ACTmap VHDL Synthesis

Methodology Guide

Actel

Windows® & UNIX® Environments

Actel Corporation, Sunnyvale, CA 94086

Printed in the United States of America

Part Number: 5579007-2

Release: April 1999

Trademarks

Table of Contents

List of Figures

Introduction

VHDL is a high-level description language for system and circuit design that supports various abstraction levels, including system design without regard to a specific technology. However, to achieve optimal performance and area from your target device, you must become familiar with the architecture of the device and code your design for the architecture.

The *ACTmap VHDL Synthesis Methodology Guide* contains information and techniques for using ACTmap VHDL to design an Actel device. This includes information about writing VHDL code for ACTmap, optimization techniques, and sample code. This guide also includes information about using the ACTmap VHDL Synthesis tool in batch mode. Refer to the *Designing with Actel* manual and the ACTmap on-line help for information about the ACTmap user interface. Refer to the *Actel HDL Coding Style Guide* for additional information about HDL coding.

Document Organization

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

Chapter 1 - ACTmap Design Flow describes the design flow for creating Actel designs with the ACTmap VHDL synthesis tool.

Chapter 2 - Using ACTmap VHDL describes how to write VHDL for use with the ACTmap VHDL synthesis tool.

Chapter 3 - Advanced Optimization Techniques contains information about the optimization features of ACTmap and describes how to implement optimization techniques in a design.

Appendix A - Using ACTmap in Batch Mode contains information about using command-line commands and command files in the ACTmap VHDL synthesis tool.

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

Document Assumptions

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

1. You have installed the Designer Series software, including ACTmap.
2. You are familiar with UNIX workstations and operating systems.
3. You are familiar with PCs and Windows operating environments.
4. You are familiar with FPGA design software, including design synthesis and simulation tools.

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
```

VHDL code appear as follows, with VHDL keywords 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
```

The <act_fam> variable represents an Actel device family. To reference an actual family, substitute the name of the Actel device when you see this variable. Available families are act1, act2 (for ACT 2 and 1200XL devices), act3, 3200dx, 40mx, 42mxand 54sx.

Actel 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 "/manuals" directory. These manuals are also installed onto your system when you install the Designer software. To view the on-line manuals, you must install Adobe® Acrobat Reader® from the CD-Rom.

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.

DeskTOP Interface Guide. This guide contains information about using the integrated VeriBest® and Synplicity® CAE software tools with the Actel Designer Series FPGA development tools to create designs for Actel Devices.

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™ 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 Guide. 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.

Actel FPGA Data Book. 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.

A Guide to ACTgen Macros. This Guide provides descriptions of macros that can be generated using the Actel ACTgen Macro Builder software.

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, Silicon Explorer, Silicon Sculptor, and APSW.

1

ACTmap Design Flow

This chapter illustrates and describes the design flow for creating Actel designs using the ACTmap VHDL synthesis tool and third party tools.

Design Flow Illustrated

Figure 1-1 illustrates the design flow for creating an Actel device using the Designer Series, ACTmap, and 3rd party CAE software[1].

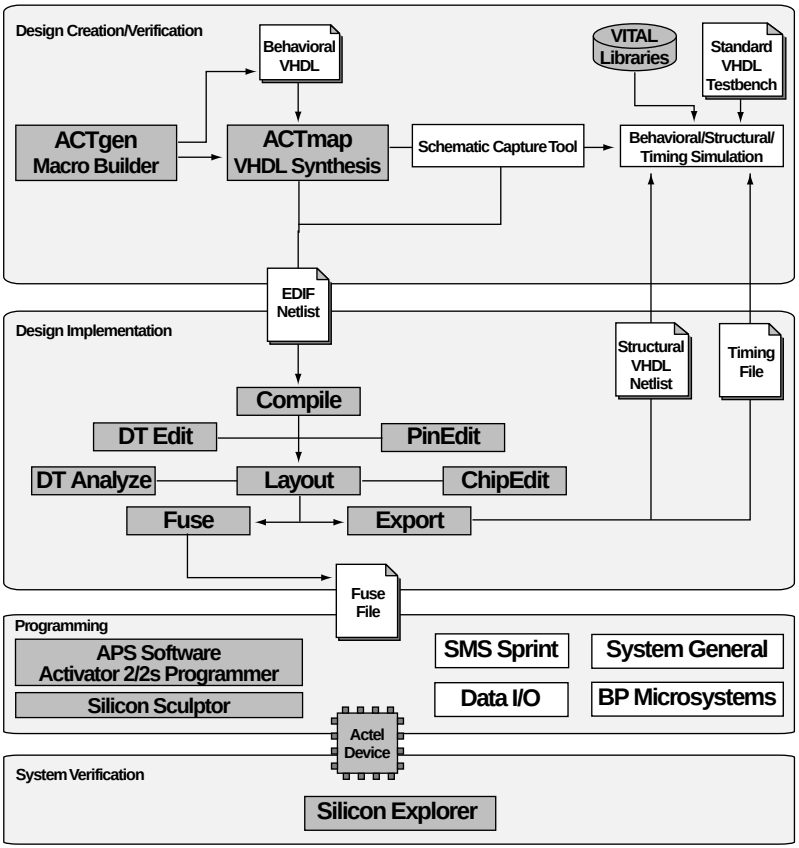


Figure 1-1. ACTmap Design Flow

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

Design Flow Overview

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

Design Creation/ Verification

During design creation/verification, a design is captured in an RTL-level (behavioral) VHDL source file. After capturing the design, behavioral simulation of the VHDL file can be performed to verify that the VHDL code is correct. The code is then synthesized into an Actel gate-level (structural) VHDL netlist using ACTmap. After synthesis, structural simulation of the design can be performed. Finally, an EDIF netlist is generated for use in Designer or a third party CAE tool. A structural VHDL netlist is also generated for timing simulation.

VHDL Design Source Entry

Enter your VHDL design source using a text editor or a context-sensitive VHDL editor. Your VHDL design source can contain RTL-level constructs as well as instantiations of structural elements, such as ACTgen macros. Refer to Chapter 2, "Using ACTmap VHDL" on page 5, for information about ACTmap VHDL coding techniques.

Behavioral Simulation

Perform a behavioral simulation of your design before synthesis. Behavioral simulation verifies the functionality of your VHDL code. Typically, unit delays are used and a standard VHDL test bench can be used to drive simulation. Refer to the documentation included with your simulation tool for information about performing behavioral simulation.

Synthesis

After you have created your behavioral VHDL design source, create a project in ACTmap and synthesize your design before placing and routing it in Designer. Synthesis transforms the behavioral VHDL file into a gate-level netlist and optimizes the design for a target technology. Refer to the *Designing with Actel* manual for information about synthesizing a design in ACTmap.

EDIF Netlist Generation

After you have created, synthesized, and verified your design, you must generate an EDIF netlist for place and route in Designer. ACTmap also can be used to create symbols and wire files for use in Viewlogic tools. Most third party CAE tools can also import the EDIF netlist as a block for use in a schematic capture tool. Refer to the Actel Interface Guides and the documentation included with your CAE tools for information about importing EDIF blocks.

This EDIF netlist is also used to generate a structural VHDL netlist. Refer to the *Designing with Actel* manual for information about generating an EDIF netlist from ACTmap.

Structural VHDL Netlist Generation

Generate a structural VHDL netlist from your EDIF netlist for use in structural and timing simulation by exporting it from ACTmap. Refer to the *Designing with Actel* manual for information about generating a structural netlist from ACTmap.

Structural Simulation

Perform a structural simulation of your design before placing and routing it. Structural simulation verifies the functionality of your post-synthesis structural VHDL netlist. Default unit delays included in the compiled Actel VITAL libraries are used for every gate. Refer to the documentation included with your simulation tool 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 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 VHDL as the Naming Style when importing the EDIF netlist into Designer. Refer to the *Designing with Actel* manual for information about using Designer.

Timing Analysis

Use the DT Analyze tool in Designer to perform static timing analysis on your design. Refer to the *Designer with Actel* manual for information on using DT Analyze.

Timing Simulation

Perform a timing simulation of your design after placing and routing it. Timing simulation verifies that the design meets timing requirements. Timing simulation requires information extracted from Designer, which overrides default unit delays in the compiled Actel VITAL libraries. Refer to the documentation included with your simulation tool for information about performing timing simulation and the *Designing with Actel* manual for information about extracting timing information from Designer.

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.

2

Using ACTmap VHDL

This chapter provides descriptions and examples of how to write VHDL for use with the ACTmap VHDL synthesis tool. This includes information about VHDL naming conventions and keywords, about declaring circuits and signals in VHDL, and a description of supported operators. Also included is information about using logic conditions and repetitive operations, assigning attributes, and instantiating cells.

Other sections include how to create hierarchical designs for ACTmap, how to infer ACTgen macros, and information about writing processes. State machine design is described and guidelines for using procedures are also given. Finally, supported packages and limitations are listed.

VHDL Naming Conventions and Keywords

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

Naming Conventions

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.
- An entity cannot have the same name as an Actel Library macro.

Keywords

The following is a list of the VHDL reserved keywords that cannot be used for signal or entity names:

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	
configuration	inout	out	sla	
constant	is	package	sra	
disconnect	label	port	srl	

Declaring a Circuit

A circuit description consists of the interface defining the signal connections of the circuit and a description of the circuit's behavior or composition. The interface is referred to as an entity and the signal connections are ports. The section of code that defines the entity behavior or composition is referred to as the architecture. The entity in VHDL is equivalent to a symbol. The architecture is equivalent to a schematic.

Entity Description

An entity consists of the entity name, the names of entity's ports, the direction of the ports (input, output, etc.), and a VHDL signal type for each of the ports. Below is an example entity description for a half adder, illustrated in Figure 2-1.

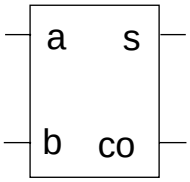


Figure 2-1. Half Adder

```
entity halfadder is
    port (a, b: in bit;
      s, co: out std_logic);
end halfadder;
```

Architecture Description

The behavior or composition of the entity is described in the architecture section of code. The level of the VHDL description of the architecture can be behavioral, register transfer level (RTL), or structural.

A behavioral description describes how the system behaves in response to input signals without regard for hardware implementation.

An RTL description defines the circuit behavior, much like a detailed block diagram describes traditional logic design. Clock and reset signals are defined, and data busses and storage devices (registers, counters, memory, etc.) have specific numbers of bits assigned. However, the level of abstraction used to describe the logic functions is higher than the gate-level details of a conventional schematic.

A structural description is used in the same manner as a netlist.

The structure of the design is described by components interconnected by signals. Regardless of the VHDL description level, the syntax of the architecture must have a type and an entity association, as in the following example:

```
architecture behavioral of example is
    ... signals and constants are declared...
begin
    ... lines of code describing the behavior of entity exam-
ple...
end example;
```

Below is an architectural description of the functionality of a half adder entity. The half adder is illustrated in Figure 2-2.

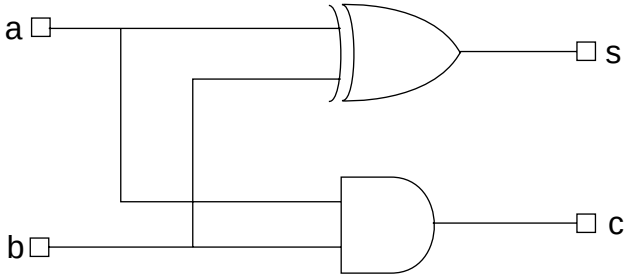


Figure 2-2. Half Adder

```
architecture behavioral of halfadder is
begin
      s <= a xor b;
      c <= a and b;
end behavioral;
```

Signals

Signal declarations are made in the declaration section of the architecture. This is the section of code that appears after the architecture type and entity association have been defined and before the begin statement. Constants are also defined in the declaration section, often after the signals are declared.

