Actel HDL Coding

Style Guide
Table of Contents

Introduction ................................................. ix
Document Organization ..................................... ix
Document Assumptions ..................................... x
Document Conventions ..................................... x
HDL Keywords and Naming Conventions ................... xi
VHDL ................................................................ xii
Verilog ................................................................ xiii
Related Manuals ........................................... xiii

1 Design Flow ..................................................... 1
   Design Flow Illustrated .................................... 1
   Design Flow Overview ..................................... 2
   Design Creation/Verification ............................. 2
   Design Implementation ................................... 3
   Programming ................................................ 4
   System Verification ....................................... 4

2 Technology Independent Coding Styles ................. 5
   Sequential Devices ....................................... 5
   Flip-Flops (Registers) ...................................... 5
   D-Latches ................................................ 13
   Datapath .................................................. 17
   Priority Encoders Using If-Then-Else .................. 17
   Multiplexors Using Case ................................. 19
   Decoders .................................................. 24
   Counters .................................................. 25
   Arithmetic Operators ..................................... 29
   Relational Operators ..................................... 31
   Equality Operator ........................................ 32
   Shift Operators .......................................... 33
   Finite State Machine ..................................... 35
   Mealy Machine ........................................... 37
   Moore Machine .......................................... 41
### Table of Contents

- Input-Output Buffers ........................................ 44
- Tri-State Buffer ............................................. 45
- Bi-Directional Buffer ........................................ 47
- Generics and Parameters ...................................... 49

#### 3 Performance Driven Coding ................................ 51
- Reducing Logic Levels on Critical Paths ..................... 51
- Resource Sharing .............................................. 53
- Operators Inside Loops ......................................... 55
- Coding for Combinability ....................................... 56
- Register Duplication .......................................... 57
- Partitioning a Design .......................................... 60

#### 4 Technology Specific Coding Techniques .................. 61
- Multiplexors .................................................... 61
- Internal Tri-State to Multiplexor Mapping .................... 62
- Registers ....................................................... 64
  - Synchronous Clear or Preset ................................ 65
  - Clock Enabled ................................................ 66
  - Asynchronous Preset ......................................... 68
  - Asynchronous Preset and Clear ................................ 71
- Registered I/Os ................................................ 71
- CLKINT/CLKBUF for Reset and/or High Fanout Networks .... 73
- QCLKINT/QCLKBUF for Medium Fanout Networks ............. 75
- ACTgen Counter ................................................ 75
- Dual Architecture Coding in VHDL ............................. 77
- SRAM .......................................................... 80
  - Register-Based Single Port SRAM .......................... 80
  - Register-Based Dual-Port SRAM ............................. 82
  - ACTgen RAM .................................................. 84
- FIFO .......................................................... 86
  - Register-Based FIFO ......................................... 86
Table of Contents

ACTgen FIFO .................................................. 92

A  Product Support ........................................ 95
   Actel U.S. Toll-Free Line .............................. 95
   Customer Service ...................................... 95
   Customer Applications Center ........................ 96
   Action Facts ........................................... 96
   Web Site .............................................. 96
   Guru Automated Technical Support .................. 97
   HLD Web Site ......................................... 97
   FTP Site ............................................. 97
   Electronic Mail ....................................... 97

Index ....................................................... 99
# List of Figures

<table>
<thead>
<tr>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actel HDL Synthesis-Based Design Flow</td>
<td>1</td>
</tr>
<tr>
<td>D Flip-Flop</td>
<td>6</td>
</tr>
<tr>
<td>D Flip-Flop with Asynchronous Reset</td>
<td>7</td>
</tr>
<tr>
<td>D Flip-Flop with Asynchronous Preset</td>
<td>8</td>
</tr>
<tr>
<td>D Flip-Flop with Asynchronous Reset and Preset</td>
<td>9</td>
</tr>
<tr>
<td>D Flip-Flop with Synchronous Reset</td>
<td>10</td>
</tr>
<tr>
<td>D Flip-Flop with Synchronous Preset</td>
<td>11</td>
</tr>
<tr>
<td>D Flip-Flop with Asynchronous Reset and Clock Enable</td>
<td>12</td>
</tr>
<tr>
<td>D-Latch</td>
<td>13</td>
</tr>
<tr>
<td>D-Latch with Gated Asynchronous Data</td>
<td>14</td>
</tr>
<tr>
<td>D-Latch with Gated Enable</td>
<td>15</td>
</tr>
<tr>
<td>D-Latch with Asynchronous Reset</td>
<td>16</td>
</tr>
<tr>
<td>Priority Encoder Using an If-Then-Else Statement</td>
<td>17</td>
</tr>
<tr>
<td>Multiplexor Using a Case Statement</td>
<td>19</td>
</tr>
<tr>
<td>Basic Structure of a Moore FSM</td>
<td>35</td>
</tr>
<tr>
<td>Basic Structure of a Mealy FSM</td>
<td>36</td>
</tr>
<tr>
<td>Mealy State Diagram</td>
<td>37</td>
</tr>
<tr>
<td>Tri-State Buffer</td>
<td>45</td>
</tr>
<tr>
<td>Bi-Directional Buffer</td>
<td>47</td>
</tr>
<tr>
<td>Single Module Implementation of a Synchronous Clear or Preset Register</td>
<td>65</td>
</tr>
<tr>
<td>Single Module Implementation of a Clock Enabled Register</td>
<td>66</td>
</tr>
<tr>
<td>Asynchronous Preset</td>
<td>69</td>
</tr>
<tr>
<td>Equivalent Asynchronous Preset</td>
<td>69</td>
</tr>
<tr>
<td>Registered I/O Cell</td>
<td>71</td>
</tr>
<tr>
<td>RAM Behavioral Simulation Model</td>
<td>80</td>
</tr>
<tr>
<td>FIFO Behavioral Simulation Mode</td>
<td>86</td>
</tr>
</tbody>
</table>


Introduction

VHDL and Verilog® HDL are high level description languages for system and circuit design. These languages support various abstraction levels of design, including architecture-specific design. At the higher levels, these languages can be used for system design without regard to a specific technology. To create a functional design, you only need to consider a specific target technology. However, to achieve optimal performance and area from your target device, you must become familiar with the architecture of the device and then code your design for that architecture.

Efficient, standard HDL code is essential for creating good designs. The structure of the design is a direct result of the structure of the HDL code. Additionally, standard HDL code allows designs to be reused in other designs or by other HDL designers.

This document provides the preferred coding styles for the Actel architecture. The information is to be used as reference material with instructions to optimize your HDL code for the Actel architecture. Examples in both VHDL and Verilog code are provided to illustrate these coding styles and to help implement the code into your design.

Document Organization

The Actel HDL Coding Style Guide is divided into the following chapters:

- **Chapter 1 - Design Flow** describes the basic design flow for creating Actel designs with HDL synthesis and simulation tools.

- **Chapter 2 - Technology Independent Coding Styles** describes basic high level HDL coding styles and techniques.

- **Chapter 3 - Performance Driven Coding** illustrates efficient design practices and describes synthesis implementations and techniques that can be used to reduce logic levels on a critical path.

- **Chapter 4 - Technology Specific Coding Techniques** describes how to implement technology specific features and technology specific macros for optimal area and performance utilization.

- **Appendix A - Product Support** provides information about contacting Actel for customer and technical support.
Document Assumptions

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

- You are familiar with Verilog or VHDL hardware description language, and HDL design methodology for designing logic circuits.
- 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

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

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

Messages that are displayed on the screen appear as follows:

Screen Message

HDL Keywords and Naming Conventions

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

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

The following is a list of the VHDL reserved keywords:

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

The following naming conventions apply to Verilog HDL designs:

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

The following is a list of the Verilog reserved keywords:

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

This guide refers to the Designer Series documentation, which provides information about designing and programming Actel devices. The Designer Series documentation is available in PDF format on the Designer Series CD ROM and the Actel Web Site.

The following manuals provide additional information about designing and programming Actel FPGAs using HDL design methodology:

ACTmap VHDL Synthesis Methodology Guide. This guide contains information about designing an Actel FPGA using the ACTmap VHDL synthesis tool, including preferred HDL coding styles and design considerations. This guide is available in PDF format on the Designer Series CD ROM and the Actel Web Site.

Synopsys Synthesis Methodology Guide. This guide contains information about designing an Actel FPGA using Synopsys synthesis tools, including preferred HDL coding styles, design considerations, and synthesis constraints. This guide is available in PDF format on the Designer Series CD ROM and the Actel Web Site.


Design Flow

This chapter illustrates and describes the basic design flow for creating Actel designs using HDL synthesis and simulation tools.

Design Flow Illustrated

Figure 1-1 illustrates the HDL synthesis-based design flow for an Actel FPGA using third party CAE tools and Designer software\(^1\).

\(^1\) Actel-specific utilities/tools are denoted in black in Figure 1-1.
Design Flow Overview

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

HDL Design Source Entry

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

Behavioral Simulation

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

Synthesis

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

After you have created, synthesized, and verified your design, you must generate an Actel EDIF netlist for place and route in Designer. This EDIF netlist is also used to generate a structural HDL netlist for use in structural simulation. Refer to the Designer Series documentation for information about generating an EDIF netlist.

**Structural Netlist Generation**

You can generate a structural HDL netlist from your EDIF netlist for use in structural simulation by either exporting it from Designer or by using the Actel “edn2vhdl” or “edn2vlog” program. Refer to the Designer Series documentation for information about generating a structural netlist.

**Structural Simulation**

You can perform a structural simulation of your design before placing and routing it. Structural simulation verifies the functionality of your post-synthesis structural HDL netlist. Default unit delays included in the compiled Actel VITAL libraries are used for every gate. Refer to the documentation included with your simulation tool for information about performing structural simulation.

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

**Place and Route**

Use Designer to place and route your design. Refer to the Designer Series documentation 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 Series documentation for information about using DT Analyze.
Timing Simulation

After placing and routing your design, you perform a timing simulation to verify that the design meets timing constraints. Timing simulation requires timing information exported from Designer, which overrides default unit delays in the compiled Actel VITAL libraries. Refer to the Designer Series documentation for information about exporting timing information from Designer.

Programming

Programming a device requires software and hardware from Actel or a supported 3rd party programming system. Refer to the Designing with Actel manual and the Activator Installation and APS Programming Guide for information on programming an Actel device.

System Verification

You can perform system verification on a programmed device using the Actel ActionProbe or Silicon Explorer. Refer to the Activator Installation and APS Programming Guide or Silicon Explorer Quick Start for information on using Action Probe or Silicon Explorer.
Technology Independent Coding Styles

This chapter describes basic, HDL coding styles and techniques. These coding styles are essential when writing efficient, standard HDL code and creating technology independent designs.

Sequential Devices

A sequential device, either a flip-flop or a latch, is a one-bit memory device. A latch is a level-sensitive memory device and a flip-flop is an edge-triggered memory device.

Flip-Flops (Registers)

Flip-flops, also called registers, are inferred in VHDL using wait and if statements within a process using either a rising edge or falling edge detection expression. There are two types of expressions that can be used, a 'event attribute or a function call. For example:

- \( (\text{clk}'\text{event and clk='1'}) \) -- rising edge 'event attribute
- \( (\text{clk}'\text{event and clk='0'}) \) -- falling edge 'event attribute
- \( \text{rising_edge(clock)} \) -- rising edge function call
- \( \text{falling_edge(clock)} \) -- falling edge function call

The examples in this guide use rising edge 'event attribute expressions, but falling edge expressions could be used. The 'event attribute expression is used because some VHDL synthesis tools may not recognize function call expressions. However, using a function call expression is preferred for simulation because a function call only detects an edge transition (0 to 1 or 1 to 0) but not a transition from X to 1 or 0 to X, which may not be a valid transition. This is especially true if using a multi-valued data type like std_logic, which has nine possible values (U, X, 0, 1, Z, W, L, H, -).

This section describes and gives examples for different types of flip-flops. Refer to “Registers” on page 64 for additional information about using specific registers in the Actel architecture.
**Rising Edge Flip-Flop**

The following examples infer a D flip-flop without asynchronous or synchronous reset or preset. This flip-flop is a basic sequential cell in the Actel antifuse architecture.

