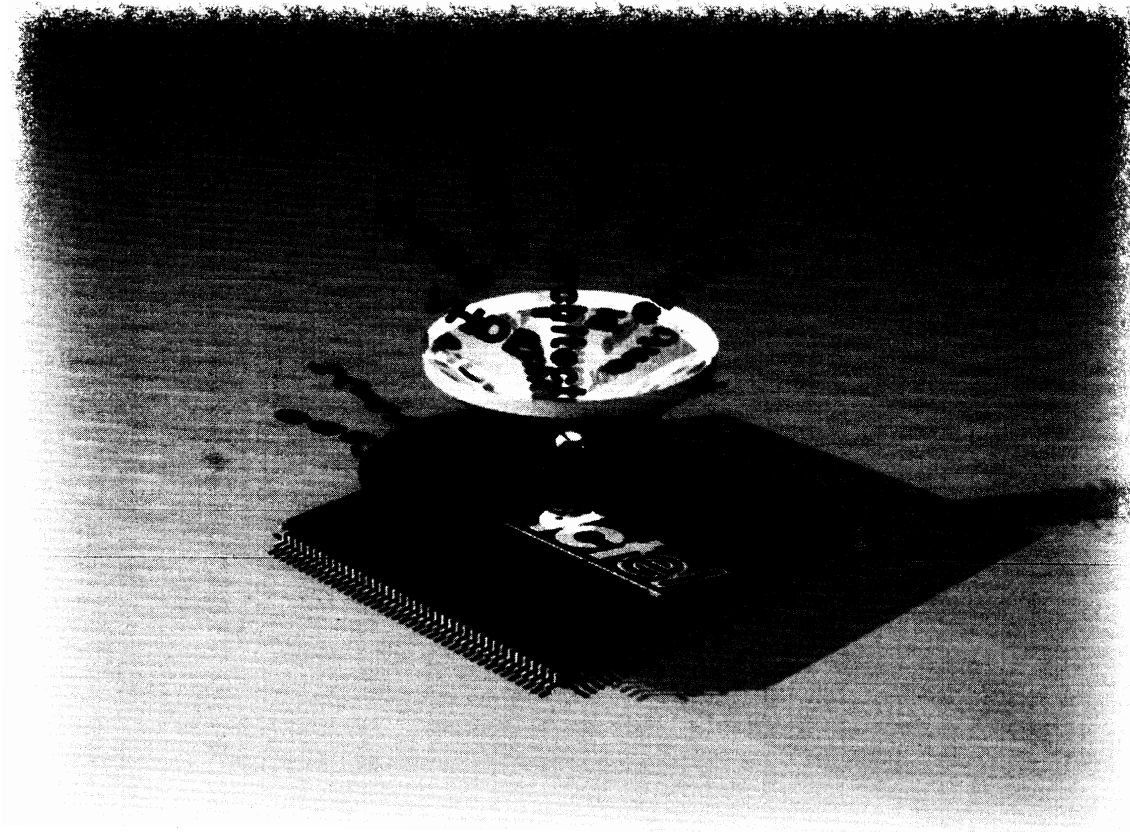
Synopsys VHDL Methodology Handbook

A Book of *Hints and Tips*



SYNOPSYS

Actel/Synopsys VHDL Methodology Handbook

A Book of Hints and Tips

Actel Corporation, Sunnyvale, CA 94086
© 1995 Actel Corporation. All rights reserved.

Part Number: 5172029-0 February 1995

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

Actel Corporation 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 of Actel Corporation.

Trademarks

The Actel logo, Action Logic, ACT, ACTmap, and ACTgen are trademarks or registered trademarks of Actel Corporation.

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

Table of Contents

1 Actel/Synopsys VHDL Methodology Handbook Overview 1
Common Problems Addressed 2
Methodology Handbook Stasis 2
Prerequisites 3
Synthesis and Actel 3
In This Manual 4

2 Summary of Hints and Tips 5
Simulate Before Place and Route 5
Know the Device/Family Resources 5
Use Scripts Properly 6
Use Design Constraints 6
Using Actel's TDPR 6
Use "Case" When Possible 7
Use ACTgen for Structured Macros 7

3 Overview of Actel Device Families. 9
ACT 1 Family 9
ACT 1 Logic Resources 10
ACT 1 I/O Resources 12
ACT 1 Special Resources 12
ACT 2 Family 13
ACT 2 Logic Resources 13
ACT 2 I/O Resources 15
ACT 2 Special Resources 16

ACT 3 Family 17

ACT 3 Logic Resources 17

ACT 3 I/O Resources 18

ACT 3 Special Resources 20

Mapping and Combinability 20

Speed Grades and Timing Characteristics 22

4 Tips Using Synopsys Design Compiler. 23

Setting Proper Design Constraints 23

Grouping Paths for Maximum Delay 26

Setting the Area Constraint 26

Why Set Constraints 27

Effectively Using Scripts 27

Effectively Using Hierarchy 28

Characterizing the Subdesigns to the Parent. 28

Using Global Signals and Special Resources. 29

Using ACT 3 I/O registers 29

Running ALS from Within Design Compiler 31

Verifying Synopsys Performance with ALS 32

5 Hints On Designing with Synopsys VHDL. 33

General State Machines Description 33

State Machine Design Hints 35

State Encoding 37

Extracting an FSM from a Sequential Design 37

Read Design 39

Map Design 39

Group FSM 39

Extract FSM. 40

Optional FSM Optimization Styles 41

Using One-Hot Encoding Strategy 42

Manual One Hot State Encoding 44

Automatic FSM Encoding Styles 44
Power-up And Reset 45
"If-then-else" Versus "Case" 46
Arithmetic Elements And DesignWare 3.1+ 46
Designware 3.1 Performance (Post Routed) 47

6 Component Instantiation 49
Basic Component Instantiation 49
Technology Independent Component Instantiation 50
Actel Library Cell Instantiation 50
Using "ACTgen" Macros In Your Design 51
ACTgen/Synopsys Design Flow 51
I/O Instantiation 53

7 Most Frequently Asked Questions 55

1

Actel/Synopsys VHDL Methodology Handbook Overview

VHDL is a high level description language for system and circuit design. The language supports various levels of abstraction. At higher levels of abstraction the user can conceptually design a system without regard to a specific technology. Traditionally, only when the system (design) is functioning and validated through high level simulation does a designer need to consider a specific target technology (Figure 1).

Note The code is translated to RTL code- not to be confused with behavioral code. It is the desire of designers to keep technology specific code to a minimum. In this way, a designer may migrate from one technology to another while keeping source code changes to a minimum.

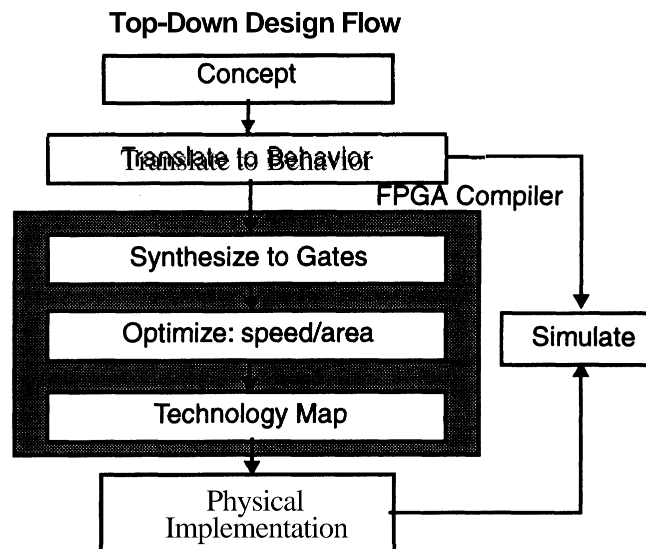


Figure 1. Top-Down Design Flow

In general, designers write their code very generic and hope that the design tools will ultimately make smart technology specific choices. This method works, but to get the most optimum performance (read: high speed) from your devices some things must be considered. It is those considerations and trade-offs that will be covered in this manual. It is also the focus of this manual to provide the designer with enough insight (hints) to be able to make trade-offs when coding their system (or design). A selection of trade-offs will be listed. A designer, depending on how much they're pushing the density or performance envelopes, may choose to employ none, some, or all of them. Some considerations are subtle, some obvious, some technology specific, and some technology independent.

Common Problems Addressed

The classic problem that we, at Actel, see is a designer debugging Actel devices before the technology *independent* simulations are complete. If your simulation strategy does not support this methodology this manual will try and lessen the cycle time. This will be achieved by recognizing the special resources available in each device family and properly using them.

Methodology Handbook Stasis

This document is meant to be a "living" document and by no means the "know-all" repository of all Field Programmable Gate Arrays (FPGA) specific hints. If you have examples and guidelines that have worked for you please pass them on to Actel by e-mailing them to tech@actel.com or FAXing them to (408) 739-1540.

Prerequisites

This manual is NOT a VHDL tutorial! Knowledge of the basic constructs of VHDL is assumed. If you need more details, training in VHDL and top-down design in general is available from a number of different sources. Even though knowledge of the Actel, Action Logic System (ALS), design suite is recommended (but not required) be sure to have an Actel Data Book and Macro Library Manual handy.

Also required is access to Synopsys Design Compiler Version. 3.1+ with Actel Technology Libraries (Version. 3.1).