As in a physical hardware system, signals can be single bits, such as a clock or a reset, or they can be busses of a specified width. All signals are declared with both a name and a data type. VHDL by itself does not predefine characteristics of signals such as logic states or driving strengths. Instead it makes provisions for doing so by means of VHDL files grouped together into packages and libraries, which are normally shipped with VHDL simulators or synthesis tools. This section lists the data types that ACTmap supports.

Enumerated Types

Types that have a fixed number of unique states are called enumerated types. You can choose to use one of the standard types or define your own.

Bit and Boolean

The bit and boolean types are standard enumerated types defined as part of ACTmap VHDL and have two states each.

- bit can be ‘0’ or ‘1’

```
signal a: bit;
```

- boolean can be true or false

```
signal a: boolean;
```

User-Defined

Enumerated types are often defined by the user for a specific purpose, such as declaring the states of a state machine. Each state of the defined type must be a unique identifier. The enumerated type must be defined using the following syntax before a signal can be declared of that type.

```
type speedtype is (stop, fast, faster);
```

Once a signal of type speedtype is defined, it can only contain one of the three values. Below is an example of a signal defined as type speedtype:

```
signal speed: speedtype;
```

The following example shows how user defined types are defined and used.

```
architecture behavioral of drive is
    signal light: bit_vector(0 to 1);
    type speedtype is (stop, fast, faster);
    signal speed: speedtype;

    constant red: bit_vector(0 to 1) := “00”;
    constant yellow: bit_vector(0 to 1) := “01”;
    constant green: bit_vector(0 to 1) := “10”;

begin
    with light select
      speed <= stop when red;
        fast when green,
        faster when others;
end behavioral;
```

Std_Logic

Two state types are often not sufficient for most simulations. For unknown values and varying signal strengths, a 9-state logic system, often referred to as MVL9, was adopted as a standard by IEEE. This standard enumerated type is called std_logic. The following states are defined:

- U for uninitialized
- X for unknown
- Z for tri-state
- W for weak strength
- H for high (resistive) - used for open collector outputs
- L for low (resistive) - used for open emitter outputs
- - for don't care

Note: During VHDL compilation, ACTmap treats '0' and 'L' as low, '1' and 'H' as high, and 'U', 'X', 'W' and '-' as don't care.

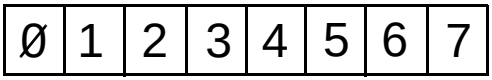
Std_Ulogic

Actel does not recommend using the std_ulogic data type. ACTmap, ACTgen, and Designer do not have the capability to write VHDL netlists using the std_ulogic data type. All VHDL netlists are written using the std_logic data type. Test benches written using the std_ulogic data types generally do not work with gate-level VHDL netlists created by the Designer Series tools.

Vectors

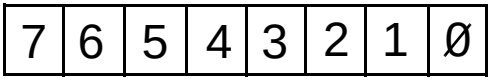
Busses or multibit signals are referred to as vectors. The data types bit and std_logic are definable as vectors. When vectors are defined, a range for the vector array must be declared. The range can be either ascending or descending. For an ascending range, the most significant bit is on the left and is defined using the "to" keyword as follows:

```
signal databus: std_logic_vector(0 to 7);
```



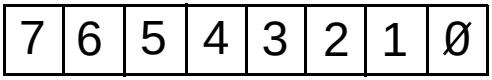
For descending range, the most significant bit is on the right, and is defined using the “downto” keyword as follows:

```
signal databus: std_logic_vector(7 downto 0);
```



To use the entire vector, use the name of the vector as shown:

```
databus
```

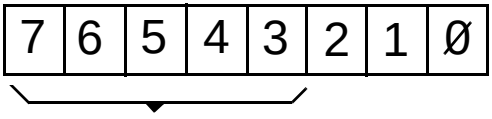


Individual bits of a bus or register are used as shown:

```
databus(4) -- for bit 4
```

A portion of a bus or register is referred to as a slice and is used as shown:

```
databus(7 downto 3)
```




```
Databus (7 downto 3)= “slice”
```

Note: The “to” or “downto” statement of the slice must agree with the to or downto statement of the vector declaration.

Integer Types

An integer type defines the set of integer values in its range. When designing arithmetic behavior, it is very helpful to work with integer types. VHDL pre-defines an integer type called “integer” that covers a range of integer values that can be represented in two’s complement with 32 bits:

```
signal s32_int: integer;
```

An object can also be defined to be a sub-range of an integer:

```
signal s4_int: integer range 0 to 15;
```

The sub-range should be always be defined, otherwise, ACTmap will automatically transfer the integer signal into a 32-bit bus during synthesis.

Constants

Similar to a signal declaration, constants can be declared and given names. The following is an example of a constant declaration:

```
constant yellow: bit_vector := “01”;
```

In this example, the name of the constant is yellow and it is a bit_vector whose value is always “01.” The constant declaration uses a similar syntax to the signal declaration with the addition of the “:=” and the assigned value of the constant. Binary values of vectors are always enclosed in double quotes as shown, and are referred to as bit string literals. You can, but you do not need to, specify how many bits are in the constant with a to or downto statement.

Operators

The standard logical and arithmetic operations that are supported by ACTmap are shown in Table 2-1.

Table 2-1. Supported Arithmetic Operators

Type	Symbol	Operation	Operand Type
Arithmetic	+	addition	integer, bit_vector, std_logic_vector
	-	subtraction	integer, bit_vector, std_logic_vector
	*	multiplication	integer, bit_vector, std_logic_vector
Logical	and	logical and	bit, boolean
	or	logical or	bit, boolean
	nand	logical nand	bit, boolean
	nor	logical nor	bit, boolean
	xor	logical excusive-or	bit, boolean
	not	logical compliment	bit, boolean
Relational	=	equal	any type
	/=	not equal	any type
	<	less than	any type
	>	greater than	any type
	<=	less than or equal	any type
	>=	greater than or equal	any type

Table 2-1. Supported Arithmetic Operators (Continued)

Type	Symbol	Operation	Operand Type
Concatenation	&	concatenation	bit, bit_vector, std_logic, std_logic_vector

The following guidelines should be used when using the operators in your VHDL design:

- Parentheses must be included where the intended function may be ambiguous. The expression “a and b or not (c)” could be interpreted as “a and (b or not(c))” or “(a and b) or not(c)”, which are not equivalent. For example:

  ```
  y <= (a and b) or not (c);
  ```

- The following example would mean a signal assignment without the if keyword. Within the if statement, the operator means less than or equal.

  ```
  if databus_1 <= databus_2 then
  ```

Logic Conditions

The various means of testing for logic conditions include the if, when, select, and case statements. If and case statements are used only in processes. Select and when statements are used only outside of processes.

If Statement

An if statement is a conditional statement that may only be used in a process. The syntax for an if statement is as follows:

```
if condition 1 then
  some action;
elsif condition 2 then
```

```
    some action;
end if;
```

The following is an example of an if-then-else statement that synthesizes a 3 stage shift register and is illustrated in Figure 2-3:

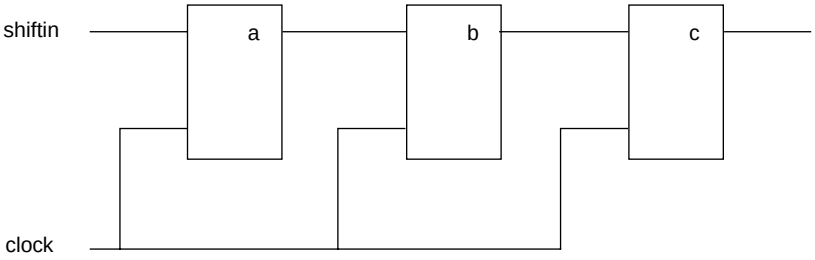


Figure 2-3. 3 Stage Shift Register

```
if (reset = '1') then
    a <= '0';
    b <= '0';
    c <= '0';
elsif (clock'event and clock = '1') then
    a <= shiftin;
    b <= a;
    c <= b;
end if;
```

When Statement

A when statement is a conditional state assignment that synthesizes as combinatorial logic. The syntax for a when statement is as follows:

```
signal <= 'value' when condition
  else 'value';
```

Below is an example of a when statement that synthesizes as an N bit "greater than" comparator:

```
signal <= '1' when databus > register
  else '0';
```

Select Statement

A select statement is a selected signal assignment that synthesizes into combinatorial logic. The syntax for a select statement is as follows:

```
with signal select
  target output <=waveform 1
when condition 1,
  waveform 2 when condition 2,
  waveform N when condition N;
```

Below is an example of a select statement that synthesizes as a four to one multiplexer controlled by "sel," a two-bit control signal, whereby "y" is switched to one of the four data lines:

```
with sel select
  y <= a when "00",
  b when "01",
  c when "10",
  d when "11";
```

Case Statement

A case statement is a selected signal assignment within a process. The syntax for a case statement is as follows:

```
case state is
      when condition 1 => target output <= waveform 1;
      when condition 2=> target output <= waveform 2;
      when condition N=> target output <= waveform N;
      when others => target output <= default waveform;
end case;
```

Below is an example of a case statement that synthesizes the same logic as the select code above when used within an unclocked process:

```
case sel is
      when "00" => y <= a;
      when "01" => y <= b;
      when "10" => y <= c;
      when others => y <= d;
end case;
```

Case statements must specify all possible cases. The “when others =>” statement should be added to VHDL case statements using std_logic data types. The following error message is displayed in ACTmap if all cases are not specified in a case statement:

```
ERROR:   (VHP_0812). Line 29. A value is missing in case
```

Repetitive Operations

Repetitive structures are declared with a generate statement.

If-Generate Statement

The if-generate statement is supported for static (non-dynamic) conditions.

```
loop label: if condition generate
      loop actions;
end generate;
```

For-Generate Statement

The for-generate statement is supported for static (non-dynamic) conditions.

```
loop label: for variable in start condition to end condition
generate
      loop actions;
end generate;
```

For-Generate Loop

For component instantiations, function or procedure calls inside a for generate loop, a block statement has to be used inside the loop to be accepted by the ACTmap VHDL Compiler. The following is an example of a for generate loop inside a block statement:

```
entity example is
...
end example;

architecture arch of example is
  component small
```

```
  ...
  end component;
...
for all: small use entity work.small(arch);
begin
  loop1: for i in 0 to 3 generate
-- begin block statement
    block1: block
    begin
      instance1: small port map ( a(i), b(i), s(i) );
    end block; -this line is added
  end generate loop1;
end arch;
```

Attributes

The ACTmap VHDL Compiler uses the “donttouch” attribute to control synthesis of the described circuit. The “donttouch” attribute directs ACTmap not to optimize a given instance. Before the attribute can be used, it must first be declared with a type. The attribute can only be attached to instances of previously optimized macros or modules.

The syntax for declaring the donttouch attribute is as follows:

```
attribute donttouch : string;
```

The syntax for attaching the “donttouch” attribute is as follows:

```
attribute donttouch of instance label: label is “attribute
value”;
```

Below is an example of adding the value true to the donttouch attribute:

```
for instance actgen_1:
attribute donttouch : string;
attribute donttouch of actgen_1: label is “true”;
```

Note: The value of the attribute is not important in this case.

Instantiating Cells

In order to instantiate an entity into a VHDL description, you must first declare a component for it. If you use a component instantiation in your VHDL design, ACTmap tries to find the definition of that component. There are three possibilities for defining an instantiated cell:

1. The component is a cell in the specified Actel macro library.
2. The component has a matching entity in the VHDL source file.
3. The component has no definition.