![Figure 2-1. D Flip-Flop](image)

**VHDL**

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

entity dff is
port (data, clk : in std_logic;
     q : out std_logic);
end dff;

architecture behav of dff is
begin
process (clk)
begin
if (clk'event and clk = '1') then
q <= data;
end if;
end process;
end behav;
```

**Verilog**

```verilog
module dff (data, clk, q);
    input data, clk;
    output q;
    reg q;
    always @(posedge clk)
    q = data;
endmodule
```
Rising Edge Flip-Flop with Asynchronous Reset

The following examples infer a D flip-flop with an asynchronous reset. This flip-flop is a basic sequential cell in the Actel antifuse architecture.

### VHDL

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

data, clk, reset, q;

entity dff_async_rst is
port (data, clk, reset : in std_logic;
      q : out std_logic);
end dff_async_rst;

architecture behav of dff_async_rst is
begin
process (clk, reset) begin
  if (reset = '0') then
    q <= '0';
  elsif (clk'event and clk = '1') then
    q <= data;
  end if;
end process;
end behav;
```

### Verilog

```verilog
module dff_async_rst (data, clk, reset, q);
  input data, clk, reset;
  output q;
  reg q;
  always @(posedge clk or negedge reset)
    if (~reset)
      q = 1'b0;
    else
      q = data;
endmodule
```

Figure 2-2. D Flip-Flop with Asynchronous Reset
Rising Edge Flip-Flop with Asynchronous Preset

The following examples infer a D flip-flop with an asynchronous preset. Refer to “Registers” on page 64 for additional information about using preset flip-flops with the Actel architecture.

**VHDL**

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

entity dff_async_pre is
  port (data, clk, preset : in std_logic;
        q : out std_logic);
end dff_async_pre;

architecture behav of dff_async_pre is
begin
  process (clk, preset)
  begin
    if (preset = '0') then
      q <= '1';
    elsif (clk'event and clk = '1') then
      q <= data;
    end if;
  end process;
end behav;
```

**Verilog**

```verilog
module dff_async_pre (data, clk, preset, q);
input data, clk, preset;
output q;
reg q;
always @(posedge clk or negedge preset)
  if (~preset)
    q = 1'b1;
  else
    q = data;
endmodule
```
Rising Edge Flip-Flop with Asynchronous Reset and Preset

The following examples infer a D flip-flop with an asynchronous reset and preset. Refer to “Registers” on page 64 for additional information about using preset flip-flops with the Actel architecture.

![Figure 2-4. D Flip-Flop with Asynchronous Reset and Preset](image)

VHDL

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

entity dff_async is
port (data, clk, reset, preset : in std_logic;
    q : out std_logic);
end dff_async;

architecture behav of dff_async is
begin
process (clk, reset, preset) begin
    if (reset = '0') then
        q <= '0';
    elsif (preset = '1') then
        q <= '1';
    elsif (clk'event and clk = '1') then
        q <= data;
    end if;
end process;
end behav;
```
Chapter 2: Technology Independent Coding Styles

Verilog

module dff_async (reset, preset, data, q, clk);
  input clk;
  input reset, preset, data;
  output q;
  reg q;
always @ (posedge clk or negedge reset or posedge preset)
  if (~reset)
    q = 1'b0;
  else if (preset)
    q = 1'b1;
  else q = data;
endmodule

Rising Edge Flip-Flop with Synchronous Reset

The following examples infer a D flip-flop with a synchronous reset.

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity dff_sync_rst is
  port (data, clk, reset : in std_logic;
        q : out std_logic);
end dff_sync_rst;

architecture behav of dff_sync_rst is
begin
  process (clk) begin
    if (clk'event and clk = '1') then
      if (reset = '0') then
        q <= '0';
      else q <= data;
      end if;
    end if;
  end process;
end behav;

Figure 2-5. D Flip-Flop with Synchronous Reset
Sequential Devices

**Verilog**

```verilog
module dff_sync_rst (data, clk, reset, q);
    input data, clk, reset;
    output q;
    reg q;
    always @(posedge clk)
        if (~reset)
            q = 1'b0;
        else
            q = data;
endmodule
```

**Rising Edge Flip-Flop with Synchronous Preset**

The following examples infer a D flip-flop with a synchronous preset.

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

entity dff_sync_pre is
    port (data, clk, preset : in std_logic;
        q : out std_logic);
end dff_sync_pre;

architecture behav of dff_sync_pre is
begin
    process (clk)
        begin
            if (clk'event and clk = '1') then
                if (preset = '0') then
                    q <= '1';
                else
                    q <= data;
                end if;
            end if;
        end process;
end behav;
```

![Figure 2-6. D Flip-Flop with Synchronous Preset](image_url)
Verilog

module dff_sync_pre (data, clk, preset, q);
  input data, clk, preset;
  output q;
  reg q;
  always @ (posedge clk)
    if (~preset)
      q = 1'b1;
    else
      q = data;
endmodule

Rising Edge Flip-Flop with Asynchronous Reset and Clock Enable

The following examples infer a D type flip-flop with an asynchronous reset and clock enable.

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity dff_ck_en is
  port (data, clk, reset, en : in std_logic;
        q : out std_logic);
end dff_ck_en;

architecture behav of dff_ck_en is
begin
  process (clk, reset)
  begin
    if (reset = '0') then
      q <= '0';
    elsif (clk'event and clk = '1') then
      if (en = '1') then
        q <= data;
      end if;
    end if;
  end process;
end behav;
Verilog

module dff_ck_en (data, clk, reset, en, q);
  input data, clk, reset, en;
  output q;
  reg q;
  always @ (posedge clk or negedge reset)
    if (~reset)
      q = 1'b0;
    else if (en)
      q = data;
endmodule

D-Latches

This section describes and gives examples of different types of D-latches.

D-Latch with Data and Enable

The following examples infer a D-latch with data and enable inputs.

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
  port(enable, data: in std_logic;
       y : out std_logic);
end d_latch;

architecture behave of d_latch is
begin
  process (enable, data)
  begin
    if (enable = '1') then
      y <= data;
    end if;
  end process;
end behave;
Verilog

```verilog
module d_latch (enable, data, y);
    input enable, data;
    output y;
    reg y;
    always @(enable or data)
        if (enable)
            y = data;
endmodule
```

D-Latch with Gated Asynchronous Data

The following examples infer a D-latch with gated asynchronous data.

![Figure 2-9. D-Latch with Gated Asynchronous Data](image)

VHDL

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

entity d_latch_e is
port (enable, gate, data : in std_logic;
     q : out std_logic);
end d_latch_e;

architecture behave of d_latch_e is
begin
    process (enable, gate, data) begin
        if (enable = '1') then
            q <= data and gate;
        end if;
    end process;
end behave;
```
**Verilog**

```verilog
module d_latch_e(enable, gate, data, q);
    input enable, gate, data;
    output q;
    reg q;
    always @ (enable or data or gate)
        if (enable)
            q = (data & gate);
endmodule
```

**D-Latch with Gated Enable**

The following examples infer a D-latch with gated enable.

**VHDL**

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

entity d_latch_en is
    port (enable, gate, d: in std_logic;
          q : out std_logic);
end d_latch_en;

architecture behave of d_latch_en is
begin
    process (enable, gate, d) begin
        if ((enable and gate) = '1') then
            q <= d;
        end if;
    end process;
end behave;
```
Chapter 2: Technology Independent Coding Styles

Verilog

module d_latch_en(enable, gate, d, q);
    input enable, gate, d;
    output q;
    reg q;
    always @ (enable or d or gate)
        if (enable & gate)
            q = d;
endmodule

D-Latch with Asynchronous Reset

The following examples infer a D-latch with an asynchronous reset.

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch_rst is
port (enable, data, reset: in std_logic;
    q: out std_logic);
end d_latch_rst;

architecture behav of d_latch_rst is
begin
process (enable, data, reset)
begin
    if (reset = '0') then
        q <= '0';
    elsif (enable = '1') then
        q <= data;
    end if;
end process;
end behav;

Figure 2-11. D-Latch with Asynchronous Reset
Verilog
module d_latch_rst (reset, enable, data, q);
    input reset, enable, data;
    output q;
    reg q;
    always @ (reset or enable or data)
        if (~reset)
            q = 1'b0;
        else if (enable)
            q = data;
endmodule

Datapath

Datapath logic is a structured repetitive function. These structures are modeled in various different implementations based on area and timing constraints. Most synthesis tools generate optimal implementations for the target technology.

Priority Encoders Using If-Then-Else

An if-then-else statement is used to conditionally execute sequential statements based on a value. Each condition of the if-then-else statement is checked in order against that value until a true condition is found. Statements associated with the true condition are then executed and the rest of the statement is ignored. If-then-else statements should be used to imply priority on a late arriving signal. In the following examples, shown in Figure 2-12, signal c is a late arriving signal.

Figure 2-12. Priority Encoder Using an If-Then-Else Statement
Chapter 2: Technology Independent Coding Styles

**VHDL**

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

entity my_if is
port (c, d, e, f: in std_logic;
     s : in std_logic_vector(1 downto 0);
     pout : out std_logic);
end my_if;

architecture my_arc of my_if is
begin
myif_pro: process (s, c, d, e, f)
begin
if s = "00" then
    pout <= c;
elsif s = "01" then
    pout <= d;
elsif s = "10" then
    pout <= e;
else
    pout <= f;
end if;
end process myif_pro;
end my_arc;
```

**Verilog**

```verilog
module IF_MUX (c, d, e, f, s, pout);
    input c, d, e, f;
    input [1:0]s;
    output pout;
    reg pout;
always @(c or d or e or f or s) begin
    if (s == 2'b00)
        pout = c;
    else if (s ==2'b01)
        pout = d;
    else if (s ==2'b10)
        pout = e;
    else
        pout = f;
end
endmodule
```
A case statement implies parallel encoding. Use a case statement to select one of several alternative statement sequences based on the value of a condition. The condition is checked against each choice in the case statement until a match is found. Statements associated with the matching choice are then executed. The case statement must include all possible values for a condition or have a default choice to be executed if none of the choices match. The following examples infer multiplexors using a case statement. Refer to “Multiplexors” on page 61 for additional information about using multiplexors with the Actel architecture.

VHDL synthesis tools automatically assume parallel operation without priority in case statements. However, some Verilog tools assume priority, and you may need to add a directive to your case statement to ensure that no priority is assumed. Refer to the documentation provided with your synthesis tool for information about creating case statements without priority.

![Figure 2-13. Multiplexor Using a Case Statement](image-url)
Chapter 2: Technology Independent Coding Styles

4:1 Multiplexor

The following examples infer a 4:1 multiplexor using a case statement.

**VHDL**

```vhdl
--4:1 Multiplexor
library IEEE;
use IEEE.std_logic_1164.all;

entity mux is
    port (c, d, e, f : in std_logic;
          s       : in std_logic_vector(1 downto 0);
          muxout  : out std_logic);
end mux;

architecture my_mux of mux is
begin
    mux1: process (s, c, d, e, f)
    begin
        case s is
            when "00" => muxout <= c;
            when "01" => muxout <= d;
            when "10" => muxout <= e;
            when others => muxout <= f;
        end case;
    end process mux1;
end my_mux;
```

**Verilog**

```verilog
//4:1 Multiplexor
module MUX (C, D, E, F, S, MUX_OUT);
    input C, D, E, F;
    input [1:0] S;
    output MUX_OUT;
    reg MUX_OUT;
    always @(C or D or E or F or S)
    begin
        case (S)
            2'b00 : MUX_OUT = C;
            2'b01 : MUX_OUT = D;
            2'b10 : MUX_OUT = E;
            default : MUX_OUT = F;
        endcase
    end
endmodule
```
12:1 Multiplexor

The following examples infer a 12:1 multiplexor using a case statement.

**VHDL**

```vhdl
-- 12:1 mux
library ieee;
use ieee.std_logic_1164.all;

-- Entity declaration:
entity mux12_1 is
port
(
  mux_sel: in std_logic_vector (3 downto 0);-- mux select
  A: in std_logic;
  B: in std_logic;
  C: in std_logic;
  D: in std_logic;
  E: in std_logic;
  F: in std_logic;
  G: in std_logic;
  H: in std_logic;
  I: in std_logic;
  J: in std_logic;
  K: in std_logic;
  M: in std_logic;
  mux_out: out std_logic -- mux output
);
end mux12_1;

-- Architectural body:
arithmetic synth of mux12_1 is
begin
  begin
    case mux_sel is
      when "0000" => mux_out <= A;
      when "0001" => mux_out <= B;
      when "0010" => mux_out <= C;
      when "0011" => mux_out <= D;
      when "0100" => mux_out <= E;
      when "0101" => mux_out <= F;
    end case;
  end process procl;
end;
```
22

when "0110"  => mux_out <= G;
when "0111"  => mux_out <= H;
when "1000"  => mux_out <= I;
when "1001"  => mux_out <= J;
when "1010"  => mux_out <= K;
when "1011"  => mux_out <= M;
when others => mux_out <= '0';
end case;
end process proc1;
end synth;

Verilog

// 12:1 mux
module mux12_1(mux_out,
               mux_sel,M,L,K,J,H,G,F,E,D,C,B,A);

output mux_out;
input [3:0] mux_sel;
input M;
input L;
input K;
input J;
input H;
input G;
input F;
input E;
input D;
input C;
input B;
input A;
reg mux_out;

// create a 12:1 mux using a case statement
always @ ([mux_sel[3:0]] or M or L or K or J or H or G or F or E or D or C or B or A)
begin: mux_blk
  case {mux_sel[3:0]} // synthesis full_case parallel_case
    4'b0000 : mux_out = A;
    4'b0001 : mux_out = B;
    4'b0010 : mux_out = C;
    4'b0011 : mux_out = D;
    4'b0100 : mux_out = E;
    4'b0101 : mux_out = F;
  endcase
end
Datapath

Case X Multiplexor

The following Verilog example infers a multiplexor using a don’t care case x statement. Actel does not recommend using don’t care case x statements in VHDL. VHDL synthesis tools do not typically support the don’t care value as well as Verilog tools.

```verilog
// 8 bit 4:1 multiplexor with don't care X, 3:1 equivalent mux
module mux4 (a, b, c, sel, q);
input [7:0] a, b, c;
input [1:0] sel;
output [7:0] q;
reg [7:0] q;

always @ (sel or a or b or c)
case (sel)
  2'b00: q = a;
  2'b01: q = b;
  2'b1x: q = c;
  default: q = c;
endcase
endmodule
```
Decoders

Decoders are used to decode data that has been previously encoded using binary or another type of encoding. The following examples infer a 3-8 line decoder with an enable.