Note Some enhanced features available with FPGA Compiler are NOT available in Design Compiler. These features can (and will) affect maximum density and performance. If you have Design Compiler it may not be necessary to obtain a license for FPGA Compiler if the results obtained meet the design constraints. Also, throughout this manual Design Compiler is used to denote the complete set of Synopsys tools (Design Analyzer, FPGA Compiler, VHDL Compiler, etc.)

Synthesis and Actel

In general, technology mappers want one thing—small homogenous building blocks. It is for that reason that most ASICs are mapped reasonably well when it comes to speed and density. The basic building block is typically an equivalent of a 2 input NAND gate. Also, the ASIC interconnect is a metal-to-metal "via" that doesn't represent a significant amount of the circuit delay. Therefore the mapper can produce less than optimal solutions and the ASIC technology will be much more forgiving. FPGAs, on the other hand, are much less forgiving. Yes, it is true that the tools have dramatically improved recently, but an FPGA design that pushes either the speed or density envelope will require more consideration.

Actel, unlike any other leading FPGA manufacturer, makes devices that have small homogeneous building blocks. This makes for easier coding and fewer technology specific considerations. Ultimately you want the synthesis tool to achieve near the schematic performance with the ease of high-level design. With Actel it will be shown that this is possible.

In This Manual

Chapter 2: Summary of Hints and Tips covered in this manual.

Chapter 3: Actel device architectures is discussed here to familiarize the user with the resources available.

Chapter 4: Covers tips on using Synopsys Design Compiler.

Chapter 5: Some VHDL constructs and styles will be covered.

Chapter 6: Covers basic cell instantiation.

Chapter 7: A compilation of Actel's most commonly asked questions about the Synopsys/Actel technology kit.

2

Summary of Hints and Tips

Below is a summary of the hints and tips covered in this manual. To see more detail on each hint/tip please refer to the associated chapter.

Simulate Before Place and Route

In Chapter 1, high-level technology independent simulation was discussed (see Figure 1). In general, the cycle time from EDIF netlist, place/route/timing-extract to back-annotated timing can be anywhere from thirty minutes to two hours. It is our experience, here at Actel, that designers spend many unnecessary hours in this loop before the design has been "wrung-out". We believe that only after the designer has successfully proven the functionality should the device be mapped. As stated earlier, if the designer's current simulation strategy does not support this then this manual will try to lessen the cycle time by providing helpful hints before the design starts.

Know the Device/Family Resources

Another observation by the team, here at Actel, is the improper usage of the special resources available in each family (refer to Chapter 3). It is through these resources that designers can realize fast, compact, and reliable designs. See Chapter 4 and Chapter 6 for more details.

Use Scripts Properly

Another common mistake is not using Design Compiler's scripts properly. It will be shown in Chapter 5 that without modifying the source VHDL code the state machine order, encoding style and optimization parameters can be modified.

Use Design Constraints

As discussed in Chapter 4, it is most important that Design Compiler knows the design goals. Also, remember it is just as important to identify what is *not* critical as well as what is.

Using Actel's TDPR

In the near future, Actel will be releasing a Timing Driven Place and Route (TDPR) tool. Synopsys design constraints will automatically be converted from Synopsys format to Actel format. It makes it even more imperative to use proper, complete design constraints.

Use "Case" When Possible

In Chapter 5, it will be suggested that the designer use `case` statements instead if long nested `if-then-else`. The reasoning is two-fold;

1. Case statements are compared to a single vector. In general, this vector can be thought of as the "select" inputs to a multiplexer. Actel FPGAs are built on multiplexer technology (see Figure 3 and Figure 5). It is a natural fit to use muxes when possible.
2. Case statements force structure to complex decoding situations. `If-then-alee's` do allow maximum flexibility but they may not produce compact logic. It has been our experience, here at Actel, that on numerous occasions (incorrectly coded) complex `if-then-else` statements were the cause of some logic (state-machine) problems.

Use ACTgen for Structured Macros

In general, ACTgen uses the Actel technology the most efficiently. It was this technology that was used to construct the DesignWare libraries (Chapter 5). But, for macros not automatically inferred by DesignWare ACTgen will should be used as the designer's most effective path to high-density, high-speed design.

3

Overview of *Actel* Device Families

Currently, Actel FPGAs are antifuse based. Actel is the home of antifuse technology. Technically the type of interconnect technology should not matter to the synthesis user. This would be true if the type of interconnect did not dictate the architecture (which is of some concern to the synthesis user). Physically antifuses are a two-terminal via type device. They are small (they reside underneath the routing lines) and have minimal impedance. The small area and low impedance allows Actel to build devices with a structure similar to channeled gate arrays (lots of small "simple" building blocks with an abundance of routing). This in turn allows the synthesis user to be less concerned with device architecture and more concerned with coding hints and resource types available.

The following data is covered in detail in the Actel data books. The intent of this section is to familiarize the VHDL designer with Actel specific features and resources of each device family.

ACT 1 Family

The ACT 1 family represents Actel's first generation of FPGAs. Currently, there are two devices available (Table 1). An ACT 1 device is structured much like a channeled gate array (Figure 2). Logic modules (logic resources) are distributed in an X,Y array surrounded by I/O modules (I/O resources). All logic and I/O resources are identical in structure. The only exception is the clock module.

Table 1. ACT 1 Family Profile

Device	Gates	Module	Registers	User I/O
A1010	1200	295	147	57
A1020	2000	547	273	69

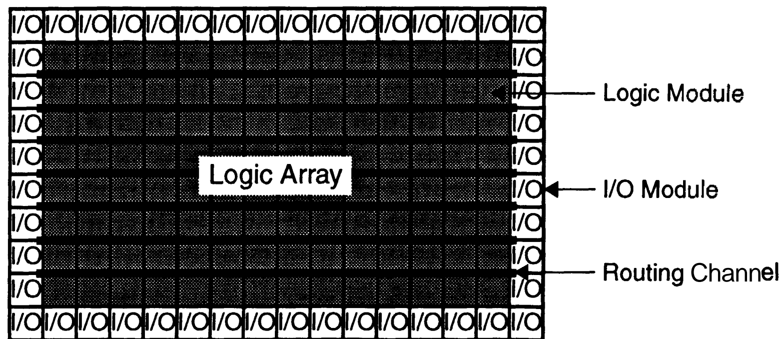


Figure 2. Channeled Gate Array Overview

ACT 1 Logic Resources

The ACT 1 logic module is an 8-input, one-output logic circuit chosen for the wide range of functions it implements and for it's efficient use of interconnect routing resources (Figure **3).**

This logic module can implement the four basic logic functions (NAND, AND, OR, and NOR) in gates of two, three, or four inputs. The logic module can also implement a variety of D-latches, exclusivity functions and complex AND-ORs type functions. No dedicated registers are available in the array. Instead registers are built with two latches configured as a master-slave (hence the register count is half that of the module count—Table 1).

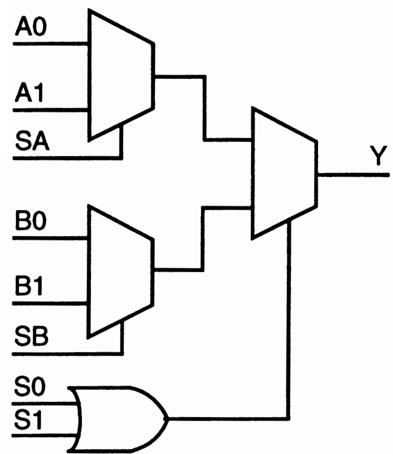


Figure 3. ACT 1 Logic Module (Logic Resource)

ACT 1 I/O Resources

Each I/O pin is available as an input, output, three-state or bi-directional buffer (Figure 4). Input and output levels are compatible with standard TTL and CMOS specifications. See Electrical Specifications in the Actel data book for additional UO buffer specifications.

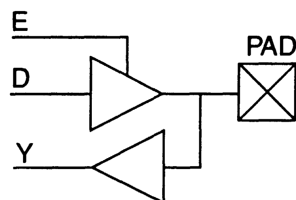


Figure 4. ACT 1 I/O module

ACT 1 Special Resources

Each ACT 1 type device has a single high drive, low skew clock buffer (CLKBUF). This is treated as an I/O module that can drive many internal loads. This resource may derive it's source from within the array. When used in this manner an I/O pin must be used in conjunction with a special resource (CLKBIBUF). This is essentially a bi-directional buffer (Figure 4) with the output (Y) connected to the clock network. The regular clock buffer (CLKBUF) is the same device with the D and E inputs disabled.

ACT 2 Family

The ACT 2 family represents Actel's second generation of FPGAs. Currently there are three devices available (Table 2). Like ACT 1, an ACT 2 device is structured much like a channeled gate array. Logic modules (logic resources) are distributed in an X,Y array surrounded by I/O modules (I/O resources). The ACT 2 family's logic resources come in two styles. The I/O have been enhanced with bi-directional latches. There are two clock networks on each device of the ACT 2 family.

Table 2. ACT2 Family Profile

Device	Gates	Module	Registers	User I/O
A1225	2500	451	>231	83
A1240	4000	684	>348	104
A1280	8000	1232	>624	140

ACT 2 Logic Resources

The ACT 2 logic resources are classified into two types: combinatorial (C-modules) and sequential (S-modules). The ACT 2 C-module (Figure 5) is an enhanced version of the ACT 1 style module and can now implement up to 5 input functions.