Library Macros

Components in the Actel macro library are considered black boxes during synthesis since there is no entity/architecture description for them. Actel library cells defined in the VHDL code are not optimized in VHDL Compiler, but are treated as black boxes.

When the optimized EDIF netlist is written, the contents for each macro is completed. An added benefit is that the time needed for optimization of the whole circuit can be reduced, since ACTmap does not have to optimize the implementation of the dedicated functions.

Note: Components found in the specified Actel macro library do not need component statements. ACTmap maintains a compiled version of these component statements.

The following example, illustrated in Figure 2-4, instantiates the ACT 3 “clkint” macro:

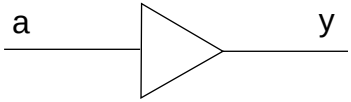


Figure 2-4. CLKINT Symbol

```
clkint_1: clkint port map (signal_a, signal_y);
```

Port mapping may be positional as in the example or it may be done by name. In positional mapping, the signals are associated to the ports by the order the ports are declared in the component declaration. In name mapping, explicitly specify the name of the port followed by the signal tied to it. For name mapping, use any port order. Positional and name mapping are functionally identical. The following is an example of a component instantiation for the clkint macro using name mapping:

```
clkint_1: clkint port map (a => signal_a, y => signal_y);
```

ACTgen Macros

Use the following procedure to instantiate an ACTgen macro into a VHDL description:

1. **Invoke ACTgen.**
2. **Select the family, macro type, and macro options.**
3. **Generate your macro as a VHDL description.** Make sure you specify VHDL as the Netlist/CAE Format when generating the macro.
4. **Add a component declaration in the entity of your VHDL description for the macro.** For example:

```
entity cnt4 is
  port(
        data : in std_logic_vector (3 downto 0);
        enable, sload, aclr, clock : in std_logic;
        q : out std_logic_vector (3 downto 0));
end cnt4;
```

5. **Instantiate the macro into your VHDL description using a port map statement.** For example:

```
u1 : cnt4 port map (signaldata => data, signalenable =>
  enable, signalsload => sload, signalaclr => aclr,
  signalclock =>clock, signalq => q);
```

6. **Compile your VHDL description.** Refer to "Implementing a Hierarchical Project" in the *Designing with Actel* manual for information about compiling a VHDL description in ACTmap.

Refer to the *Designing With Actel* manual or the ACTgen on-line help for information about using ACTgen.

Creating Hierarchy

Up to this point, the discussions have focussed on logic circuits that most likely would be part of a single VHDL entity or functional block of logic. These blocks should generally be limited in size so that they can be synthesized and simulated relatively quickly. Most FPGA designs consist of multiple entities or logic blocks. Hierarchical designs can be created using a structural VHDL description.

Consider the schematic of a full adder that consists of two half adders, shown in Figure 2-5.

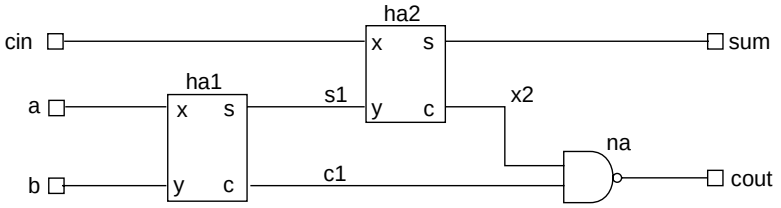


Figure 2-5. Full Adder Schematic

This design uses two separate instances of half adders, designated as components “ha1” and “ha2.” Like any other schematic, the actual signals connected to the component pins may have different names than the individual entity’s port names. Without this capability, you would not be able to use an entity more than once in a design.

The following is the entity and architecture description for the half adder logic block:

```
entity halfadder is
  port (x, y: in bit;
    s, c: out bit);
end halfadder ;

architecture behavioral of halfadder is
begin
    s <= x xor y;
    c <= x and y;
end behavioral;
```

The architecture of the full adder, top-level logic block, is a structural VHDL interconnection of the entity components and signals. The architecture consists of two parts: the declaration and instantiation sections.

Declaration Section

The declaration section includes Signal declarations for signals such as "c1" that are internal to the full adder entity, component declarations for the "parts" used, constant declarations, if needed, and configuration declarations to bind all instances to the desired architecture description.

Instantiation Section

The instantiation section includes Component Instances, such as "ha1," "ha2," etc., with their specific signal connections defined in a port map declaration.

The entity and architecture for a full adder are as follows:

```
architecture structural of fulladder is
signal c1, c2, s1: bit;

component halfadder
  port (
      x, y: in bit;
      s, c: out bit);
end component;

begin
      ha1: halfadder port map (a, b, s1, c1);
      ha2: halfadder port map (x=>cin, y=>s1, s=>sum, c=>c2);
      cout <= c1 nand c2;
end structural;
```

Note: Notice that instance "ha1" was instantiated using positional mapping, and instance "ha2" was instantiated using name mapping.

Inferring ACTgen Macros

Macros such as counters, multiplexers, adders and subtractors can be described in your VHDL code and created using the ACTgen Macro Builder. This section describes how to infer different types of ACTgen macros.

Counters

ACTmap recognizes counters from the VHDL specification and calls ACTgen to generate an optimized counter for the final design. The following guidelines apply to the previous example for inferring a counter:

- The data_load and the data signals can be of type bit_vector, std_logic_vector, or unsigned.
- The reset, sload, and updown signals are optional. However, when used they must be a simple name comparison to '1' or '0'.
- The data must be set to a constant value upon reset.
- The count may only increment or decrement by one.
- An ACTgen macro is not be inferred if the asynchronous load signals exist.
- Counters that use both enable and synchronous load can not be inferred.

Counters, including those requiring an asynchronous reset, a synchronous load, count enable, and up-down count facilities, can be inferred with the following general counter template:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity gencount is
    generic ( msb : integer := 7);
      port (
        clk: in    std_logic ;
        reset: in    std_logic ;
        sload: in    std_logic ;
        updown: in    std_logic;
        load_data: in std_logic_vector ( msb downto 0);
```

```
        data: inout std_logic_vector ( msb downto 0));
end gencount;

architecture template of gencount is
    signal data: std_logic_vector (msb downto 0);
begin
      counter_1: process (clk, reset, sload)
      begin
        if ( reset =  '0' ) then
          data <= "00000000";
        elsif ( clk'event and clk = '1' ) then
          if (load = '0') then
            data <= load_data;
          elsif (updown = '1') then
            data <= data + '1' ;
          else
            data <= data - '1';
          end if;
        end if;
      end process counter_1;
end template;
```

The template can be more flexible. For example, the “reset” statement can be:

```
“if (reset1=’0’ and reset2=’1’)”
```

Multiplexers

ACTmap recognizes multiplexers from the VHDL specification and calls ACTgen to generate optimized macros for the final design. From 2 to 32 inputs may be multiplexed with busses of up to 24 bits for every input in ACT 1 and 40MX, and busses up to 32 bits for every input in all other device families. There are three templates that may be used to infer multiplexers in VHDL and generate them with the ACTgen Macro Builder.

The following guidelines apply to the examples for inferring a multiplexer:

- If more than half of the data inputs are constant values, the macro block is not inferred, because simplifications are possible in the netlist due to this constant value.
- Although std_logic_vector is allowed in the template, the “-” (don’t

care value) is not considered.

- When using “if” or “case” statements to infer a multiplexer, some values of the selector may be unspecified (no “else” or no “when others” statement).

Multiplexer Using an If Statement

In this example, a 6 to 1 multiplexer, illustrated in Figure 2-6, is generated by ACTgen. Some logic is generated and connected to the selection port of the multiplexer in order to select the correct inputs according to the values of signals “a,” “b,” “c,” and “d.”

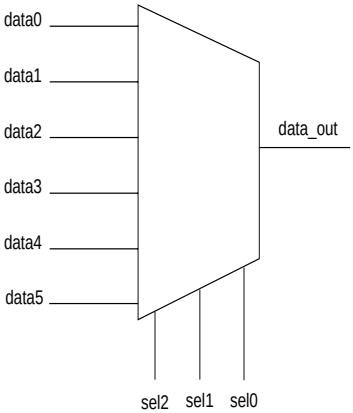


Figure 2-6. Multiplexer Using an If Statement

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity genmx1 is
port (
      data0: in    bit_vector(11 downto 0);
      data1: in    bit_vector(11 downto 0);
      data2: in    bit_vector(11 downto 0);
      data3: in    bit_vector(11 downto 0);
      data4: in    bit_vector(11 downto 0);
      data5: in    bit_vector(11 downto 0);
```

```
      data_out: out bit_vector(11 downto 0);
      a: in    bit_vector(2 downto 0);
      b: in    bit_vector(1 downto 0);
      c: in    bit;
      d: in    bit);
end genmx1;

architecture template of genmx1 is
begin
      mux: process
(a,b,c,d,data0,data1,data2,data3,data4,data5)
      begin
        if ( a(0) =  '0' ) then
          data_out <= data0;
        elsif ( b = '0' ) then
          data_out <= data1;
        elsif ( c = '1' ) then
          data_out <= data2;
        elsif ( a(2 downto 1) = "11" ) then
          data_out <= data3;
        elsif ( d = '0' ) then
          data_out <= data4;
        else
          data_out <= data5;
        end if;
      end process mux;
end template;
```

Multiplexer Using a With Statement

In this example, an 8 to 1 multiplexer, illustrated in Figure 2-7, is generated by ACTgen. The “sel” signals are connected to the selection ports directly.

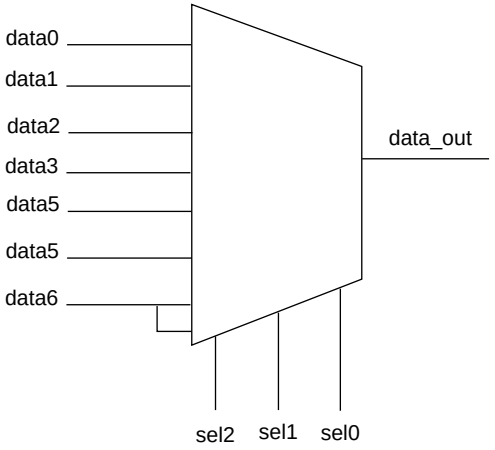


Figure 2-7. Multiplexer using a With or Case Statement

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity genmx2 is
port (
  data0: in    bit_vector(11 downto 0);
  data1: in    bit_vector(11 downto 0);
  data2: in    bit_vector(11 downto 0);
  data3: in    bit_vector(11 downto 0);
  data4: in    bit_vector(11 downto 0);
  data5: in    bit_vector(11 downto 0);
  data6: in    bit_vector(11 downto 0);
  data_out: out bit_vector(11 downto 0);
  sel: in    bit_vector(2 downto 0));
end genmx2;

architecture template of genmx2 is
begin
  with sel select
    data_out <= data6 when “110” | “111”,
        data2 when “010”,
```

```
        data1 when "001",
        data5 when "101",
        data0 when "000",
        data3 when "011",
        data4 when others;
end template;
```

Multiplexer Using a Case Statement

In this example, an 8 to 1 multiplexer, illustrated in Figure 2-7, is generated by ACTgen. The "sel" signals are connected to the selection ports directly.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity genmx3 is
port (
      data0: in    bit_vector(11 downto 0);
      data1: in    bit_vector(11 downto 0);
      data2: in    bit_vector(11 downto 0);
      data3: in    bit_vector(11 downto 0);
      data4: in    bit_vector(11 downto 0);
      data5: in    bit_vector(11 downto 0);
      data6: in    bit_vector(11 downto 0);
      data_out: out bit_vector(11 downto 0);
      sel: in    bit_vector(2 downto 0));
  end genmx3;

architecture template of genmx3 is
begin
  mux: process (data0,data1,data2,data3,data4,data5,data6,sel)
begin
    case sel is
      when "110" | "111" =>
        data_out <= data6;
      when "010"=>
        data_out <= data2;
      when "001"=>
        data_out <= data1;
      when "101"=>
        data_out <= data5;
      when "000"=>
        data_out <= data0;
      when "011"=>
        data_out <= data3;
```

```
      when others =>
        data_out <= data4;
    end case;
end process;
end template;
```

Adders and Subtractors

ACTmap recognizes adders and subtractors from the VHDL specification and calls ACTgen to generate optimized macros for the final design. The following guidelines apply when inferring adders and subtractors:

- The data_a, data_b, and data_out signals can be of type bit_vector, std_logic_vector, or unsigned.
- The output data, “data_out,” must have a size equal to the input data busses, “data_a” and “data_b.”

