of multiplexer inferencing variables and constraints result in an efficient implementation for the Actel architecture. For Example:

Verilog

```
module org_mux_32_1 (data, s, y);
input[4:0] s;
input[31:0] data;
output y;
reg y;
//synopsys infer_mux "mux"
always @(data or s)
begin: mux
  case(s)
    5'h00 : y = data[0];
    5'h01 : y = data[1];
    5'h02 : y = data[2];
    5'h03 : y = data[3];
    5'h04 : y = data[4];
    5'h05 : y = data[5];
    5'h06 : y = data[6];
    5'h07 : y = data[7];
    5'h08 : y = data[8];
    5'h09 : y = data[9];
    5'h0a : y = data[10];
    5'h0b : y = data[11];
    5'h0c : y = data[12];
    5'h0d : y = data[13];
    5'h0e : y = data[14];
    5'h0f : y = data[15];
    default: y = 1'd0;
  endcase
end
endmodule
```

VHDL

```
library ieee;
library synopsys;
use ieee.std_logic_1164.all;
use synopsys.attributes.all;
entity org_mux_32_1 is
port (data: in std_logic_vector(31 downto 0);
       s : in std_logic_vector(4 downto 0);
       y : out std_logic );
end org_mux_32_1;
architecture behave of org_mux_32_1 is
-- include the next line to use mux_op inferencing
attribute infer_mux of mux1: label is "true";
begin
  mux1: process (s, data)
    begin
      case s is
        when "00000" => y <= data(0);</pre>
        when "00001" => y <= data(1);</pre>
        when "00010" => y <= data(2);</pre>
        when "00011" => y <= data(3);</pre>
        when "00100" => y <= data(4);</pre>
        when "00101" => y <= data(5);</pre>
        when "00110" => y <= data(6);</pre>
        when "00111" => y <= data(7);</pre>
        when "01000" => y <= data(8);</pre>
        when "01001" => y <= data(9);</pre>
        when "01010" => y <= data(10);</pre>
        when "01011" => y <= data(11);</pre>
        when "01100" => v <= data(12);</pre>
        when "01101" => y <= data(13);</pre>
        when "01110" => y <= data(14);</pre>
        when "01111" => y <= data(15);</pre>
        when others => y <= '0';
    end case;
  end process;
end behave;
```

Synopsys Script

To force (V)HDL Compiler to infer an area efficient multiplexer, the compile constraints must be set properly. The following example compile script sets the constraints.

```
/* Script file for org_mux_32_1.vhd design */
sh date
actlib = get_unix_variable ("ACT_SYNOPDIR")
include actlib + /scripts/<act_fam>/actsetup.scr.
/* Set Mux inferencing switches */
compile_mux_no_boundary_optimization = true
hdlin_dont_infer_mux_for_resource_sharing = false /* allows
best implementation of mux */
/* read design files - use the following for VHDL */
read -f vhdl org_mux_32_1.vhd
/* read design files - use the following for Verilog */
read -f vhdl org_mux_32_1.v
current_design = org_mux_32_1
/* force shortest delay on s[4]*/
set_max_delay 5 -to { "y "} -from { "s<4>" }
/* force short delay on s[3:2] so designer will remove
unneeded mux
              */
set_max_delay 9 -to { "y" } -from { "s<3>" }
set_max_delay 9 -to { "y" } -from { "s<2>" }
/* set longer delay from s[1:0] */
set_max_delay 20 -to { "y" } -from { "s<1>" }
set_max_delay 20 -to { "y" } -from { "s<0>" }
set_operating_conditions COMWCSTD
compile -map_effort medium
set_port_is_pad all_inputs()
set_port_is_pad all_outputs()
insert_pads
write -f edif -h -o org_mux_32_1.edn
```

Finite State Machine Design

Because of sequential element abundance in the Actel architecture, the one-hot encoding for state machines can generate an area and speed optimized design.

You must code the state machine as a regular compact encoding, and the Finite State Machine (FSM) optimization must be used to extract the states and generate the bit per state methodology. If you code the bit per state technique in the HDL code, Synopsys does not generate an area and performance optimized state machine. For example, consider a simple Mealy FSM illustrated in Figure 3-2 and Table 3-1.



Figure 3-2. FSM Diagram

Table 3-1.	FSM	Table	2
------------	-----	-------	---

Present State	Next State		Output (z)	
	a=0	a=1	a=0	a=1
S0	S0	S1	0	0
S1	S1	S2	1	0
S2	S0	S1	0	1

The state machine is described with two processes. One process defines the next state assignments (state registers) and the other process describes the combinatorial portion of the design that determines the state assignment.

In the following examples, the signal type, "pres_state," defines the current state of the state machine and the signal, "next_state," defines the next state of the state machine, depending on the current state and input. A reset assignment sets the state machine to the state "S0."

Verilog

```
module mealy_ent ( a, clock, reset, z );
  input a, clock, reset;
  output z;
  reg [1:0] pres_state, next_state;
  reg z;
  parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10;
  always @ (posedge clock or negedge reset )
    if (!reset) pres_state = 2'b00;
     else
     pres_state = next_state;
  always @ (pres_state)
   begin
    case (pres_state) //synopsys parallel_case full_case
    S0 : if (a)
       begin
       next_state = S1;
       z = 1'b0;
       end
     else
       begin
       next_state = S0;
       z = 1'b0;
       end
    S1 :if (a)
       begin
       next_state = S2;
       z = 1'b0;
     else
       begin
```

```
next_state = S1
       z = 1'b1;
       end
   S2 : if (a)
       begin
       next_state = S1;
       z = 1'b1;
       end
     else
       begin
       next_state = S0;
       z = 1'b0;
       end
   endcase
 end
endmodule
```

VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity mealy_ent is
 port (
              : in std_logic;
       а
       clock : in std_logic;
       reset : in std_logic;
               : out std_logic);
       z
end mealy_ent;
architecture mealy_arc_d of mealy_ent is
  type state_type is (s0, s1, s2);
  signal pres_state, next_state :state_type;
begin
--next state assignment; synchronizing process
sync: process (reset, clock)
begin
  if ( reset = '0' ) then
      pres_state <= s0;--async reset
 else
      if (clock'event and clock='1')then
        pre_state <= next_state;--assign next state</pre>
     end if;
  end if;
```

```
end process sync;
--process to hold combinatorial logic that determines the
--next_state
comb: process (pres_state, a)
begin
  case pres_state is
    when s0 =>
      if (a='0')then
         z <= '0';
         next_state <= S0;</pre>
      else
         z <= '0';
         next_state <= s1;</pre>
      end if;
    when s1 =>
      if (a='0')then
         z <= '1';
         next_state <= s1;</pre>
      else
         z <= '0';
         next_state <= s2;</pre>
      end if;
    when s2 =>
      if (a='0')then
         z <= '0';
         next_state <= s0;</pre>
      else
         z <= '1';
         next_state <= 1;</pre>
    end if;
  end case;
end process comb;
```

```
end mealy_arc_d;
```

Extracting an FSM from a Sequential Design

After completing the behavioral description of the state machine, you must select the state machine style. Use the following procedure to extract and optimize designs. You can use this procedure when the entire design is the state machine or when the state machines registers are buried within the context of a larger design:

1. **Read design.** Type one of the following commands at the prompt:

```
read -f vhdl state.vhd /* for vhdl */
read -f verilog state.v /* for verilog */
```

2. Set the current design to your state machine. Type the following command at the prompt:

current_design = mealy_ent

3. Set the clock constraints. Type the following commands at the prompt:

create_clock -period 10 -waveform {0 5} clock
dont_touch_network clock
set drive 0 clock

4. Map the design. If the design is not mapped, run compile to map the design to gates. Type the following command at the prompt:

compile -map_effort medium

5. Define variables. Define the variables that identify the state register names and find the state registers in the design. Type the following commands at the prompt:

```
state_reg_name = pres_state
state_regs = find (cell, state_reg_name + "*")
```

6. Group the FSM. If the entire design is a state machine, this step is optional. Grouping the FSM section of a circuit produces a new level of hierarchy containing just the FSM state vector flip-flops and their associated logic. The new design is still in netlist format. If all flip-flops in the design are not part of the state machine, Type the following command at the prompt to allow the compiler to identify the state registers:

set_fsm_state_vector state_regs

7. Group the FSM subset of the design. Type the following command at the prompt:

group -fsm -design_name mealy_ent_fsm

8. Set the current design to the FSM section of the design. Type the following command at the prompt:

current_design = mealy_ent_fsm

9. Extract FSM. Extracting an FSM from a circuit changes the representation from a netlist format to FSM "state_table" format. For specific optimization methods, refer to the *Design Compiler Reference Manual*. Type the following command at the prompt:

extract

The remaining steps only describe the one-hot encoding methods.

10. Specify the state vector flip-flops. Type the following command at the prompt:

set_fsm_state_vector state_regs

The order of the flip-flops must agree with the order of the state vector bits (i.e., each bit in the state vector represented by one flipflop order.

11. (Optional) Set the state encoding. Type the following command at the prompt:

set_fsm_encoding "s0=0", "s1=1", ...

State encoding provides the Design Compiler with the names and values of each state.

12. Extract the FSM. Extracting the FSM converts the circuit into an FSM. Type the following command at the prompt:

extract

13. Minimize the state-transition logic. Type the following command at the prompt:

reduce_fsm

Manual One- Hot State Encoding	Actel does not recommend using the manual one-hot encoding in HDL. This method produces inefficient state machines.
Automatic FSM Encoding Styles	The (V)HDL Compiler can automatically select the most appropriate encoding style by selecting auto as the encoding style. Type the following commands to let (V)HDL Compiler select the encoding style: <pre>set_fsm_encoding { } set_fsm_encoding_style auto</pre>
	This encoding style uses a proprietary algorithm. This algorithm's primary objective is to determine a set of encoding that reduces the complexity of the combinatorial logic while using minimum number of encoding bits. Consequently, this encoding style is targeted for area optimization as smaller area reduces delay. The maximum supported state vector length for automatic encoding is 30 bits.
Multiple Resets in FSM	The FSM compiler may not generate optimal results with multiple resets in the FSM. If you are using multiple resets, add "AND" statements to the reset signals in a separate module so that the state machine has a single reset.
Moore Machines	The Moore state machine includes fewer states than the Mealy state machine because the outputs are derived directly from the present state and input. The Mealy state machine's outputs change during transitions from state to state.
Power On and Reset	For simulation, the state machine initializes into the left most value of the enumeration type. However, for synthesis, the state where the machine powers on is not clear. Because (V)HDL Compiler performs state encoding on the machine's enumeration type, the state machine may power on in a state not defined in HDL. Therefore, to achieve simulation and synthesis consistency, it is important to supply a reset to the state machine.

If you want to perform one-hot encoding, you must supply the state machine with a reset. Remember that only one register must be active. All other registers must be reset or inactive. Make sure that no logic exists on the reset network. You can use the "dont_touch_network" command to ensure that no logic is generated for the reset network.

DesignWare Module Coding

Through inference, Synopsys can synthesize efficient design modules from HDL operators. However, in most cases the designs are not optimal implementations for the Actel architecture. To maximize performance, use the Actel DesignWare libraries.

The Actel DesignWare libraries support synthetic modules such as adders, subtractors, comparators, and counters. These modules are optimized for the Actel architecture.

Note: To target the 1200XL family, synthesize using the ACT 2 library and use the "XL" operating condition for timing Refer to "Synthesis Library Operating Conditions" on page 94 for information.

You can use Actel DesignWare library components in the following ways.

Method A: You can allow Synopsys to infer the best implementation for the HDL operator. Synopsys makes a decision based on the design's area and timing constraints. Actel DesignWare library modules are selected only if there are strict timing constraints and if the modules are in the critical path.

Method B: You can select a specific arithmetic implementation (+, -) or logical operator $(<, \le, >, \ge)$ by explicit statements enclosed within comment characters.

Method C: You can manually instantiate the DesignWare library modules by name. You must already know the parameters for module instantiation. Currently, Actel only offers counters for instantiation.

The following examples show how Method A, Method B, and Method C are applied to select the Actel DesignWare library modules. The examples in this chapter are located in the following directory:

```
$ACT_SYNOPDIR/tutorial/designware
```

Refer to "DesignWare Library Information" on page 97 for additional information.

Adders

There are three types of DesignWare adders: RIPADD (ripple carry adder), MFADD (medium fast adder), and FADD (fast adder). Bit vectors range from 2 to 32.

Method A

The following example shows how to implement a Verilog 16-bit adder using Method A.