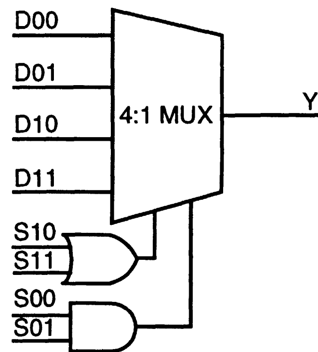


Figure 5. ACT 2 C-module (Logic Resource)

The S-module (Figure 6) is designed to implement high-speed register functions within a single logic resource. The S-module is configured as a C-module followed by a storage element, either a D-type register or latch. The storage element can be disabled by configuring the storage element as a latch and permanently enabling it. Likewise, the combinatorial logic can be disabled by connecting the select inputs to zero and bringing data in through a mux input. Notice the CLR and S01 inputs share a pin. This affects combinability.

Synopsys Design Compiler takes full advantage of the S-module. This is essential for fast, compact designs.

Note The S-module storage element's clear input is ***active low only!***

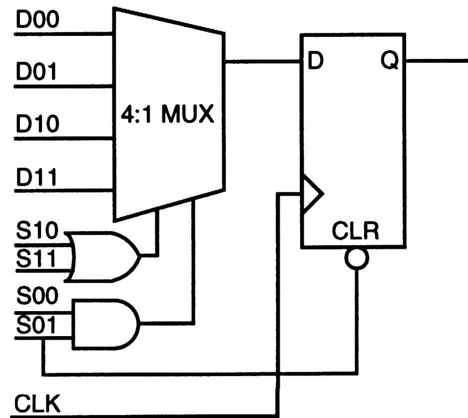


Figure 6. ACT 2 S-module Implementations (Logic Resource)

ACT 2 I/O Resources

The ACT 2 I/O resource is an enhanced version of the ACT 1 I/O module (Figure 7). In addition to each I/O pin as an input, output, three-state or bi-directional buffer a latch is available on both input and output. This latch can be combined with a second latch within the array to construct registered inputs and outputs (for faster input set-up and clock-to-out times). Input and output levels are compatible with standard TTL and CMOS specifications. See Electrical Specifications in the Actel data book for additional I/O buffer specifications.

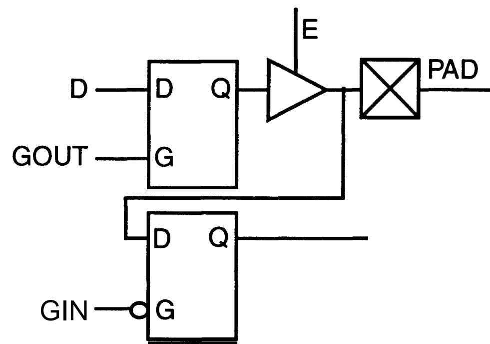


*Figure 7. **ACT 2** I/O module*

ACT 2 Special Resources

Each ACT 2 type device has two high drive, low skew clock buffers. There are treated as I/O modules that can drive many internal loads. Unlike the ACT 1 version, the ACT 2 clock buffers need not use an external I/O pin when driving an internal load. The internal clock (CLKINT) is used in this case.

ACT 3 Family

The ACT 3 family represents Actel's third generation of high performance FPGAs. Currently there are five devices available (Table 3). Like ACT 1 and ACT 2, an ACT 3 device is structured much like a channeled gate array. Logic modules (logic resources) are distributed in an X,Y array surrounded by I/O modules (I/O resources). Like ACT 2, ACT 3 family's logic resources come in two styles. The I/O have been enhanced with bi-directional registers with a dedicated I/O clock for fast Clock-to-Out times (<10ns) and low input set-up times. A special high-speed clock network (HCLK) has been added in addition to the two clock networks identical to the ACT 2 family.

Table 3. ACT3 Family Profile

Device	Gates	Module	Registers	User I/O
A1415	1500	200	>104	80
A1425	2500	310	>160	100
A1440	4000	564	>288	140
A1460	6000	848	>432	168
A14100	10000	1377	>697	228

ACT 3 Logic Resources

Like ACT 2, the ACT 3 logic resources are also classified into two types: combinatorial (C-modules) and sequential (S-modules). The C-module is exactly like the ACT 2 C-module (Figure 5). The ACT 3 S-module (Figure 8) is an improved version of the ACT 2 S-module.

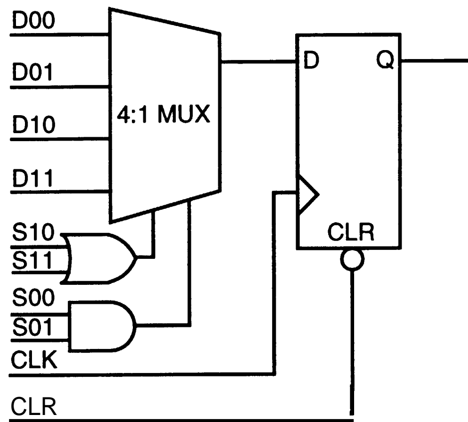


Figure 8. ACT 3 S-module Implementations (Logic Resource)

The ACT 3 S-module has a separate CLR input. This allows all combinatorial functions (C-modules) that are followed by a simple data storage element to be combinable.

ACT 3 I/O Resources

Like ACT 2, ACT 3 I/Os perform simple buffer configurations. In addition, a high-speed register is available on both input and output for very fast input set-up and clock-to-out times (Figure 9). The data input may be derived from the output register output or directly from the pad (Figure 10). Input and output levels are compatible with standard TTL and CMOS specifications. See Electrical Specifications in the Actel data book for additional I/O buffer specifications.

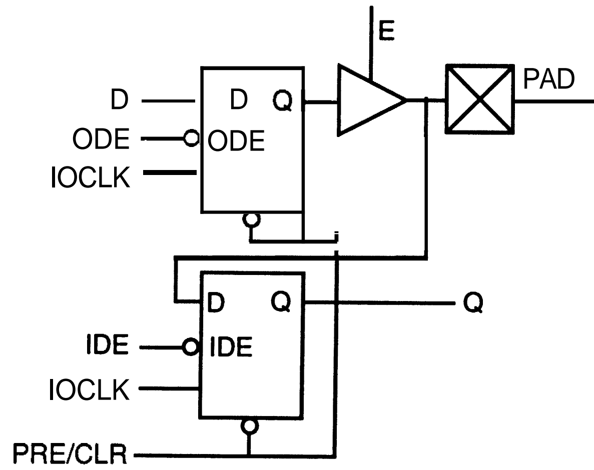


Figure 9. ACT 3 I/O module (with bi-directional registers)

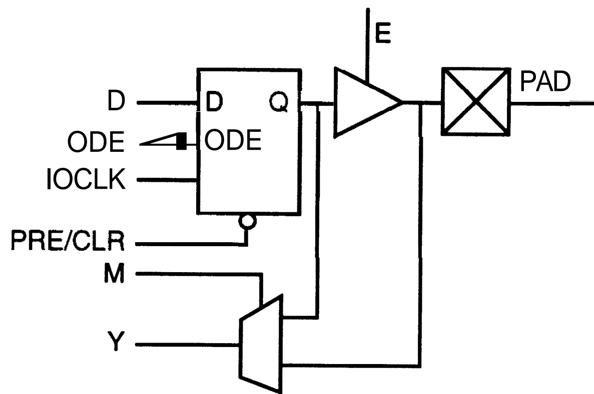


Figure 10. ACT 3 I/O module (registered with dual feedback)

ACT 3 Special Resources

Each ACT 3 type device has 4 special I/O resources. Use of these is key to compact, high-speed and reliable designs. Like ACT 2, an ACT 3 device has two high drive, low skew clock buffers (CLKBUF). These special I/O buffers need not use an external I/O pin when driving an internal load. The internal clock (CLKINT) is used in this case. In addition, there is a very high-speed clock buffer network. (HCLKBUF). This clock buffer should be used for signals faster than 75 MHz. The I/O registers (Figure 9 and Figure 10) use a dedicated I/O clock (IOCLKBUF) and reset/preset network (IOPCLBUF).

Note Synopsys Design Compiler V3.1+ will automatically use the I/O registers where ever possible. There are some limitations. See Chapter 4 for more details.

Mapping and Combinability

Actel FPGAs have a simple combinatorial building block used to construct from simple to complex functions using mapping. For example, in Figure 11 one of the many possible mappings of a three input AND (AND3) gate is shown. Logic that can fit in an ACT 2 or ACT 3 C-module (Figure 5 on page 14) with an output load of 1 and followed by a simple storage element is called combinable (Figure 12). This eliminates one level of logic (delay) and doesn't increase logic resources used (logic is already in front of S-module). In ACT 2 there is an exception. If the C-module logic must use both of it's local AND gate's inputs to synthesize the desirable function then this function can *not* be combined. Synopsys Design Compiler 3.1+ resolves this by using DFM7x's and DFM8x's (complex flip-flops) for ACT 2 and ACT 3 respectively.