**VHDL**

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

entity decode is
  port ( Ain : in std_logic_vector (2 downto 0);
         En : in std_logic;
         Yout : out std_logic_vector (7 downto 0));
end decode;

architecture decode_arch of decode is
begin
  process (Ain)
  begin
    if (En='0') then
      Yout <= (others => '0');
    else
      case Ain is
        when "000" => Yout <= "00000001";
        when "001" => Yout <= "00000010";
        when "010" => Yout <= "00000100";
        when "011" => Yout <= "00001000";
        when "100" => Yout <= "00010000";
        when "101" => Yout <= "00100000";
        when "110" => Yout <= "01000000";
        when "111" => Yout <= "10000000";
        when others => Yout <= "00000000";
      end case;
    end if;
  end process;
end decode_arch;
```
Verilog

module decode (Ain, En, Yout);
    input En;
    input [2:0] Ain;
    output [7:0] Yout;
    reg [7:0] Yout;

    always @(En or Ain)
    begin
        if (!En)
            Yout = 8'b0;
        else
            case (Ain)
                3'b000 : Yout = 8'b00000001;
                3'b001 : Yout = 8'b00000010;
                3'b010 : Yout = 8'b00000100;
                3'b011 : Yout = 8'b00001000;
                3'b100 : Yout = 8'b00010000;
                3'b101 : Yout = 8'b00100000;
                3'b110 : Yout = 8'b01000000;
                3'b111 : Yout = 8'b10000000;
                default : Yout = 8'b00000000;
            endcase
    end
endmodule

Counters

Counters count the number of occurrences of an event that occur either randomly or at uniform intervals. You can infer a counter in your design. However, most synthesis tools cannot infer optimal implementations of counters higher than 8-bits. If your counter is in the critical path of a speed and area critical design, Actel recommends that you use the ACTgen Macro Builder to build a counter. Once generated, instantiate the ACTgen counter in your design. Refer to “ACTgen Counter” on page 75 for examples of ACTgen counter instantiation. The following examples infer different types of counters.
8-bit Up Counter with Count Enable and Asynchronous Reset

The following examples infer an 8-bit up counter with count enable and asynchronous reset.

**VHDL**

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

entity counter8 is
  port (clk, en, rst : in std_logic;
        count : out std_logic_vector (7 downto 0));
end counter8;

architecture behav of counter8 is
  begin
    process (clk, en, cnt, rst)
    begin
      if (rst = '0') then
        cnt <= (others => '0');
      elsif (clk'event and clk = '1') then
        if (en = '1') then
          cnt <= cnt + '1';
        end if;
      end if;
    end process;
    count <= cnt;
  end behav;
```

**Verilog**

```
module count_en (en, clock, reset, out);
  parameter Width = 8;
  input clock, reset, en;
  output [Width-1:0] out;
  reg [Width-1:0] out;

  always @(posedge clock or negedge reset)
    if(!reset)
      out = 8'b0;
    else if(en)
      out = out + 1;
endmodule
```
8-bit Up Counter with Load and Asynchronous Reset

The following examples infer an 8-bit up counter with load and asynchronous reset.

**VHDL**

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

entity counter is
  port (clk, reset, load: in std_logic;
        data: in std_logic_vector (7 downto 0);
        count: out std_logic_vector (7 downto 0));
end counter;

architecture behave of counter is
begin
  process (clk, reset)
  begin
    if (reset = '0') then
      count_i <= (others => '0');
    elsif (clk'event and clk = '1') then
      if load = '1' then
        count_i <= data;
      else
        count_i <= count_i + '1';
      end if;
    end if;
  end process;
  count <= count_i;
end behave;
```

**Verilog**

```verilog
module count_load (out, data, load, clk, reset);
  parameter Width = 8;
  input load, clk, reset;
  input [Width-1:0] data;
  output [Width-1:0] out;
  reg [Width-1:0] out;

  always @(posedge clk or negedge reset)
  if(!reset)
    out = 8'b0;
  else if(load)
    out = data;
  else
    out = out + 1;
endmodule
```
Chapter 2: Technology Independent Coding Styles

8-bit Up Counter with Load, Count Enable, Terminal Count and Asynchronous Reset

The following examples infer an 8-bit up counter with load, count enable, terminal count, and asynchronous reset.

**Verilog**

```verilog
module count_load (out, cout, data, load, clk, en, reset);
    parameter Width = 8;
    input load, clk, en, reset;
    input [Width-1:0] data;
    output cout; // carry out
    output [Width-1:0] out;
    reg [Width-1:0] out;

    always @(posedge clk or negedge reset)
        if (!reset)
            out = 8'b0;
        else if (load)
            out = data;
        else if (en)
            out = out + 1;
        // cout=1 when all out bits equal 1
        assign cout = &out;

endmodule
```

**N-bit Up Counter with Load, Count Enable, and Asynchronous Reset**

The following examples infer an n-bit up counter with load, count enable, and asynchronous reset.

**VHDL**

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

entity counter is
    generic (width : integer := n);
    port (data : in std_logic_vector (width-1 downto 0);
          load, en, clk, rst : in std_logic;
          q : out std_logic_vector (width-1 downto 0));
end counter;
```
Datapath

architecture behave of counter is
begin
    process(clk, rst)
    begin
        if rst = '1' then
            count <= (others => '0');
        elsif (clk'event and clk = '1') then
            if load = '1' then
                count <= data;
            elsif en = '1' then
                count <= count + 1;
            end if;
        end if;
        q <= count;
    end process;
end behave;

Arithmetic Operators

Synthesis tools generally are able to infer arithmetic operators for the target technology. The following examples infer addition, subtraction, division and multiplication operators.

VHDL

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

entity arithmetic is
port (A, B: in std_logic_vector(3 downto 0);
    Q1: out std_logic_vector(4 downto 0);
    Q2, Q3: out std_logic_vector(3 downto 0);
    Q4: out std_logic_vector(7 downto 0));
end arithmetic;

architecture behave of arithmetic is
begin
    process (A, B)
    begin
        Q1 <= ('0' & A) + ('0' & B); --addition
        Q2 <= A - B; --subtraction
        Q3 <= A / B; --division
        Q4 <= A * B; --multiplication
    end process;
end behave;
If the multiply and divide operands are powers of 2, replace them with shift registers. Shift registers provide speed optimized implementations with large savings in area. For example:
\[ Q \leftarrow C/16 + C*4; \]
can be represented as:
\[ Q \leftarrow \text{shr } (C, "100") + \text{shl } (C, "10"); \]
or
\[ \text{VHDL } Q \leftarrow "0000" \& C (8 \text{ downto } 4) + C (6 \text{ downto } 0) \& "00"; \]

The functions “shr” and “shl” are available in the IEEE.std_logic_arith.all library.

**Verilog**

```verilog
module arithmetic (A, B, Q1, Q2, Q3, Q4);
    input [3:0] A, B;
    output [4:0] Q1;
    output [3:0] Q2, Q3;
    output [7:0] Q4;
    reg [4:0] Q1;
    reg [3:0] Q2, Q3;
    reg [7:0] Q4;
    always @ (A or B)
        begin
            Q1 = A + B; //addition
            Q2 = A - B; //subtraction
            Q3 = A / 2; //division
            Q4 = A * B; //multiplication
        end
endmodule
```

If the multiply and divide operands are powers of 2, replace them with shift registers. Shift registers provide speed optimized implementations with large savings in area. For example:
\[ Q \leftarrow C/16 + C*4; \]
can be represented as:
\[ Q \leftarrow \{4b'0000 C[8:4]\} + \{C[6:0] 2b'00\}; \]
Relational operators compare two operands and indicate whether the comparison is true or false. The following examples infer greater than, less than, greater than equal to, and less than equal to comparators. Synthesis tools generally optimize relational operators for the target technology.

**VHDL**

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity relational is
  port (A, B : in std_logic_vector(3 downto 0);
        Q1, Q2, Q3, Q4 : out std_logic);
end relational;

architecture behav of relational is
begin
  process (A, B)
  begin
    -- Q1 <= A > B; -- greater than
    -- Q2 <= A < B; -- less than
    -- Q3 <= A >= B; -- greater than equal to
    if (A <= B) then
      -- less than equal to
      Q4 <= '1';
    else
      Q4 <= '0';
    end if;
  end process;
end behav;
```

**Verilog**

```verilog
module relational (A, B, Q1, Q2, Q3, Q4);
  input [3:0] A, B;
  output Q1, Q2, Q3, Q4;
  reg Q1, Q2, Q3, Q4;

  always @ (A or B)
  begin
    // Q1 = A > B; //greater than
    // Q2 = A < B; //less than
    // Q3 = A >= B; //greater than equal to
    if (A <= B) //less than equal to
      Q4 = 1;
    else
      Q4 = 0;
  end
endmodule
```
The equality and non-equality operators indicate a true or false output based on whether the two operands are equivalent or not. The following examples infer equality operators.

**VHDL**

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

entity equality is
  port (A: in STD_LOGIC_VECTOR (3 downto 0);
         B: in STD_LOGIC_VECTOR (3 downto 0);
         Q1: out STD_LOGIC;
         Q2: out STD_LOGIC);
end equality;

architecture equality_arch of equality is
begin
  process (A, B)
  begin
    Q1 <= A = B; -- equality
    if (A /= B) then -- inequality
      Q2 <= '1';
    else
      Q2 <= '0';
    end if;
  end process;
end equality_arch;
```

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

entity equality is
  port (A: in STD_LOGIC_VECTOR (3 downto 0);
         B: in STD_LOGIC_VECTOR (3 downto 0);
         Q1: out STD_LOGIC;
         Q2: out STD_LOGIC);
end equality;

architecture equality_arch of equality is
begin
  Q1 <= '1' when A = B else '0'; -- equality
  Q2 <= '1' when A /= B else '0'; -- inequality
end equality_arch;
```

**OR**

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

entity equality is
  port (A: in STD_LOGIC_VECTOR (3 downto 0);
         B: in STD_LOGIC_VECTOR (3 downto 0);
         Q1: out STD_LOGIC;
         Q2: out STD_LOGIC);
end equality;

architecture equality_arch of equality is
begin
  Q1 <= '1' when A = B else '0'; -- equality
  Q2 <= '1' when A /= B else '0'; -- inequality
end equality_arch;
```
Verilog

module equality (A, B, Q1, Q2);
  input [3:0] A;
  input [3:0] B;
  output Q1;
  output Q2;
  reg Q1, Q2;

always @ (A or B)
begin
  Q1 = A == B; //equality
  if (A != B) //inequality
    Q2 = 1;
  else
    Q2 = 0;
end
endmodule

Shift Operators

Shift operators shift data left or right by a specified number of bits. The following examples infer left and right shift operators.