```
// verilog/add16.v
module adder (c, a, b);
input[15:0]a, b;
output[15:0]c;
wire[15:0]a, b, c;
assign c=a+b;
```

endmodule

The following example shows how to implement a VHDL 8-bit adder with carry in "cin" and carry out "cout" using Method A:

```
-- vhdl/add_c.vhd
library ieee;
use ieee.std_logic_ll64.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity adder is
  generic (width : integer := 8);
  port (a, b: in unsigned (width-1 downto 0);
      cin: in std_logic;
      cout: out std_logic;
```

```
y: out unsigned (width-1 downto 0));
end adder;
architecture rtl of adder is
begin
  process (a,b,cin)
    variable temp_a,temp_b,temp_y:unsigned(a'length downto 0);
    begin
      temp_a := '0' & a;
      temp_b := '0' & b;
      temp_y := temp_a + temp_b + cin;
      y <= temp_y(a'length-1 downto 0));
      cout <= temp_y(a'length);
end process
end rtl;
```

The following design script file compiles the preceding HDL descriptions. The constraint forces Synopsys to use a Actel DesignWare adder instead of a Synopsys adder:

```
/* adder.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog add16.v
/* use the following line to read VHDL source code */
read -format vhdl add_c.vhd
current_design = adder
set_max_delay 10 -from all_inputs() -to all_outputs()
set_operating_conditions MILBC-3
compile
write -hierarchy -format db -output adder.db
exit
```

Method B

The following example shows how to implement a Verilog 32-bit adder using Method B. The text enclosed in the comment line (/* */) allows Synopsys to choose the adder's FADD implementation. You can select "a1" as the arithmetic operation of the FADD description (DWACT_ADD).

```
module adder (clk, reset, dataa, datab, sum, cout);
output [31:0] sum;
output cout;
input [31:0] dataa, datab;
input clk, reset;
// ********declare output types*********
wire [31:0] sum;
wire cout;
// ********declare internal types********
reg [31:0] suminternal;
assign sum[31:0] = suminternal[30:0];
assign cout = suminternal[31]
always @(posedge clk or negedge reset)
 begin
    if (reset == 0)
      suminternal[30:0] <= 32'b0;</pre>
    else
     begin : lab_1
       /*
                 synopsys resource r0:
                 map_to_module = "DWACT_ADD"
                 implementation = "FADD"
                 ops = "lab_2";
                 /*
      suminternal[31:0] <= dataa[31:0] + datab[31:0];</pre>
      //synopsys label lab_2
    end
  end
endmodule
```

The following example shows how to implement a VHDL 4-bit adder using Method B:

```
-- vhdl/fadd4.vhd
library ieee, dwact, synopsys;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use dwact.dwact_components.all;
entity adder is
   port ( a, b : in unsigned (3 downto 0);
```

DesignWare Module Coding

```
c :out unsigned (3 downto 0)
    );
end adder;
architecture impl1 of adder is
begin
    process(a,b)
    constant r0: resource :=0;
    attribute map_to_module of r0: constant is "dwact_add";
    attribute implementation of r0: constant is "fadd";
    attribute ops of r0: constant is "al";
    begin
    c <= a+b; -- pragma label al
    end process;
end impl1;</pre>
```

The following design script file compiles the preceding HDL descriptions:

```
/* fadd.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog fadd32.v
/* use the following line to read VHDL source code */
read -format vhdl fadd4.vhd
current_design = adder
set_operating_conditions INDTC-2
compile
report_area > adder.area
write -hierarchy -format db -output adder.db
exit
```

Subtractors

There are three types of DesignWare subtractors: RIPSUB (ripple carry subtractor), MFSUB (medium fast subtractor), and FSUB (fast subtractor). Bit vectors range from 2 to 32.

Method A

The following example show how to implement a Verilog 16-bit subtractor using Method A:

```
// verilog/sub16.v
module sub (c, a, b);
input[15:0]a,b;
output[15:0]c;
wire[15:0]a,b,c;
    assign c = a-b;
endmodule
```

The following example shows how to implement a VHDL 10-bit subtractor using Method A.

```
-- vhdl/sub10.vhd
library ieee;
use ieee.std_logic_l164.all;
use ieee.std_logic_arith.all;
entity sub is
    port (in1, in2: unsigned (9 downto 0);
        diff: out std_logic_vector (9 downto 0));
end sub;
architecture impl1 of sub is
begin
    diff <= in1 - in2 ;
end impl1;</pre>
```

The following design script file compiles the preceding HDL descriptions. This script file allows timing constraint on input to output. The constraint forces synthesis of an Actel DesignWare subtractor instead of a Synopsys subtractor.

```
/* subt.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog sub16.v
/* use the following line to read VHDL source code */
read -format vhdl sub10.vhd
current_design = sub
set_max_delay 10 -from all_inputs() -to all_outputs()
set_operating_conditions COMWCSTD
compile
write -hierarchy -format db -output sub.db
exit
```

Method B

The following example shows how to implement a Verilog 16-bit subtractor using Method B. The text enclosed in the comment line (/* */) allows Synopsys to choose the subtractor's FSUB implementation. You can select "a1" as the arithmetic operation of the FSUB description (DWACT_SUB).

```
// verilog/fsub16.v
module sub (c, a, b);
input[15:0]a, b;
output[15:0]c;
wire[15:0]a, b;
reg[15:0]c;
always @ (a or b)
  begin: blk01
    /* synopsys resource r0:
   map_to_module = "DWACT_SUB",
    implementation = "FSUB",
     ops = "al";
    * /
    c = a - b; //synopsys label a1
  end
endmodule
```

The following example shows how to implement a VHDL 4-bit subtractor using Method B.

```
-- vhdl/fsub4.sub
library ieee, dwact, synopsys;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use synopsys.attributes.all;
use dwact.dwact_components.all;
entity sub is
 port ( a, b : in unsigned (3 downto 0);
         c :out unsigned (3 downto 0)
      );
end sub;
architecture impl1 of sub is
begin
 process(a,b)
   constant r0: resource :=0;
    attribute map_to_module of r0: constant is "dwact_sub";
    attribute implementation of r0: constant is "mfsub";
    attribute ops of r0: constant is "a1";
 begin
    c <= a-b; -- pragma label al
  end process;
end impl1;
```

The following design script file compiles the preceding HDL descriptions:

```
/* fsub.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog fsub16.v
/* use the following line to read VHDL source code */
read -format vhdl fsub4.vhd
current_design = sub
set_operating_conditions COMWCSTD
compile
write -hierarchy -format db -output sub.db
exit
```

Comparators

There are four types of DesignWare comparators available for different HDL operations: less than, greater than, less than equal to, and greater than equal to. Refer to "DesignWare Library Comparators" on page 100 for additional information.

Method A

The following example shows how to implement a Verilog 16-bit comparator using Method A:

```
// verilog/comp16.v
module comp (c, a, b);
input [15:0]a, b;
output c;
wire [15:0]a,b;
wire c;
   assign c = (a < b);
endmodule</pre>
```

The following design script file compiles the preceding HDL descriptions. The constraint forces synthesis of an Actel DesignWare comparator instead of a Synopsys comparator.

```
/* compl6.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
read -format verilog compl6.v
current_design = comp
set_max_delay 10 -from all_inputs() -to all_outputs()
set_operating_conditions COMWCSTD
compile
write -hierarchy -format db -output comp.db
exit
```

Method B

The following example shows how to implement a Verilog 16-bit comparator using Method B. The text enclosed in the comment line (/* */) allows Synopsys to choose the comparator's FCOMP implementation. You can select "a1" as the arithmetic operation of the FCOMP description (DWACT_CMPLT).

```
// verilog/fcomp16.v
module comp (c, a, b);
input [15:0]a,b;
output c;
wire [15:0]a, b;
regc;
always @ (a or b)
begin : blk01
  /* synopsys resource r0:
 map_to_module = "DWACT_CMPLT",
  implementation = "FCOMP",
  ops = "al"
  */
  c <= (a < b); //synopsys label al
end
endmodule
```

The following design script file compiles the preceding HDL descriptions:

```
/* fcompl6.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
read -format verilog fcompl6.v
current_design = comp
set_operating_conditions COMWCSTD
compile
write -hierarchy -format db -output comp.db
exit
```

Counters

Counters are only available for instantiation. Refer to "DesignWare Library Counters" on page 101 for names and descriptions of the available DesignWare counters.

Method C

The following example shows how to instantiate a Verilog 8-bit up counter using Method C.

```
// verilog/count8.v
module count (data, q, sload, enable, aclr, clock);
input [7:0]data;
input sload, enable, aclr, clock;
output [7:0]q;
  dwact_up_ctr #(8) u0( data, q, sload, enable, aclr, clock);
endmodule
```

Note: The variable "# (8)" defines the counter's bit width. Refer to "DesignWare Library Counters" on page 101 or the "DWACT_components.vhd" file located in the "\$ACT_SYNOPDIR/syn/<act_fam>/dwact" directory for pin ordering information.

The following example shows how to implement a VHDL 10-bit counter using Method C.

```
-- vhdl/count10.vhd
library ieee, dwact, synopsys;
use ieee.std_logic_1164.all;
use ieee.std logic arith.all;
use synopsys.attributes.all;
use dwact.dwact_components.all;
entity count is
  port ( data :in std_logic_vector (9 downto 0);
       load, cen, reset, clk:in std_logic;
       count :out std_logic_vector (9 downto 0)
      );
end sub;
architecture impl1 of count is
  attribute implementation of u0: label is "tlacnt";
begin
 u0: dwact_dn_ctr
```

```
generic map (width => 10)
port map (
    data => data
    count => count,
    load => load,
    cen => cen,
    reset => reset,
    clk => clk
    );
end impl1;
```

The following design script file compiles the preceding HDL descriptions:

```
/* counter.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog count8.v
/* use the following line to read VHDL source code */
read -format vhdl count10.vhd
current_design = count
dont_touch count
set_operating_conditions COMTC-1
compile
write -hierarchy -format db -output count.db
exit
```

Incrementers There is one type of DesignWare incrementer: FINC (fast incrementer). Bit vectors range from 2 to 32.

Method A

The following example shows how to implement a Verilog 16-bit incrementer using Method A.

```
// verilog/inc16.v
module incrementer (c, a) ;
input [15:0]a;
output [15:0]c;
wire [15:0]a, c;
    assign c=a+1;
```

endmodule

The following example shows how to implement a VHDL 8-bit incrementer using Method A:

The following design script file compiles the preceding HDL descriptions. The constraint forces Synopsys to use an Actel DesignWare incrementer instead of a Synopsys incrementer:

```
/* incrementer.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog incl6.v
/* use the following line to read VHDL source code */
read -format vhdl incrementer8.vhd
current_design = incrementer
set_max_delay 10 -from all_inputs () -to all_outputs ()
set_operating_conditions MILBC-3
compile
write -hierarchy -format db -output incrementer.db
exit
```

Method B

The following example shows how to implement a Verilog 32-bit incrementer using Method B. The text enclosed in the comment line (/ **/) allows Synopsys to choose the incrementer's FINC implementation. You can select "a1" as the arithmetic operation of the FINC description (DWACT_INC).

```
always @ (posedge clk or negedge reset)
   begin
     if (reset == 0)
       suminternal [30:0] <= 32'b0;</pre>
     else
       begin : lab_1
         /* synopsys resource r0:
              map_to_module = "DWACT_INC"
               implementation = "FINC"
               ops = "lab_2";
               /*
       suminternal [31:0] <= dataa[31:0] + 1;</pre>
       //synopsys label lab_2
     end
   end
endmodule
```

The following example shows how to implement a VHDL 4-bit incrementer using Method B:

```
-- vhdl/finc4.vhd
library ieee, dwact, synopsys;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use synopsys.attributes.all;
use dwact.dwact_components.all;
entity incrementer is
 port (a : in unsigned (3 downto 0) ;
       c :out unsigned (3 downto 0)
  );
end incrementer;
architecture impl1 of incrementer is
begin
  process(a)
   constant r0: resource :=0;
   attribute map_to_module of r0: constant is "dwact_inc";
   attribute implementation of r0: constant is "finc";
   attribute ops of r0: constant is "al";
 begin
   c <= a+1; -- pragma label a1
  end process;
end impl1;
```

The following design script file compiles the preceding HDL descriptions:

/* incrementer.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog finc32.v
/* use the following line to read VHDL source code */
read -format vhdl finc4.vhd
current_design = incrementer
set_operating_conditions INDTC-2
compile
report_area > incrementer.area
write -hierarchy -format db -output incrementer.db
exit

```
Decrementers
```

There is one type of DesignWare decrementer: FDEC (fast decrementer). Bit vectors range from 2 to 32.

Method A

The following example shows how to implement a Verilog 16-bit decrementer using Method A.

```
// verilog/decl6.v
module decrementer (c, a) ;
input [15:0]a;
output [15:0]c;
wire [15:0]a, c;
   assign c=a-1;
endmodule
```

The following example shows how to implement a VHDL 8-bit decrementer using Method A:

```
-- vhdl/decrementer8.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity decrementer is
port (a: in unsigned (7 downto 0));
y: out unsigned (7 downto 0));
end decrementer;
architecture impl of decrementer is
begin
y <= a - 1;
end impl;
```

The following design script file compiles the preceding HDL descriptions. The constraint forces Synopsys to use an Actel DesignWare decrementer instead of a Synopsys decrementer:

```
/* decrementer.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog dec16.v
/* use the following line to read VHDL source code */
read -format vhdl decrementer8.vhd
current_design = decrementer
set_max_delay 10 -from all_inputs () -to all_outputs ()
set_operating_conditions MILBC-3
compile
write -hierarchy -format db -output decrementer.db
exit
```

Method B

The following example shows how to implement a Verilog 32-bit decrementer using Method B. The text enclosed in the comment line (/ **/) allows Synopsys to choose the decrementer's FDEC implementation. You can select "a1" as the arithmetic operation of the FDEC description (DWACT_DEC).