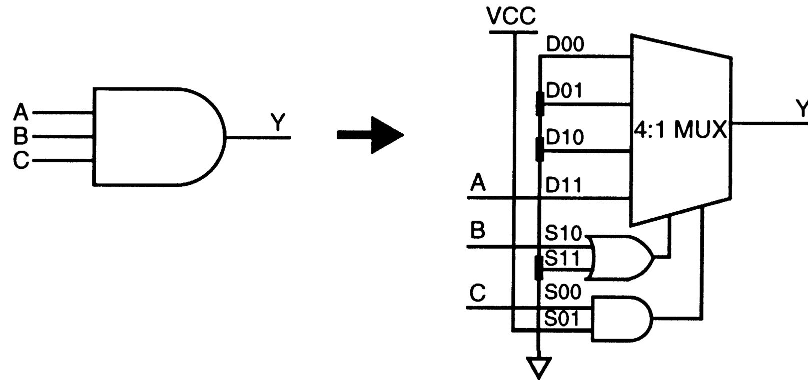


Figure 11. One (of many) AND3 Mappings

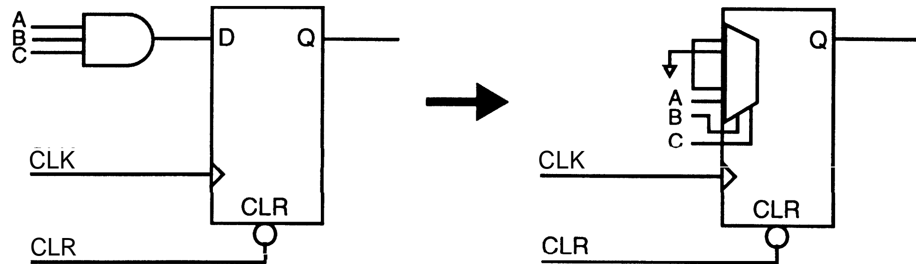


Figure 12. Combining of C-mod and Simple Register

When doing timing analysis or simulation, this reduced level of logic will represent no delay, To keep the timing characteristics coherent, ALS will back-annotate 0 ns for the combined function block into the simulator or the ALS timing tool (timer).

Speed Grades and Timing Characteristics

All Actel Device families come in multiple speed grades. A standard part represents a device that meets the minimum family timing profile (see data book). A part with a '–1' modifier is a device measured during the device sorting process to be 15 percent faster, a '–2' is 25 percent faster, etc. Actel uses *worst case* derating to specify device timing (worst case voltage, temperature, and process) in the data book and ALS software. If the designer is using the Design Compiler's default typical derating factor then all timing numbers must be divided by 1.25. For example, if the *worst case* performance of a device is 30 MHz (33.33 ns) then to derate for Design Compiler, at typical derating, in multiple speed grades is:

---

a) 33.33ns / **1.25** => 26.67ns (derate from typical to worst case)

b) for '–1' (15% faster device) – 26.67ns * **1.15** => 30.67ns

c) for '–2' (25% faster device) – 26.67ns * **1.25** => 33.33ns

d) for '–3' (35% faster device) – 26.67ns * **1.35** => 36.00ns

---

Example 1. Sample Derating Calculations

4

Tips Using Synopsys Design Compiler

It is our experience, here at Actel, that the most commonly under utilized group of commands are constraints. Constraints are used identify what your design goals are.

Setting Proper Design Constraints

Constraints refer to measurable circuit characteristics such as area and timing. Design Compiler calculates design area and path delays using the area and timing values from the Actel technology library.

There are two kinds of constraints: *design-rule constraints* and *optimization constraints.* In general, design-rule constraints reflect technology-specific restrictions that must be met for a functional design, (such as maximum loading on a net). Optimization constraints represent design goals that are desirable, but not crucial, to the operation of a design (such as the desired maximum circuit area or delay). Design Compiler tries to meet both types of constraints but gives emphasis to design-rule constraints, as they are requirements for a functional design.

Design Compiler uses constraints to guide optimization and implementation of a design. Constraints define the goals of the synthesis process. Design Compiler tries to meet these goals when synthesizing a design.

Design Compiler has a plethora of constraint-based commands. This manual will only focus on a subset of two types of optimization constraints: Speed and Area.

In general, the most important optimization constraint is maximum delay (`max_delay`). There are four types of delay categories (Figure 13). They are:

Tcq **Clock to Q :** This is generally known as the synchronous speed of the circuit or the register to register delay. These paths are constrained by specifying the clock(s) for the registers.

Tsu **Set-Up :** The delay from the input to valid data at the D-input of first flip-flop. These paths are usually constrained by specifying the clock for the register, and setting an input delay relative to a clock on the input port(IN1).

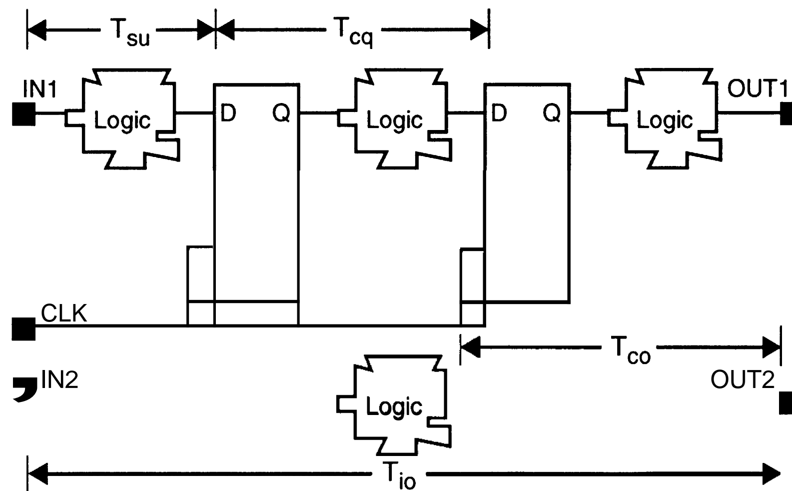


Figure 13. Four types of timing paths

Tco **Clock to Out** : The delay from the clock to valid data at the output port (OUT1). These paths are usually constrained by specifying the clock for the register, and setting an output delay relative to a clock on the output port(OUT1).

Tio **In to Out :** The delay from the input, through logic, then to the output. These paths can be constrained by setting an input delay on the input port (IN2), and either an output delay relative to a clock, or maximum and minimum delay targets on the output port (OUT2).

For the purposes of simplicity this manual will focus on the register to register delays. Maximum delay target values for each timing path in the design are automatically determined after considering clock waveforms and skew, library setup times, external delays, multicycle or false path specifications and `max_delay` commands. Load, drive, operating conditions, wire load models, and other factors are also taken into consideration.

Design Compiler has a built-in *static timing analyzer* for evaluating timing constraints. A static timing analyzer calculates path delays from local gate and interconnect delays but does not simulate the design. That is to say, *it does not check the design for functionality.* The Design Compiler timing analyzer performs critical path tracing to check minimum and maximum delay for every timing path in the design.

Note The most critical path is not necessarily the longest combinatorial path in a sequential design, since paths can be relative to different clocks at path start and endpoints. On the other hand, the Actel static timing analyzer (Timer), included with every ALS system, defaults to check only for the longest combinatorial paths.

Grouping Paths for Maximum Delay

Path groups affect the way the maximum delay cost is computed. You can assign paths or endpoints to named groups. Paths are placed in groups with **group_path** and create-clock commands. For further detail on the **group_path** command(s) see Design Compiler Reference Manual. The following example uses the **create_clock** command to create a groupgath with all register to register paths, a delay constraint of 50ns (20 MHz), 50% duty cycle, and group (signal) name of **clk**.

```
create-clock -period 50 -waveform {0 25) clk

                    or

create-clock -period 50 clk
```

Setting the Area Constraint

The **max_area** command specifies the maximum allowable area for the current design. Design Compiler computes the area of a design by adding together the areas of each of its components on the lowest level of the design hierarchy. The area of a cell is obtained from the Actel technology libraries. Maximum area is an optimization constraint and is therefore secondary to design-rule constraints (such as maximum fanout).

```
max_area 1200
```

To properly determine the maximum area in a given Actel device family use the "family profile" tables in Chapter **3.** For example, to constrain a design into an Actel A1280 (ACT 2 family) Table 2 indicates a total maximum area of 1232 modules. Design Compiler will attempt to meet this constraint.

Why Set Constraints

Design Compiler, in many cases, can synthesize and optimize circuitry with many different configurations. If, for example, the desired speed was 20 MHz (50ns) but not specified, Design Compiler may (and will) default to area optimization and possibly provide a less than desirable result. On the other hand, if the area was overly restrictive to, say 800 modules in A1280, then Design Compiler may not have enough room to use a much faster, but larger, version of a function because it was too large.

When setting constraints, setting the "don't cares" (the slow/static parts of circuit) is just as important (maybe more) than setting the target speed. In this way, Design Compiler does *not* use up precious high-speed resources for circuitry that may be static or control lines. In summary, setting the design constraints properly is the easiest way to realize correctly coded designs achieving the systems goals.

Effectively Using Scripts

The power of Design Compiler is it's flexibility through scripts. The source VHDL code need not be modified at all to achieve from a small, medium speed, compact design to a much faster, but larger, design. All throughout this manual you will see references to (**design compiler** shell or **dc_shell**) commands. In the previous section, it was shown how to set the target clock frequency and target area of a design. In the following sections it will be shown how, using scripts, a design can be manipulated to achieve higher performance with a minimal amount of effort and without changing the VHDL source.

Effectively Using Hierarchy

Hierarchy in a VHDL design is a form of design partitioning performed with component instantiation and multiple entity/architecture pairs that define the contents of the components.

Design Compiler supports automatic preservation of hierarchy in your design. The designer has the choice of either preserving the block functions or flattening the design at any branch to achieve group optimization. For example, a design may be fully flattened to attempt optimization of the complete design as one large block. Flattening a design may improve area and speed results but the final output could be hopelessly flattened beyond recognition as individually traceable/simulateable blocks.