VHDL

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

entity shift is
port (data : in std_logic_vector(3 downto 0);
  q1, q2 : out std_logic_vector(3 downto 0));
end shift;

architecture rtl of shift is
begin
  process (data)
  begin
    q1 <= shl (data, "10"); -- logical shift left
    q2 <= shr (data, "10"); --logical shift right
  end process;
end rtl;
Chapter 2: Technology Independent Coding Styles

OR

library IEEE;
use IEEE.std_logic_1164.all;

entity shift is
port (data : in std_logic_vector(3 downto 0);
     q1, q2 : out std_logic_vector(3 downto 0));
end shift;

architecture rtl of shift is
begin
  process (data)
  begin
    q1 <= data(1 downto 0) & "10"; -- logical shift left
    q2 <= "10" & data(3 downto 2); --logical shift right
  end process;
end rtl;

Verilog

module shift (data, q1, q2);
  input [3:0] data;
  output [3:0] q1, q2;
  parameter B = 2;
  reg [3:0] q1, q2;

always @ (data)
begin
  q1 = data << B; // logical shift left
  q2 = data >> B; //logical shift right
end
endmodule
**Finite State Machine**

A finite state machine (FSM) is a type of sequential circuit that is designed to sequence through specific patterns of finite states in a predetermined sequential manner. There are two types of FSM, Mealy and Moore. The Moore FSM has outputs that are a function of current state only. The Mealy FSM has outputs that are a function of the current state and primary inputs. An FSM consists of three parts:

1. **Sequential Current State Register**: The register, a set of n-bit flip-flops (state vector flip-flops) clocked by a single clock signal is used to hold the state vector (current state or simply state) of the FSM. A state vector with a length of n-bit has $2^n$ possible binary patterns, known as state encoding. Often, not all $2^n$ patterns are needed, so the unused ones should be designed not to occur during normal operation. Alternatively, an FSM with m-state requires at least $\log_2(m)$ state vector flip-flops.

2. **Combinational Next State Logic**: An FSM can only be in one state at any given time, and each active transition of the clock causes it to change from its current state to the next state, as defined by the next state logic. The next state is a function of the FSM's inputs and its current state.

3. **Combinational Output Logic**: Outputs are normally a function of the current state and possibly the FSM's primary inputs (in the case of a Mealy FSM). Often in a Moore FSM, you may want to derive the outputs from the next state instead of the current state, when the outputs are registered for faster clock-to-out timings.

Moore and Mealy FSM structures are shown in Figure 2-14 and Figure 2-15.

* State Vector Flip-flops

*Figure 2-14. Basic Structure of a Moore FSM*
Use a reset to guarantee fail safe behavior. This ensures that the FSM is always initialized to a known valid state before the first active clock transition and normal operation begins. In the absence of a reset, there is no way of predicting the initial value of the state register flip-flops during the “power up” operation of an Actel FPGA. It could power up and become permanently stuck in an unencoded state. The reset should be implemented in the sequential current state process of the FSM description.

An asynchronous reset is generally preferred over a synchronous reset because an asynchronous reset does not require decoding unused states, minimizing the next state logic.

Because FPGA technologies are register rich, “one hot” state machine implementations generated by the synthesis tool may generate optimal area and performance results.
The following examples represent a Mealy FSM model for the Mealy state diagram shown in Figure 2-16.

![Figure 2-16. Mealy State Diagram](image)

**VHDL**

```vhdl
-- Example of a 5-state Mealy FSM

library ieee;
use ieee.std_logic_1164.all;

entity mealy is
  port (clock, reset: in std_logic;
        data_out: out std_logic;
        data_in: in std_logic_vector (1 downto 0));
end mealy;

architecture behave of mealy is
  type state_values is (st0, st1, st2, st3, st4);
  signal pres_state, next_state: state_values;
begin
  -- FSM register
  statereg: process (clock, reset)
  begin
```
if (reset = '0') then
  pres_state <= st0;
elsif (clock'event and clock = '1') then
  pres_state <= next_state;
end if;
end process statereg;

-- FSM combinational block
fsm: process (pres_state, data_in)
begin
  case pres_state is
  when st0 =>
    case data_in is
      when "00"  => next_state <= st0;
      when "01"  => next_state <= st4;
      when "10"  => next_state <= st1;
      when "11"  => next_state <= st2;
      when others => null;
    end case;
  when st1 =>
    case data_in is
      when "00"  => next_state <= st0;
      when "10"  => next_state <= st2;
      when others => next_state <= st1;
    end case;
  when st2 =>
    case data_in is
      when "00"  => next_state <= st1;
      when "01"  => next_state <= st1;
      when "10"  => next_state <= st3;
      when "11"  => next_state <= st3;
      when others => null;
    end case;
  when st3 =>
    case data_in is
      when "01"  => next_state <= st4;
      when "11"  => next_state <= st4;
      when others => next_state <= st3;
    end case;
  when st4 =>
    case data_in is
      when "11"  => next_state <= st4;
      when others => next_state <= st0;
    end case;
  when others => next_state <= st0;
  end case;
end process fsm;
-- Mealy output definition using pres_state w/ data_in
outputs: process (pres_state, data_in)
begin
  case pres_state is
    when st0 =>
      case data_in is
        when "00"   => data_out <= '0';
        when others => data_out <= '1';
      end case;
    when st1 => data_out <= '0';
    when st2 =>
      case data_in is
        when "00"   => data_out <= '0';
        when others => data_out <= '1';
      end case;
    when st3 => data_out <= '1';
    when st4 =>
      case data_in is
        when "10"   => data_out <= '1';
        when others => data_out <= '0';
      end case;
    when others => data_out <= '0';
  end case;
end process outputs;
end behave;

Verilog

// Example of a 5-state Mealy FSM

module mealy (data_in, data_out, reset, clock);
output data_out;
input [1:0] data_in;
input reset, clock;
reg data_out;
reg [2:0] pres_state, next_state;

parameter st0=3'd0, st1=3'd1, st2=3'd2, st3=3'd3, st4=3'd4;

// FSM register
always @ (posedge clock or negedge reset)
  begin: statereg
    if(!reset)// asynchronous reset
      next_state <= st0;
    else
      next_state <= pres_state;
    end
  end

begin
  case pres_state is
    when st0 =>
      case data_in is
        when "00"   => data_out <= '0';
        when others => data_out <= '1';
      end case;
    when st1 => data_out <= '0';
    when st2 =>
      case data_in is
        when "00"   => data_out <= '0';
        when others => data_out <= '1';
      end case;
    when st3 => data_out <= '1';
    when st4 =>
      case data_in is
        when "10"   => data_out <= '1';
        when others => data_out <= '0';
      end case;
    when others => data_out <= '0';
  end case;
end process outputs;
end behave;
pres_state = st0;
else
    pres_state = next_state;
end // statereg

// FSM combinational block
always @(pres_state or data_in)
begnin: fsm
case (pres_state)
    st0: case(data_in)
        2'b00: next_state=st0;
        2'b01: next_state=st4;
        2'b10: next_state=st1;
        2'b11: next_state=st2;
    endcase
    st1: case(data_in)
        2'b00: next_state=st0;
        2'b10: next_state=st2;
        default: next_state=st1;
    endcase
    st2: case(data_in)
        2'b0x: next_state=st1;
        2'b1x: next_state=st3;
    endcase
    st3: case(data_in)
        2'bx1: next_state=st4;
        default: next_state=st3;
    endcase
    st4: case(data_in)
        2'b11: next_state=st0;
    default: next_state=st0;
endcase
end // fsm

// Mealy output definition using pres_state w/ data_in
always @(data_in or pres_state)
begnin: outputs
case(pres_state)
    st0: case(data_in)
        2'b00: data_out=1'b0;
        default: data_out=1'b1;
    endcase
    st1: data_out=1'b0;
    st2: case(data_in)
The following examples represent a Moore FSM model for the Mealy state diagram shown in Figure 2-16 on page 37.

**VHDL**

-- Example of a 5-state Moore FSM

library ieee;
use ieee.std_logic_1164.all;

entity moore is
    port (clock, reset: in std_logic;
          data_out: out std_logic;
          data_in: in std_logic_vector (1 downto 0));
end moore;

architecture behave of moore is
    type state_values is (st0, st1, st2, st3, st4);
    signal pres_state, next_state: state_values;
begin
    -- FSM register
    statereg: process (clock, reset)
    begin
        if (reset = '0') then
            pres_state <= st0;
        elsif (clock = '1' and clock'event) then
            pres_state <= next_state;
        end if;
    end process statereg;

    2'b0x: data_out='0';
    default: data_out='1';
    endcase
    st3: data_out='1';
    st4: case(data_in)
        2'b1x: data_out='1';
        default: data_out='0';
    endcase
    default: data_out='0';
    endcase
endmodule
Chapter 2: Technology Independent Coding Styles

-- FSM combinational block
fsm: process (pres_state, data_in)
begin
  case pres_state is
    when st0 =>
      case data_in is
        when "00" => next_state <= st0;
        when "01" => next_state <= st4;
        when "10" => next_state <= st1;
        when "11" => next_state <= st2;
        when others => null;
      end case;
    when st1 =>
      case data_in is
        when "00" => next_state <= st0;
        when "10" => next_state <= st2;
        when others => next_state <= st1;
      end case;
    when st2 =>
      case data_in is
        when "00" => next_state <= st1;
        when "01" => next_state <= st1;
        when "10" => next_state <= st3;
        when "11" => next_state <= st3;
        when others => null;
      end case;
    when st3 =>
      case data_in is
        when "01" => next_state <= st4;
        when "11" => next_state <= st4;
        when others => next_state <= st3;
      end case;
    when st4 =>
      case data_in is
        when "11" => next_state <= st4;
        when others => next_state <= st0;
      end case;
    when others => next_state <= st0;
  end case;
end process fsm;

-- Moore output definition using pres_state only
outputs: process (pres_state)
begin
  case pres_state is
    when st0 => data_out <= '1';
    when st1 => data_out <= '0';
  end case;
end process outputs;
Finite State Machine

when st2 => data_out <= '1';
when st3 => data_out <= '0';
when st4 => data_out <= '1';
when others => data_out <= '0';
end case;
end process outputs;
end behave;

Verilog
// Example of a 5-state Moore FSM

module moore (data_in, data_out, reset, clock);
    output data_out;
    input [1:0] data_in;
    input reset, clock;
    reg data_out;
    reg [2:0] pres_state, next_state;

    parameter st0=3'd0, st1=3'd1, st2=3'd2, st3=3'd3, st4=3'd4;

    // FSM register
    always @(posedge clock or negedge reset)
    begin: statereg
        if(!reset)
            pres_state = st0;
        else
            pres_state = next_state;
    end // statereg

    // FSM combinational block
    always @(pres_state or data_in)
    begin: fsm
        case (pres_state)
            st0: case(data_in)
                2'b00: next_state=st0;
                2'b01: next_state=st4;
                2'b10: next_state=st1;
                2'b11: next_state=st2;
            endcase
            st1: case(data_in)
                2'b00: next_state=st0;
                2'b10: next_state=st2;
                default: next_state=st1;
            endcase
        endcase
    end // fsm
endmodule
Chapter 2: Technology Independent Coding Styles

```verilog
st2:   case(data_in)
   2'b0x:   next_state=st1;
   2'b1x:   next_state=st3;
   endcase
st3:   case(data_in)
   2'b1x:   next_state=st4;
   default: next_state=st3;
   endcase
st4:   case(data_in)
   2'b11:   next_state=st4;
   default: next_state=st0;
   endcase
   default:                     next_state=st0;
   endcase
end  // fsm

// Moore output definition using pres_state only
always @(pres_state)
begin: outputs
   case(pres_state)
      st0:   data_out=1'b1;
      st1:   data_out=1'b0;
      st2:   data_out=1'b1;
      st3:   data_out=1'b0;
      st4:   data_out=1'b1;
      default: data_out=1'b0;
   endcase
end  // outputs
endmodule  // Moore
```

**Input-Output Buffers**

You can infer or instantiate a I/O buffers in your design. The following examples represent both techniques. Regardless of which method you use, all I/O buffers should be declared at the top level of the design.
**Tri-State Buffer**

A tri-state buffer is an output buffer with high-impedance capability. The following examples show how to infer and instantiate a tri-state buffer.

![Tri-State Buffer Diagram](image)

*Figure 2-17. Tri-State Buffer*

**Inference**

**VHDL**

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

entity tristate is
port (e, a : in std_logic;
     y : out std_logic);
end tristate;

architecture tri of tristate is
begin
    process (e, a)
    begin
        if e = '1' then
            y <= a;
        else
            y <= 'Z';
        end if;
    end process;
end tri;
```

**OR**

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

entity tristate is
port (e, a : in std_logic;
     y : out std_logic);
end tristate;

architecture tri of tristate is
begin
    y <= a when (e = '1') else 'Z';
end tri;
```
Chapter 2: Technology Independent Coding Styles

Verilog

module TRISTATE (e, a, y);
    input a, e;
    output y;
    reg y;
always @(e or a) begin
    if (e)
        y = a;
    else
        y = 1'bz;
end
endmodule

OR

module TRISTATE (e, a, y);
    input a, e;
    output y;
assign y = e ? a : 1'bZ;
endmodule

Instantiation

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

generate
    entity tristate is
        port (e, a : in std_logic;
              y : out std_logic);
    end tristate;

architecture tri of tristate is

component TRIBUFF
    port (D, E: in std_logic;
          PAD: out std_logic);
end component;

begin
    U1: TRIBUFF port map (D => a,
                           E => e,
                           PAD => y);
end tri;
Verilog

module TRISTATE (e, a, y);
input a, e;
output y;
TRISBUFF U1 (.D(a), .E(e), .PAD(y));
endmodule

Bi-Directional Buffer

A bi-directional buffer can and input or output buffer with high impedance capability. The following examples show how to infer and instantiate a bi-directional buffer.