Adders and subtractors can be inferred by using the following subtractor example (to infer an adder, change the “-” operator to “+”):

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity gensub is
  generic ( msb : integer := 7);
port (
      data_a: in std_logic_vector ( msb downto 0);
      data_b: in std_logic_vector ( msb downto 0);
      data_out: out std_logic_vector ( msb downto 0));
end gensub;

architecture template of gensub is
begin
      data _out <= data_a - data_b;
end template;
```

Multipliers

ACTmap recognizes multipliers from the VHDL specification and calls ACTgen to generate optimized macros for the final design. The following guidelines apply when inferring multipliers:

- Multiplier inferring is not supported for ACT1, and 40 MX devices.

- the “A” and “B” signals can be of type bit_vector, std_logic_vector, or unsigned.
- An ACTgen macro is inferred when “A * B” is found (if both “A” and “B” are not a constant).
- An ACTgen macro is only inferred if the following constraints are met; 2 <= WidthA <= 29, 2 <= WidthB <= 29, and WidthA + WidthB <= 32.
- If WidthA < 2 or WidthB < 2, ACTmap does not work.
- If WidthA > 29, or WidthB > 29, or WidthA + WidthB > 32, which is beyond the ACTgen limitation, ACTmap does not infer an ACTgen macro. Library based synthesis is employed instead.
- Although in ACTgen WidthA must be greater than or equal to WidthB, this limitation does not apply when inferring a multiplier.

Multipliers can be inferred by using the following example:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity testmultiplier is

  port  (a : in std_logic_vector ( 3 downto 0 ) ;
         b : in std_logic_vector ( 4 downto 0 ) ;
         product : out std_logic_vector ( 8 downto 0 ) ) ;
end testmultiplier;

architecture template of testmultiplier is
  begin
    product <= a * b ;
end template;
```

Incrementers and Decrementers

ACTmap recognizes incrementers and decrementers from the VHDL specification and calls ACTgen to generate optimized macros for the final design. The following guidelines apply when inferring incrementers and decrementers:

- Incrementer and Decrementer inferring is not supported in ACT1 and 40 MX devices.

- The “a” and “p” signals can be of type bit_vector, std_logic_vector, or unsigned.
- An ACTgen macro is inferred when p <= a + 1, p <= a - 1, a <= a + 1, and a <= a - 1, when the width of “a” and “p” is 1 < width <= 32.
- An ACTgen macro is inferred when “a + 1” or “a - 1” is found in a top level VHDL specification, function, procedure, and hierarchy model.

Incrementers and decrementers can be inferred by using the following example:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity testinc is
port  (s : in std_logic;
       a : in std_logic_vector(7 downto 0 ) ;
       p : out std_logic_vector(7 downto 0));
end testinc;

architecture template of testinc is
  begin
    proc: process(s, a)
      begin
        if (s = '1') then
            p <= a  + 1 ;
          else
            p <= a - 1 ;
          end if;
        end process;
end template;
```

Accumulator Inferring

The following example infers an Accumulator with only asynchronous clear, enable, and clock:

```
accum : process (clk, reset, enable)

      begin
          if (reset = ‘0’ ) then
              data <= (others=>’0’);
          elsif( clk’event and clk = ‘0’ ) then
              if (enable = ‘0’ ) then
```

```
                data <=data + load_data;
            end if;
        end if;
    end process
```

- The width of the “data” and “load_data” is 2 <= Width <= 24 for act1 and 40 MX devices, and 2 <= Width <= 32 for other families.
- The condition can also be defined using logical parameters as follows: “reset1 = ‘1’ and reset2 = ‘1’.” For example, even if the process is defined as follows, an accumulator can still be inferred:

```
accum : process (clk, reset1, reset2)

    begin
        if (reset1 = ‘1’ and reset2 = ‘0’) then
            data <= (others=>’0’);
        elsif( clk’event and clk = ‘1’ ) then
            if (clk (0) = ‘1’ or clt(1) = ‘0’) then
                data <=data + load_data;
            end if;
        end if;
    end process
```

Comparator Inferring

The following examples infer a comparator:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;


entity testcomp is
port( a : in  std_logic_vector(  8 downto 0) ;
      b : in  std_logic_vector(  8 downto 0) ;
      p : inout std_logic_vector(  8 downto 0) ) ;
end testcomp;
```

```
architecture template of testcomp is
    begin
      processes(a,b)
        begin
          if  ( a > b ) then
            p <=  p + '1';
          else
            p <=  p - '1';
          end if;
        end process;
  end template;
```

or,

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity testcomp is
port( a : in  std_logic_vector(  8 downto 0);
      b : in  std_logic_vector(  8 downto 0);
      p : out std_logic);
end testcomp;

architecture template of testcomp is
    begin
      processes(a,b)
        begin
          if  ( a >=  b ) then
            p <=  '1';
          else
            p <=  '0';
          end if;
        end process;
end template;
```

- All 6 compressions ">, >=,<, <=, =, /=" can be inferred

XOR Tree Inferring

The following example infers an XOR tree with positive polarity:

```
XorP:   Process ( Data )
    Variable t  : Std_Logic;
    Begin
      t :=  '0';
      For i In (Data'Length-1)DownTo 0 Loop
        t :=  t xor Data(i)
        End Loop;
      Result <= t;
    End Process XorP;
or
Result<=d(0) xor d(1) xord(2) xord(3)xor d(4) xor d(5)
xor d(6) xor d(7)
```

The following example infers an XOR tree with the negative polarity:

```
XorP:   Process ( Data )
    Variablet   : Std_Logic;
    Begin
      t :=  '1';
      For i In (Data'Length-1)DownTo 0 Loop
        t :=  t xor Data(i);
        End Loop;
      Result <= t;
    End Process XorP;
```

or

```
Result<= not ( (0)xor d(1) xor d(2) xor d(3) xor d(4) xor d(5)
xor d(6) xor d(7) );
```

The following example infers an xor tree with the negative polarity:

- The width of the data is 4 <= width <= 64.

Processes

Processes are sections of sequentially executed statements. While in the dataflow syntax, all statements are executed concurrently. In a process, the order of the statements does not matter. Processes

resemble the sequential coding style of high-level programming languages.

A process can be called from the dataflow section of VHDL code. Each process is a sequentially executed program, but all processes run concurrently. Processes communicate with each other via signals that are declared in the declaration section of the architecture. The signals that the process waits for are included in the sensitivity list of the process. During the normal flow of a VHDL simulation, the process waits for a change to occur on one of the signals in the sensitivity list. It executes the statements between the begin and the end of the process.

Processes are labeled and use the following syntax:

```
label: process (sensitivity list)
begin
... lines of code describing the behavior of the process...
end process label;
```

The actions described in the process can be of two forms; a clocked process that is synthesized into clocked or sequential logic, or an unclocked process that produces combinatorial logic. Clocked processes always include the clock signals in the sensitivity list.

There are two types of expressions that can be used to infer clocked logic, a 'event attribute or a function call. For example:

```
(clk'event and clk='1')       --rising edge 'event attribute
(clk'event and clk='0')       --falling edge 'event attribute
rising_edge(clock)            --rising edge function call
falling_edge(clock)           --falling edge function call
```

A clock signal cannot use the rising edge procedure if it has been defined as type bit. In order to use the rising edge procedure, the clock must be defined as type std_logic. The following error message is displayed in ACTmap if a rising edge is not properly defined:

```
ERROR:   (VHP_0808). Line 17. rising_edge can not have such
operands in this context.
```

Inferring Multiplexers

The following example, illustrated in Figure 2-8, infers a 2 to 1 multiplexer:

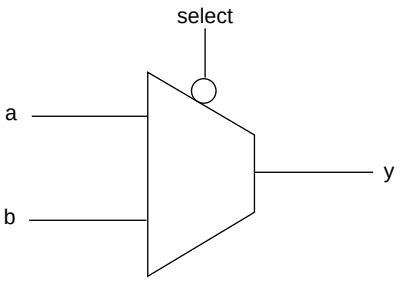


Figure 2-8. 2 to 1 Multiplexer

```
architecture behavioral of mx2 is
begin
  mx2_1: process (a, b, select)
  begin
    if (select = '0') then
      y <= a;
    else
      y <= b;
    end if;
  end process mx2_1;
end behavioral;
```

Inferring Flip-Flops

The following example, illustrated in Figure 2-9, infers a single bit D flip-flop with an active low asynchronous clear:

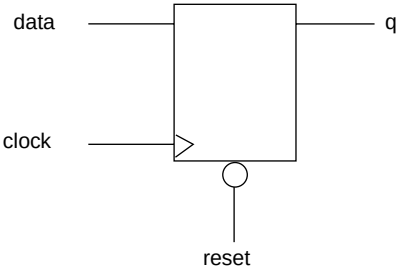


Figure 2-9. Single Bit D Flip-Flop

```
architecture behavioral of flipflop is
```

```
begin
  dff_1: process (clock, reset)
  begin
    if (reset = '0') then
       q <= '0';
    elsif (clock'event and clock = '1') then
       q <= data;
    end if;
  end process dff_1;
end behavioral;
```

Inferring Latches

The following example, illustrated in Figure 2-10, infers a multi-bit D latch with an active high enable and an active low asynchronous clear:

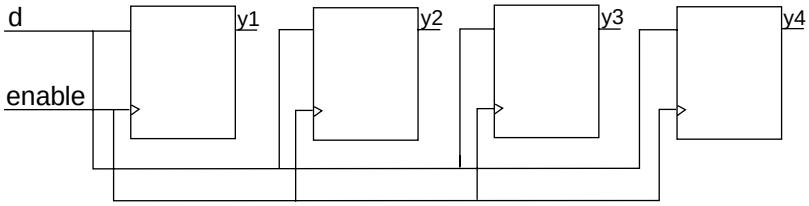


Figure 2-10. Multi-bit D Latch

```
architecture behavioral of latches is
begin
  dlc_1: process (enable, reset, d)
  begin
    if (reset = '0') then
      y <= “0000”;
    elsif (enable = '1') then
      y <= d;
    end if;
  end process dlc_1;
end behavioral;
```

Incomplete Sensitivity Lists

Incomplete sensitivity lists in a process may cause differences in the pre and post synthesis behavior. The following example demonstrates the incorrect way to synthesize a three-input AND gate, because the signal “c” is not in the sensitivity list. Therefore, the variable “y” is not re-evaluated when c changes.