Characterizing the Subdesigns to the Parent

In Actel devices, local (non-global) net loading directly affects performance of a given signal path. If only a single net is highly loaded in a design and this net is the slowest path in a register to register group then this path will become the critical path (remember, it is the *slowest* delay that determines circuit speed).

Hierarchical designs are composed of *subdesigns.* You can describe subdesigns independently of the parent design by *characterizing* or by *modeling* (modeling creates a characterized design as a library cell and will not be covered in this manual—please refer to the Design Compiler Reference Manual). Characterizing determines a Subdesign's port timing values, logical connection attributes, and

constraints by examining its context in the parent design. The `characterize` command automatically derives the boundary condition of subdesigns on its context in a parent design. Three types of boundary conditions are computed timing conditions, constraints, and connection relations.

```
characterize instance_name     /* for example -> "U100" */
```

Using Global Signals and Special Resources

Global signals, such as clocks, resets, and enables may be easier to design using high drive resources such as CLKBUFs and HCLKBUFs. Each Actel device has at least one and as many as four high drive resources. The following example will not buffer the reset line so that a special resource, like CLKBUF may be used.

```
dont_touch_network find(port, rst)
```

Using ACT 3 I/O registers

The ACT 3 family of devices from Actel have dedicated registers in the I/O modules to facilitate fast input set-up times (<1.5ns) and fast clock to out times (<10ns). These resources are not automatically utilized by Design Compiler, a post-processing step is required. The following post-processing script (Example 2) is used to instantiate the special ACT 3 I/O buffers into a given design.

Note Two new ports are created during this process "IOCLK and "IOPCL". To make sure simulation is coherent, the designer must add them to the simulation model (vectors).

```
/* create ports IOCLK & IOPCL and instantiate buffer cells */
createqort IOCLK
create-cell "ioclk" act3/IOCLKBUF
create-net {ioc
lk_net}
connect-net ioclk_net find(port, (IOCLK))
connect-net ioclk_net find(pin, (ioclk/PAD))
createqort IOPCL
create-cell "iopcl" act3/IOPCLBUF
create-net {iopcl_net}
connect-net iopcl-net find (port, (IOPCL))
connect-net iopcl-net find(pin, (iopcl/PAD))

create-net {ioclk-net-out)
connect-net ioclk-net-out find(pin, ioclk/Y)
create-net {iopcl-net-out]
connect-net iopcl-net-out find(pin, iopcl/Y)

/* find cell names  for reference */
filter(find(cell, "*"),       "@ref_name == ORECTH_NO_TRI ||
@ref-name == IREC_SY        || @ref-name == IREP_SY         || \
@ref-name == ORECTH_SY      || @ref-name == ORECTL_SY       || \
@ref-name == OREPTH_SY      || @ref-name == OREPTL_SY       || \
@ref-name == ORECTL_NO_TRI || @ref-name == OREPTH_NO_TRI || \
@ref-name == OREPTL_SY ")

foreach(seq_cell, seqio_1)
  {
  /* to find name of clk net */
  all-connected seq_cell + /CLK
  clock-net = dc_shell_status

  /* to find name of set/reset net */
  all-connected seq_cell + /IOPCL
  clear-net = dc_shell_status

  /* disconnect this clock net & connect to dedicated net  */
  disconnect-net clock-net find(pin, seq_cell + /CLK)
  connect-net ioclk-net-out  find(pin, seq_cell + /CLK)

  /* disconnect this reset net & connect to dedicated net  */
  disconnect-net clear-net find(pin, seq_cell + /IOPCL)
  connect-net iopcl-net-out  find(pin, seq_cell + /IOPCL)
  }
```

Example 2. Post processing script for special ACT 3 I/O

Note A link library **link.v** is also required to use this script file. Contact the Actel Technical Support Hotline to obtain a copy of both scripts at (800) 262-1060 or e-mail at tech@actel.com.

Running ALS from Within Design Compiler

Actel's ALS tool suite supports simple command line operations that can be run from within the Design Compiler environment. Design Compiler has the ability to call UNIX and execute external commands. This is accomplished through the **sh** (shell) command. The following example (Example 3) will run all necessary ALS 2.3 commands to convert Design Compiler EDIF output to ALS format, validate the design (rule checker), place, route, and then optionally invoke the Actel static timing analyzer (Timer). For a brief description of the ALS "Timer".

```
sh cae2adl -edn2adl fam:act3 ednin:TEST.edn NObadOrigName:T TEST
sh als -set fam act3 TEST
sh als -set die 1460 TEST
sh als -set package qfp208 TEST
sh als -validate TEST
sh als -place TEST
sh als -route TEST
sh als -xtract TEST
sh als -timer TEST < TIMER.SCR (optional invoking of Timer)
```

Example 3. DC_SHELL Script to run ALS from within Design Compiler

Note TEST is the name of the top level design. In the above example, the design TEST is set to a A1460 (ACT 3) in a QFP208 package. Device, family and package only have to be set once.

Verifying Synopsys Performance with ALS

The ALS software contains a robust static timing analysis tool called the Timer. The Timer works on the principle of sets, much like Design Compiler's groups. The Timer will automatically build default sets using input buffers, output buffers, and registers. The default group for timing is register to register delays (using the clock net as a starting reference port). The following is the contents of TIMER.SCR from Example 4.

```
orstop inpads    adds input pads to stop paths at borders
orstop outpads   adds outputs pads to stop set
longest          get longest paths (defaults: Tcq)
expand 0         expand worst path '0'
```

Example 4. Simple Timer command sequence to extract longest Tcq

5

Hints On Designing with Synopsys VHDL

In order to determine the most common issues with customer VHDL-based designs we, at Actel, did a wide study of our customer's design implementations. State machines construction and custom arithmetic functions had the highest percentage of recurrence. The following hints in this section are presented in a technology independent manner wherever possible.

General State Machines Description

The possible states of the state machine are listed in an enumerated type. A signal of this type`(present-state)` defines in which state the state machine is currently in. In a case statement of one process, a second signal`(next-state)` is updated depending on `present-state` and the inputs. In the same `case` statement, the outputs are also updated. A second process updates `present-state` with `next-state` every clock cycle except when the rst is asserted.

Here is the VHDL code (Example 5) for such a typical state machine description. This design implements a post food processing unit.

```
entity food-cycle is
   port (clk, rst, switch      : in bit;
         full, gte_half, empty : in bit;
         water, purge          : out bit);
end food-cycle;

architecture behavior of food-cycle is
-- define possible states for state machine
  type state-type is ( reset, uwait, flush, fill, half);
  signal present-state, next-state : state-type;
```

***Example** 5. Food Psot-Processor Unit*

```
begin
  process(RST, CLK)          -- registers
  begin
    if (rst = '0')then
      present-state <= reset;
    elsif( CLK='1' and CLK'event) then
      present-state <= next-state;
    end if;
  end process;

process(present_state, rst, switch, full, gte_half, empty)
  begin                      -- transitions
  case present-state is
    when reset =>            -- reset state
      if(rst='1' and full='1') then
        next-state <= uwait;
      end if;
      water <= '1'; purge <= '0';
    when uwait =>            -- wait for button push
      if(switch='1') then
        next-state <= flush;
      end if;
      water <= '0';  purge <= '0';
    when flush =>            -- open purge
      if(empty='1') then
        next-state <= fill;
      end if;
      water <= '0';  purge <= '1';
    when fill =>             -- fill water / ignore switch
      if(gte_half='1') then
        next-state <= half;
      end if;
      water <= '1'; purge <= '0';
    when half =>             -- cont. fill / examine switch
      if(full='1') then
        next-state <= uwait;
      elsif(switch='1') then
        next-state <= flush;
      end if;
      water <= '1'; purge <= '0';
  end case;
  end process;
end behavior;
```

Example 5. Food Psot-Processor Unit (Continued)

State Machine Design Hints

There are various issues of coding style for state machines that might affect performance of the synthesized result. State machine style can have the most profound affect on FPGA performance. The following section will give the designer hints on technology independent coding styles.

A first issue is the form of state machine that will be created. There are basically two forms of state machines, Mealy machines and Moore machines. In general, Moore machines have fewer states than a Mealy machine. In a Mealy machine, the outputs change during transitions from state to state. In a Moore machine, the outputs are derived directly from the present state and the inputs.

To demonstrate this we will use the example of the food post-processor unit from the previous section. The Moore version is shown in Figure 14.