Inference

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity bidir is
port (y : inout std_logic;
    e, a : in std_logic;
    b : out std_logic);
end bidir;

architecture bi of bidir is
begin
    process (e, a)
    begin
        case e is
            when '1' => y <= a;
            when '0' => y <= 'Z';
            when others => y <= 'X';
        end case;
    end process;
    b <= y;
end bi;
Chapter 2: Technology Independent Coding Styles

Verilog

module bidir (e, y, a, b);
  input a, e;
  inout y;
  output b;
  reg y_int;
  wire y, b;

always @ (a or e)
begin
  if (e == 1'b1)
    y_int <= a;
  else
    y_int <= 1'bz;
end
assign y = y_int;
assign b = y;
endmodule

Instantiation

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity bidir is
port (y : inout std_logic;
  e, a: in std_logic;
  b : out std_logic);
end bidir;

architecture bi of bidir is

component BIBUF
  port (D, E: in std_logic;
        Y : out std_logic;
        PAD: inout std_logic);
end component;

begin
U1: BIBUF port map (D => a,
                      E => e,
                      Y => b,
                      PAD => y);
end bi;
Generics and Parameters

Verilog

module bidir (e, y, a, b);
  input a, e;
  inout y;
  output b;

  BIBUF U1 ( .PAD(y), .D(a), .E(e), .Y(b) );
endmodule

Generics and Parameters

Generics and parameters are used to define the size of a component. This allows the design of parameterized components for which size and feature sets may be defined by values of the instantiation parameters. The following examples show how to use generics and parameters when describing a parameterized adder. Furthermore, this adder is instantiated for varying widths.

VHDL

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

entity adder is
  generic (WIDTH : integer := 8);
  port (A, B: in UNSIGNED(WIDTH-1 downto 0);
    CIN: in std_logic;
    COUT: out std_logic;
    Y: out UNSIGNED(WIDTH-1 downto 0));
end adder;

architecture rtl of adder is
begin
  process (A,B,CIN)
  variable TEMP_A,TEMP_B,TEMP_Y:UNSIGNED(A'length downto 0);
  begin
    TEMP_A := '0' & A;
    TEMP_B := '0' & B;
    TEMP_Y := TEMP_A + TEMP_B + CIN;
    Y <= TEMP_Y (A'length-1 downto 0);
    COUT <= TEMP_Y (A'length);
  end process;
end rtl;
“Width” indicates the width of the adder. The instantiation for this parameterized adder for a bit width of 16 is:

U1: adder generic map(16) port map (A_A, B_A, CIN_A, COUT_A, Y_A);

**Verilog**

```verilog
module adder (cout, sum, a, b, cin);
    parameter Size = 8;
    output cout;
    output [Size-1:0] sum;
    input cin;
    input [Size-1:0] a, b;

    assign {cout, sum} = a + b + cin;

endmodule
```

“Size” indicates the width of the adder. The instantiation for this parameterized adder for a bit width of 16 is:

adder #(16) adder16(cout_A, sum_A, a_A, b_A, cin_A)
Unlike ASICs, FPGAs are module based arrays. Each logic level used on a path can add delay. As a result, meeting timing constraints on a critical path with too many logic levels becomes difficult. Using an efficient coding style is very important because it dictates the synthesis logic implementation. This chapter describes synthesis implementations, techniques, and efficient design practices that can be used to reduce logic levels on a critical path.

Reducing Logic Levels on Critical Paths

Each logic level on the critical path in an FPGA can add significant delay. To ensure that timing constraints can be met, logic level usage must be considered when describing the behavior of a design. The following examples illustrate how to reduce logic levels on critical paths.

**Example 1**

In the following VHDL example, the signal “critical” goes through three logic levels.

```vhdl
if ((( Crtical='0' and Obi='1' and Sar='1')
or CpuG='0') and CpuR='0') then
  Des <= Adr;
elsif (((Crtical='0' and Obi='1' and Sar='1')
or CpuG='0') and CpuR='1') then
  Des <= Bdr;
elsif (Sar='0' and ........
```
The signal “critical” is a late arriving signal. To reduce the logic level usage on “critical”, imply priority by using an if-then-else statement. As a result, the signal “critical” goes through one logic level, as shown below.

Example 2
In the following example, the signal “critical” goes through two logic levels.
To reduce the logic level usage on “critical”, multiplex inputs “in1” and “in2” based on “non_critical”, and call this output “out_temp”. Then multiplex “out_temp” and “in2” based on “critical”. As a result, the signal “critical” goes through one logic level, as shown below.

```
signal out_temp : std_logic
  if (non_critical)
    out_temp <= in1;
  else out_temp <= in2;
  if (clk'event and clk = '1') then
    if (critical) then
      out1 <= out_temp;
    else out1 <= in2;
    end if;
  end if;
end if;
```

Resource Sharing

Resource sharing can reduce the number of logic modules needed to implement HDL operations. Without it, each HDL description is built into a separate circuit. The following VHDL examples illustrate how to use resource sharing to reduce logic module utilization.

**Example 1**

This example implements four adders.

```
if (...) (siz == 1) ...
  count = count + 1;
else if (...) (siz == 2) ...
  count = count + 2;
else if (...) (siz == 3) ...
  count = count + 3;
else if (...) (siz == 0) ...
  count = count + 4;
```
By adding the following code, two adders can be eliminated:

```c
if (... (siz == 0) ...)
    count = count + 4;
else if (...)
    count = count + siz
```

**Example 2**

This example uses poor resource sharing to implement adders.

```
if (select)
    sum <= A + B;
else
    sum <= C + D;
```

Adders use valuable resources. To reduce resource usage, rewrite the code to infer two multiplexors and one adder, as shown below.

```
if (select)
    temp1 <= A;
    temp2 <= B;
else
    temp1 <= C;
    temp2 <= D;
    sum <= temp1 + temp2;
```

**Note:** This example assumes the select line is not a late arriving signal.
Operators Inside Loops

Operators are resource intensive compared to multiplexors. If there is an operator inside a loop, the synthesis tool has to evaluate all conditions. In the following VHDL example, the synthesis tool builds four adders and one multiplexor. This implementation is only advisable if the select line “req” is a late arriving signal.

```vhdl
operator := sum;
for i in 0 to 3 loop
  if (req(i)='1') then
    vsum <= vsum + offset(i);
  end if;
end loop;
```

If the select line “req” is not critical, the operator should be moved outside the loop so the synthesis tool can multiplex the data before performing the adder operation. The area efficient design is implemented in a larger multiplexor and a single adder, as shown below.
Chapter 3: Performance Driven Coding

```vhdl
vsum := sum;
for i in 0 to 3 loop
    if (req(i)='1') then
        offset_1 <= offset(i);
    end if;
end loop;
end process;
```

Coding for Combinability

Combinatorial modules can be merged into sequential modules in the antifuse architecture. This results in a significant reduction in delay on the critical path as well as area reduction. However, cells are only merged if the combinatorial module driving a basic flip-flop has a load of 1. In the following VHDL example, the AND gate driving the flip-flop has a load of 2. As a result, the AND gate cannot be merged into the sequential module.

```
one : process (clk, a, b, c, en) begin
    if (clk'event and clk = '1') then
        if (en = '1') then
            q2 <= a and b and c;
        end if;
        q1 <= a and b and c;
    end if;
end process one;
```

To enable merging, the AND gate has to be duplicated so that it has a load of 1. To duplicate the AND gate, create two independent processes, as shown below. Once merged, one logic level has been removed from the critical path.
Register Duplication

Note: Some synthesis tools automatically duplicate logic on the critical path. Other synthesis tools detect the function “a & b & c” in the two processes and share the function on a single gate. If the function is shared, the logic is not duplicated and instantiation should be considered.

part_one: process (clk, a, b, c, en) begin
if (clk'event and clk = '1') then
  if (en = '1') then
    q2 <= a and b and c;
  end if;
end if;
end process part_one;

part_two: process (clk, a, b, c) begin
if (clk'event and clk = '1') then
  q1 <= a and b and c;
end if;
end process part_two;

Register Duplication

The delay on a net rises as the number of loads increase in the antifuse architecture. This may be acceptable for networks such as reset, but not others such as tri-state enable, etc. It is important to keep the fanout of a network below 16. In the following VHDL example, the signal “Tri_en” has a fanout of 24.
architecture load of four_load is
    signal Tri_en std_logic;
begin
loadpro: process (Tri_en, Clk)
begin
    if (clk'event and clk = '1') then
        Tri_end <= Tri_en;
    end if;
end process loadpro;
endpro : process (Tri_end, Data_in)
begin
    if (Tri_end = '1') then
        out <= Data_in;
    else
        out <= (others => 'Z');
    end if;
end process endpro;
end load;

To decrease the fanout by half, registers are duplicated on the signal "Tri_en" so the load is split in half, as shown in the following example.

Note: Some synthesis tools duplicate registers to resolve timing and fanout violations and do not need to use this coding technique.
architecture loada of two_load is
signal Tri_en1, Tri_en2 : std_logic;
begin
loadpro: process (Tri_en, Clk)
begin
if (clk'event and clk = '1') then
Tri_en1 <= Tri_en;
Tri_en2 <= Tri_en;
end if;
end process loadpro;

process (Tri_en1, Data_in)
begin
if (Tri_en1 = '1') then
out(23:12) <= Data_in(23:12);
else
out(23:12) <= (others => 'Z');
end if;
end process;

process (Tri_en2, Data_in)
begin
if (Tri_en2 = '1') then
out(11:0) <= Data_in(11:0);
else
out(11:0) <= (others => 'Z');
end if;
end process;
Partitioning a Design

Most synthesis tools work best when optimizing medium size blocks, approximately two to five thousand gates at a time. To reduce synthesis time, you should partition designs so that module block sizes do not exceed the recommendations of the synthesis tool vendor. When partitioning a design into various blocks, it is good design practice to have registers at hierarchical boundaries. This eliminates the need for time budgeting on the inputs and outputs. The following example shows how to modify your HDL code so that registers are placed at hierarchical boundaries.

Registers Embedded Within a Module

process (clk, a, b) begin
    if (clk'event and clk = '1') then
        a1 <= a;
        b1 <= b;
    end if;
end process;

process (a1, b1)
begin
c <= a1 + b1;
end process;

Registers Pushed Out at the Hierarchical Boundary

process (clk, a, b) begin
    if (clk'event and clk = '1') then
        c <= a + b;
    end if;
end process;
In addition to technology independent coding and performance driven coding, there are coding techniques that can be used to take advantage of the Actel architecture to improve speed and area utilization of your design. Additionally, most synthesis tools can implement random logic, control logic and certain datapath macros. However, they may not generate technology optimal implementations for datapath elements that cannot be inferred using operators, such as counters, RAM, FIFO, etc. This chapter describes coding techniques to take advantage of technology specific features and how to instantiate technology specific macros generated by the ACTgen Macro Builder tool for optimal area and performance.

**Multiplexors**

Using case statements with the multiplexor based Actel architecture provides area and speed efficient solutions and is more efficient than inference of priority encoders using if-then-else statements. Actel recommends that you use case statements instead of long, nested if-then-else statements to force mapping to multiplexors in the Actel architecture. Refer to “Multiplexors Using Case” on page 19 for examples of multiplexor coding.

VHDL synthesis tools automatically assume parallel operation without priority in case statements. However, some Verilog tools assume priority, and you may need to add a directive to your case statement to ensure that no priority is assumed. refer to the documentation provided with your synthesis tool for information about creating case statements without priority.
**Internal Tri-State to Multiplexor Mapping**

All internal tri-states must be mapped to multiplexors. The antifuse technology only supports tri-states as in/out ports, but not internal tri-states. The following examples show an internal tri-state followed by a multiplexor that the internal tri-state should be changed to.