```
// verilog/fdec32.v
module decrrementer (clk, reset, dataa, sum, cout) ;
output [31:0] sum;
output cout;
input [31:0] dataa;
input clk, reset;
// *********declare output types*********
wire [31:0] sum;
wire cout;
// *********declare internal types*********
reg [31.0] suminternal ;
assign sum [31:0] = suminternal [30:0];
assign cout = suminternal [31]
always @ (posedge clk or negedge reset)
 begin
   if (reset == 0)
      suminternal [30:0] <= 32'b0;</pre>
    else
     begin : lab_1
        /* synopsys resource r0:
             map_to_module = "DWACT_DEC"
             implementation = "FDEC"
             ops = "lab_2";
            /*
      suminternal [31:0] <= dataa[31:0] - 1;</pre>
      //synopsys label lab_2
    end
  end
endmodule
```

The following example shows how to implement a VHDL 4-bit decrementer using Method B:

```
-- vhdl/fdec4.vhd
library ieee, dwact, synopsys;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use synopsys.attributes.all;
use dwact.dwact_components.all;
entity decrementer is
port (a : in unsigned (3 downto 0);
     c : out unsigned (3 downto 0)
);
end decrementer;
architecture impl1 of decrementer is
begin
  process(a)
   constant r0: resource :=0;
    attribute map_to_module of r0: constant is "dwact_dec";
    attribute implementation of r0: constant is "fdec";
    attribute ops of r0: constant is "a1";
  begin
    c <= a-1; -- pragma label al
  end process;
end impl1;
```

The following design script file compiles the preceding HDL descriptions:

```
/* decrementer.scr */
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
/* use the following line to read Verilog source code */
read -format verilog fdec32.v
/* use the following line to read VHDL source code */
read -format vhdl fdec4.vhd
current_design = decrementer
set_operating_conditions INDTC-2
compile
report_area > decrementer.area
write -hierarchy -format db -output decrementer.db
exit
```

Synthesis Constraints

This chapter contains descriptions and usage examples of design constraints that can be applied to Actel designs to improve performance. This includes information about using constraints to remove attributes, set operating conditions, and meet design goals. Also included are methodologies for using constraints to maintain or flatten design hierarchy.

This chapter discusses using constraints to infer buffers and to reduce the maximum fanout value. Information about using constraints to set the register type for a design, to avoid certain cells, to balance registers, and to use wide decode cells in 3200DX or 42MX is also provided. Finally, this chapter contains information about where to find internal tri-state and (q)clkint usage recommendations. Refer to the *Synopsys Command Reference Manual* for additional information.

Operating Conditions

Actel silicon is characterized for temperature, voltage and speed grade information. The characterization information is included in the Actel libraries so that Synopsys uses proper modeling during compilation. You should always set an operating condition to instruct Synopsys to use the correct timing models. For example, if your device is designed to operate at 3.0V and 85° C, type the following command:

set_operating_condition INDWCSTDV

Refer to "Synthesis Library Operating Conditions" on page 94 for information about default and available operating conditions.

Design Constraints

Design goals, such as area and performance, determine the constraints and compile options of a design. This section lists some design constraints and gives usage examples of the constraint.

Clock Constraint	The clock constraint is specified by the clock in a sequential design and determines the maximum register to register delay in the design. The clock must be specified after reading in the HDL description. The following is an example command for specifying a clock:
	create_clock -name clk -period 20 -waveform {0 10} clk
Delay Constraints	The delay constraint sets the path delay on ports relative to a clock edge. The following are delay constraint compile options.
	set_input_delay
	Input ports have zero input delay unless specified. Path delays can be specified for both input and output modes on bidirectional ports. The following is an example command for setting input delays:
	<pre>set_input_delay -clock clock 5 all_inputs()</pre>
	set_output_delay
	Output ports have zero output delay unless specified. Path delays can be specified for both input and output modes on bidirectional ports. The following is an example command for setting output delays:
	<pre>set_output_delay -clock clock 4 all_outputs()</pre>
No Buffering on Reset Network	The "set_dont_touch_network" constraint assigns a "dont_touch" attribute to all cells and nets on the specified network to prevent objects from being modified or replaced during optimization. The following is an example command for setting a "dont_touch" network:
	<pre>set_dont_touch_network find(port, reset)</pre>
Logic Level Constraints	The optimization style is determined by the compile attributes and options. The following are the logic level compile attributes.

set_flatten

The "set_flatten" constraint attempts to create a 2-level sum of product implementation. It does not remove hierarchy. Existing structure is removed and the constraint can be used when you have timing goals and "dont_care" attributes in the design. The constraint is off by default and has three options: effort, minimize, and phase.

Effort - The effort option has three settings: low, medium, and high.

Minimize - The minimize option has three settings: none, single, and multiple. The single setting works on each output individually, does not share terms, but may produce the fastest implementation. The multiple setting is area efficient since it shares terms between outputs.

Phase - The phase option has two settings: false and true. The true setting evaluates both the "1" and "0" of the Karnaugh map and uses the best solution.

Note: You must only flatten random logic. Do not flatten structured blocks such as adders, subtracters, etc.

The following is an example of the "set_flatten" constraint for performance optimization:

set_flatten true -effort medium -minimize single -phase true

set_structure

The "set_structure" constraint maintains the structure of the design and is on by default. The constraint has two options: timing and boolean. Never set timing and boolean to "true" at the same time.

Timing - The timing option has two settings: true and false. The true setting allows timing-driven structuring.

Boolean - The Boolean option has two settings: true and false. The true setting uses Boolean algebra to reduce the size of the design.

The following is an example of the "set_structure" compile attribute for performance optimization:

```
set_structure true -timing true -boolean false
```

Chapter 4: Synthesis Constraints

The following is an example of the "set_flatten" compile attribute for area optimization:

set_structure true -timing false -boolean true

Note: The attribute may provide better results if applied on a gate-level netlist.

The following are some common compile options:

-only_design_rule

The "-only_design_rule" option instructs Synopsys to only check design rules during compile. The following is an example command for compiling with the option set:

compile -only_design_rule

set_map_effort

The "set_map_effort" option sets the effort level of Synopsys during compile. There are three options: "low" for low effort, "medium" for medium effort, and "high" for high effort. The following command is an example command that would instruct Synopsys to perform a medium (default) effort compile:

compile -map_effort medium

The quality of results for some Actel families (e.g. 54SX) can be improved using a high effort. Using a high effort on a second compile is recommend if you use a two compile design methodology.

boundary_optimization

The "boundary_optimization" option allows Synopsys to optimize a design on the boundaries of the design. The following is an example command of setting the boundary optimization option during compile:

```
compile -boundary_optimization
```
Area Constraint	The area constraint, "set_max_area" is default to 0. For best results, use
	realistic area and timing constraints.

Design Hierarchy

This section discusses methodologies for maintaining and flattening design hierarchy and recommendations for when to use each.

Maintaining the
Design
Hierarchy

Maintaining the hierarchy of a design is always preferable because the schematics are easier to read for debugging. The four hierarchical compile strategies are:

- top-down compile
- bottom-up compile
- compile-characterize-write script-recompile
- time budget-compile

Top-Down Compile Methodology

The top-down approach is an easy, push button approach without inter-module dependencies. The following example script compiles a design using a top-down methodology:

```
actlib = get_unix_variable ("ACT_SYNOPDIR")
include actlib + /scripts/<actel_fam>/actsetup.scr.
read -f vhdl lower1.vhd
read -f vhdl lower2.vhd
read -f vhdl top.vhd
current_design top
set_max_fanout 12 top
set_max_fanout 12 find(design, -hier, "*")
current_design top
uniquify
create_clock CLK25 -period 40 -waveform {0 20}
set_dont_touch_network CLK25
set_operating_condition COMWCSTD
compile -map_effort high -boundary_optimization
```

```
/* Optional - Will flatten design hierarchy */
/* ungroup -all -flatten
compile -map_effort high */
set_port_is_pad top
insert_pads
write -f db -h -o top.db
write -f edif -h -o top.edn
write -f vhdl -h -o top.vhd
report_timing >> report.timing
report_area >> report.area
```

Bottom-Up Compile Methodology

The bottom-up approach gives users more control over the design hierarchy and allows for module-based constraints. The following example script compiles a design using a bottom-up methodology:

```
actlib = get_unix_variable ("ACT_SYNOPDIR")
include actlib + /scripts/<actel_fam>/actsetup.scr.
read -f vhdl lower1.vhd
read -f vhdl lower2.vhd
read -f vhdl top.vhd
current_design top
set_max_fanout 12 top
set_max_fanout 12 find(design, -hier, "*")
current_design top
create_clock CLK25 -period 40 -waveform {0 20}
set_dont_touch_network CLK25
current_design lower1
create_clock CLK25 -period 40 -waveform {0 20}
set_dont_touch_network CLK25
compile -boundary_optimization
current_design lower2
create_clock CLK25 -period 40 -waveform {0 20}
set_dont_touch_network CLK25
compile -boundary_optimization
current_design top
uniquify
set_operating_condition INDWCSTDV
```

```
compile -map_effort high -boundary_optimization
set_port_is_pad top
insert_pads
write -f db -h -o top.db
write -f edif -h -o top.edn
report_timing >> report.timing
report_area >> report.area
```

Compile-Characterize-Write Script Recompile Methodology

This approach characterizes the module with a loading problem and recompiles it with the actual requirements. In the following example, and procedure, illustrated in Figure 4-1, the module "Top" is recompiled and characterized:



Figure 4-1. Compile-Characterize-Recompile Methodology Diagram

- 1. Invoke your synthesis tool.
- **2. Read in the entire compiled design hierarchy.** Type the following command:

current_design Top report_timing

- 3. Characterize the instance "U1." Type the following command: characterize -constraints {U1}
- Set the "current_design" to "A." Type the following command: current_design A
- 5. Write script. Type the following command: write_script > A_1.wscr

Chapter 4: Synthesis Constraints

6. Remove all designs from memory. Type the following command at the prompt:

remove_design -all

7. Analyze the module, "A." Type the following command at the prompt:

analyze -format verilog A.v

8. Elaborate the module, "A." Type the following command at the prompt:

elaborate A

9. Include the script, "A_1.wscr." Type the following command at the prompt:

include A_1.wscr

10. Compile the module, "A." Type the following command at the prompt:

compile

Time-Budget Methodology

A time-budgeting approach defines the accurate timing specification for each module, creates a script file containing the attributes and constraints to implement this specification, and compiles each module with its corresponding script file.

Flattening the Design Hierarchy

When both timing and area are critical, you may want to collapse the hierarchy of the design before optimization. This is only recommended for designs with a gate count less than 10,000, since the compile is CPU and memory intensive. The command for collapsing the hierarchy after invoking Design Compiler or FPGA Compiler before compile is:

ungroup -all -flatten

You may also see further timing and area improvements by ungrouping the design after compiling and then recompiling the design. For example:

```
compile
ungroup -all -flatten
compile
```

Reporting all Cells in the Hierarchy

Use the following procedure to report all cells in the hierarchy:

- 1. Set the "current_design" path to top level.
- 2. Execute the following:

```
m = find (design, "*")
foreach (m1,m)
{current_design m1}
report_cell >> cells
```

Removing all
Designs from
Hierarchy

To remove all designs from the hierarchy, execute the following command:

remove_design find(design,"*")

Internal Tri-State

The antifuse technology does not support internal tri-states. All tristates must be connected to pads and internal tri-states should be recoded to map to multiplexors. Refer to the *Actel HDL Coding Style Guide* for additional information about internal tri-state usage. Chapter 4: Synthesis Constraints

Inferring Buff	fers			
	This section describes the methods to use when inferring input/output buffers in your design.			
Input and Output Buffers	You can infer the INBUF, OUTBUF, TRIBUF, BIBUF, and CLKBUF macros using one of the following commands: <pre>set_port_is_pad insert_pads</pre>			
	Note: These commands infer BIBUFs and TRIBUFs if the "Z" state is defined in your HDL.			
Clock Buffer Macro	You can use the CLKBUF macro to drive clock, preset, and clear inputs of registers and latches. To force the inference of a CLKBUF macro, identify the clock port name and use the following command: <pre>set_port_is_pad <portname> set_port_is_pad <portname> insert_pads</portname></portname></pre>			
HCLKBUF Macro	Use the following commands to infer an HCLKBUF macro. Remember that this macro has a "dont_use" attribute attached to it. remove_attribute act3/HCLKBUF dont_use set_port_is_pad clk /* clk is the name of the clock port */ set_pad_type -clock clk -exact HCLKBUF insert_pads set_dont_use act3/HCLKBUF dont_touch HCLKBUF			
	Note: You must instantiate complex I/O buffers in a gate level format in your HDL file. Refer to "Complex Act 3 I/O Mapping" on page 72 for information about inferring specific ACT 3 I/O cells.			

Reducing the Maximum Fanout Value

By default, the ACT 1 and 40MX family fanout limits are set to 10. The ACT 2, ACT 3, 3200DX, 42MX and 54SX family limits are set to 16. You can change a design's maximum fanout value using the "set_max_fanout" constraint.

You can specify your design's maximum fanout limit to "n." However, Synopsys ignores this limit for logic blocks that have a "dont_touch" attribute. This procedure only fixes design rules violations such as "max_fanout." The design is not re-optimized. Use the following procedure to force Synopsys to apply the "max_fanout" constraint to blocks with a "dont_touch" attribute:

Note: You should not apply the new fanout limit to the clock buffer network.

- 1. Remove the "dont_touch" attribute from blocks that do not obey the fanout constraints.
- 2. Specify the maximum fanout value for your top level design. Use the "set_max_fanout" attribute:

set_max_fanout <value> <design_name>
set_dont_touch_network <name_of_clock_port>

You can specify the maximum fanout value for your complete design with the following command:

set_max_fanout <value> find (design, -hier, "*")
set_dont_touch_network find(port, clock)

3. Apply fanout constraints to "dont_touch" blocks. Because fanout constraints are not applied to blocks with a "dont_touch" attribute, you must remove the "dont_touch" attribute by using the following command:

```
remove_attribute <block_name> dont_touch
```

4. Perform a "design_rule" compile for your top level design. Type the following command at the prompt:

compile -only_design_rule

Chapter 4: Synthesis Constraints

Register Type Preferences

The "set_register_type" command can be used to select your desired flip-flops. The following example specifies "DFC1B" as the flip-flop:

```
...
current_design = top
set_register_type -flip_flop DFC1B
```

Avoid Using Certain Cells

Use the "set_dont_use" command to select cells that you do not wish (V)HDL Compiler to infer. The following commands force the (V)HDL Compiler to ignore all "DFP*" cells:

read act.db
set_dont_use act3/DFP*

Register Balancing

Retiming a design for pipelining moves registers to achieve minimum cycle time. The "balance_registers" command balances delays on sequential designs that can accept a latency and continuously get data. Figure 4-2 illustrates a schematic before register balancing.



Figure 4-2. Schematic Before Register Balancing

The following Verilog example shows sample code and script:

```
module mult_pipe (clock, ain, bin, qout);
input clock;
input [3:0] ain, bin;
output [7:0] qout;
reg [7:0] pipe1, pipe2, pipe3, qout;
always @ (ain or bin)
pipe1 <= ain * bin;</pre>
```

```
// DO NOT USE ASYNCHRONOUS RESET
always @ (posedge clock) begin
pipe2 <= pipe1;
pipe3 <= pipe2;
qout <= pipe3;
end
endmodule</pre>
```

The following design script file compiles the preceding HDL description and forces Synopsys to perform register balancing. Figure 4-3 illustrates the schematic after register balancing.

```
include actsetup.scr
read -f verilog mult_pipe.v
/* Specify the number of pipeline stages */
/* In this case, there are 3 pipeline stages */
create_clock clock -period 3
/* Compile at low effort before balance_registers */
/* because balance_registers rearranges the gates */
/\,\star and performs another map
                                                   */
compile -map_effort low
/* Specify the desired post-pipelining clock period */
create_clock clock -period 10
/* Issue the balance_registers command */
balance_registers
/* Perform a high map effort compile */
compile -map_effort high
/* Always insert_pads after balance_registers */
set_port_is_pad
insert_pads
/* Write the db and edif netlist */
write -f db -h -o mult_pipe1.db
write -f edif -h -o mult_pipe1.edn
        clk
```

Figure 4-3. Schematic after Register Balancing

Limitations

The main limitations of this command are listed below:

- The "balance_registers" command cannot perform retiming between flip-flops that have a "dont_touch" attribute.
- The "balance_registers" command treats any asynchronous set/reset flip-flop as "dont_touch" and does not move it. This is because "balance_registers" does not determine the initial state of the flip-flop when it is asynchronously set through the set and reset pins.
- The "balance_registers" command cannot perform retiming if the design has gated clocks. For example, if you use the "insert_pads" command before the "balance_registers" command, Synopsys considers the CLKBUF as a gate and issues errors.
- Use only edge triggered D flip-flops when you use the "balance_registers" command, not latches. All clock inputs of the flip-flops in the design must be connected to the same edge of the same clock.

Refer to the Synopsys documentation for additional information about register balancing.

Removing Attributes

The "dont_use" attribute is attached to several Actel sequential macros that use one sequential or one combinational logic module. You can remove the "dont_use" attribute from these modules by using the "remove_attribute" command.

Using (Q)CLKINT

At times, designs may need to use high fanout drivers to drive internally generated clocks, reset networks, enable networks, etc. Refer to the *Actel HDL Coding Style Guide* for additional information about using (Q)CLKINT drivers.

Wide Decode Cells in 3200DX and 42MX

All 3200DX devices and some 42MX devices have a limited number of wide decode cells. To utilize these cells efficiently and to avoid excess usage, compile your design hierarchically and set or remove the "dont_use" attribute from these cells before compiling selected modules. Refer to the *Integrator Series FPGAs: 40MX and 42MX Families* Data Sheet for information about which 42MX devices have wide decode cells.

By default, the cells in the 3200DX family do not have the "dont_use" attribute set and the cells in the 42MX family do have the "dont_use" attribute set. Actel has provided scripts in the "\$ACT_SYNOPDIR/ sctipts/<act_fam>" directory that toggle the wide decode "dont_use" and "dont_touch" attributes. The "WD_use" script sets wide decode cells as usable. The "WD_dont_use" script sets the cells as not usable.

To use wide decode cells:

Run the "WD_use" script on the design.

To not use wide decode cells:

Run the "WD_dont_use" script on the design.

You can also instantiate these cells in Verilog or VHDL, or use ACTgen to generate macros that use these wide decode cells.

Note: These cells are located on the periphery of the device, thereby reducing output delay.

Actel-Synopsys Design Considerations

This chapter contains information and procedures to assist you in creating Actel designs with Synopsys tools. This includes information about compiling designs that use DesignWare components, translating designs and timing constraints, and assigning pins in Synopsys. Also included is information about using ACTmap to optimize I/O placement, correcting bus array syntax, and running Designer in batch mode through Synopsys.

Other sections include using control flow commands, complex ACT 3 I/O mapping, and how to instantiate ACTgen macros. Procedures to generate EDIF and structural HDL netlists are also provided. Finally, information about where to find radiation environment design techniques and where to find information about maintaining technology independence is included.

Compiling Designs with DesignWare Components

Actel recommends that you do not ungroup and flatten the DWACT library components before compiling a design. Ungrouping and flattening before compilation can offset gains in area or timing you expect from using the DWACT library. If you must flatten the design before compiling it, you must use the following command:

```
current_design <design_name>
ungroup -all -flatten
```

If you do not want to use the Actel DesignWare library, you can add the following lines to your compile script:

```
script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr
read dwact.sldb
set_dont_use dwact.sldb/DWACT*
```

If you do not wish to use the Synopsys DesignWare library, you can add the following lines to your script:

Verilog

```
read dw01.sldb
read -f verilog design.v
dont_use standard.sldb/DW0*
```

VHDL

read dw01.sldb
read -f vhdl design.vhd
dont_use standard.sldb/DW0*

Translating Designs from Other Technologies

You may have EDIF netlists or a Synopsys design database for other technologies that you wish to translate to an Actel design. If the behavioral HDL is not available, you cannot compile your design. However, you can translate your design using the following procedure:

1. Set the target library of the other vendor. Type the following command at the prompt:

target_library = ASICLIB.db link_library = ASICLIB.db

2. Read the design netlist or database. Type one of the following commands at the prompt:

read -f edif <design_name>.edif
read -f db <design_name>.db

3. Set the current design. Type the following command at the prompt:

current_design = <design_name>

4. Point to the Actel library. Type the following commands at the prompt:

```
actlib = get_unix_variable ("ACT_SYNOPDIR") + "/syn"
search_path = search_path +{actlib +/<act_fam>}
target_library = {act.db}
symbol_library = {actsym.sdb}
```

5. Translate the design. Type the following command at the prompt:

translate -verify -verify_effort high

6. Write a new netlist. Type one of the following commands at the prompt:

write -f edif -h -o <name>.edn
write -f db -h -o <name>.db

Translating Timing Constraints into Designer

The "synop2dcf" program is an Actel interface program that converts Synopsys timing constraints to DCF format. This program can only convert top-level Synopsys design constraints. If you specify design constraints at a lower level and you want to specify the constraints for a layout tool, use DirectTime Edit or edit the DCF file. Refer to the *Designing with Actel* manual for information about using DirectTime Edit. The "synop2dcf" program uses your EDIF netlist to generate timing constraints. To translate timing constraints using "synop2dcf," type the following command at the prompt:

synop2dcf fam:<act_fam> ednin:<design_name>.edn <design_name>

Note: Make sure that you have established reasonable constraints when compiling your design. An over-constrained design may not improve layout results.

Assigning Pins in Synopsys

After inserting pads and compiling the design, use the following command to assign pin locations in Synopsys:

set_attribute find(net, en3) "ALSPIN" -type string "4"

The "en3" is the net/port name and "4" is the pin location. When generating your EDIF netlist, make sure the following switches are set in your ".synopsys_dc.setup" file:

```
edifout_dc_script_flag = "als"
edifout_write_attributes = "true"
edifout_write_properties_list = {ALSPIN}
```

When you import the EDIF netlist into Designer, the port/net "en3" is automatically assigned to pin number "4."

Using ACTmap to Optimize I/O Placement

You may be able to improve your Synopsys design results by optimizing your netlist using the ACTmap VHDL Synthesis tool. ACTmap performs the following features to optimize a Synopsys generated netlist:

- automatic absorption of internal flip-flops into ACT 3 I/O flip-flops
- · netlist optimization for either area or speed

Refer to the *Designing with Actel* manual and the *ACTmap VHDL Synthesis Methodology Guide* for additional information.

Bus Array Syntax

The "read_array_naming_style" script reads in Verilog and VHDL description languages. This script converts the square brackets ([]) to angle brackets (< >). However, Synopsys can only recognize script files that use "[]" in array notation. To read in script files that include "< >," use the following script:

define_name_rules ARRAY -restricted "<" -replacement_char "["
define_name_rules ARRAY -restricted ">" -replacement_char "]"
change_names -rules ARRAY -hierarchy

Use the following command to list the rules that are applied:

report_name_rules ARRAY

Script Mode Place and Route

You can run Designer in the Synopsys environment by creating a DSF script and invoking Designer in batch mode through Design Compiler. Use the following procedure to invoke Designer in Design Compiler:

1. Invoke dc_shell. Type the following command at the prompt:

dc_shell

2. Generate an EDIf netlist. Type the following command at the prompt:

write -f edif -h -o <design_name>.edn

3. Run Designer in batch mode. Type the following command at the prompt:

sh wdesigner script_file:<design_name>.dsf
script_mode:batch"

The following is an example DSF file:

```
/*Session script file*/
main()
{
    new_design();
    setup_design("<design_name>","ACT3");
    import_netlist("<design_name>.edn","EDIF");
    set_device( "DIE = A1415A, PACKAGE = 100 PQFP" );
    compile();
    import_aux_file("<design_name>.dcf","DCF");
    set("LAYOUT_MODE","TIMING_DRIVEN);
    layout();
    save_as("<design_name>.adb");
}
```

Control Flow Commands

This section contains example control flow commands. These examples demonstrate how to check the dc shell status and how to use a string variable inside a "foreach" statement.

Testing the DC	To test the dc shell status, add the following lines to your compile				
Shell Status	script:				
	check_design				
	if (dc_shell_status == 0)				
	{				
	quit				
	} else {				
	rest of script				
	}				

Using String
Variables
Inside a
Foreach
Statement

To use a string variable inside a "foreach" statement, add the following lines to your compile script:

```
all_modules = {.....}
foreach (module, all_modules)
{
  current_design = module
  create_clock -name .....
dont_touch_network
  compile
}
```

Complex Act 3 I/O Mapping

For fast clock to out times, or if you have depleted array resources, you can synthesize ACT 3 sequential I/O macros that contain registers with asynchronous preset or clear. Inference of these cells is limited to a small subset of cells and is only successful if the HDL code and scripts match the examples provided in this section. You must make sure that these macros are driven by the dedicated buffers, such as IOCLKBUF and IOPCLBUF. The sequential macros' IOPCL pin and CLK pin must be driven by IOPCLBUF and IOCLKBUF, respectively. This section provides you with information regarding synthesizing ACT 3 sequential I/O macros.

Figure 5-1 shows the directory structure for the ACT 3 I/O macros.



Figure 5-1. ACT 3 I/O Macros Directory Structure

The "act3io.db" file is a sequential I/O library file. The "act3io_linklib.db" file is the link library file.

You can synthesize ACT 3 I/O macros from HDL descriptions using a script file provided by Actel when working with Synopsys tools. Synopsys inserts the relevant I/O pads and connects them to the macros. This process infers the sequential macros and connects them to the IOPCLBUF and IOCLKBUF. Figure 5-2 is an example of an ACT 3 macro with sequential macros driven by IOPCLBUF and IOCLKBUF.



Figure 5-2. IOPCLBUF and IOCLKBUF driven sequential cells

Synthesizing ACT 3 I/O Macros

Synthesizing macros involves reading the design into Synopsys, executing commands before compilation, and executing a script after compilation.

Table 5-1 and Table 5-2 illustrate the sequential I/O cells that Synopsys can synthesize. These cells are mapped onto Actel macros through a link library named "act3io_linklib.db."

Cells	Description			
IREC_SY	An input register with clear			
IREP_SY	An input register with preset			

Table 5-1. Sequential Input Cells Available for Synthesis

Table 5-2. Sequential Output Cells Available for Synthesis

Cells	Description			
ORECTH_SY	An output register with clear, tri-enable, high slew			
ORECTL_SY	An output register with clear, tri-enable, low slew			
OREPTH_SY	An output register with preset, tri-enable, high slew			
OREPTL_SY	An output register with preset, tri-enable, low slew			
ORECTH_NO_TRI	An output register with clear, high slew			
ORECTL_NO_TRI	An output register with clear, low slew			
OREPTH_NO_TRI	An output register with preset, high slew			
OREPTL_NO_TRI	An output register with preset, low slew			

To automatically synthesize sequential cells:

1. Configure your ".synopsys_dc.setup" file. Make sure your ".synopsys_dc.setup" file includes the following lines. These lines add the sequential I/O library file "act3io.db" to the target and link libraries:

```
link_library = {act3.db, act3io.db}
target_library = {act3.db, act3io.db}
```

- 2. Read the HDL design into Synopsys. Refer to the *Synopsys Command Reference Manual* for additional information.
- **3.** Set pad types and insert pads. Type the following commands at the prompt:

```
set_port_is_pad <design_name>
set_pad_type -clock clk
insert_pads -thru_hierarchy
```

In this command, the "clk" variable is the clock pin name in the design. The "-thru_hierarchy" option forces Synopsys to insert pads at the hierarchy level where the flip-flops exist. Otherwise, the sequential I/O cell synthesis may not occur.

4. Set switches and compile the design. Figure 5-3 shows the compilation results. This example illustrates that the pad that drives the CLK inputs of the sequential output cells does not drive any other non-sequential I/O cells.



5. Ungroup and flatten the design. The following command ungroups and flattens the design, and removes the hierarchy. This command also allows the "act3io.script" script to automatically perform net connections for the sequential I/O cells.

ungroup -all -flatten

6. Run automatic insertion. The following command runs the script "act3io.script" for performing IOPCLBUF and IOCLKBUF automatic insertion, and for setting proper net connections to sequential I/O cells:

include act3io.script



Figure 5-4 shows the design results after executing the script:

Figure 5-4. Script Execution Results

Note: The port names are not changed when the IOCLKBUF and the IOPCLBUF are inserted. However, if a design has the "clk" pad driving cells other than the sequential I/O cells, as shown in Figure 5-5, then the script creates additional ports, namely IOPCL_NEW and IOCLK_NEW. These new ports are connected to IOPCLBUF and IOCLKBUF that drive the sequential I/O cells. The I/O cells that are not sequential are driven by the user defined ports, as shown in Figure 5-6.



Figure 5-5. "CLK" Pad Driving Sequential I/O Cells and Other Logic

Complex Act 3 I/O Mapping



Figure 5-6. Corrected Design After Script Implementation

7. Set links to local link library in your file. This command sets a link to the local link library:

set_local_link_library act3io_linklib.db
link

This command maps the sequential I/O cells to ACT 3 I/O macros, and links the design. The link that maps the sequential I/O cells on to ACT 3 I/O cells is shown in Figure 5-7.



Figure 5-7. Sequential I/O Cell to ACT 3 I/O Cell Link

8. Write and Save your design.

Automatic Synthesis

The following script demonstrates how to automatically synthesize the ACT 3 I/O cells. You can insert more Synopsys commands without offsetting the control flow.

search_path = seach_path + {.}
search_path = search_path + {<path to act3 library>}
/* Stage 1 */
link_library = {act.db, act3io.db}
target_library = {act.db, act3io.db}
/* Stage 2 */
read -format verilog your_design.v
current_design = your_design
/* Stage 3 */
set_port_is_pad <design_name>
set_pad_type -clock CLK /*clock port CLK*/
insert_pads -thru_hierarchy
/* Stage 4 */
max_area 0

```
create_clock -period 40 clk
compile
/* Stage 5 */
ungroup -all -flatten
/* Stage 6 */
include act3io.script
/* Stage 7 */
set_local_link_library act3io_linklib.db
link
/* Stage 8 */
report_reference > your_design.ref
write -format edif -hierarchy -output your_design.edn
report_hierarchy > your_design.hierarchy
report_cell > your_design.cell
```

```
quit
```

Inferring a Sequential Cell

The following example is a flip-flop model with active low reset. If an input pad is attached to this flip-flop and the design is compiled, the inference result is the "IREC_SY," a sequential input cell. This cell can be mapped on to the ACT 3 I/O cell "IREC," with the link library. Actel recommends that you follow the procedure for automatically synthesizing sequential cells described on page 75.

```
module your_design (data, clk, reset, q );
input data, clk, reset;
output q;

    reg q;
    always @ (posedge clk or negedge reset)
        if (!reset)
        q = 1'b0;

    else
        q = data;
```

endmodule

ACT 3 I/O Synthesis Notes

This section describes notes to remember when synthesizing the design:

• The script "act3io.script" will work *only* if all the IOPCL pins of the sequential I/O cells are driven by a single net. You must make sure that the reset or preset net names you chose in the HDL description are the same. This ensures that all the IOPCL pins are driven by one signal IOPCLBUF.

Note: This also applies to sequential I/O cell CLK pins. All CLK pins must be driven by the same net.

- The flip-flops modeling must be done so they can be easily inferred by Synopsys and must use an active high clock. The *Synopsys HDL Reference Manuals* provide you with few examples to model the flip-flops and to achieve the desired results. In some cases, because of Synopsys limitations, bus logic or vectors may not be inferred. Refer to the Synopsys *HDL Coding Styles: Sequential Devices Application Note* for additional information.
- A sequential input macro is inferred when an input pad drives a flip-flop.
- A sequential output macro is inferred when a flip-flop drives an output pad.
- In order for the "act3io.script" script to function, you must ungroup and flatten your design. This script is located in the "\$ACT_SYNOPDIR/scripts/act3" directory.
- There should be no logic gates present between the IOPCLBUF and IOPCL pin of the sequential I/O cells, and between the IOCLKBUF and the CLK pins of the sequential I/O cells.

Instantiating ACTgen Macros

The ACTgen Macro Builder can create macros to be instantiated into a Verilog or VHDL design that effectively use the Actel architecture to achieve optimum performance and minimal module count to improve designer productivity. Recommendations on when to instantiate and the procedure for instantiating an ACTgen macro are described in this section.

When to Instantiate

ACTgen can generate optimized macros for the Actel architecture for several functions. Actel recommends that you instantiate the following functions:

- RAM cells
- FIFOs
- Multipliers
- Counters

Generate an ACTgen Macro

Use the following procedure to generate an ACTgen macro that can be instantiated into a design:

- 1. Invoke ACTgen.
- 2. Select the family, macro type, and macro options.
- **3.** Generate your macro as an HDL description. Make sure you specify VHDL or Verilog as the Netlist/CAE Format when generating the macro.

Instantiate the Macro After you have generated the macro, you must instantiate it into your HDL design. The following examples show how to instantiate the macro.

Verilog

Verilog provides two instantiation methods. One method is explicit port name specification that allows signals to be defined in any order. The second method is implied port reference by position, where the instantiation ports correspond to the library cell pins by the order the port names given in the instantiation. The following is and example of

a 32 x 32 bit dual-port RAM macro. This example uses explicit port name specification. The macro is illustrated in Figure A-8.



Figure A-8. ACTgen Generated 32 x 32 bit Dual Port RAM

```
RAM32x32 U0 (.RAddress[4:0](read[4:0]),.WE(writeen),
.RClock(clk1), .WAddress[4:0](write[4:0]), .RE(readen),
.WClock(clk2), .Data[31:0](d[31:0]), .Q[31:0](q[31:0]));
```

In the example, the pins "RAddress[4:0]," "WAddress[4:0]," "RE," "WE," "RClock," "WClock," "Data[31:0]," "Q[31:0]" are connected to nets "read[4:0]," "write[4:0]," "readen," "writen," "clk1," "clk2," "d[31:0]," "q[31:0]," respectively. "U0" is the instance name.

VHDL

The following is an example of a $32 \ge 32$ bit FIFO macro. The macro is illustrated in Figure A-9.



Figure A-9. ACTgen Generated 32 x 32 bit FIFO

```
/*DECLARATION*/
component FIF032x32
   port (RE, WE, Aclr, Clock, REF, WEF: in std_logic;
       Data: in std_logic_vector(31 downto 0);
       FF, EF: out std_logic;
       Q: out std_logic_vector (31 downto 0);
end component;
/*CONCURRENT STATEMENT*/
U1: FIF032x32 port map ( RE => renable,
                         WE => wenable,
                         Aclr => asynclr,
                         Clock => clkin
                         REF => refin,
                         WEF => wefin,
                         Data => din
                         Q => qout,
                         FF => ffout,
                         EF => efout);
```

In the example, "/*DECLARATION*/" is the macro declaration, and "/ *CONCURRENT STATEMENT*/ is the macro concurrent statement. U1

is the instance name. Pins "Data0" through "Data31," "Q0" through "Q31," "RE," "WE," "Aclr," "Clock," "REF," "WEF," "FF," and "EF" are connected to nets "din0" through "din31," "q0" through "q31," "renable," "wenable," "asynclr," "clkin," "refin," "wefin," "ffout," and "efout" respectively.

Add "dont_touch" Attributes

You must add the "dont_touch" attribute to the instantiated macro to make sure that Synopsys does not attempt to synthesize the code. To verify that the "dont_touch" attribute has been added to the macro, type the following command:

report -cell <macro_name>

Compile the	After you have compiled the design, the macro is instantiated into your
Design	design.

Generating an EDIF Netlist

Use the following procedure to generate an EDIF netlist from Synopsys:

1. Configure your ".synopsys_dc.setup" file. Make sure your ".synopsys_dc.setup" file includes the appropriate Actel family "actsetup.scr" file. Add the following lines to your ".synopsys_dc.setup" file:

script_lib = get_unix_variable ("ACT_SYNOPDIR")
include script_lib + /scripts/<act_fam>/actsetup.scr

If you have assigned pins in your design, you must also include the following lines to your ".synopsys_dc.setup" file:

```
edifout_dc_script_flag = "als"
edifout_write_attributes = "true"
edifout_write_properties_list = {ALSPIN}
```

2. Generate the EDIF netlist. Type the following command at the prompt:

write -f edif -h -output <design_name>.edn

Generating a Structural HDL Netlist

Generate a structural HDL netlist from your EDIF netlist by either exporting it from Designer or by using the "edn2vhdl" or "edn2vlog" program. The structural HDL netlist generated by Designer and "edn2vhdl" use std_logic for all ports. The bus ports are in the same bit order as they appear in the EDIF netlist. You can also generate a structural VHDL netlist directly from Synopsys. Actel does not support the use of structural Verilog netlists generated by Synopsys.

To generate a netlist using Designer:

- 1. Invoke Designer.
- 2. Import your EDIF netlist. Select the Import Netlist command from the File menu. The Import Netlist dialog box is displayed. Specify EDIF as the Netlist Type, GENERIC as the Edif Flavor, and VHDL or Verilog as the Naming Style. Type the full path name of your EDIF netlist or use the Browse button to select your design. Click OK.
- **3. Export a structural HDL netlist.** Select the Export command from the File menu. The Export dialog box is displayed. Specify Netlist File as the File Type and VHDL or Verilog as the Format. Click OK.

To generate a structural HDL netlist using edn2vhdl or edn2vlog:

Type the following command at the prompt:

VHDL

edn2vhdl fam:<act_fam> <design_name>

Verilog

edn2vlog fam:<act_fam> <design_name>

To generate a structural VHDL netlist using Synopsys:

Execute the following commands in Synopsys:

```
default_name_rules = "sge_vhdl"
change_names -hierarchy
vhdlout_write_components = FALSE
vhdlout_write_top_configuration = TRUE
vhdlout_use_package = {IEEE.std_logic_1164 <act_fam>}
write -f vhdl -h -o <design_name>.vhd
```

Designing for Radiation Environments

Actel has macros and scripts that allow designers to create Actel designs for radiation environments. Refer to *Enhanced Tools for Minimizing Single Event Upset Effects* and the *Using Synopsys to Design Actels Radiation-Hardened FPGAs* Application Note on the Actel Web site (http://www.actel.com) for additional information.

Maintaining Technology Independence

Instantiating a macro implies that you are limited to a particular technology. However, you can remain technology independent by creating a behavioral model of the instantiated function. Refer to the *Actel HDL Coding Style Guide* for additional information about dual architecture coding.

Synthesis Library Information

This appendix contains information about the Actel Synopsys synthesis libraries. This includes information about timing parameters in the libraries and about which cells in the libraries have been marked with certain attributes. The maximum fanout value, and how to change it, is provided as well. Also included in this appendix are sections with ACT 3 and 54SX specific information, some guidelines to remember when using the synthesis libraries, and information about the synthesis library operating conditions.

Timing Parameters

The libraries include temperature and voltage derating factors. The library default operating are not reliable. Always set an operating condition before synthesizing. All sequential modules include setup and clock pulse width specifications according to the *FPGA Databook and Design Guide*. Refer to "Synthesis Library Operating Conditions" on page 94 for a list of operating conditions for Actel family devices.

Attributes

Some macros in the synthesis libraries have been marked with a "dont_use" or "dont_touch" attribute. Use the "remove_attribute" command to map to one of these cells. For example, if to remap to an ACT 2 DFC1 cell, use the following command:

remove_attribute act2/DFC1 dont_use

Clock buffer Interface macros (Gxxxx) are marked with "dont_use" and "dont_touch" attributes. "Dont_touch" stops the compiler from removing these macros from existing designs. "Dont_use" stops the compiler from using clock buffer Interface macros. Gating the clock is not a recommended design practice.

A "dont_use" attribute is attached to the CLKINT macro so that Synopsys does not infer this macro instead of a regular buffer. The CLKINT macro must only be used for driving the high fanout global clock networks. A "dont_use" attribute is attached to all complex I/Os, as Synopsys may not map them efficiently. A "dont_use" attribute is also attached to inefficient register and latch implementations. The following tables list the macros in each Actel device family with the "dont_touch" and "dont_use" attribute attached.

Appendix A: Synthesis Library Information

ACT 1/40MX Table A-1 lists the ACT1/40MX macros that have the "dont_touch" and "dont_use" attribute attached.

"dont_touch" Macros			"dont_use" Macros
CLKBIBUF	GMX4	HA1B	CLKBIBUF
DFPCA	GNAND2	HA1C	GAND2
FA1	GNOR2		GMX4
FA1A	GOR2		GNAND2
FA1B	GXOR2		GNOR2
FA2A	HA1		GOR2
GAND2	HA1A		GXOR2

Table A-1. ACT 1/40MX "dont_touch" and "dont_use" Macros

ACT 2/1200XL Table A-2 lists the ACT 2/1200XL macros that have the "dont_touch" and "dont_use" attribute attached.

Table A-2. ACT 2/1200XL "dont_touch" and "dont_use" Macros

"dont_touch" Macros		"dont_use" Macros			
BBDLHS	GMX4	HA1B	BBDLHS	GAND2	OBDLHS
CLKINT	GNAND2	HA1C	BUF	GMX4	TBDLHS
CY2B	GNOR2	IBDL	CLKINT	GNAND2	
FA1A	GOR2	OBDLHS	DFC1	GNOR2	
FA1B	GXOR2	TBDLHS	DFC1A	GOR2	
FA2A	HA1		DLC1	GXOR2	
GAND2	HA1A		DLC1A	IBDL	

3200DX/42MX

Table A-3 lists the 3200DX/42MX macros that have the "dont_touch" and "dont_use" attribute attached.

"dont_touch" Macros			"dont_use" Macros		
BBDLHS	GOR2	RAM4FR	BBDLHS	GXOR2	RAM8FF
CLKBIBUF	GXOR2	RAM4RA	CLKBIBUF	IBDL	RAM8FR
CLKINT	HA1	RAM4RF	CLKINT	OBDLHS	RAM8RA
CY2B	HA1A	RAM4RR	DFC1	QCLKBUF	RAM8RF
DXAX7	HA1B	RAM8FA	DFC1A	QCLKINT	RAM8RR
FA1A	HA1C	RAM8FF	DLC1	RAM4FA	TBDLHS
FA1B	IBDL	RAM8FR	DLC1A	RAM4FF	
FA2A	OBDLHS	RAM8RA	GAND2	RAM4RF	
GAND2	QCLKBUF	RAM8RF	GMX4	RAM4FR	
GMX4	QCLKINT	RAM8RR	GNAND2	RAM4RA	
GNAND2	RAM4FA	TBDLHS	GNOR2	RAM4RR	
GNOR2	RAM4FF		GOR2	RAM8FA	

Table A-3. 3200DX/42MX "dont_touch" and "dont_use" Macros

Appendix A: Synthesis Library Information

ACT 3

The following table lists the ACT 3 macros that have the "dont_touch" and "dont_use" attribute attached.

"dont_touch" Macros		"dont_use" Macros		
BIECTH	FEPTMH	BIECTH	DLC1G	IOCLKBUF
BIECTL	FEPTML	BIECTL	DLE2C	IODFE
BIEPTH	GAND2	BIEPTH	DLE3B	IODFEC
BIEPTL	GMX4	BIEPTL	DLE3C	IODFEP
BRECTH	GNAND2	BRECTH	DLP1	IOPCLBUF
BRECTL	GNOR2	BRECTL	DLP1A	IREC
BREPTH	GOR2	BREPTH	DLP1B	IREP
BREPTL	GXOR2	BREPTL	DLP1C	OBUFTL
CLKBIBUF	HA1	CLKBIBUF	FECTH	ORECTH
CLKINT	HA1A	CLKINT	FECTL	ORECTL
CS2	HA1B	DECETH	FECTMH	OREPTH
CY2B	HA1C	DECETL	FECTML	OREPTL
DECETH	IBUF	DEPETH	FEPTH	
DECETL	IODFE	DEPETL	FEPTL	
DEPETH	IODFEC	DFC1	FEPTMH	
DEPETL	IODFEP	DFC1A	FEPTML	
FA1A	IREC	DFC1E	GAND2	
FA2A	IREP	DFC1G	GMX4	
FECTH	OBUFTL	DFP1	GNAND2	
FECTL	ORECTH	DFP1A	GNOR2	
FECTMH	ORECTL	DFP1B	GOR2	
FECTML	OREPTH	DFP1D	GXOR2	
FEPTH	OREPTL	DLC1A	HCLKBUF	
FEPTL		DLC1F	IBUF	

Table A-4. ACT 3 "dont_touch" and "dont_use" Macros
54SX

Table A-5 lists the 54SX macros that have the "dont_touch" and "dont_use" attribute attached.

"dont_toue	"dont_touch" Macros		"dont_use" Macros		
CLKINT	GNOR2	CLKINT	DLP1A	HCLKBUF	
DFPCA	GOR2	CM8F	DLP1B		
FA1	GXOR2	CM8INV	DLP1C		
FA1A	HA1	DFPCA	GAND2		
FA1B	HA1A	DLC1A	GMX4		
FA2A	HA1B	DLE2C	GNAND2		
GAND2	HA1C	DLE3B	GNOR2		
GMX4	HCLKBUF	DLE3C	GOR2		
GNAND2		DLP1	GXOR2		

Table A-5. 54SX "dont_touch" and "dont_use" Macros

Max Fanout

The "max_fanout" attribute can be added to input ports or designs. You can set the "max_fanout" attribute on ports or designs using the "set_max_fanout" command. For example, to set the max_fanout globally to 12, execute the following command:

set_max_fanout 12 <design_name>

The maximum fanout for the ACT 1 and 40MX family libraries is 10. The maximum fanout for the ACT 2, 1200XL, ACT 3, 3200DX, 42MX, and 54SX family libraries is 16. The maximum fanout for any cell is less than or equal to 16. The clock buffer has a maximum fanout of 2000.

ACT 3 Specific Information

The following is a list of ACT 3 specifics within the Synthesis libraries:

- The IOCLKBUF cell is hardwired to the CLK input of the I/O modules.
- The IOPCLBUF cell is hardwired to the PRESET/CLEAR inputs of the I/O modules.
- CLKBUF and HCLKBUF cannot drive I/O modules.
- HCLKBUF is a special clock buffer that is hard-wired to clock pins of the sequential logic modules. Since Synopsys infers the HCLKBUF macro every time a clock buffer is needed, a "dont_use" attribute is attached to the HCLKBUF.

The sequential elements listed in Table A-6 cannot be connected to HCKLBUF, as they are built from combinatorial modules.

DFP1	DFPC	DLC1G	DLP2C	DFP1A
DFP1B	DLC1	DLE2C	DLP1A	
DFP1D	DLC1A	DLE3B	DLP1B	
DFPCA	DLC1F	DLE3C	DLP1C	

Table A-6. Macros that Cannot be Connected to HCLKBUF

All other sequential elements can connect to HCLKBUF. The macros listed in previous table have a "dont_use" attribute attached to them in the ACT 3 library. Refer to "Inferring Buffers" on page 60 for information about how to use the ACT 3 library clock buffers.

54SX Specific Information

The HCLKBUF clock buffer can drive the clock pin of all flip flops, but no latches. Since Synopsys infers the HCLKBUF macro every time a clock buffer is needed, a "dont_use" attribute is attached to the HCLKBUF. All other sequential elements can connect to HCLKBUF. Refer to "Inferring Buffers" on page 60 for information about how to use the 54SX library clock buffers.

Additional Information

The following is a list of guidelines to remember when using the Actel Synopsys Synthesis libraries:

- Design Compiler does not enforce fanout restrictions across hierarchical boundaries and black boxed macros.
- Synopsys may create black boxes for some Actel cells due to their complexity. These cells have the "removable" property and should not be used by Synopsys.
- Inverter removal optimization is only available with FPGA Compiler. It is not a feature of Synopsys DC Expert or DC Professional. This feature may have the added benefit of improving your results by removing unnecessary inverters.

Appendix A: Synthesis Library Information

Synthesis Library Operating Conditions

The operating condition is a concatenation of the application, operation condition, speed grade, and (optionally) any family specific conditions. For example, to set worst case commercial temperature and voltage range (70°C, 4.75 V), standard speed grade conditions, type the following command:

set_operating_conditions COMWCSTD

Default Operating Conditions

Default operating conditions have been set in each Actel device family. The default operating condition has been set as the fastest common speed grade for each device in the family. Table A-7 lists the default operating condition for each family

Family	Default Operating Condition
ACT 1	COMWC-3
ACT 2/1200XL	COMWC-2
ACT 3	COMWC-3
3200DX	COMWC-1
40MX	COMWC-3
42MX	COMWC-2
548X	COMWC-2

Table A-7. Default Operating Conditions

Synthesis Library Operating Conditions

Table A-8 describes the nomenclature and descriptions of the Synthesis library operating conditions. Not all operating conditions are available for all Actel devices or families.

Condition Type	Nomenclature	Description	
	СОМ	Commercial Range	
	IND	Industrial Range	
Application	MIL	Military Range	
	RH0	Simulation at OKR Radiation	
	RH3	Simulation at 300KR Radiation	
	BC	Best Case	
Operating Conditions	WC	Worst Case	
Conditions	TC	Typical Case	
	-1	-1 Speed Grade	
	-2	-2 Speed Grade	
Speed Grade	-3	-3 Speed Grade	
	-F	-F Speed Grade	
	STD	Standard Speed Grade	
	XL	1200XL	
	XLV	1200XL with 3.3V Operating Voltage	
	V	3.3V Operating Voltage	
Family Specific	В	Mixed 3.3/5V Operating Voltage (42MX only)	
	_D3265	3265DX Device Conditions	
	_D3265V	3265DX Device Conditions with 3.3V Operating Voltage	

Table A-8. Synthes	is Library O	perating	Conditions
--------------------	--------------	----------	------------

DesignWare Library Information

This appendix provides information about the Actel DesignWare libraries. This includes a description of the DesignWare library, as well as descriptions of DesignWare adders, subtractors, comparators, counters, incrementers, and decrementers. This appendix also contains guidelines for improving compilation time when using DesignWare components and module and count performance for each of the DesignWare modules. Refer to "DesignWare Module Coding" on page 31 for information about implementing DesignWare modules into a design.

DesignWare Library Description

The Actel DesignWare libraries currently support the ACT 2, 1200XL, 3200DX, 42MX, ACT 3, and 54SX families. All modules except counters can be inferred in Synopsys. Counters must be instantiated because Synopsys does not yet support DesignWare counter inference.

When one of the inputs is a constant in cases such as c=a+1 or c=a-1, there is a reduction in the number of modules. There could also be a gain in area or time due to the constant pushing performed by Synopsys. Table B-1 lists the Actel DesignWare library modules supported for the Actel device families.

Module Name	Implementation	Binding Operator	Reference
DWACT_ADD	RIPADD, MFADD, FADD	+	page 98
DWACT_SUB	RIPSUB, MFSUB, FSUB	-	page 99
DWACT_CMPGT DWACT_CMPLT DWACT_CMPLE DWACT_CMPGE	FCOMP	> < <= >=	page 100
DWACT_UP_CTR	TLACNT, COMPCNT	NA	page 101
DWACT_DN_CTR	TLACNT, COMPCNT	NA	page 101
DWACT_INC	FINC	+	page 102
DWACT_DEC	FDEC	-	page 103

Table B-1. Supported Modules

Appendix B: DesignWare Library Information

DesignWare Library Adders

Figure B-1 shows an example DesignWare adder symbol. In this example, "a" is the addend, "b" is the augend, "sum" signal is the result, and "co" is the carry out. The DesignWare adder is defined as "DWACT_ADD" and is available in three implementations:

- "RIPADD" Ripple Adder
- "MFADD" Medium Fast Adder
- "FADD" Very Fast Adder



Figure B-1. DesignWare Adder Symbol

Table B-2. Adder Pin Description

Pin Name	Size	Туре	Function
a	2-32 bits	input	Input data bus
b	2-32 bits	input	Input data bus
со	1	output	carry out
sum	2-32 bits	output	sum

DesignWare Library Subtractors

Figure B-2 shows an example DesignWare Subtractor symbol. In this example, "a" is the minuend, "b" is the subtrahend, "diff" is the result, and "co" is the carry out. The DesignWare subtractor is defined as "DWACT_SUB" and is available in three implementations:

- "RIPSUB" Ripple Subtractor
- "MFSUB" Medium Fast subtractor
- "FSUB" Very Fast Subtractor



Figure B-2. DesignWare Subtractor Symbol

Table B-3. Subtractor Pin Description

Pin Name	Size	Туре	Function
а	2-32 bits	input	Input data bus
b	2-32 bits	input	input data bus
со	1	output	carry out
diff	2-32 bits	output	difference

Appendix B: DesignWare Library Information

DesignWare Library Comparators

Figure B-3 shows an example DesignWare comparator symbol. In this example, "a" and "b" are the numbers that are compared. The output "<fn>" represents the greater than, less than, less than or equal to, or greater than or equal to function. The DesignWare Comparators are defined as "DWACT_CMPGT" (greater than), "DWACT_CMPLT" (less than), "DWACT_CMPLE" (greater than or equal to), or "DWACT_CMPLE" (less than or equal to). All comparators are implemented as "FCOMP."



Figure B-3. DesignWare Comparator Symbol

<i>Table B-4. Comparator Pin Description</i>	Table B-4.	Comparator	Pin	Descri	ption
--	------------	------------	-----	--------	-------

Pin Name	Size	Туре	Function
а	2-32 bits	input	Input data bus
b	2-32 bits	input	Input data bus
gt	1	output	A>B
lt	1	output	A <b< td=""></b<>
ge	1	output	A <u>≥</u> B
le	1	output	A <u>≤</u> B

DesignWare Library Counters

Figure B-4 shows an example DesignWare counter symbol. This symbol refers to both the up and down counters. The up counter is defined as "DWACT_UP_CTR." The down counter is defined as "DWACT_DN_CTR." The counter is available in two implementations: TLACNT (toggle look ahead counter) and COMPCNT (compact counter).



Figure B-4. DesignWare Counter Symbol

	Tuble D-9. Counter 1 in Description				
Pin Name	Size	Туре	Function		
data	2-32 bits	Input	Input data bus		
count	2-32 bits	Output	Output count bus		
load	1	Input	Counter, asynchronous load enable, active high		
cen	1	Input	Counter enable, active high		
reset	1	Input	Asynchronous counter reset, active low		
clk	1	Input	Clock		

Table B-5. Counter Pin Description

Table B-6. Counter Operation Truth Table

reset	load	cen	Operation
0	Х	Х	reset
1	1	Х	load
1	0	0	hold
1	0	1	down count or up count

Appendix B: DesignWare Library Information

DesignWare Library Incrementer

DesignWare Incrementers are only available for the 54SX family. Figure B-5 shows an example DesignWare incrementer symbol. In this example, "a" is the addend, "sum" is the incremented output, and "co" is the carry out. The DesignWare incrementer is defined as DWACT_DEC and is available in the FINC (fast incrementer) implementation.



Figure B-5. DesignWare Incrementer Symbol

Table B-7. In	crementer Pin	Description
---------------	---------------	-------------

Pin Name	Size	Туре	Function
а	2-32 bits	input	Input data bus
со	1	output	Carry out
sum	2-32 bits	output	sum

DesignWare Library Decrementer

DesignWare Incrementers are only available for the 54SX family. Figure B-6 shows an example DesignWare decrementer symbol. In this example, "a" is the minuend, "sum" is the decremented output, and "co" is the carry out (or in the context of decrementing, the borrow out). The DesignWare decrementer is defined as DWACT_DEC and is available in the FDEC (fast decrementer) implementation.



Figure B-6. DesignWare Decrementer Symbol

Table B-8. Decrementer Pin Description

Pin Name	Size	Туре	Function
а	2-32 bits	input	Input data bus
со	1	output	Carry out
sum	2-32 bits	output	sum

Appendix B: DesignWare Library Information

Improving Compilation Time

To decrease the synthesis compilation time, you can selectively attach a "dont_use" attribute to the DesignWare Library. If the Actel DesignWare library provides better results than the Synopsys DesignWare library, you may attach a "dont_use" attribute to the Synopsys DesignWare library using the following command:

```
dont_use standard.sldb/DW02*
dont_use standard.sldb/DW01*
```

If you wish to use the Synopsys DesignWare library, you can attach a "dont_use" attribute to the Actel DesignWare library using the following command:

read dwact.sldb
set_dont_use dwact.sldb/*

Module Count and Performance

This section describes the performance of the DesignWare Library Modules.

Adders

The fast adder offers the shortest delay. If you require fewer modules for your design, you can implement the medium fast adder. If you implement the medium fast adder, you will increase the number of logic levels. The ripple adder offers the least number of modules, but it has a module delay that is proportional to the bit width.

Figure B-7 and Figure B-8 show module count and logic levels required for a given bit for the ACT 1, ACT 2, 1200XL, ACT 3, 3200DX,

Module Count and Performance



40MX, 42MX families. Figure B-9 and Figure B-10 and show module count and logic levels required for a given bit for the 54SX family.

Figure B-7. Adder Module Count



Figure B-8. Adder Logic Level



Figure B-9. 54SX Adder Module Count

Module Count and Performance



Figure B-10. 54SX Adder Logic Level

Subtractors The fast subtractor offers the shortest delay. If you require fewer modules for your design, you can implement the medium fast subtractor. If you implement the medium fast subtractor, you will increase the number of logic levels. The ripple subtractor offers the least number of modules, but it has a module delay that is proportional to the bit width.

Figure B-11 and Figure B-12 show module count and logic levels required for a given bit for the ACT 1, ACT 2, 1200XL, ACT 3, 3200DX, 40MX, 42MX families. Figure B-13 and Figure B-14 and show module count and logic levels required for a given bit for the 54SX family.





Figure B-12. Subtractor Logic Level

Module Count and Performance



Figure B-13. 54SX Subtractor Module Count



Figure B-14. 54SX Subtractor Logic Level

Appendix B: DesignWare Library Information

Comparators Figure B-15 and Figure B-16 show the logic levels and module count required for a given bit for the ACT 1, ACT 2, 1200XL, ACT 3, 3200DX, 40MX, 42MX families. No information is currently available for the 54SX family.



Figure B-15. Comparator Module Count



Figure B-16. Comparator Logic Levels

Counters

Figure B-17 and Figure B-18 show the logic levels and module count required for a given bit for the ACT 1, ACT 2, 1200XL, ACT 3, 3200DX, 40MX, 42MX families. No information is currently available for the 54SX family.



Figure B-17. Counter Module Count



Figure B-18. Counter Logic Levels

Appendix B: DesignWare Library Information

Incrementer DesignWare Incrementers are only available for the 54SX family. Figure B-19 and Figure B-20 show module count and logic levels required for a given bit.



Figure B-19. Incrementer Module Count



Figure B-20. Incrementer Logic Levels

Decrementer

DesignWare Decrementers are only available for the 54SX family. Figure B-21 and Figure B-22 show module count and logic levels required for a given bit.



Figure B-21. Decrementer Module Count



Figure B-22. Decrementer Logic Levels

С

Common Problems

This appendix describes problems that may occur during design and synthesis, and solutions to the problems. This includes library error messages, problems when inferring DesignWare, using internal tristates, and problems that may occur during multiplexer inferencing.

Library Errors

This section describes error messages that are displayed when there are errors with the Actel synthesis libraries.

Problem: The library path is incorrect or the library does not exist and the following error message is displayed:

Unable to resolve reference

Solution: Make sure the library references in your ".synopsys_dc.setup" script are correct.

DWACT Component

Unable to

Reference

Resolve

Problem: The target library has not been regenerated for the Synopsys version you are using and the following error message is displayed:

The package 'DWACT_components depends on the package 'std_logic_1164' which has been analyzed more recently. Please re-analyze the source file for 'DWACT_components' and try again (LBR-28)

Solution: Reanalyze the DesignWare library for the current Synopsys version. Refer to "Reanalyzing DesignWare Libraries" on page 2 for regeneration procedures.

Appendix C: Common Problems

Inferring DesignWare

Problem: Synopsys does not infer a DWACT FADD with Cin to implement SUM = A + B + Cin. It uses 2 ripple adders, one adder to implement X = A + B, and another adder to implement SUM = X + Cin.

Solution: The design has not been constrained, hence Synopsys is using a ripple implementation. Apply a max delay constraint to the adder using the following command:

```
set_max_delay 5 -from all_inputs() -to all_outputs()
```

You can also set the following switch in your .synopsys_dc.setup

hdlin_use_cin = true

Internal Tri-State

Problem: The antifuse architecture does not support internal tri-states. The following message is displayed when you have described the internal tri-states in HDL.

Mapping to **TSGEN**

Solution: Modify your HDL to use multiplexor logic.

Multiplexer Inferencing

This section describes problems that may occur during multiplexer inferencing.

Compile Switches *Problem:* Setting the "compile_create_mux_op_hierarchy" switch to "false" eliminates the MUX_OP hierarchy in the design as well as removing the MX4 mapping and uses random logic to build the MUX structure. This random logic mapping is inefficient for area and timing. Why can't Synopsys just remove the MUX_OP hierarchy and not touch the MX4 mapping.

Cause: MUX_OPs are created when the infer_mux attribute is set to a full case statement in the HDL code. For each full case statement with infer_mux, a generic MUX_OP design is created. If compile_create_mux_op_hierarchy is true, Design Compiler maps the MUX_OP to a tree of multiplexers. Keeping the hierarchy is necessary so that the multiplexer information is preserved for the next compile. Once the multiplexer hierarchy is removed, i.e. setting "compile_create_mux_op_hierarchy" switch to "false" all traces of the multiplexer tree are gone and the next pass of compile could generate a less optimal design.

Solution: By default "compile_create_mux_op_hierarchy" switch is always set to "true." Never set it to "false." If you want to eliminate the MUX_OP from the hierarchy, ungroup the MUX_OP hierarchy levels and then save the design. Do not re-run compile after ungrouping the MUX_OP hierarchy level.

Appendix C: Common Problems

Inefficient Optimization	<i>Problem:</i> When the input signals or wires driving the inputs of muxes are tied to logic 0 or 1, Synopsys does constant pushing and no longer maps to the MX4 basic cell. The random logic implementation is not efficient.		
	<i>Cause:</i> Since Synopsys does boundary optimization by default, Synopsys will map to discrete gates when it sees muxes driven by constants.		
	<i>Solution:</i> Turn off boundary optimization for multiplexors by adding the following switch in the .synopsys_dc.setup (actsetup.scr) file:		
	compile_mux_no_boundary_optimization = "true"		
	If the design has input ports that are tied to logic 0 or 1, you should remove port from the port list and use signals or wires to tie in the constants.		
EDIF Netlist Errors	<i>Problem:</i> The EDIF netlist uses both angle and square brackets for buses. How can one force Synopsys to use only angle brackets.		
	<i>Solution:</i> Set the following switch in your ".synopsys_dc.setup" or "actsetup.scr" file:		
	<pre>bus_dimension_separator_style ="><"</pre>		
	With this setup you will get an EDIF netlist with the correct naming style. For example:		
	(port (rename CFG_ADDR_MIN1_ "CFG <addr_min><1>") (direction INPUT))</addr_min>		
	instead of		
	(port (rename CFG_ADDR_MIN1_ "CFG[ADDR_MIN][1]") (direction INPUT))		

Product Support

Actel backs its products with various support services including Customer Service, a Customer Applications Center, a fax back service, a Web and FTP site, electronic mail, and worldwide sales offices. This appendix contains information about using these services and contacting Actel for service and support.

Actel U.S. Toll-Free Line

Use the Actel toll-free line to contact Actel for sales information, technical support, requests for literature about Actel and Actel products, Customer Service, investor information, and using the Action Facts service.

The Actel Toll-Free Line is (888) 99-ACTEL.

Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

From Northeast and North Central U.S.A., call (408) 522-4480. From Southeast and Southwest U.S.A., call (408) 522-4480. From South Central U.S.A., call (408) 522-4474. From Northwest U.S.A., call (408) 522-4434. From Canada, call (408) 522-4480. From Europe, call (408) 522-4474 or +44 (0) 1256 305600. From Japan, call (408) 522-4252. From the rest of the world, call (408) 522-4252. Fax, from anywhere in the world (408) 522-8044.

Customer Applications Center

The Customer Applications Center is staffed by applications engineers who can answer your hardware, software, and design questions.

All calls are answered by our Technical Message Center. The center retrieves information, such as your name, company name, phone number and your question, and then issues a case number. The Center then forwards the information to a queue where the first available application engineer receives the data and returns your call. The phone hours are from 7:30 a.m. to 5 p.m., Pacific Standard Time, Monday through Friday.

The Customer Applications Center number is (800) 262-1060.

European customers can call +44 (0) 1256 305600.

Guru Automated Technical Support

Guru is a Web based automated technical support system accessible through the Actel home page (http://www.actel.com). Guru provides answers to technical questions about Actel products. Many answers include diagrams, illustrations and links to other resources on the Actel Web site. Guru is available 24 hours a day, seven days a week.

Action Facts

Action Facts is a 24-hour fax-back service that allows you to have technical literature faxed directly to you. The literature includes current applications notes, device diagrams, radiation testing reports, CAErelated documents, training release schedules, and much more.

To receive your free copy of the catalog or any of our literature, contact us at (800) 262-1062 or (408) 739-1588.

Web Site

Actel has a World Wide Web home page where you can browse a variety of technical and non-technical information. Use a Net browser (Netscape recommended) to access Actel's home page.

The URL is http://www.actel.com. You are welcome to share the resources we have provided on the net.

Be sure to visit the "Actel User Area" on our Web site, which contains information regarding: products, technical services, current manuals, and release notes.

HLD Web Site

Actel also has an HLD web site at http://www.actel.com/hld/ that you can browse for HLD design information. The site provides detailed information and examples on coding styles and synthesis and simulation flow, as well as links to our synthesis and simulation partners.

FTP Site

Actel has an anonymous FTP site located at ftp://ftp.actel.com. You can directly obtain library updates, software patches, design files, and data sheets.

Electronic Mail

You can communicate your technical questions to our e-mail address and receive answers back by e-mail, fax, or phone. Also, if you have design problems, you can e-mail your design files to receive assistance. The e-mail account is monitored several times per day.

The technical support e-mail address is tech@actel.com.

Worldwide Sales Offices

Actel Corporation 955 East Arques Avenue Sunnyvale, California 94086 USA Toll Free: 888.99.ACTEL Tel: 408.739.1010 Fax: 408.739.1540

California (North) Tel: 408.328.2200 Fax: 408.328.2358

California (South) Tel: 714.727.0470 Fax: 714.727.0476

Georgia Tel: 770.844.4495 Fax: 770.844.1395

Illinois Tel: 847.259.1501 Fax: 847.259.1572

Massachusetts

Tel: 508.635.0010 Fax: 508.635.9510

Texas Tel: 972.235.8944 Fax: 972.235.9659 Actel Europe, Ltd. Daneshill House, Lutyens Close Basingstoke, Hampshire RG24 8AG United Kingdom Tel: +44 (0)1256.305600 Fax: +44 (0)1256.355420

France Tel: +33 (0)1.40.83.11.00 Fax: +33 (0)1.40.94.11.04

Germany Tel: +49 (0)8165.9584.0 Fax: +49 (0)8165.9584.10

Italy Tel: +39 (0)2.38.09.32.59 Fax: +39 (0)2.38.09.32.60 Actel Asia-Pacific EXOS Ebisu Building 4F 1-24-14 Ebisu Shibuya-ku Tokyo 150 Japan Tel: +81 (0)3.3445.7671 Fax: +81 (0)3.3445.7668

Korea Tel: +82 (0)2.555.7425 Fax: +82 (0)2.555.5779

Taipei, Taiwan R.O.C. Tel: +886 (0)2.698.2448 ext. 104 Fax: +886 (0)2.698.2548

(Q)CLKINT 64

.synopsys_dc.setup File 3, 5, 77, 84 <act_fam> variable xiii

Α

Accessing Synthesis Libraries 5 ACT 3 I/O Macros Automatic Synthesis 78–79 Inference 79 Synthesis 72-79 Synthesis Notes 80 act3io.script 76 Actel Device Families xiii FTP Site 121 Web Based Technical Support 120 Web Site 121 ACTgen dont_touch attribute 84 Generating a Macro 81 Instantiating in Verilog 81 Instantiating in VHDL 83 Instantiation Recommendations 81 Macros 80 Action Facts 120 ACTmap 70 actsetup.scr File 3, 5 Adder, DesignWare 32 Assigning Pins 69 Assumptions xii Attributes, Removing 64 Automatic ACT 3 I/O insertion 76 Finite State Machine Encoding 30 Avoiding Cells 62

В

Balancing Registers 62 Limitations 64 Behavioral Simulation 8 BIBUF 60 Bottom-Up Compile 56 Buffering on a Reset Network 52 Bus Array Syntax 70

С

Capturing a Design 8 Case, Under Utilized 21 Cells Avoiding 62 MUX OP 11 CLKBUF 60 Clock Buffer 87 Inference 60 Coding DesignWare Adder 32 DesignWare Comparator 39 DesignWare Counter 41 DesignWare Incrementer 43 DesignWare Subtractor 36 Technology Independent 86 Command Control Flow 72 dc shell status 72 foreach 72 **Common Problems** DesignWare 116 Internal Tri-State 116 Library Errors 115 Multiplexer Inferencing 117 Comparator, DesignWare 39 **Compilation Variables 12**

compile_create_mux_op_hierarchy 12 compile_mux_no_boundary_optimization 13 Compile Options 54 Complex ACT 3 I/O Mapping 72-80 ACT 3 I/O Synthesis 73 ACT 3 I/O Synthesis Notes 80 Automatic I/O Synthesis 78–79 Inferring a Sequential Cell 79 Constraining a Design 51-65 (Q)CLKINT 64 Area 55 Avoiding Cells 62 Buffering on a Reset Network 52 Clock Buffers 60 **Compile Options 54** Delays 52 Fanout 61 Flattening 53 Flattening Hierarchy 58 Hard Clock Buffers 60 Hierarchy 55-59 I/O Buffers 60 Inferring Buffers 60 Input Delays 52 Internal Tri-State 59 Logic Levels 52 Maintaining Hierarchy 55 Maintaining Structure 53 **Output Delays 52 Register Balancing 62** Register Types 62 Removing Attributes 64 Removing Designs from Hierarchy 59 Reporting Cells in Hierarchy 59 **Contacting Actel** Customer Service 119

Electronic Mail 121 Technical Support 120 Toll-Free 119 Web Based Technical Support 120 Conventions xii <act_fam> variable xiii Naming, Verilog xiv Naming, VHDL xiii Counter, DesignWare 41 Customer Service 119

D

dc_shell_status 72 dcf File 69 Delays 52 Input 52 Output 52 Design Hierarchy 55-59 Layout 9 **Optimization** 70 Synthesis 8 Translating 68 Design Constraint 51-65, 69 (Q)CLKINT 64 Area 55 balance_registers 62 Buffering on Reset Networks 52 Clock Buffers 60 **Clock Constraint 52 Compile Options 54** Delays 52 dont_use 64 Flattening Hierarchy 58 Hard Clock Buffers 60 Hierarchy 55-59

I/O Buffers 60 Inferring Buffers 60 Internal Tri-State 59 Logic Level 52 Maintaining Hierarchy 55 **Operating Condition 51** Removing Designs from Hierarchy 59 Reporting Cells in Hierarchy 59 set_dont_use 62 set_flatten 53 set_input_delay 52 set_output_delay 52 set_register_type 62 set structure 53 set-max-fanout 61 **Design Creation/Verification 8 Behavioral Simulation 8 EDIF Netlist Generation 9** HDL Source Entry 8 Structural Netlist Generation 9 Structural Simulation 9 Synthesis 8 Design Flow 8-10 **Design Creation/Verification 8 Design Implementation 9** Programming 10 System Verification 10 **Design Implementation 9** Place and Route 9 **Timing Analysis 9 Timing Simulation 10** Designer DirectTime 69 DT Analyze Tool 9 Manuals xv Place and Route 9

Script Mode Place and Route 71 Software Installation Directory 1 Timing Analysis 9 DesignWare 67 Adder 32, 98 Coding 31-49 Comparator 39, 100 Compilation Time 104 Compiling Designs 67 Counter 41, 101 Errors 116 Incrementer 43 Instantiation 41 Libraries 2 Setup 3 Subtractor 36, 99 DesignWare Libraries 97-113 Accessing 3 Reanalyzing 2 DesignWare Module Count and Performance 104-113 Adder 104, 107 Comparator 110 Counter 111 Subtractor 104, 107 Device Debugging 10 Families xiii Programming 10 Document Assumptions xii Conventions xii Organization xi DT Analyze 9 **Dual Architecture Coding 86**

Ε

EDIF Netlist Generation 9, 84 edn2vhdl 85 Electronic Mail 121 Extracting a FSM from a Sequential Design 28

F

Fanout 61 Limit 87 Networks 64 File .synopsys_dc.setup 3, 5, 77, 84 actsetup.scr 3, 5 dcf 69 Finite State Machine (FSM) Design 24-31 Automatic Encoding 30 Extracting from a Sequential Design 28 Mealv 24 Moore 30 Multiple Results 30 One-Hot Encoding 30 Power On and Reset 30 Flattening Designs 53 Hierarchy 58 foreach statement 72

G

Gate-Level Netlist 8 Generating ACTgen Macros 81 EDIF Netlist 9, 84 Gate-Level Netlist 8 Structural Netlist 9, 85

Η

Hard Clock Buffer Inference 60 HCLK 60 HDL Source Entry 8 Hierarchy 55–59 Bottom-Up Methodology 56 Characterize Methodology 57 Flattening 58 Maintaining 55 Removing Designs 59 Reporting 59 Time-Budget Methodology 58 Top-Down Methodology 55 High Fanout Networks 64 HLD Web Site 121

1

I/O Buffer Inference 60 I/O Mapping 72 **INBUF 60** Incrementer, DesignWare 43 Inference Variables 11 hdlin 12 Inferring Buffers 60 Multipelxers 13 Inferring Buffers Clock 60 Hard Clock 60 I/O 60 Installation Directory Designer 1 Synopsys 1 Instantiating ACTgen Macros 80 Instantiating DesignWare Counters 41
Internal Tri-State 59 Errors 116

Κ

Keywords Verilog xiv VHDL xiii

L

Libraries Verifying Version 5 Library DesignWare 2, 97–113 Errors 115 Setup 3, 5 Synthesis 5, 87–95 Logic Level Constraints 52

Μ

Macros (Q)CLKINT 64 ACT 3 I/Os 92 **BIBUF 60** CLKBUF 60 Complex ACT 3 I/O Mapping 72-80 HCLK 60 **INBUF 60 OUTBUF 60 TRIBUF 60** Maintaining Hierarchy 55 Bottom-Up Methodology 56 Characterize Methodology 57 Time-Budget Methodology 58 Top-Down Methodology 55 Maintaining Structure 53 Mapping I/Os 72

Maximum Fanout 61 Moore State Machine 30 Multiple Results 30 Multiplexer Encoding 11–23 Compilation Variables 12 compile_create_mux_op_hierarchy 12 compile_mux_no_boundary_optimization 13 Errors 117 hdlin Variable 12 Inference Variables 11 Inferencing 13 Registered Multiplexer 18 Under Utilized Case Statements 21 Wide Multiplexer 15 MUX_OP Cell 11

Ν

Naming Conventions Verilog xiv VHDL xiii Netlist Generation EDIF 9, 84 Gate-Level 8 Structural 9, 85 Networks, High Fanout 64

0

On-Line Documentation xv On-Line Help xviii Operating Conditions Setting 51 Synthesis Libraries 95 Optimizing a Design with ACTmap 70 OUTBUF 60

Index

Ρ

Pins, Assigning 69 Place and Route 9 DirectTime 69 Script Mode 71 Power On and Reset 30 Product Support 119-121 Action Facts 120 **Customer Applications Center 120** Customer Service 119 Electronic Mail 121 FTP Site 121 HLD Web Site 121 **Technical Support 120** Toll-Free Line 119 Web Site 121 Programming a Device 10

R

Reanalyzing DesignWare Libraries 2 Register Balancing 62 Limitations 64 Register Types 62 Registered Multiplexer 18 Related Manuals xv Removing Attributes 64 Designs from Hierarchy 59 Reporting Hierarchy 59 Required Software 1

S

Script act3io.script 76 Place and Route 71 read_array_naming_style 70 Setting **Operating Conditions 51** Setting Environment Variables 1 Setup Procedures Setting **Environment Variables 1** System Setup 1 User Setup 2-5 Simulation Behavioral 8 Structural 9 Timing 10 Software Requirements 1 State Machine Design 24-31 Automatic Encoding 30 Extracting from a Sequential Design 28 Mealy 24 Moore 30 Multiple Results 30 One-Hot Encoding 30 Power On and Reset 30 Static Timing Analysis 9 String Variables 72 Structural Netlist Generation 9, 85 edn2vhdl 85 Structural Simulation 9 Structure 53 Subtractor, DesignWare 36 synop2dcf 69 Synopsys, Software Installation Directory 1 Syntax 70 Synthesis 8 Synthesis Libraries 87-95 Accessing 5 ACT 3 I/Os 92 Fanout Limit 87

Index

Operating Conditions 95 Setup 5 Timing Constraints 87 Synthesizing ACT 3 I/Os 73 System Requirements 1 System Setup 1 System Verification 10 Silicon Explorer 10

Т

Technical Support 120 Technology Independent Coding 86 Time-Budget Compile 58 Timing Analysis 9 Timing Constraints 69 dcf file 69 synop2dcf 69 Synthesis Libraries 87 **Timing Simulation 10** Toll-Free Line 119 Top-Down Compile 55 Translating Designs 68 Timing Constraints into Designer 69 Translating Timing into Designer 69 **TRIBUF 60** TroubleShooting DesignWare 116 Internal Tri-State 116 Library Errors 115 Multiplexer Inferencing 117

U

Under Utilized Case Statement 21 Unit Delays 8 User Setup 2–5 Using (Q)CLKINT 64

V

variable, <act_fam> xiii Verilog Naming Conventions xiv Reserved Words xiv VHDL Naming Conventions xiii Reserved Words xiii

W

Web Based Technical Support 120 Wide Multiplexer 15