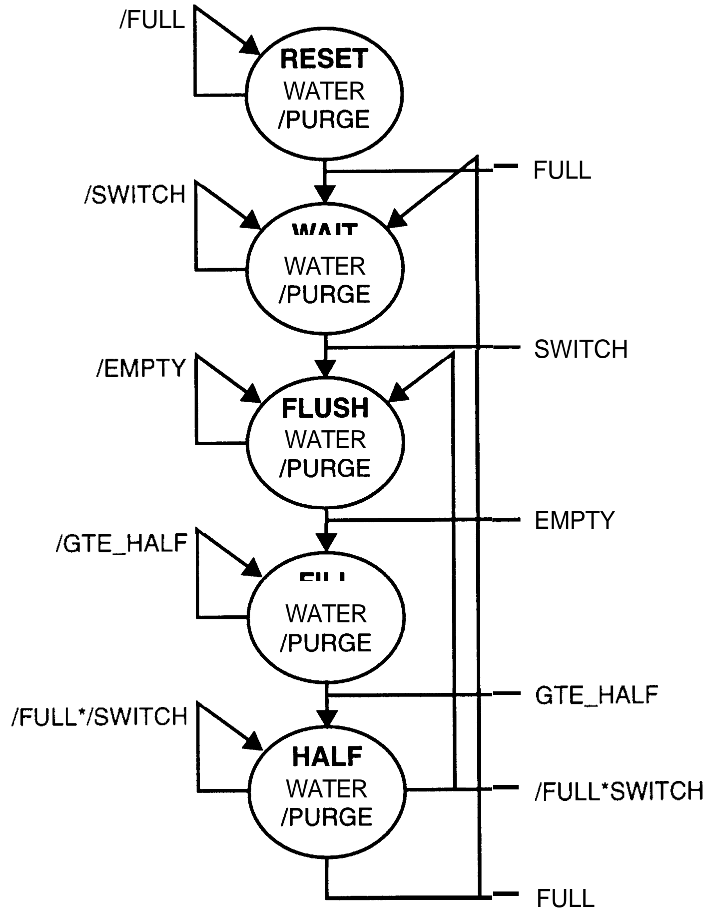


Figure 14. Graphical State Diagram of Example 5

State Encoding

The present-state signal in Example 5 is of a user-defined enumerated type. There are five values (states) in this type: reset, uwait, flush, fill and half. This means for simulation, present-state can appear in each of these five states. For synthesis, state assignment has to be done on a number of state bits to represent each of these states. By default, Design Compiler implements compact (binary) encoding. This means that three state bits are needed. Encoding is done based on the definition of the state type, counting left to right. In this example, the following state values are assigned to each state:

Table 4. Compact Encoded State Bits

state	state-bits 012
reset	000
uwait	001
flush	010
fill	011
half	100

If you require different binary encoding values for each of the states, you must change the order of the enumerated type accordingly.

Extracting an FSM from a Sequential Design

Why does an FSM need to be extracted? So Design Compiler can know which registers are to be optimized as that FSM. In this way, the designer may changed encoding styles or simply let Design Compiler choose use the auto command.

Figure 15 show the flow from a design or netlist to a Finite State Machine (FSM). Each step is shown with the relevant Design Compiler commands. ***Optional statements are shaded.***

There are two classes of state machines design that must be considered. When the entire design is the state machine or when the state machine registers are buried within the context of a larger design. The complete flow must be followed for the latter case.

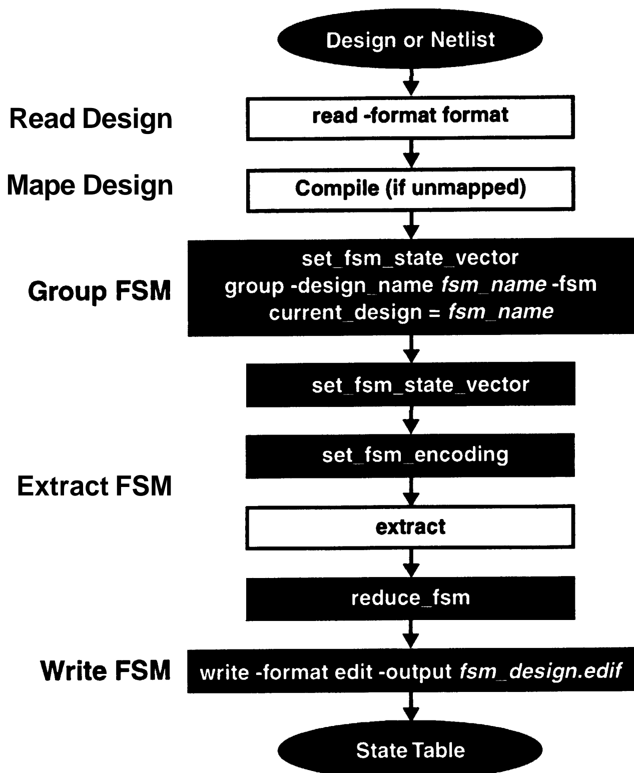


Figure 15. Extracting a Finite-State Machine From a Design

The following steps will be required to extract and optimize this design.

Read Design

Read the design into Design Compiler with the appropriate **read** command. For example, to read Example 5:

```
read -format vhdl food_cycle.vhd
```

Map Design

If the design is not mapped, run **compile** to map the design to gates:

```
compile -map_effort low  /* map—effort shown is optional */
```

Group FSM

If the entire design is a state machine, this step is optional. Grouping the FSM part 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. The next step is to extract the FSM information to create an FSM representation.

If not all flip-flops in the design are part of the state machine, use the **set_fsm_state_vector** command to name the FSM state vector flip-flops:

```
set_fsm_state_vector (reset-reg, uwait_reg, ....)
```

use the **group -fsm** command to group the subset of the design that includes just the FSM flip-flops and their associated logic into a new level of design:

```
group -fsm -design—name food—cycle—fern
```

Finally, move down into the FSM part of the design by setting the current-design variable to the new FSM name:

```
current—design = food—cycle—fsm
```

Extract FSM

Extracting an FSM from a circuit changes the representation from a netlist format to FSM `state-table` format. Consult the Design Compiler Reference Manual for specific optimizations that can be supplied. This section will only cover **`binary`** and `one-hot` encoding methods.

First, set the order of the state vector flip-flops with the **`set_fsm_state_vector`** command. The order of the flip-flops must agree with order of the state vector bits, that is, each bit in the state vector represented by one flip-flop in order:

```
set-fsm-state-vector { reset-reg, uwait_reg,...}
```

Next, you can optionally set the state encoding with the **`set_fsm_encoding`** command (see Design Compiler Reference Manual —*Efficient* FSM Extraction). State encoding tells Design Compiler the names and values of each state:

```
set_fsm_encoding {"reset=0","uwait=2",...}
```

Extract the FSM (convert the circuit into an FSM):

```
extract
```

You can optionally use the **`reduce_fsm`** command to minimize the state-transition logic:

```
reduce-fsm
```

Optional FSM Optimization Styles

Now that the design is optimized and represented internally as a state table, we can optimize for a different coding strategy. In this case, we will choose **one_hot**. First, re-extract design and then set new strategy as follows:

```
/* re -extract */
set_fsm_state_vector { reset-reg, uwait_reg,...}
set_fsm_encoding {"reset=0","uwait=2",...}
```

Set FSM *Order*

To manually define the ordering of states and have Design Compiler automatically number the state vectors in binary or gray encoding:

```
set_fsm_order { reset, uwait, flush,...}
set-fsm-encoding-style gray

          or

set-fsm-encoding-style binary
```

Use One Hot *Encoding*

To use **one_hot** encoding and to have Design Compiler automatically generate the state vectors for a very fast FSM, where only one bit per state is used:

```
set-fsm-encoding-style one-hot
```

Re-Optimize New FSM Design

Now we re-optimize the FSM design, minimize and compile:

```
set-fsm_minimize true
compile
```

Pop back a hierarchy level to write the design:

```
current-design = food-cycle
```

Write FSM

Once the is in FSM format, you can optionally **write** it to a file on the appropriate file format. In this example will write it out in the EDIF format:

```
write -format edif -output food-cycle.edu
```

Using One-Hot Encoding Strategy

One-Hot (or Bit-Per-State) encoded state machines are state machines that will get one flip-flop per enumeration value (state).

Table 5. Compact Encoded State Bits

state	state-bits
01234	
reset	10000
uwait	01000
flush	00100
fill	00010
half	00001

Only one bit is '1' at any given time (hence "one-hot") in objects of one-hot encoded enumerated types. For example, on the previous example (Example 5) we saw that the five states produce a resultant state machine using three register bits (Table 4 on page **37).** With one-hot encoding the register count would increase to five, one for each state (Table 5).

This encoding strategy often improves the speed of the synthesized circuit and, in Actel FPGAs, flip-flops have very low area cost, the area may improve as well. A complete listing of a typical script file used for the design from Example 5 is shown in Example 6. We will assume that the state machine is but a small part of the design.

```
/* sets the level of hierarchy to FSM design */
current-design = food-cycle
/* create clock (see section 3.1) */
create-clock -name clk -period "50" -waveform {"0" "25"} {clk}
/* make sure no logic is on reset line (port) */
dont_touch_network find(port, rst)
/* compile design  as shown with steps above */
compile

set_fsm_state_vector {reset-reg, uwait_reg, ....}
group -fun -design-name food-cycle-fsm
/* sets the level of hierarchy to FSM design */

current-design = food_cycle_fsm
set-fern-state-vector {reset_reg,uwait_reg,flush_reg...}
set-fun-encoding
{"reset=0","uwait=2","flush=4","fill=8",half=16"}
extract
/* reduce fsm logic, set vector, encode, assign FSM order */
reduce-fsm
set_fsm_state_vector {reset_reg,uwait_reg,flush_reg,...}
set_fsm_encoding
{"reset=0","uwait=2","flush=4","fill=8","half=16"}
set-fern-order { reset, uwait, flush, fill, half }
set-fun-encoding-style one-hot
set-fsm_minimize true
compile
current-design = food-cycle
write -format db -heirarchy -output ./food-cycle-hot.&
```

Example 6. Commands (script) for One-Hot Design

These steps should produce smaller faster machines due to the large number of registers in Actel devices. Refer to the FSM chapter in the Design Compiler Reference Manual for more details.