**Note:** Some synthesis tools automatically map internal tri-states to multiplexors.

**VHDL Tri-State**

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity tribus is
port (A, B, C, D : in std_logic_vector(7 downto 0);
    E0, E1, E2, E3 : in std_logic;
    Q : out std_logic_vector(7 downto 0));
end tribus;

architecture rtl of tribus is
begin
    Q <= A when (E0 = '1') else "ZZZZZZZZ";
    Q <= B when (E1 = '1') else "ZZZZZZZZ";
    Q <= C when (E2 = '1') else "ZZZZZZZZ";
    Q <= D when (E3 = '1') else "ZZZZZZZZ";
end rtl;
```
VHDL Multiplexor

library IEEE;
use IEEE.std_logic_1164.all;

entity muxbus is
port (A, B, C, D : in std_logic_vector(7 downto 0);
     E0, E1, E2, E3 : in std_logic;
     Q : out std_logic_vector(7 downto 0));
end muxbus;

architecture rtl of muxbus is
signal E_int : std_logic_vector(1 downto 0);
begin
process (E0, E1, E2, E3)
variable E : std_logic_vector(3 downto 0);
beg
E := E0 & E1 & E2 & E3;
case E is
 when "0001" => E_int <= "00";
 when "0010" => E_int <= "01";
 when "0100" => E_int <= "10";
 when "1000" => E_int <= "11";
 when others => E_int <= "--";
end case;
end process;

process (E_int, A, B, C, D)
begin
 case E_int is
 when "00" => Q <= D;
 when "01" => Q <= C;
 when "10" => Q <= B;
 when "11" => Q <= A;
 when others => Q <= (others => '-');
end case;
end process;
end rtl;

Verilog Tri-State
module tribus (A, B, C, D, E0, E1, E2, E3, Q);

 input [7:0]A, B, C, D;
 output [7:0]Q;
 input E0, E1, E2, E3;

 assign Q[7:0] = E0 ? A[7:0] : 8'bzzzzzzzz; 
 assign Q[7:0] = E1 ? B[7:0] : 8'bzzzzzzzz;
 assign Q[7:0] = E2 ? C[7:0] : 8'bzzzzzzzz; 
 assign Q[7:0] = E3 ? D[7:0] : 8'bzzzzzzzz;
endmodule
Module Multiplexor

module muxbus (A, B, C, D, E0, E1, E2, E3, Q);
  input [7:0] A, B, C, D;
  output [7:0] Q;
  input E0, E1, E2, E3;
  wire [3:0] select4;
  reg [1:0] select2;
  reg [7:0] Q;

  assign select4 = {E0, E1, E2, E3};

  always @ (select4)
  begin
    case (select4)
      4'b0001 : select2 = 2'b00;
      4'b0010 : select2 = 2'b01;
      4'b0100 : select2 = 2'b10;
      4'b1000 : select2 = 2'b11;
      default : select2 = 2'bxx;
    endcase
  end

  always @ (select2 or A or B or C or D)
  begin
    case (select2)
      2'b00 : Q = D;
      2'b01 : Q = C;
      2'b10 : Q = B;
      2'b11 : Q = A;
    endcase
  end
endmodule

Registers

The XL, DX, MX, and ACT 3 families have dedicated asynchronous reset registers in the sequential modules (SMOD). In each SMOD is a 4:1 multiplexor with some gating logic on the select lines. Implementing a simple register or an asynchronous reset register allows the gating logic in front of the register to be pulled into the SMOD, reducing the path delay by one level. This is called full combinability. Full combinability offers improved speed, increasing a 50MHz operation to 75MHz in some designs. The following examples show how to use registers for combinability and discuss any speed or area penalty associated with using the register.
Synchronous Clear or Preset

The synchronous clear or preset register only uses part of the SMOD multiplexor, allowing for some combinability. The following example shows how to share a synchronous register with a 2:1 multiplexor.

**VHDL**

```vhdl
-- register with active low sync preset shared with a 2-to-1 mux.
library ieee;
use ieee.std_logic_1164.all;
entity dfm_sync_preset is
PORT (d0, d1: in std_logic;
    clk, preset, sel: in std_logic;
    q: out std_logic);
end dfm_sync_preset;
architecture behav of dfm_sync_preset is
signal tmp_sel: std_logic_vector(1 downto 0);
signal q_tmp: std_logic;
begin
process (clk) begin
    tmp_sel <= preset & sel;
    if (clk'event and clk = '1') then
        case tmp_sel is
        when "00" => q_tmp <= '1';
        when "01" => q_tmp <= '1';
        when "10" => q_tmp <= d0;
        when "11" => q_tmp <= d1;
        when others => q_tmp <= '1';
        end case;
    end if;
end process;
q <= q_tmp;
end behav;
```

Figure 4-1. Single Module Implementation of a Synchronous Clear or Preset Register
Verilog

/* register with active-low synchronous preset shared with 2-to-1 mux */

module dfm_sync_preset (d0, d1, clk, sync_preset, sel, q);
input d0, d1;
input sel;
input clk, sync_preset;
output q;
reg q;
always @ (posedge clk)
beginn
  case ({sync_preset, sel})
    2'b00: q = 1'b1;
    2'b01: q = 1'b1;
    2'b10: q = d0;
    2'b11: q = d1;
  endcase
end
endmodule

Clock Enabled

The clock enabled register uses a 2:1 multiplexor with output feedback, which uses some of the SMOD multiplexor. The following example shows how to share a clock enabled register with the input logic.

![Figure 4-2. Single Module Implementation of a Clock Enabled Register](image-url)
library ieee;
use ieee.std_logic_1164.all;

entity dfm_clken is
PORT (d0, d1: in std_logic;
    clk, reset, clken, sel: in std_logic;
    q: out std_logic);
end dfm_clken;

architecture behav of dfm_clken is
signal tmp_sel: std_logic_vector(1 downto 0);
signal q_tmp: std_logic;

begin
process (clk, reset) begin
    tmp_sel <= clken & sel;
    if (reset = '0') then
        q_tmp <= '0';
    elsif (clk'event and clk = '1') then
        case tmp_sel is
            when "00" => q_tmp <= d0;
            when "01" => q_tmp <= d1;
            when "10" => q_tmp <= q_tmp;
            when "11" => q_tmp <= q_tmp;
            when others => q_tmp <= q_tmp;
        end case;
    end if;
end process;
q <= q_tmp;
end behav;
Verilog

/* register with asynchronous set, clock enable, shared with a 2-to-1 mux */

module dfm_clken (d0, d1, clk, reset, clken, sel, q);
input d0, d1;
input sel;
input clk, reset, clken;
output q;
reg q;
always @(posedge clk or negedge reset)
begin
  if (!reset)
    q = 8'b0;
  else
    case ({clken, sel})
    2'b00: q = d0;
    2'b01: q = d1;
    2'b10: q = q;
    2'b11: q = q;
    endcase
end
endmodule

Asynchronous Preset

Some synthesis tools automatically translate an asynchronous preset register into an asynchronous reset register without performance penalties. The bubbled logic can then be pushed into the surrounding logic without any delay penalty. There are various types of preset registers in the Actel libraries. Some of the registers use two combinatorial modules (CMOD) and most use an inverter, which consumes part of the SMOD multiplexor. If your synthesis tool does not automatically translate an asynchronous preset register into a functionally equivalent asynchronous preset register using an asynchronous reset register, use the following examples to design an asynchronous reset register.
Verilog Asynchronous Preset

```verilog
module dfp (q, d, clk, preset);
input d, clk, preset;
output q;
reg q;

always @ (posedge clk or negedge preset)
if (!preset)
  q = 1'b1;
else
  q = d;
endmodule
```

Verilog Equivalent Asynchronous Preset

```verilog
module dfp_r (q, d, clk, preset);
input d, clk, preset;
output q;
wire d_inv, reset;
reg q_inv;

assign d_inv = !d;
assign q = !q_inv;
assign reset = preset;
always @ (posedge clk or negedge reset)
if (!reset)
  q_inv = 1'b0;
else
  q_inv = d_inv;
endmodule
```
Chapter 4: Technology Specific Coding Techniques

VHDL Asynchronous Preset

-- register with active low async preset.

library ieee;
use ieee.std_logic_1164.all;
entity dfp is
  port (d, clk, preset : in std_logic;
        q : out std_logic);
end dfp;

architecture behav of dfp is
begin
process (clk, preset)
begin
  if (preset = '0') then
    q <= '1';
  elsif (clk'event and clk = '1') then
    q <= d;
  end if;
end process;
end behav;

VHDL Equivalent Asynchronous Preset

-- register with active low async preset.

library ieee;
use ieee.std_logic_1164.all;
entity dfp_r is
  port (d, clk, preset : in std_logic;
        q : out std_logic);
end dfp_r;

architecture behav of dfp_r is
signal reset, d_tmp, q_tmp : std_logic;
begin
  reset <= preset;
  d_tmp <= NOT d;
process (clk, reset)
begin
  if (reset = '0') then
    q_tmp <= '0';
  elsif (clk'event and clk = '1') then
    q_tmp <= d_tmp;
  end if;
end process;
q <= NOT q_tmp;
end behav;
Asynchronous Preset and Clear

This is the most problematic register for the ACT 2, XL, DX, MX, and ACT 3 architectures. Only one cell can be used, the DFPC cell, to design an asynchronous preset and clear register. The DFPC uses two CMODs to form a master latch and a slave latch that together form one register. This uses two CMODs per register and it offers no logic combinability with the SMOD. The DFPC requires more setup time and no combinability. The net timing loss can often be as high as 10 ns. Actel recommends that you do not use any asynchronous preset and clear registers on critical paths. Use a synchronous preset with asynchronous clear or a synchronous clear register instead.

An asynchronous preset and clear register can be used if it does not affect a critical path or cause high utilization in the design.

Registered I/Os

The ACT 3 technology has registers in the I/O ring, with both reset and preset, which allow for fast input setup and clock-to-out delays. Because most synthesis tools do not infer these special resources, the following example shows how to instantiate a registered I/O cell, BREPTH, in your design.

![Registered I/O Cell](image)

Figure 4-5. Registered I/O Cell
Chapter 4: Technology Specific Coding Techniques

**VHDL**

library IEEE;
use IEEE.std_logic_1164.all;

entity regio is
  port (data, en, Tri_en, clock, preset : in std_logic;
        bidir : inout std_logic;
        q_pad : out std_logic);
end regio;

architecture rtl of regio is
  -- Component Declaration
  component BREPTH
    port (D, ODE, E, IOPCL, CLK : in std_logic;
          Y : out std_logic;
          PAD : inout std_logic);
  end component;

begin
  -- Concurrent Statement
  U0 : BREPTH port map (D => data,
                           ODE => en,
                           E => Tri_en,
                           IOPCL => preset,
                           CLK => clock,
                           Y => q_pad,
                           PAD => bidir);
end rtl;

**Verilog**

module regio (data, Q_pad, clock, preset, Tri_en, en, bidir);

  input  data, clock, preset, Tri_en, en;
  output Q_pad;
  inout  bidir;

BREPTH U1 (.PAD(Q_pad), .D(data), .CLK(clock), .IOPCL(preset),
            .E(Tri_en), .ODE(en), .Y(bidir));
endmodule

**Note:** As a good design practice, instantiate all input/output cells at the top level of your design.
CLKINT/CLKBUF for Reset and/or High Fanout Networks

Many designs have internally generated clocks, high fanout control signals, or internally generated reset signals. These signals need a large internal driver, CLKINT, to meet both area and performance goals for the circuit. If the high fanout signals come directly into the design through an I/O, a CLKBUF driver is used. Most synthesis tools do not automatically use these drivers. Instead, the synthesis tool builds a buffer tree that consumes one module per driver. On a high fanout net this can affect both the area and timing for that signal. If the global drivers for a given array are still available, you should instantiate the CLKINT or CLKBUF driver into the design. The following example shows how to instantiate these drivers.

CLKINT

The following examples instantiate the CLKINT driver.

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity design is
  port (………………… : in std_logic;
        …………………… : out std_logic);
end design;

  architecture rtl of design is
  -- Component Declaration
  component CLKINT
    port (A : in std_logic;
          Y : out std_logic);
  end component;

  begin
    -- Concurrent Statement
    U2 : CLKINT port map ( A => neta,
                            Y => int_clk);
  end rtl;
Chapter 4: Technology Specific Coding Techniques

Verilog

module design ();

    input __________;
    output __________;
    CLKINT U2 (.Y(int_rst), .A(neta));
    __________
endmodule

CLKBUF

The following examples instantiate a CLKBUF driver.

VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity design is
    port (PAD : in std_logic;
          Y : out std_logic);
end component;

begin
    -- Concurrent Statement
    U2 : CLKBUF port map (PAD => neta, Y => int_rst);
end rtl;

Verilog

module design ();

    input __________;
    output __________;
    CLKBUF U2 (.Y(rst), .PAD(reset));
    __________
endmodule
**QCLKINT/QCLKBUF for Medium Fanout Networks**

The 32100DX, 32200DX, and 32300DX have four quadrant clocks that can be used to drive internally generated high fanout nets (QCLKINT) or high fanout nets generated from I/O ports (QCLKBUF). The methodology and instantiation are similar to the CLKINT/CLKBUF drivers. However, the QCLK drivers can only drive within a quadrant. Although the placement of the cells into a quadrant is automated by the Designer place and rout software, you must limit the number of fanouts and prevent the use of multiple QCLK signals to drive the same cell or gate. Table 4-1 lists fanout limits for the devices.

You can double your fanout limit and drive half the chip by combining two drivers into one to drive 2 quadrants. However, each time you combine drivers, you reduce the number of available QCLKs by one. The Designer place and route software automatically combines QCLKs when necessary.

**Table 4-1. Fanout Limits**

<table>
<thead>
<tr>
<th></th>
<th>32100DX</th>
<th>32200DX</th>
<th>32300DX</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quadrant QCLK</strong></td>
<td>175</td>
<td>307</td>
<td>472</td>
</tr>
<tr>
<td><strong>Fanout Limit</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Half Chip QCLK</strong></td>
<td>350</td>
<td>614</td>
<td>944</td>
</tr>
<tr>
<td><strong>Fanout Limit</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ACTgen Counter**

Several synthesis tools cannot build an optimal counter implementation for the Actel architecture. If a counter is on a critical path, this implementation can increase logic level usage and decrease performance. To reduce critical path delays and to achieve optimal results from your design, Actel recommends that you instantiate counters generated by the ACTgen Macro Builder. The ACTgen Macro Builder supports a wide variety of counters for area and performance needs.
The following example uses a 5-bit counter with load, count enable, and asynchronous reset that has been generated with ACTgen and saved as a structural HDL netlist called “CNT5”. The counter is instantiated as follows:

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

entity counter is
    port (bus_d : in std_logic_vector(4 downto 0);
          bus_q : out std_logic_vector(4 downto 0);
          net_clock, net_aclr, net_enable : in std_logic;
          net_sload : in std_logic);
end counter;

architecture rtl of counter is

    -- Component Declaration
    component CNT5
        port (Data : in std_logic_vector(4 downto 0);
              Sload, Enable, Aclr, Clock : in std_logic;
              Q : out std_logic_vector(4 downto 0));
    end component;

begin
    -- Concurrent Statement
    U0 : CNT5 port map (Data => bus_d,
                         Sload => net_sload,
                         Enable => net_enable,
                         Aclr => net_aclr,
                         Clock => net_clock,
                         Q => bus_q);
end rtl;
```
Verilog