```
architecture behavioral of tand3 is
begin
  and3_1: process (a, b)
  begin
    if (reset = '0') then
      y <= a and b and c;
  end process and3_1;
end behavioral;
```

The correct method to synthesize a three-input and gate is as follows:

```
architecture behavioral of tand3 is
begin
  and3_1: process (a, b, c)
  begin
    if (reset = '0') then
      y <= a and b and c;
  end process and3_1;
end behavioral;
```

Note: ACTmap does not always correctly report missing signals in the sensitivity list of a process. To avoid erroneous results during simulation, make sure that all sensitivity lists do not have missing signals.

Incomplete Construct Value Specification

The if then else and case statements can infer latches instead of multiplexers if all possible states or values are not specified. The following example, illustrated in Figure 2-11, infers a 2 to 1 multiplexer:

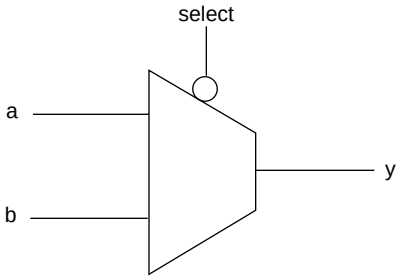


Figure 2-11. 2 to 1 Multiplexer

```
if (select = '0') then
    y <=a;
else
    y <= b;
end if;
```

If you do not specify the else statement, a latch, illustrated in Figure 2-12, is inferred:

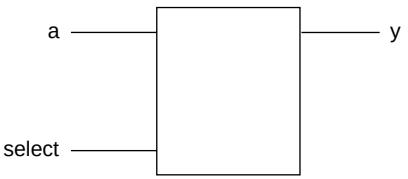


Figure 2-12. Latch Diagram

```
if (select) then
    y <= a;
end if;
```

State Machine Design

A state machine is a sequencer that is organized as a finite set of states. Each state represents one set of actions, such as enabling a counter to increment or generating an acknowledge output. Almost all states also contain a method of transferring control to another state based on certain conditions. Any state that does not have a means of going to another state would have to be the last state of the state machine, and the system would remain in this state forever. Transferring between states can be conditional (based on the values of other signals in the system) or non-conditional.

There are many ways to describe a finite state machine (FSM) in VHDL. The important point is that the synthesis tool should optimize the corresponding logic in an efficient way for both speed and area. This is done by applying optimized automatic state assignments. This section describes three types of FSM, the single-process, the double process, and a user defined FSM.

Single Process FSM

Figure 2-13 illustrates a single-process FSM that controls a traffic light. The sensitivity list of the process contains only two signals: the clock and the reset signals. To describe the transitions between states, a case statement identifies which state is considered. In the case statement, all state register values have to be enumerated in when statements. The state registers may be assigned conditionally in an if statement or not. The conditions are boolean expressions of the input ports.

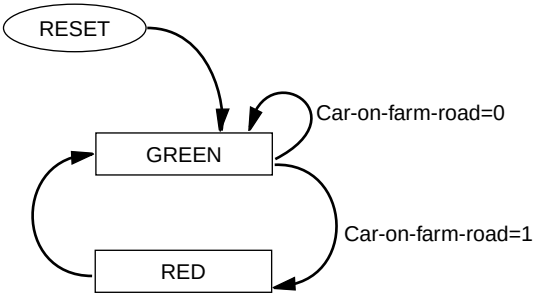


Figure 2-13.

```
library ieee;
use ieee.std_logic_1164.all;

entity light is
  port (
    clock: in std_logic;
    reset: in std_logic;
    car_on_farm_road: in std_logic;
    set_green_on_highway: out std_logic);
end tlight;

architecture state_machine of light is
  type state_type is (red, green);
  signal next_state: state_type;
begin
sequencer: process (reset, clock)
begin
    if (reset = '0') then
      next_state <= green;
      set_green_on_highway <= '1';
    elsif (clock'event and clock = '1') then
      case next_state is
        when green =>
          set_green_on_highway <= '1';
          if (car_on_farm_road = '1') then
```

```
            next_state <= red;
          else
            next_state <= green;
          end if;
        when red =>
          set_green_on_highway <= '0';
          next_state <= green;
      end case;
    end if;
end process sequencer;
end state_machine;
```

Double Process FSM

A common approach to describing an FSM uses two processes. One process defines registers or synchronization. The other process describes the combinatorial logic to define the next state and the outputs.

The first process sets the current state and the registered outputs of the FSM. This process is triggered by the clock and the reset signals. Therefore, it is executed when either signal changes. The FSM must be triggered on the clock edge. You can trigger on either a rising or falling edge. It is not necessary to have a reset, but if a reset signal exists, it must be asynchronous. It may be active high or active low. The process should use the following template:

```
registers: process (clock, reset)
begin
  if (reset = <'1','0'>) then
    ... reset the value of the state ...
    ... optionally reset the registered outputs ...
  elsif (clock'event and clock = <'1','0'>) then
    ... Set the new FSM state ...
    ... Assign values to the registered outputs ...
  end if;
end process registers;
```

Another process updates the present state with the next state and takes care of any combinatorial logic. The process is sensitive to all of the input signals and the signal that maintains the current state. It must also include all the internal signals that affect the output of the process. A case statement typically calculates the next state and the outputs as in the following template:

```
transitions : process (clock, reset)
begin
  ... Assign default values to all unregistered outputs ...
    case present_state is
      when state_0 =>
        output <= <value>;
        next_state <= <value>;
      when state_1 => ...;
      .
      .
      .
      when others => ...;
    end case;
end process registers;
```

Note: A value must be assigned to all unregistered outputs for each state. If you do not assign output values, the FSM maintains the previous values and creates unnecessary latches during synthesis. To avoid this problem, assign a default value to all unregistered outputs at the beginning of this process before the case statement. Default assignments of an if statement within a case statement must be declared explicitly.

The following is an example of a simple Mealy FSM using two processes. Figure 2-14 and Table 2-2 illustrate the example:

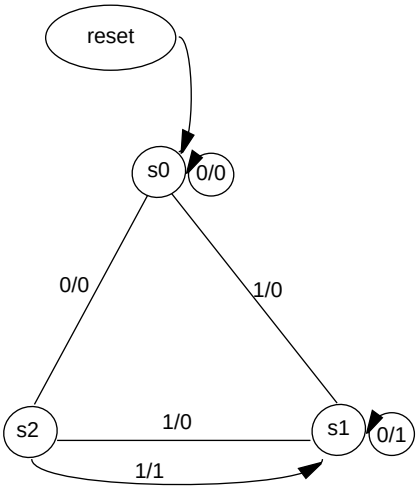


Figure 2-14. Mealy FSM

Table 2-2. Mealy FSM State Table

Present State	Next State		Output	
	x=0	x=1	x=0	x=1
s0	s0	s1	0	0
s1	s1	s2	1	0
s2	s0	s1	0	1

```
library ieee;
use ieee.std_logic_1164.all;

entity mealy is
  port (
      a: in std_logic;
```

```
      clock: in std_logic;
      reset: in std_logic;
      z: out std_logic);
end mealy;

architecture state_machine of mealy is
  type state_type is (s0, s1, s2);
  signal current_state, next_state: state_type;
begin
registers: process (reset, clock)
begin
      if (reset = '0') then
        current_state <= s0;
      else
        if (clock'event and clock = '1') then
          current_state <= next_state;
        end if;
      end if;
end process registers;

combinatorial: process (current_state)
begin
  case current_state is
    when s0 =>
      if (a = '0') then
        z <= '0';
        next_state <= s0;
      else
        z <= '0';
        next_state <= s1;
      end if;
    when s1 =>
      if (a = '0') then
        z <= '1';
        next_state <= s1;
      else
        z <= '0';
        next_state <= s2;
      end if;
    when s2 =>
      if (a = '0') then
        z <= '0';
        next_state <= s0;
      else
        z <= '1';
        next_state <= s1;
      end if;
    end case;
```

```
end process combinatorial;
end state_machine;
```

User Defined FSM

An FSM can also be defined to your specification. Each state is defined as a constant with a unique value. Below is an example of a counter defined as an FSM using user-defined states:

```
entity count4c is
  port (
      reset: in bit;
      clock: in bit;
      s : out bit_vector(3 downto 0);
      load_data : in bit_vector(3 downto 0);
      load : in bit;
      enable : in bit);
end count4c;

architecture state_machine of count4c is
  signal state, next_state: bit_vector(3 downto 0);
  -- usually this can be an integer
  constant state0: bit_vector ( 3 downto 0) := "1111";
  constant state1: bit_vector ( 3 downto 0) := "1110";
  constant state2: bit_vector ( 3 downto 0) := "1101";
  constant state3: bit_vector ( 3 downto 0) := "1100";
  constant state4: bit_vector ( 3 downto 0) := "1011";
  constant state5: bit_vector ( 3 downto 0) := "1010";
  constant state6: bit_vector ( 3 downto 0) := "1001";
  constant state7: bit_vector ( 3 downto 0) := "1000";
  constant state8: bit_vector ( 3 downto 0) := "0111";
  constant state9: bit_vector ( 3 downto 0) := "0110";
  constant statea: bit_vector ( 3 downto 0) := "0101";
  constant stateb: bit_vector ( 3 downto 0) := "0100";
  constant statec: bit_vector ( 3 downto 0) := "0011";
  constant stated: bit_vector ( 3 downto 0) := "0010";
  constant statee: bit_vector ( 3 downto 0) := "0001";
  constant statef: bit_vector ( 3 downto 0) := "0000";
begin
sequential: process (clock, reset)
begin
    if (reset = '1') then
      state <= state0;
    elsif (clock'event and clock = '1') then
      if ( load = '1' ) then
        state <= load_data;
      elsif ( enable = '1' ) then
```

```
        state <= next_state;
      else
        state <= state;
      end if;
    end if;
end process sequential;

combinatorial: process (state)
begin
    case state is
      when state0 =>
        next_state <= state1;
        s <= x"0";
      when state1 =>
        next_state <= state2;
        s <= x"1";
      when state2 =>
        next_state <= state3;
        s <= x"2";
      when state3 =>
        next_state <= state4;
        s <= x"3";
      when state4 =>
        next_state <= state5;
        s <= x"4";
      when state5 =>
        next_state <= state6;
        s <= x"5";
      when state6 =>
        next_state <= state7;
        s <= x"6";
      when state7 =>
        next_state <= state8;
        s <= x"7";
      when state8 =>
        next_state <= state9;
        s <= x"8";
      when state9 =>
        next_state <= statea;
        s <= x"9";
      when statea =>
        next_state <= stateb;
        s <= x"a";
      when stateb =>
        next_state <= statec;
        s <= x"b";
      when statec =>
        next_state <= stated;
```

```
        s <= x"c";
      when stated =>
        next_state <= statee;
        s <= x"d";
      when statee =>
        next_state <= statef;
        s <= x"e";
      when statef =>
        next_state <= state0;
        s <= x"f";
      when others => null;
    end case;
end process combinatorial;
end state_machine;
```

Supported Packages

There are a number of operations in VHDL that occur regularly. An example is translation of vectors to integers and back. For this reason, ACTmap provides packages that define attributes, functions and procedures that are often used. Using the functions and procedures reduces the amount of initial circuitry that is generated, compared to writing the behavior explicitly in a user-defined function or procedure. This reduces the time for compilation and also could result in a smaller circuit implementation due to improved optimization.

Actel Component Packages

Users instantiating Actel macros in their designs do not need to declare the Actel components. ACTmap maintains a compiled version of the component. The VHDL source for these packages is located in the "<actel_install_directory>/lib/actel/vhdl/<act_fam>" directory. The macro component declarations are included by using the following use statement:

```
library <act_fam>;
use <act_fam>.components.all;
```

Note: The component package for the Actel family being targeted should be compiled before the VHDL code referencing the components.

IEEE Packages

ACTmap supports the following IEEE packages:

- std_logic_1164
- std_logic_unsigned
- std_logic_arith

The textio package is not supported.

Using Procedures

This section lists guidelines to follow when using procedures in ACTmap.

Intermediate Signals

When using procedures, ACTmap requires the use of intermediate signals.

The following example does not work:

```
user_procedure(conv_integer(aaa));
```

The procedure should be written as follows:

```
int_aaa <= conv_integer(aaa);
user_procedure(int_aaa);
```

Inout Parameters Not Supported

ACTmap does not support the use of inout parameters for procedures. The parameters must be either in or out. ACTmap displays the following error message if a procedure has an inout parameter:

```
ERROR:   (VHDL_1768). Line 88. Inout parameter not yet
supported.
```

Limitations

This section lists known limitations and unsupported features in ACTmap VHDL

Bi-Directional Buffers

When using bi-directional buffers, make sure the feedback signal connects to internal logic. If the feedback signal is not connected to internal logic, ACTmap changes the BIBUF to OUTBUF.

Preserving Character Case

The “AMP_EDIFUPPER” environment variable, which forces all characters to upper case, is set to “YES” in ACTmap. If you want ACTmap to preserve character case in your VHDL code, you must set the “AMP_EDIFUPPER” environment variable to no.

Event Construct

ACTmap does not support the event construct for a vector bit. The following example produces an error:

```
architecture behavioral of bug is
signal vector : std_logic_vector( 7 downto 0);
begin
    process (vector)
    begin
      if (vector(0)'event )then
        k <= jj;
      else
        k(6 downto 0) <= jj and vector(6 downto 0);
        k(7) <= '1';
      end if
    end process;
end behavioral;
```

The above example should be written as follows:

```
architecture behavioral of bug is
signal vector : std_logic_vector( 7 downto 0);
signal e_vector: std_logic; -this line is added
begin
e_vector <= vector(0); -this line is added
    process (e_vector, vector) -this line is changed
    begin
```

```
      if (e_vector'event )then -this line is changed
        k <= jj;
      else
        k(6 downto 0) <= jj and vector(6 downto 0);
        k(7) <= '1';
      end if
    end process;
end behavioral;
```

Multiple Clock Events

Multiple clock events are not supported. The following examples do not work:

```
if (rst'event and rst = '0') then
    ...
elsif (clk'event and clk = '1') then
    ...
end if;
```

```
if ((rst'event and rst = '0') and (clk'event and clk = '1'))
then
    ...
end if;
```

VHDL 92 and 93

The set of constructs added in the VHDL 92 and 93 specification were intended to be used for simulation purposes and are not supported in ACTmap.

Bus Width Errors

ACTmap does not always correctly report bus width errors. Make sure that all of your bus widths match in your VHDL code or ACTmap may produce erroneous results.

Multi-Dimensional Buses

ACTmap does not support multi-dimensional buses.

Unsupported Data Types

The following data types are not supported: physical, floating point, signed arrays, access, and file. User defined arrays are only supported for a dimension not exceeding 2.

Wait For Time Construct

The “wait for time” construct is not supported.

Loop Statements

The “while...loop... end loop” and the “loop...end loop” statements are not supported.

3

Advanced Optimization Techniques

This chapter describes optimization guidelines and features in ACTmap. This includes ACTmap VHDL and general optimization guidelines, information about optimizing state machines, and using design constraints during optimization. Also included is information about the ACTmap automatic global I/O insertion and sequential remapping features, and information about using special cells in 3200DX and 42MX devices to improve performance. Finally, information about gated macro usage and about where to find radiation environment design techniques is given.

ACTmap VHDL Guidelines

The following are ACTmap VHDL guidelines to assist you in obtaining the best synthesis results possible:

- For the ACT 3 FPGA family devices, Actel recommends that you avoid describing reset and clear flips-flops and latches in your VHDL descriptions. The ACT 3 preset and clear flip-flops cannot be connected to the hardwired global clock networks driven by HCLKBUF. For ACT 2 and ACT 3 family devices, use asynchronous clear latches and flip-flops. The active low asynchronous clear flip-flops and latches are ACTmap's basic building blocks. Using them may result in better optimization.
- Actel recommends that you avoid describing any flips flops and latch configurations that do not have an asynchronous clear input. The JK or toggle flip-flops, without any asynchronous clear or preset (with sequential remapping) feature may not be simulated by your CAE simulator tools.

General Optimization Guidelines

One of the most powerful features of the ACTmap program is its optimization capability. The optimization and mapping technique used in ACTmap is designed to improve the area or speed of most designs targeted for Actel devices. However, this does not mean that the algorithm can improve all designs. This section lists guidelines to keep in mind when optimizing your design with ACTmap.

Logic Design Type

The type of logic used in the design affects how much of the design can be optimized. ACTmap's algorithm produces excellent results when optimizing random logic, but it does not work as well for structured logic blocks such as adders, subtractors, comparators, and accumulators. Many counters, adders, subtractors, decoders and multiplexers can be inferred through VHDL descriptions or generated using the ACTgen Macro Builder. Other logic blocks can also be created by the ACTgen Macro Builder and added to a design. Refer to "Inferring ACTgen Macros" on page 24 and "ACTgen Macros" on page 21 for information about integrating an ACTgen macro into your design.

Design Optimization Level

You must consider whether to optimize the complete design, or only a part of the design (generally, Actel recommends that you use smaller blocks for better optimization). At times it is beneficial to optimize the whole chip because chip optimization can break down the boundaries between the functional blocks. This allows ACTmap to globally consider the logic that is optimized, which often produces better results.

Note: ACTmap may not produce improved results when optimizing highly structured or optimized designs.

Limited Optimization

If you are using an original design that has only structured or optimized sections, you can instruct ACTmap to ignore the optimal sections.

ACTmap ignores optimal sections when you add a donttouch attribute to an instance by editing the EDIF, ADL, or VHDL netlist files. The following example shows the "DONTTOUCH:TRUE" attribute used in an ADL design file:

```
USE FLIP; I1I386; DONTTOUCH:TRUE.
USE ADLIB:OR3; DONTTOUCH:TRUE
```

This example shows an EDIF design file with the “donttouch:true” property added.

```
(instance (rename i1i235”)
(viewref Netlist (cellref or3 (libraryref act2)))
(property donttouch (string “true”) )
)
instance (rename i1i386 “i1i386”)
(viewref netlist (cellref flip (libraryref this_design)))
(property donttouch (string “true”)
)
```

This example shows an ACTmap VHDL design description with the “donttouch:true” property added:

```
architecture structural of example is
  attribute donttouch : string;
  attribute donttouch of u0 : label is “true”;

  component inva
  port (
    a: in bit;
    y: out bit);
  end component;

begin
    u0: inva port map (a, s1);
end structural;
```

Structural VHDL

ACTmap does not optimize structural VHDL netlists. Structural netlists are treated as though they have the “donttouch” attribute added to them. VHDL netlists created by ACTgen are not optimized when they are added as part of a project, they are merged into the project.

Design Size

Memory requirements and ACTmap run time vary with design type. If the ACTmap functions exceed hardware limitations, you may want to optimize by functional block, rather than the whole design. Actel recommends limiting VHDL blocks to less than 1500 logic modules and netlists to less than 800 logic modules.

Memory Size

ACTmap's two primary goals are efficient memory usage and short run-times. However, optimizing certain designs, such as designs with adders, multipliers, and some counters, causes ACTmap to use large amounts of memory. These designs contain highly structured logic blocks. Use ACTgen to build these macros with donttouch attributes, and instantiate them directly into your ACTmap design. You can also use ACTmap to merge the top-level netlist with ACTgen macros after optimization.

State Machine Optimization

ACTmap allows you to select between five state machine encoding algorithms. ACTmap uses the following methods to generate state machines from VHDL source files to netlists.

- **One-Hot**. The One-Hot algorithm reduces each bit in the state machine to a single register for maximum speed.
- **Compact**. The Compact algorithm produces decoded states for minimum area.
- **Gray**. The Gray Code algorithm identifies long paths without branching. It applies successive Gray codes on path nodes.
- **Johnson**. Like the Gray Code, the Johnson algorithm identifies long paths and applies successive Johnson codes on the path nodes.
- **Sequential**. Sequential encoding identifies the long paths and applies successive radix 2 codes on the nodes of the paths. The radix 2 code helps in minimizing area because it can efficiently minimize next-state equation complexity of paths.
- **User Defined**. The User Defined encoding is based on the states defined in the VHDL.

When optimizing smaller designs, optimizing the design for area frequently produces the greatest speed. Actel recommends that you first optimize small designs for area and save the results. You can then optimize the design for speed and compare the results.

Setting Design Constraints

During netlist optimization, preferential treatment can be assigned to a given global design constraint (through the Set Constraint command in the Options menu). The ACTmap Netlist Optimizer will optimize the netlist so that the preferred global constraint receives the greatest amount of optimization. You can only set global constraints through the Netlist Optimizer window. It is not possible to set specific values for global constraints.

The following Global Constraints can be specified:

- **Clock Frequency**. Synthesis is targeted toward the optimization of the clock frequency.
- **Inpad to Outpad**. Synthesis is targeted toward the optimization of paths, starting at an input port and ending at an output port.
- **Inpad to Setup**. Synthesis is targeted toward the optimization of paths, starting at an input port and ending at the setup for latches and flip-flops.
- **Clock to Outpad**. Synthesis is targeted toward the optimization of paths, starting at the clock and ending at an output port.
- **All to Setup**. Synthesis is targeted toward the optimization of all paths, ending at the setup for latches and flip-flops.
- **All to Outpad**. Synthesis is targeted toward the optimization of paths, ending at an output port.
- **Maximum Delay**. Synthesis is targeted toward the minimization of the maximum path delay for the design.

Automatic Global I/O Insertion

ACTmap automatically inserts global I/Os and buffers in all Actel family devices. ACTmap inserts CLKBUF macros to drive the CLKA global network in ACT 1 and 40MX devices, the CLKA and CLKB global networks in ACT 2, 1200XL, 3200DX, and 42MX devices, and the CLKA, CLKB, and HCLK global networks in ACT 3 and 54SX devices.

During insertion, ACTmap inserts CLKBUF macros in all dangling clock network input ports. It inserts INBUF macros in all other dangling input ports, and OUTBUF macros in all dangling output ports.

You can set your I/O insertion commands, and set the automatic I/O insertion commands at the command line or in your .ami file. Refer to "I/O Macros" on page 70 for a description of the commands.

3200DX and 42MX

The 3200DX and 42MX device families have specialized cells and clocks that can be used to improve performance. This section describes how to utilize those specialized cells and clocks.

Wide Decoders and RAM Cells

You can use the wide decoder modules and the RAM cells in ACTmap, but they must be instantiated and their utilization must be monitored by the user. They cannot be inferred. Actel recommends that logic blocks using wide decoders and RAM cells are generated using ACTgen and instantiated into the design.

Quad Clocks

The quad clock modules may be utilized using ACTmap, but they must be instantiated, and their utilization must be monitored by the user. They cannot be inferred.

Sequential Remapping in Netlist Optimization

For almost all ACT 3 flip-flops and some ACT 2 flips-flops, ACTmap performs pre-optimized, sequential remapping. The sequential remapping feature enhances the optimizer performance to take advantage of combinatorial and sequential combining features. It divides sequential library elements into smaller and more basic elements that may generate better results during optimization. Sequential remapping applies to both VHDL synthesis and optimization. This feature is available for the ACT2, 3200DX, 42MX, and ACT 3 families.

The following are the sequential remapping options available.

- **All -** All sequential logic modules are remapped into basic ACT 2 and ACT 3 flip-flops before they are optimized. These cells (DFC1B, DFC1D, DF1B, DL1, DL1B, DLC and DLCA) are combinable sequential elements. For example, DFE1C, a D-type flip-flop with active enable and clock, remaps into MX2, a two-to-one multiplexer, and DF1B, a D-type flip-flop with an active low clock. The two input multiplexer can be combined with other combinatorial logic.
- **No -** No sequential remapping is performed.

The following figures demonstrate a sequential remapping process. Figure 3-1 shows two library cells, DFC1E and DFM7A before remapping. Figure 3-2 shows the library cells instantiated and remapped to other sequential cells that are easier to route and combine.

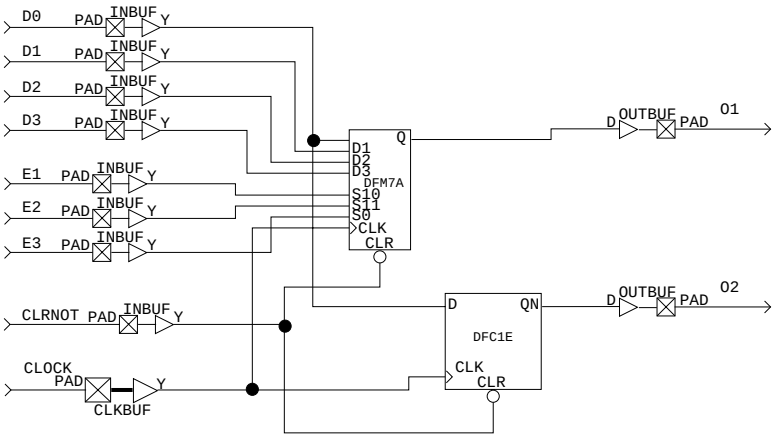


Figure 3-1. Library Cells DFC1E and DFM7A before Remapping

- **Basic** - All sequential logic modules are remapped to basic ACT 2 and ACT 3 flip-flops.
- .**Complex** - Complex flip-flops and latches are remapped to Actel internal logic modules.

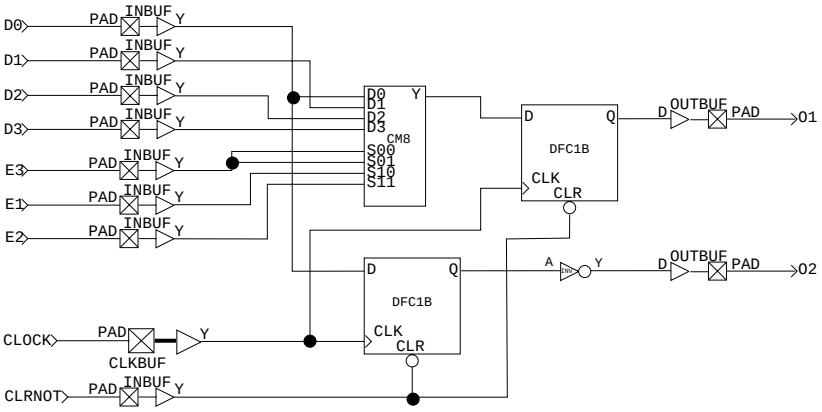


Figure 3-2. Library Cells DFC1E and DFM7A after Remapping

Gated Macros

Gated macros must be instantiated to be utilized because ACTmap cannot synthesize to gated macros. Designs that must gate clocks on the dedicated clock network should utilize the gated macros “gand2,” “gmx4,” “gnand2,” “gnor2,” and “gxor2.”

Designing for Radiation Environments

ACTmap can compile and optimize Actel designs for radiation environments. Refer to *Enhanced Tools for Minimizing Single Event Upset Effects* on the Actel Web site (http://www.actel.com) and the ACTmap On-line Help for additional information.

A

Using ACTmap in Batch Mode

This appendix contains information and procedures for using ACTmap in batch mode. This includes information about invoking ACTmap in batch mode and a description of command line format. Also included is batch file and configuration file creation information. Finally, available batch mode options are listed and usage examples for the options are given.

Invoking ACTmap in Batch Mode

This sections describes the procedures for invoking ACTmap in batch mode.

UNIX

Both ACTmap functions and option settings can be specified on the command line or in the configuration file (ami file). Type the following command at the prompt to invoke ACTmap in batch mode:

```
actmap
```

Microsoft Windows

You can execute ACTmap functions and set options in the ACTmap windows or define options in the configuration file (ami file). Use the following procedure to invoke ACTmap in batch mode:

1. **Create a batch file using a text editor.** Refer to “Creating a Batch File” on page 62 for and example batch file.
2. **Choose the Run command from the Start menu.** The Run dialog box is displayed.
3. **Invoke ACTmap with the complete path of a batch file.** Type the following command in the Run box:

```
actmapw.exe bfile:<batch_file_location>\<batch_file>.bat
```

Command Line Format

The command line format for using ACTmap in batch mode is:

```
actmap [function:{vhdl,netopt,translate}] [<option-
name>:<option_value>] <design_name>
```

The function parameter invokes a specific ACTmap utility. When the function parameter is set to “vhdl,” the ACTmap VHDL Compiler is invoked. The default function parameter is “vhdl.” The <optionname> variable is the name of one of the ACTmap options and the <option_value> variable is a legal value for that option.

All options can be specified in the configuration file. By default, ACTmap reads the <design_name>.ami file located in the project directory, if it exists. To specify another configuration file name or location, use the initfile option.

Creating a Batch File

You can use batch files with ACTmap in Windows. This allows you to run ACTmap for several designs or for one design with different options. The following is an example batch file:

```
actmap infile:\username\designs\decoder.v initfile:\user-
name\designs\decoder1.ami

actmap infile:\username\designs\decoder.vhdl initfile:\user-
name\designs\decoder2.ami

actmap infile:\username\designs\atm.vhdl

actmap infile:\username\designs\counter.vhdl informat:vhdl
state:onehot effort:lo fam:act2 outformat:edif mode:chip
cell:best mapstyle:speed maxfanout:10 counter
```

In the above example, “decoder.v,” “decoder.vhdl,” “atm.vhdl,” and “counter.vhdl” are design file names and formats and “decoder1.ami” and “decoder2.ami” are configuration files. Refer to “Specifying Input and Output Files” on page 66 for information about input and output files and “Creating a Configuration File” on page 63 for information about configuration files.

Creating a Configuration File

All options for each ACTmap function can be specified in the configuration file (ami file). The format for specifying options in the ami file is:

```
fam:act3
mapstyle:area
```

The options in the configuration files are applied to given ACTmap functions as follows:

1. All options specified after the function declaration statement and before another function declaration statement are applied to the given ACTmap function at runtime.

   ```
   function:vhdl
   infile:design.vhd
   fam:act3
   mapstyle:area
   state:onehot

   function:netopt
   infile:design.edn
   fam:act3
   mapstyle:area
   maxfanout:12

   function:translate
   infile:design.edo
   fam:act3
   outformat:verilog
   merge:on
   ```

   The options specified between the “function:vhdl” and the “function:netopt” statements are read by the ACTmap VHDL Compiler. The options specified between the “function:netopt” and the “function:translate” statements are read by the ACTmap Netlist Optimizer. The options specified after the “function:translate” statement are read by the ACTmap Translator utility.

2. When the “informat” option is set in the configuration file, all other options are applied to that ACTmap utility only. When the “informat” option is set to “vhdl,” “edif,” or “adl” the other options are applied to the VHDL Compiler, and the Netlist Optimizer respectively.

3. When neither the “informat” option nor a function declaration statement has been specified in the configuration file, all options are applied to the ACTmap VHDL Compiler only.

ACTmap Options

Table A-1 describes the available ACTmap batch mode options. The default value for the option is shown in bold.

Table A-1. ACTmap Options

Option Name	Option Values	Applicable Functions	Reference Page
actgenmacro	**on**, blackbox, off	vhdl	page 69
cell	**best**, lm, lib	vhdl, netopt	page 68
clka	<port name of first clkbuf>	vhdl, netopt	page 70
clkb	<port name of second clkbuf>	vhdl, netopt	page 70
edninflavor	**generic**, wv, mentor	netopt, translate	page 67
fam	act1, act2, act3, 3200dx, 1200xl, 40mx, **42mx,** 54sx	vhdl, netopt, translate	page 67
flatten	on, **off**	vhdl	page 69
globalconstraint	**clock_freq**, inpad_to_outpad, inpad_to_gated, clock_to_outpad, all_to_gated, all_to_outpad, max_delay	netopt	page 69

Table A-1. ACTmap Options (Continued)

Option Name	Option Values	Applicable Functions	Reference Page
hclk	<port name of hclkbuf>	vhdl, netopt	page 71
infile	<name of design file to read>	vhdl, netopt, translate	page 66
initfile	<name of configuration file>	vhdl, netopt, translate	page 66
logfile	<log file name>	vhdl, netopt, translate	page 66
mapstyle	**speed,** area	vhdl, netopt	page 68
maxfanout	2 to 24	vhdl, netopt	page 68
merge	**on**, off	translate	page 72
mode	**block**, chip	vhdl, netopt	page 68
outfile	<name of output file>	vhdl, netopt, translate	page 67
outformat	**designer**, vhdl, edif, adl, verilog, vl	translate	page 67
portinstname	match, **unique**	vhdl, netopt	page 71
seqremap	**off,** on	vhdl, netopt	page 68
state	**compact**, onehot, gray, johnson, sequential, user	vhdl	page 69
ff_type	**default**, cc (ACT 2, ACT 3, 3200DX, 42MX only), tmr (ACT 1 and 40MX are not supported)	vhdl, netopt	page 72

Batch Mode Options Usage Examples

This section describes the batch mode options, and gives an example usage of each.

Specifying the Configuration File

Use the “initfile” option to specify the name of the configuration file to be read into ACTmap. Actel recommends using the “.ami” extension. The default value is <design_name>.ami. For example:

```
initfile:config1.ami
```

Specifying the Log File

Use the “logfile” option to specify the name of your log file to store ACTmap output messages. Actel recommends using the “.aml” extension. The default value is <design_name>.aml. For example:

```
logfile:run1.aml
```

Specifying Input and Output Files

The following options are used to specify input and output file names and formats.

Specifying Input File Name

Use the “infile” option to specify the name of your input file. Actel recommends using the following extensions:

- ACTmap VHDL - use “.vhd”
- ADL input file - use “.aal” or “.adl”
- EDIF input file - use “.edn”

For example:

```
infile:<filename>.vhd
infile:<filename>.aal or .adl
infile:<filename>.edn
```

Specifying Output File Format

Use the "outformat" option to specify which output files to generate during netlist translation. The default value is "designer." For example:

```
outformat:vhdl
```

Specifying the EDIF Output Filename

Use the "outfile" option to specify the name of the EDIF file to be written by the VHDL Compiler. Actel recommends using an ".edo" extension for the optimized EDIF netlists written by the VHDL Compiler. For example:

```
outfile:<filename>.edo
```

Specifying EDIF Netlist Flavor

Use the "edninflavor" option to specify the flavor of the EDIF netlist to be read. Use "viewlogic" for Viewlogic, "mgc" for Mentor Graphics and "generic" for all other EDIF netlists. The default value is "generic." For example:

```
edninflavor:viewlogic
```

Specifying the Target Family

This section describes how to specify the target device family.

Specifying the Target Family

Use the "fam" option to specify the target family of the netlist to be retargeted. The default value is "42mx." For example:

```
fam:act3
```

In this example, the retargeted netlist is created using the ACT 3 Family architecture.

Technology Mapping

This section describes how to specify technology mapping options.

Specifying Mapping Approach

Use the "cell" option to specify the mapping. Use "lm" to map to the Actel internal module. All logic is mapped to CM8 or CM8A. Use "lib" to map to predefined Actel library cells. Since not all possible combinations have predefined cells in the library, mapping to the CM8 or CM8A module can produce better results. To allow the Compiler to choose the mapping, specify "best" for the "cell" option. The default value is "best." For example:

```
cell:lib
```

Specifying Optimization Type

Use the "mapstyle" option to specify the type of optimization. Use "area" for area optimization and "speed" for speed optimization. The default value is "speed." For example:

```
mapstyle:speed
```

Specifying the Maximum Fanout

Use the "maxfanout" option to specify the maximum fanout limit during netlist optimization. You can set a fanout range from 2 to 24. The default value is "16" for the ACT 2, ACT 3, 3200DX, 42MX, and SX families. The default value is "10" for the ACT 1 and 40 MX families. For example:

```
maxfanout:8
```

Specifying the Remapping of Sequential Elements

Use the "seqremap" option to specify which sequential elements should be remapped to basic Actel sequential elements before Netlist optimization. Use "all" to remap all sequential elements, "basic" for basic sequential elements, "complex" for complex sequential elements, and "No" for no sequential remapping. The default value is "all" or

"on." Refer to "Sequential Remapping in Netlist Optimization" on page 58 for more information. For example:

```
seqremap:all
```

Specifying the Encoding Algorithm for State Machines

Use the "state" option to specify the encoding algorithm used to map state machines by the VHDL Compiler. The following option values may be specified: "onehot," "compact," "gray," "johnson," "sequential," and "user." The default option is "compact." For a description of these options, refer to "State Machine Optimization" on page 56. For example:

```
state:user
```

In this example, the user-defined encoding specified in the VHDL file is used during optimization.

Specifying ACTgen Macro Usage

Use the "actgenmacro" option to specify when ACTgen macros are to be generated for identified templates. Use "on" to let ACTgen generate any macros it can identify, use "off" to force ACTmap to generate the logic. The default value is "on." For example:

```
actgenmacro:off
```

Preserving Hiearchy

Use the "flatten" option to specify when the design hierarchy will be flattened during VHDL compilation. Use "off" to preserve the hierarchy during VHDL compilation. Use "on" to let ACTmap flatten the design. The default value is "off." For example:

```
flatten:on
```

Specifying Global Optimization Constraints

Use the "globalconstraint" option to specify the optimization constraint. For a description of these options, refer to "Setting Design Constraints" on page 57. The default value is "clock_freq." For example:

```
globalconstraint:clock_freq
```

In this example, synthesis is targeted toward the optimization of the clock frequency.

I/O Macros

The following describes how to specify the options related to inserting I/O macros. Refer to "Automatic Global I/O Insertion" on page 57 for more information.

I/O Buffer Insertion

Use the "mode" option to specify when I/O macros are to be inserted. Use "chip" to add I/O macros to top-level ports that do not have I/O macros already added. Use "block" when no I/O insertion is desired. The default value is "block." For example:

```
mode:chip
```

Specifying the First Clock

Use the "clka" option to specify the first global routed clock port name. A CLKBUF macro is added to the specified port when the "mode" option is set to "chip." For example:

```
clka:clock1
```

In this example, a CLKBUF macro is added to the clock1 port during optimization.

Specifying the Second Clock

Use the "clkb" option to specify the second global routed clock port name. A CLKBUF macro is added to the specified port when the "mode" option is set to "chip." For example:

```
clkb:clock2
```

In this example, a CLKBUF macro is added to the clock2 port during optimization.

Specifying the Hardwired Clock

Use the “hclk” option to specify the global hardwired clock port name. An HCLKBUF macro is added to the specified port when the “mode” option is set to “chip.” This option is only available for ACT 3 and SX devices. For example:

```
hclk:hclock
```

In this example, an HCLKBUF macro will be added to the hclock port during optimization.

Port and Instance Name Matching

Use the “portinstname” option to specify a unique port name and I/O macro instance name. A number of CAE systems do not allow identical names to be used for the port and the I/O macro instance names. Use “unique” to specify that different names be used for the port name and the I/O macro instance name. Use “match” to specify that identical names be used. The default value is “unique.” For example:

```
portinstname:match
```

Block Merging

Use the “merge” option to set specify when external netlists are to be merged. The default value is “on.” For example:

```
merge:on
```

In this example, all referenced ADL and EDIF netlists are merged into the top-level design.

Sequential Type

Use the “ff_type” to specify what implementation of sequential macros to use if you are designing for radiation environments. The “cc” option implements combinatorial macros only (54SX devices to not support combinatorial macros). The “tmr” option implements triple voting macros only. The default value is “default,” which uses standard macros. For example:

```
ff_type:tmr
```

In this example, ACTmap will use triple voting macros only in the design that is being compiled/optimized.

B

Product Support

Actel backs its products with various support services including Customer Service, a Customer Applications Center, 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) 522-4434.
From Northwest U.S.A., call (408) 522-4434.
From Canada, call (408) 522-4480.
From Europe, call (408) 522-4252 or +44 (0) 1256 305600.
From Japan, call (408) 522-4743.
From the rest of the world, call (408) 522-4743.
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/**). 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.

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.

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

Headquarters

Actel Corporation
955 East Arques Avenue
Sunnyvale, California 94086
Toll Free: 888.99.ACTEL

Tel: 408.739.1010
Fax: 408.739.1540

US Sales Offices

California

Bay Area
Tel: 408.328.2200
Fax: 408.328.2358

Irvine
Tel: 949.727.0470
Fax: 949.727.0476

San Diego
Tel: 619.938.9860
Fax: 619.938.9887

Thousand Oaks
Tel: 805.375.5769
Fax: 805.375.5749

Colorado

Tel: 303.420.4335
Fax: 303.420.4336

Florida

Tel: 407.677.6661
Fax: 407.677.1030

Georgia

Tel: 770.831.9090
Fax: 770.831.0055

Illinois

Tel: 847.259.1501
Fax: 847.259.1572

Maryland

Tel: 410.381.3289
Fax: 410.290.3291

Massachusetts

Tel: 978.244.3800
Fax: 978.244.3820

Minnesota

Tel: 612.854.8162
Fax: 612.854.8120

North Carolina

Tel: 919.376.5419
Fax: 919.376.5421

Pennsylvania

Tel: 215.830.1458
Fax: 215.706.0680

Texas

Tel: 972.235.8944
Fax: 972.235.965

International Sales Offices

Canada
Suite 203
135 Michael Cowpland Dr.,
Kanata, Ontario K2M 2E9

Tel: 613.591.2074
Fax: 613.591.0348

France
361 Avenue General de Gaulle
92147 Clamart Cedex

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

Germany
Bahnhofstrasse 15
85375 Neufahrn

Tel: +49 (0)8165.9584.0
Fax: +49 (0)8165.9584.1

Hong Kong
Suite 2206,
Parkside Pacific Place,
88 Queensway

Tel: +011.852.2877.6226
Fax: +011.852.2918.9693

Italy
Via Giovanni da Udine No. 34
20156 Milano

Tel: +39 (0)2.3809.3259
Fax: +39 (0)2.3809.3260

Japan
EXOS Ebisu Building 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150

Tel: +81 (0)3.3445.7671
Fax: +81 (0)3.3445.7668

Korea
135-090, 18th Floor,
Kyoung AmBldg
157-27 Samsung-dong
Kangnam-ku, Seoul

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

Taiwan
4F-3, No. 75, Sec. 1,
Hsin-Tai-Wu Road,
Hsi-chih, Taipei, 221

Tel: +886 (0)2.698.2525
Fax: +886 (0)2.698.2548

United Kingdom
Daneshill House,
Lutyens Close
Basingstoke,
Hampshire RG24 8AG

Tel: +44 (0)1256.305600
Fax: +44 (0)1256.355420

Glossary

ACTgen Macro Builder Software A program developed by Actel to generate custom macros for a specific Actel Family architecture.

architecture VHDL name used for the section of code that defines the behavior or composition of a block.

attribute A VHDL property that can be attached to signals or instances.

behavioral VHDL VHDL code written to describe the functionality of a design without regard for a specific architecture.

binding statement VHDL declaration of entity/architecture pair.

bit Signal type having logic states 0 and 1.

case A VHDL statement used to synthesize a selected signal assignment within a process.

clocked process A VHDL statement used to synthesize circuits with flip flops, registers, latches, or any other type of clocked logic.

compact encoding When states are decoded for minimum area.

component declaration A VHDL statement that references the name and I/O ports of an entity that will be used in a block.

component instantiation The occurrence of an entity in a VHDL block, similar to the placement of a part on a schematic.

configuration file A text file used to assign values to ACTmap options.

constant declaration A VHDL statement defining the type and value of a constant.

dataflow method A style of VHDL code that represents a lower level of abstraction than behavioral VHDL while still not resorting to a true gate- level structure.

design file A text file used to describe the behavior of a design block.

entity A VHDL statement used to identify a functional piece of a system and its I/O connections.

enumerated types VHDL data types that are defined to have a fixed number of unique states.

explicit mapping A VHDL port mapping style that maps the port name to a signal, regardless of the port order. This is also referred to as Named Port Mapping.

function A VHDL subprogram that has only one output. This is used to simplify the coding of repetitive or commonly used circuit operations.

Gray encoding States are defined so that only one bit changes at a time.

implicit mapping A VHDL port-mapping style that maps the a given signal to a given port, based on the port order.

initialization file A text file used to assign values to ACTmap options. This is also referred to as a configuration file.

I/O insertion The automatic addition of I/O buffers to ports not having I/O buffers.

Johnson encoding Like the Gray Code, the Johnson algorithm identifies long paths and applies successive Johnson codes on the path nodes.

keywords Words reserved by the VHDL language.

libraries A convenient mechanism for storing commonly used VHDL functions and for defining data types.

one-hot encoding each bit in the state machine is mapped to a single register for maximum speed.

operators A VHDL keyword or symbol that causes an operation to occur between signals.

overloading A VHDL technique used to define operations between the same and different types of data, thus making it possible to mix integer, bit, and other data types.

package VHDL code that is generally used to define the names and the inputs and outputs of the functions in the library.

port map A list of the specific signals connected to the I/O ports of a instance of an entity.

procedures A VHDL subprogram having multiple outputs used to simplify the coding of repetitive or commonly used circuit operations.

processes A VHDL block of code that waits for some condition to occur and, in response, causes some other action.

reserved words Words reserved by the VHDL language.

resource sharing A style of writing VHDL that takes advantage of commonly shared functions such as adders, thus reducing the number of gates needed to implement a function.

Register Transfer Level (RTL) VHDL VHDL code written to describe the detail behavior of a design, but without regard for the gate-level details.

sensitivity list A list of the signals that a process waits for.

sequential encoding Sequential encoding identifies the long paths and applies successive radix 2 codes on the nodes of the paths. The radix 2 code helps in minimizing area because it can efficiently minimize next-state equation complexity of paths.

sequential remapping A pre-optimization technique that divides sequential library elements into smaller and more basic elements.

slices A portion of a bus or register.

std_ulogic A nine state logic value system. Also referred to as MVL9, for Multi-Valued Logic, 9 states. Actel does not recommend using this state system in VHDL designs.

test bench A VHDL entity used to generate the input signals for the design being tested and to monitor the results at the output ports or at points internal to the entity.

VHDL VHSIC Hardware Description Language developed by the United States Government during the 1980s to support the electronic design communities.

Index

E

M

N

O

P

Q

R

S

T

U

V

W

X