Manual One Hot State Encoding

If you are not using enumerated types for your state-machines, or would like to have full control over the synthesized circuit, there is a way to do manual one-hot on a state-machine in VHDL. Using Table 5 and Figure 14 you can construct a simple shift-register like state-machine. Start by encoding the equations to the registers with the equations for each arrowhead. There are some considerations from this type of state-machine encoding. Using Example 5 they are:

1. **A** case statement should not be used for state comparisons, since the state comparison has to depend on one bit of the state vector only. `present-state` used with a `case` statement can only compare the whole vector.
2. The `elsif` construct should not be used to do the same state comparisons, since it introduces additional constraints on the values of each state. Using `elsif` means that this code is only entered if all of the previous conditions are false. In the case of one-hot encoding, it is certain that all previous conditions are false already.
3. It is essential to assign `next_state` to zero before the state transitions are generated. Since only a single bit of `next_state` will be set in the transitions process, combinatorial loops would have to generated to preserve the other bits if this initial assignment was not included.

Automatic FSM Encoding Styles

Additionally, Design Compiler can automatically select the most appropriate encoding style by replacing the argument as follows:

```
set_fsm_encoding { }
set_fsm_encoding_style auto
```

The auto encoding style uses a proprietary algorithm in which the primary object is to determine a set of encodings that best reduces the complexity of the combinatorial logic while using the minimum number of encoding bits. Consequently, this encoding style is targeted toward ***area*** optimization. Speed is not addresses directly, but, in general, smaller area reduces delay.

Note The maximum supported state vector length for **auto** encoding is 30 bits.

In summary, manual **one-hot** encoding is not a trivial process. If **one-hot** encoding does not produce the best implementation then the VHDL code will have to be re-written. In general, a more practical methodology for designing state-machines is to design a generic state-machine and let Design Compiler choose the configuration best for the application or manipulate the configuration through the script file commands.

Power-up And Reset

For simulation, the state-machine will initialized into the leftmost value of the enumeration type, but for synthesis it is unknown in which state the machine powers-up. Since Design Compiler does state encoding on the enumeration type of the state machine, the state machine could power up in a state that is not even defined in VHDL. Therefore to get simulation and synthesis consistency, it is very important to supply a reset to the state machine.

If the desired state machine encoding style is **one_hot** then it is imperative that the state machine is supplied a reset. Since for **one_hot** style only a single register must be active, all others must be reset/inactive. Also, for Design Compiler, it is necessary that no logic must exists on the reset network. The **dont_touch_network** command can be used to assure that no logic is generated for the reset network

(Chapter 4). In order to achieve a high drive network (>24 loads) a high drive resource must be used. In ACT 2 and ACT 3 devices a CLKBUF is recommended (required) for highly loaded networks. See Chapter **3** for number of resources available.

"If-then-else" Versus "Case"

The **`if-then-else`** statement is used in VHDL to conditionally execute sequential statements. The `case` statement selects, for execution, one of a number of alternative sequences of statements. The chosen alternative is defined by the value of the expression. It is recommended that the designer uses the **`case`** statement in VHDL when they have a complex decoding situation. It is a more readable statement than a collection of nested `if-then-else` statements. It allows the designer to easily identify the value and associated actions. Additionally, if you want Design Compiler to generate a **`MUX`** structure for the case statement you should use the parallel/full case command. Otherwise a complex priority encoder will be synthesized (not friendly to a mux-based device).

Arithmetic Elements And DesignWare 3.1+

When arithmetic and relational logic is used for a specific VHDL design, Design Compiler provides a method to synthesize technology specific implementations for these operations. Multiple implementations of these operations (small to fast) have been developed by Actel using the Actel structured macro generation tool ACTgen. DesignWare is a tool that can build the supported, functions (for bit sizes from 2 to 32 bits), but depending on the constraints (see Chapter 3) these functions may (or may not) be used. The DesignWare macros can offer performance improvements of up to 80 percent.

Designware 3.1 Performance (Post Routed)

Table 6 contains DesignWare "Add/Subtract" data.

Table 6. ACT 3 (standard speed) Family Design Adder Performance

Bits	Area	Freq	DW Area	DW Freq	Speed Up	Post Route
2	3 modules	119 MHz	3 modules	119 MHz	+0%	124 MHz
3	9	119	9	119	+0	124
4	19	58	14	109	+89	110
5	35	57	19	104	+82	104
6	47	57	21	69	+21	73
7	50	45	26	69	+53	72
8	74	57	30	68	+19	72
9	57	38	35	66	+74	71
10	86	45	39	64	+42	68
11	87	37	49	66	+78	n/a
12	94	37	53	64	+73	n/a
13	74	32	58	63	+97	n/a
14	88	33	62	63	+91	n/a
15	124	36	71	63	+75	n/a
16	113	32	78	63	+97	n/a
17	111	28	85	63	+125	n/a
18	126	31	81	48	+55	n/a
19	133	27	86	48	+78	n/a
20	146	27	90	48	+78	n/a
21	140	30	95	47	+57	n/a
22	167	28	99	46	+64	n/a
23	156	27	109	47	+74	n/a
24	165	25	113	47	+88	n/a
25	202	24	119	48	+100	n/a
26	170	25	122	46	+84	n/a

Table 6. ACT 3 (standard speed) Family Design Adder Performance (Continued)

27	214	26	132	46	+77	n/a
28	218	25	136	46	+84	n/a
29	199	24	141	46	+92	n/a
20	225	27	145	45	+67	n/a
21	234	27	154	45	+67	n/a
32	248	24	161	45	+88	n/a

6

Component Instantiation

Components are a method of introducing structure in a VHDL description. A component represents a structural module in a design. Using components it is possible to describe a netlist in VHDL.

Basic Component Instantiation

Components are instantiated in the dataflow environment. Example 7 is an example of a structural VHDL description where a 2 bit RAM is instantiated as a design:

```
entity example-ram is
        port ( addr   : in bit_vector(1 downto 0);
               din    : in bit_vector(1 downto 0);
               wr_en  : in bit;
               dout   : out bit_vector(1 downto 0));
        end example-ram;

architecture dataflow of example-ram is

        component RAM16x4
               port ( we : in bit;
                      a0, a1 : in bit;
                      d0, d1 : in bit;
                      q0, q1 : out bit );
        end component;

begin
        U100 : RAM16x4 port map ( we => wr_en,
                      a0 => addr(0), a1 => addr(1),
                      d0 => din(0),  d1 => din(1),
                      q0 => dout(0), q1 => dout(1));
end dataflow;
```

Example 7. Simple Macro Cell Instantiation Example

User and "Soft Macro" components can be instantiated in the same manner. By defining the port list in a **component** statement and instantiating the component in the **port map** statement any user defined macro can be used in a VHDL design.

Technology Independent Component Instantiation

To implement a technology *independent* top level VHDL description a designer only needs to create a subdesign, with equivalent ports and functionality using generic VHDL constructs. Then, during technology *specific* mapping, replace the generic code with the technology specific component netlist.

Actel Library Cell Instantiation

Example 8 is the format used to instantiate a low-level gate like an Actel Library Cell:

```
component HCLKBUF port (
              PAD : in  bit ;
              Y   : out bit ) ;
end component;

UO : HCLKBUF port map ( PAD=>HCLK_P,  Y=>CLK );
```

Example 8. Actel Library Cell Instantiation Example

Where, HCLKBUF is an Actel ACT 3 family high-speed, high drive, low skew clock buffer. UO is the instance name, and HCLK_P and CLK are the nets connected to pins PAD and Y, respectively.

Using "ACTgen" Macros In Your Design

The ACTgen structured macro builder is an Actel Tool that can create macros which can be instantiated in VHDL designs for synthesis using Design Compiler. These macros are defined in a high level language from which an EDIF netlist is created. The goals of the ACTgen macro builder is to create flexible library elements that efficiently use the Actel architecture to achieve optimal performance, minimal module count and improved designer productivity.

ACTgen/Synopsys Design Flow

Currently, ACTgen generates counters, registers, adders, decoders, and multiplexers. Many more functions are being added with each release. Consult the ***ACTgen Macro Builder User Guide*** for a current listing. To integrate an ACTgen generated component into the VHDL design, a few simple step must be followed:

1. Invoke ACTgen, set the CAE system to generic.
2. Select Actel family, macro function and parameters.
3. Generate the macro (output sent to EDIF file).
4. Read EDIF file into Design Compiler, write as *.db
5. Identify function port names and directions
6. Instantiate component into design
7. Modify script to read (*.db or *.vhdl)

The design flow is described in Figure 16.