module counter (bus_q, bus_d, net_clock, net_aclr, net_enable, net_sload);
  input [4:0] data;
  input net_sload, net_enable, net_aclr, net_clock;
  output [4:0] bus_q;

CNT5 U0 (.Q(bus_q), .Data(bus_d), .Clock(net_clock),
  .Aclr(net_aclr), .Enable(net_enable), .Sload(net_sload));
endmodule

Dual Architecture Coding in VHDL

It is possible to maintain technology independence after instantiating an ACTgen macro into your design. By adding a second technology independent architecture, you can maintain two functionally equivalent architectures of the same entity in your design. The ACTgen macro is Actel specific and instantiated in your design to take advantage of the architectural features of the target Actel FPGA. This allows you to meet your design goals quickly. The technology independent architecture is functionally equivalent to the Actel specific architecture (verified by simulation) and can be used to synthesize the design to another technology if necessary. The following example shows the technology independent (RTL) and Actel specific (structural) architecture for a counter called “CNT5” and illustrates how to write your code so that you can choose which architecture to use.

RTL Architecture

This implementation of “CNT5” is written as a behavioral description directly into the design.

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

entity CNT5 is
  port (Data: in std_logic_vector(4 downto 0);
    Sload, Enable, Aclr, Clock: in std_logic;
    Q: out std_logic_vector(4 downto 0));
end CNT5;
architecture RTL of CNT5 is

signal cnt: std_logic_vector(4 downto 0);

begin
    counter : process (Aclr, Clock)
    begin
        if (Aclr = '0') then
            cnt <= (others => '0'); -- asynchronous reset
        elsif (Clock'event and Clock = '1') then
            if (Sload = '1') then
                cnt <= Data; -- synchronous load
            elsif (Enable = '1') then
                cnt <= cnt + '1'; -- increment counter
            end if;
        end if;
    end process;
    Q <= cnt; -- assign counter output to output port
end RTL;

Structural Architecture

This implementation of “CNT5” is created by the ACTgen macro builder. The port names for the RTL description must match the port names of the structural “CNT5” netlist generated by ACTgen.

library ieee;
use ieee.std_logic_1164.all;
library ACT3;

entity CNT5 is
    port (Data : in std_logic_vector(4 downto 0); Enable, Sload, Aclr, Clock : in std_logic; Q : out std_logic_vector(4 downto 0))
end CNT5;

architecture DEF_ARCH of CNT5 is

component DFM7A
    port(D0, D1, D2, D3, S0, S10, S11, CLR, CLK : in
        std_logic; Q : out std_logic);
end component;

end DEF_ARCH;
Instantiating “CNT5” in the Top Level Design

Once you have created both architectures, instantiate “CNT5” into your design, adding binding statements for both architectures. The binding statements are used to specify which architecture the synthesis tool uses in the design. The technology independent RTL architecture might not meet the performance requirements. The Actel specific DEF_ARCH architecture is optimized for the Actel FPGA architecture and may provide higher performance. By removing the comment on one of the “use” statements in the code, a particular architecture can be chosen to meet the design needs.

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

entity counter is
  port (bus_d: in std_logic_vector(4 downto 0);
         bus_q: out std_logic_vector(4 downto 0);
         net_clock, net_aclr, net_enable: in std_logic;
         net_sload: in std_logic);
end counter;

architecture rtl of counter is

  -- Component Declaration
  component CNT5
    port (Data : in std_logic_vector(4 downto 0);Enable, Sload, Aclr, Clock: in std_logic; Q : out std_logic_vector(4 downto 0));
  end component;

  -- Binding statements to specify which CNT5 architecture to use
  -- RTL architecture for behavioral CNT5
  -- DEF_ARCH architecture for structural (ACTgen) CNT5
  -- for all: CNT5 use entity work.CNT5(RTL);
  -- for all: CNT5 use entity work.CNT5(DEF_ARCH);

begin
  -- Concurrent Statement
  U0: CNT5 port map (Data => bus_d,
                        Sload => net_sload,
                        Enable => net_enable,
                        Aclr => net_aclr;
                        Clock => net_clock,
                        Q => bus_q);
end rtl;
```
Chapter 4: Technology Specific Coding Techniques

SRAM

The following examples show how to create register-based SRAM for non-SRAM based Actel devices.

![Register Array (8x8)](image)

Figure 4-6. RAM Behavioral Simulation Model

Register-Based Single Port SRAM

The following example shows the behavioral model for a 8x8 RAM cell. To modify the width or depth, simply modify the listed parameters in the code. The code assumes that you want to use posedge clk and negedge reset. Modify the always blocks if that is not the case.

VHDL

```vhdl
-- *************************************************
-- Behavioral description of a single-port SRAM with:
--     Active High write enable (WE)
--     Rising clock edge (Clock)
-- *************************************************

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity reg_sram is
generic (width : integer:=8;
depth : integer:=8;
addr : integer:=3);
```
SRAM

port (Data : in std_logic_vector (width-1 downto 0);
    Q   : out std_logic_vector (width-1 downto 0);
    Clock : in std_logic;
    WE   : in std_logic;
    Address : in std_logic_vector (addr-1 downto 0));
end reg_sram;

architecture behav of reg_sram is
  type MEM is array (0 to depth-1) of std_logic_vector(width-1 downto 0);
  signal ramTmp : MEM;
begin
  process (Clock)
  begin
    if (clk'event and clk='1') then
      if (WE = '1') then
        ramTmp (conv_integer (Address)) <= Data;
      end if;
    end if;
  end process;
  Q <= ramTmp(conv_integer(Address));
end behav;

Verilog

`timescale 1 ns/100 ps
//########################################################
// Behavioral single-port SRAM description :
//    Active High write enable (WE)
//    Rising clock edge (Clock)
//########################################################
module reg_sram (Data, Q, Clock, WE, Address);
  parameter width = 8;
  parameter depth = 8;
  parameter addr = 3;
  input Clock, WE;
  input [addr-1:0] Address;
  input [width-1:0] Data;
  output [width-1:0] Q;
  wire [width-1:0] Q;
  reg [width-1:0] mem_data [depth-1:0];
always @(posedge Clock)
  if (WE)
    mem_data[Address] = #1 Data;
assign Q = mem_data[Address];
endmodule
The following example shows the behavioral model for an 8x8 RAM cell. This code was designed to imitate the behavior of the Actel DX family dual-port SRAM and to be synthesizable to a register-based SRAM module. To modify the width or depth, modify the listed parameters in the code. The code assumes that you want to use posedge clk and negedge reset. Modify the always blocks if that is not the case.

**VHDL**

```vhdl
-- Behavioral description of dual-port SRAM with:
--  Active High write enable (WE)
--  Active High read enable (RE)
--  Rising clock edge (Clock)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity reg_dpram is
  generic (width : integer:=8;
            depth : integer:=8;
            addr : integer:=3);
  port (Data : in std_logic_vector (width-1 downto 0);
        Q : out std_logic_vector (width-1 downto 0);
        Clock : in std_logic;
        WE : in std_logic;
        RE : in std_logic;
        WAddress: in std_logic_vector (addr-1 downto 0);
        RAddress: in std_logic_vector (addr-1 downto 0));
end reg_dpram;

architecture behav of reg_dpram is
  type MEM is array (0 to depth-1) of std_logic_vector(width-1 downto 0);
  signal ramTmp : MEM;
begin
  -- Write Functional Section
  process (Clock)
  begin
    if (clk'event and clk='1') then
      if (WE = '1') then
        ramTmp (conv_integer (WAddress)) <= Data;
      end if;
    end if;
  end process;

  -- Read Functional Section
  process (Clock)
```
begin
  if (clk'event and clk='1') then
    if (RE = '1') then
      Q <= ramTmp(conv_integer (RAddress));
    end if;
  end if;
end process;
end behav;

Verilog
'timescale 1 ns/100 ps

module reg_dpram (Data, Q, Clock, WE, RE, WAddress, RAddress);
parameter width = 8;
parameter depth = 8;
parameter addr = 3;
input Clock, WE, RE;
input [addr-1:0] WAddress, RAddress;
input [width-1:0] Data;
output [width-1:0] Q;
reg [width-1:0] Q;
reg [width-1:0] mem_data [depth-1:0];

always @(posedge Clock)
begin
  if (WE)
    mem_data[WAddress] = #1 Data;
end

always @(posedge Clock)
begin
  if (RE)
    Q = #1 mem_data[RAddress];
end
endmodule
ACTgen RAM

The RAM cells in the 3200DX family of devices support asynchronous and synchronous dual-port RAM. The basic RAM cells can be configured as 32x8 or 64x4. However, most synthesis tools cannot infer technology specific features such as RAM cells. The following example shows an ACTgen structural implementation for instantiation. Although the behavioral description is synthesizable, the implementation is not optimal for speed and area.

Using ACTgen, generate a 32x16 dual port RAM with the configuration shown in the figure below. Save the structured Verilog or VHDL implementations as “ram”.

**VHDL**

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

entity ram32_16 is
  port (WAddress, RAddress: in std_logic_vector(4 downto 0);
        Data : in std_logic_vector (15 downto 0);
        Aclr, WClock, RClock,WE,RE: in std_logic;
        Q : out std_logic_vector (15 downto 0));
end ram32_16;

architecture rtl of ram32_16 is
  component ram
    port (Data    : in std_logic_vector (15 downto 0);
          Aclr    : in std_logic;
          WE      : in std_logic;
          RE      : in std_logic;
          WClock  : in std_logic;
          RClock  : in std_logic;
          WAddress: in std_logic_vector (4 downto 0);
          RAddress: in std_logic_vector (4 downto 0);
          Q        : out std_logic_vector (15 downto 0));
  end component;
  signal foo: std_logic_vector (15 downto 0);
end rtl;
```


end component;

begin

R_32_16: ram
  port map (Data => Data,
             Aclr => Aclr,
             WE => WE,
             WAddress => WAddress,
             RE => RE,
             RAddress => RAddress,
             WClock => WClock,
             RClock => RClock,
             Q => Q);

end rtl;

Verilog

module ram (WAddress, RAddress, Data, WClock, WE,
            RE, Rclock, Q);
  input [4:0] WAddress, RAddress;
  input [15:0] Data;
  input Rclock, WClock;
  input WE, RE;
  output [15:0] Q;

RAM R_32_16 (.Data(Data), .WE(WE), .RE(RE), .WClock(WClock),
               .Rclock(Rclock), .Q(Q), .WAddress(WAddress),
               .RAddress(RAddress));

endmodule
Chapter 4: Technology Specific Coding Techniques

FIFO

The following example shows how to create a register-based FIFO for non-SRAM based Actel devices.