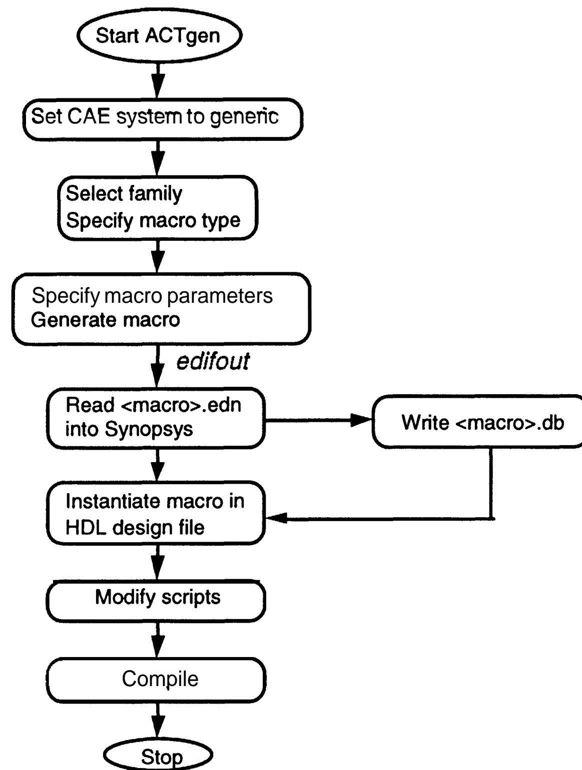


Figure 16. ACTgen/Synopsys Design Flow

I/O Instantiation

Input and output buffers are automatically inferred. The following scripts must be set prior to optimization in order to perform I/O buffer mapping with Design Compiler or dc_shell.

```
set_port_is_pad all-inputs0
set_port_is_pad all-outputs0
setgad-type all-inputs0
setgad-type all-outputs0
setgad-type -clock CLK
insertgads
```

Where, CLK is an example port name.

These scripts will cause Design Compiler to map all I/O ports specified to an Actel I/O buffer. The clock function, described in the VHDL design file, will be mapped to a CLKBUF. After each switch is set, messages appear showing the corresponding switch function being implemented for applicable I/O ports, as shown in the example below:

```
Performing set_port_is_pad on port 'in1'
Performing set_port_is_pad on port 'out1'
Performing setgad-type on port 'in1'
Performing setgad-type on port 'out1'
Performing setgad-type on port 'CLK'
```

Where, in1, out1 and CLK are examples of port names.

A simpler way of applying I/O buffers is by using the script,

```
setsort-isgad <design-name>
```

Which places an INBUF on an input, OUTBUF on an output and CLKBUF on a clock line.

7

*Most F*requenfly Asked Questions

Where do I install the Actel/Synopsys libraries and how do I read in the libraries?

The libraries may be installed anywhere on the system and the following path should be specified in the .synopsys_dc.setup file:

```
target-library = ~synopsys/library/actXX_31.db;
symbol-library = ~synopsys/library/actXXsym.sdb;
link-library = ~synopsys/library/actXX_31.db;
read ~synopsys/library/actXX_31.db;
```

Where, XX = 1,2,3 are the actel family pointers, (e.g. n=3 specifies ACT3), and ~synopsys/library is the directory where the libraries have been installed. If the libraries are installed at a different location, the library path should be modified accordingly.

What scripts do I need to set for generating an Actel specific EDIF netlist ?

The following scripts have to be specified in the .synopsys_dc.setup file in order to generate an Actel specific EDIF netlist:

```
edifout_no_array = true
edifout_netlist_only = true
edifout_power_and_ground_representation = cell
edifout_ground_name = GND
edifout_power_name = VCC
edifout_ground_pin_name = Y
edifout_power_pin_name = Y
edifoutqrettyqrint = true
read_array_naming_style = "%s<%d>"
```

Does Actel support backannotation in Viewlogic's Viewsim?

Actel is still in the process of testing this flow for backannotation. However, several customers have successfully backannotated to Viewsim. An outline of the necessary steps for backannotation, upon completion of place and route, are:

Generate a WIR file using,

```
edifneti <design_name>.edn
```

The wir files are placed in the WIR subdirectory of the primary design directory. If you wish to generate schematics for the design, use

```
viewgen <design-name>
```

Then generate a vsm file using,

```
del2vl <design-name>
```

What scripts do I need to set for Actel specific EDIF netlist generation for back annotation?

Using the **ungroup** command for flattening the design is recommended for an optimal design. However, this puts /'s in instance names in the Synopsys generated EDIF file, which is mistaken as a level of hierarchy during the Actel design flow. In order to avoid such confusion, the following scripts should be used for EDIF generation.

Using Viewsim, a Viewlogic simulator, the following scripts should be set in Design–Analyzer or dc_shell for EDIF generation:

```
define-name-rules CAPS -allowed "A-Z 0-9 _"
change-names -rules CAPS -hierarchy
```

This will capitalize all instance names in the EDIF file, thereby avoiding any conflicts that might occur due to the case of the alphabets.

```
define-name-rules RES1 -restricted "/" -replacement-char "_"
change-names -rules RES1 -hierarchy
```

This script will replace all /'s with _'s in instance names since /'s might be mistaken as a level of hierarchy at some stage of the design flow. The same script can be used for replacing \'s with another character.

*I want to change the maximum **fanout** on my design. It currently limits me to 16 but I want to reduce it,*

By default, the ACT1 library fanout limit is set to 10 and the ACT 2 and ACT 3 is set to 16. The maximum fanout can be changed for the design using the "**set_max_fanout**" script. However, the new fanout limit should not be applied to the clock buffer network. This is accomplished using:

```
set_max_fanout <new_fanout> <design_name>
dont_touch_network <name_of_clock_port>
```

Design Compiler will always choose the most restrictive!!!

*I specify my max **fanout** to be n. However, this limit is ignored for logic blocks that have the **dont_touch** attribute. Why does Synopsys ignore this design rule? How can I force **Synopsys** to obey this design rule?*

The following suggestion should address this problem:

Remove the **dont_touch** attribute on the blocks which do not obey the fanout constraints. Specify the max fanout at the top level using the **set_max_fanout** script. Then perform an incremental compile at the top level using:

```
compile -only_design_rule
```

Consequently, compilation will follow the design rules (for example, **set_max_fanout**) and individual blocks will not be recompiled.

The `read-array-naming-style` script uses < > instead of ( ) for array notation. When I write out script files while characterizing, the script files have < > instead of ( ). Synopsys cannot read in script files with < > for array notation. Synopsys can only read in ( ). What can I do?

The `read-array-naming-style` script reads in VHDL which has [ ] for array notation and converts them to < >. If you are characterizing your blocks and writing out script files, this switch writes < > for arrays in the script files. However, Synopsys cannot read script files which use < > for array notation since the array terminology has to be [ ]. In order to successfully read in the script files after characterization, the < > array terminology has to be changed to [ ] before the script file is written, using the following scripts in Synopsys version 3.0c:

```
define-name-rules ARRAY -restricted "<" -replacement-char "["
definegame-rules ARRAY -restricted ">" -replacement-char "]"
change-names -rules ARRAY -hierarchy
```

The following script will list the rules being applied,

```
report-name-rules ARRAY
```

These scripts will ensure that the script files are written out with [ ] rather than < > for array notation.

Why do some of the ACT2 and ACT3 macros have "`dont_use`" attribute attached to them and how do I remove this attribute?

The "`dont_use`" attribute has been attached to several ACT2 and **ACT3** sequential macros that use one sequential and one combinational logic module. These two module macros are considered inefficient compared to others in the library.

The "`dont_use`" attribute can be removed by using the "`remove_attribute`" command.

How do I translate from the Synopsys to the ALS environment?

After optimizing and flattening the design (using the `ungroup` command) in the Synopsys environment, and setting the appropriate EDIF scripts in the .synopsys_dc.setup file,

An EDIF netlist from Synopsys is generated by executing the following command:

```
write -f edif -o <design_name>.edn <design-name>
```

The EDIF netlist, <design_name>.edn, is converted to an Actel specific netlist, <design_name>.adl, file using the program, cae2adl. The syntax for invoking the command is:

```
cae2adl -edn2adl fam:<value> ednin:<edif netlist filename>.edn
NObadOrigName:T <design-name>
```

where,

fam	is the family type, either ACT 1, ACT 2 or ACT 3
ednin	is the EDIF file to be converted, which must be <design_name>.edn
NObadOrigName:T	ensures that the Edif name is used by the Actel suite of programs if the original name is not completely legal.
<design-name>	is the name of the design.

Note A successful generation of the <design_name>.adl file will create two additional files in the <design-name> directory: <design_name>.crt and <design_name>.ipf.

Invoke the Actel suite of programs by typing,

```
als <design-name> (version: ALS 2.3)
designer (version: Designer for X-Windows)
```

from the <design–name> directory.

How do I assign pins to my design?

The Actel place and route suite of programs offer fully automatic pin assignment, logic assignment, and routing. However, the user has the capability of manual pin assignment. This procedure is described in the Designer Series user guide.

I have optimized my design and generated an adl netlist using EDN2ADL, but when I try to use Pin Edit, an error stating "bad checksum, certify the <design_name>.ipf file" is issued.

This problem occurs when the top level of hierarchy described in the HDL is not the same name as the directory in which the EDIF netlist exists as well as the <design–name> specified in the EDN2ADL command. For example, if the top level entity in the HDL is *dma, then the EDIF netlist and HDL for the design, dma, must reside in a directory named, dma, and the <design–name> specified in the EDN2ADL statement must be dma.*

After making design changes and regenerating a netlist, Actel no longer recognizes my <design_name>.ipf file. Why?

Only if users have changed their primary I/O names will they need to follow the instructions below:

Users who have completed their board design need to fix pins on their chip. This fixed pin information is saved in the <design_name>.ipf file. After every design iteration a new edif netlist, <design_name>.edn, is created. Consequently, a new

<design_name>.ipf file has to be created since there will be mismatches between the old **<design_name>.ipf** and new **<design_name>.edn**. The **<design_name>.ipf** file will not automatically update itself to match the new edif **netlist.**