This code was designed to imitate the behavior of the Actel DX family dual-port SRAM based FIFO and to be synthesizeable to a register-based FIFO. To modify the width or depth, simply modify the listed parameters in the code. However, the code does assume that you want to use posedge clk and negedge reset. Modify the always blocks if that is not the case.

**VHDL**

```vhdl
-- *************************************************
-- Behavioral description of dual-port FIFO with :
--  Active High write enable (WE)
--  Active High read enable (RE)
--  Active Low asynchronous clear (Aclr)
--  Rising clock edge (Clock)
--  Active High Full Flag
--  Active Low Empty Flag
-- *************************************************
```

![Figure 4-7. FIFO Behavioral Simulation Mode](image-url)
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity reg_fifo is
  generic (width : integer:=8;
            depth : integer:=8;
            addr : integer:=3);
  port (Data : in std_logic_vector (width-1 downto 0);
         Q : out std_logic_vector (width-1 downto 0);
         Aclr : in std_logic;
         Clock : in std_logic;
         WE : in std_logic;
         RE : in std_logic;
         FF : out std_logic;
         EF : out std_logic);
end reg_fifo;

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

architecture behavioral of reg_fifo is

  type MEM is array(0 to depth-1) of std_logic_vector(width-1 downto 0);
  signal ramTmp : MEM;
  signal WAddress : std_logic_vector (addr-1 downto 0);
  signal RAddress : std_logic_vector (addr-1 downto 0);
  signal words : std_logic_vector (addr-1 downto 0);

  begin

    WRITE_POINTER : process (Aclr, Clock)
      begin
        if (Aclr = '0') then
          WAddress <= (others => '0');
        elsif (Clock'event and Clock = '1') then
          if (WE = '1') then
            if (WAddress = words) then
               ramTmp(WAddress) := Data;
               WAddress := RAddress;
            else
               WAddress := WAddress + 1;
            end if;
          end if;
        end if;
      end process;

default

  end behavioral;
Chapter 4: Technology Specific Coding Techniques

WAddress <= (others => '0');
else
  WAddress <= WAddress + '1';
end if;
end if;
end if;
end process;

WRITE_RAM : process (Clock)
begin
  if (Clock'event and Clock = '1') then
    if (WE = '1') then
      ramTmp (conv_integer (WAddress)) <= Data;
    end if;
  end if;
end process;

-- #########################################################
-- # Read Functional Section
-- #########################################################

READ_POINTER : process (Aclr, Clock)
begin
  if (Aclr = '0') then
    RAddress <= (others => '0');
  elsif (Clock'event and Clock = '1') then
    if (RE = '1') then
      if (RAddress = words) then
        RAddress <= (others => '0');
      else
        RAddress <= RAddress + '1';
      end if;
    end if;
  end if;
end process;

READ_RAM : process (Clock)
begin
  if (Clock'event and Clock = '1') then
    if (RE = '1') then
      Q <= ramTmp(conv_integer(RAddress));
    end if;
  end if;
end process;
-- # Full Flag Functional Section : Active high
-- ############################################################################

FFLAG : process (Aclr, Clock)
begin
  if (Aclr = '0') then
    FF <= '0';
  elsif (Clock'event and Clock = '1') then
    if (WE = '1' and RE = '0') then
      if ((WAddress = RAddress-1) or
           ((WAddress = depth-1) and (RAddress = 0))) then
        FF <= '1';
      end if;
    else
      FF <= '0';
    end if;
  end if;
end process;

-- # Empty Flag Functional Section : Active low
-- ############################################################################

EFLAG : process (Aclr, Clock)
begin
  if (Aclr = '0') then
    EF <= '0';
  elsif (Clock'event and Clock = '1') then
    if (RE = '1' and WE = '0') then
      if ((WAddress = RAddress+1) or
           ((RAddress = depth-1) and (WAddress = 0))) then
        EF <= '0';
      end if;
    else
      EF <= '1';
    end if;
  end if;
end process;
end behavioral;
Verilog

```verilog
module reg_fifo (Data, Q, Aclr, Clock, WE, RE, FF, EF);

// Behavioral description of FIFO with:
//  Active High write enable (WE)
//  Active High read enable (RE)
//  Active Low asynchronous clear (Aclr)
//  Rising clock edge (Clock)
//  Active High Full Flag
//  Active Low Empty Flag

parameter width = 8;
parameter depth = 8;
parameter addr = 3;

input Clock, WE, RE, Aclr;
input [width-1:0] Data;
output FF, EF;// Full & Empty Flags
output [width-1:0] Q;
reg [width-1:0] Q;
reg [width-1:0] mem_data [depth-1:0];
reg [addr-1:0] WAddress, RAddress;
reg FF, EF;

// Write Functional Section

always @ (posedge Clock or negedge Aclr)
begin
  if (!Aclr)
    WAddress = #2 0;
  else if (WE)
    WAddress = #2 WAddress + 1;
end

always @ (posedge Clock)
begin
  if (WE)
    mem_data[WAddress] = Data;
end

endmodule
```
// Read Functional Section
// bytesRead pointer
always @(posedge Clock or negedge Aclr)
begin
  if(!Aclr)
    RAddress = #1 0;
  else if (RE)
    RAddress = #1 RAddress + 1;
end

// Read register
always @(posedge Clock)
begin
  if (RE)
    Q = mem_data[RAddress];
end

// Full flag functional section : Active high
always @(posedge Clock or negedge Aclr)
begin
  if(!Aclr)
    FF = #1 1'b0;
  else if ((WE & !RE) && ((WAddress == RAddress-1) || ((WAddress == depth-1) && (RAddress == 1'b0))))
    FF = #1 1'b1;
  else
    FF = #1 1'b0;
end

// Empty flag functional section : Active low
always @(posedge Clock or negedge Aclr)
begin
  if(!Aclr)
    EF = #1 1'b0;
  else if (((!WE & RE) && ((WAddress == RAddress+1) || ((RAddress == depth-1) && (WAddress == 1'b0)))))
    EF = #1 1'b0;
  else
    EF = #1 1'b1;
end
endmodule
ACTgen FIFO

The RAM cells in the 3200DX family of devices can be used to implement a variety of FIFOs. The behavioral description of a 32x8 FIFO for simulation is shown below. However, most synthesis tools cannot infer technology specific features such as RAM cells. Synthesizing this model will result in high area utilization. ACTgen can generate an area and performance optimized structured HDL netlist for instantiation.

Using ACTgen, generate a 32x8 FIFO with the configuration shown in the figure below. Save it as a Verilog or VHDL netlist called “fifo_ff_ef.”

VHDL

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

entity fifo_32_8 is
    port (D : in std_logic_vector(7 downto 0);
          OUT : out std_logic_vector(7 downto 0);
          Reset : in std_logic;
          Rd_En, Wr_En : in std_logic;
          Rd_En_F, Wr_En_F : in std_logic;
          CLK : in std_logic;
          E_Flag, F_Flag : out std_logic);
end fifo_32_8;

architecture fifo_arch of fifo_32_8 is
    component fifo_ff_ef
        generic (width : integer;
                 depth : integer;
                 clrPola : integer;
                 clkEdge : integer);
        port (Data : in std_logic_vector (width-1 downto 0);
              Aclr : in std_logic;
              Clock : in std_logic;
              RE : in std_logic;
              FF : out std_logic;
              E.Flag : out std_logic;
              Q[7:0] : out std_logic);
    end component;
end fifo_arch;
```

92
FIFO

begin
F_32_8: fifo_ff_ef
  generic map (width => 8, depth => 32, clrPola => 1, clkEdge => 1)
  port map (Data => D,
             Aclr => Reset,
             WE => We_En,
             WEF => We_En_F,
             RE => Rd_En,
             REF => Rd_En_F,
             Clock => CLK,
             Q => OUT,
             FF => F_Flag,
             EF => E_Flag);
end fifo_arch;

Verilog
module fifo_32_8 (D, OUT, Reset, Rd_En, Wr_En, CLK, E_Flag,
                  Rd_En_F, Wr_En_F, F_Flag);
input [7:0] D;
output [7:0] OUT;
input Reset;
input Rd_En;
input Rd_En_F;
input Wr_En;
input Wr_En_F;
input CLK;
output E_Flag;
output F_Flag;
wire [7:0] OUT;
wire E_Flag;
wire F_Flag;

fifo_ff_ef F_32_8 (.Data(D), .Aclr(Reset), .WE(Wr_En),
                    .WEF(Wr_En_F), .RE(Rd_En), .REF(Rd_En_F),
                    .Clock(CLK), .Q(OUT), .FF(F_Flag), .EF(E_Flag));
endmodule
Product Support

Actel backs its products with various technical support services including a Web site, monthly Fax Broadcast, a fax back service, a bulletin board system, electronic mail, Customer Service, and a Customer Applications Center.

Actel U.S. Toll-Free Line

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

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

Customer Service

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

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

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.

Action Facts

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

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

Web Site

Actel has a World Wide Web home page where you can browse a variety of technical and non-technical information. Use a Web 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, Designer’s Digest, current manuals, and release notes.
HLD Web Site

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

Guru Automated Technical Support

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

FTP Site

Actel has an anonymous FTP site located at ftp://ftp.actel.com. You can directly obtain library updates (such as VITAL), 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.
Index

A
ACT 3 I/O 71
Actel FTP Site 97
Actel Web Site 96
   Technical Support 97
ACTgen
   Counter Instantiation 75
   FIFO 92
   RAM 84
Action Facts 96
Addition 29
Arithmetic Operator 29
   Shift Register Implementation 30
Assumptions x

B
Behavioral Simulation 2
BREPTH 71

C
Capturing a Design 2
Case Statement 19, 61
   Adding Directive 61
CLKBUF 73
CLKINT 73
Coding Dual Architecture 77
   Instantiating 79
   RTL 77
   Structural 78
Combining Logic 53, 56
Component
   Size 49
   Width 49
Contacting Actel
   Customer Service 95
   Electronic Mail 97
   Technical Support 96
   Toll-Free 95
   Web Based Technical Support 97
Conventions x
   Document x
   Naming, Verilog xii
   Naming, VHDL xi
Counter 25–29
   8-Bit, Count Enable, Asynchronous Reset 26
   8-Bit, Load and Asynchronous Reset 27
   8-Bit, Load, Count Enable, Terminal Count and
      Asynchronous Reset 28
   Instantiation 75
   N-Bit, Load, Count Enable, and Asynchronous
      Reset 28
   Recommendations 25, 75
Critical Path Logic Reduction 51
Customer Service 95

D
Data Shift 33
Datapath 17–34
   Arithmetic Operator 29
   Counter 25
   Decoder 24
   Equality Operator 32
   If-Then-Else 17
   Multiplexor 19
   Relational Operator 31
   Shift Operator 33
Decoder 24
Design Creation/Verification 2
   Behavioral Simulation 2
   EDIF Netlist Generation 3
   HDL Source Entry 2
Index

Structural Netlist Generation 3
Structural Simulation 3
Synthesis 2
Design Flow
  Design Creation/Verification 2
  Design Implementation 3
  Programming 4
  System Verification 4
Design Implementation 3
  Place and Route 3
  Timing Analysis 3
  Timing Simulation 4
Design Layout 3
Design Partitioning 60
Design Synthesis 2
Designer
  DT Analyze Tool 3
  Place and Route 3
  Timing Analysis 3
Device Programming 4
DFPC Cell 71
Division 29
D-Latch 5–17
  with Asynchronous Reset 16
  with Data and Enable 13
  with Gated Asynchronous Data 14
  with Gated Enable 15
Document Assumptions x
Document Conventions x
Document Organization ix
Don’t Care 23
DT Analyze 3
Dual Architecture Coding 77
  Instantiating 79
  RTL 77
  Structural 78
Dual Port SRAM 82, 84
Duplicating Logic 57

E
Edge-Triggered Memory Device 5
EDIF Netlist Generation 3
Electronic Mail 97
Equality Operator 32

F
Fanout
  High Fanout Networks 73, 75
  Reducing 57
FIFO 86–93
  ACTgen Implementation 92
  Behavioral Implementation 86
  Register-Based 86
  Structural Implementation 92
Finite State Machine 35–44
  Combinational Next State Logic 35
  Combinational Output Logic 35
  Mealy 37
  Moore 41
  One Hot 36
  Sequential Current State Register 35
  Structure 35
Flip-Flop 5–13
  See Also Register
  Positive Edge Triggered 6
  with Asynchronous Reset 8
  with Asynchronous Reset 7
  with Asynchronous Reset and Clock Enable 12
  with Asynchronous Reset and Preset 9
  with Synchronous Reset 11
  with Synchronous Reset 10
FSM. See Finite State Machine
Index

G
Gate-Level Netlist 2
Generating
   EDIF Netlist 3
   Gate-Level Netlist 2
   Structural Netlist 3
Generics 49–50
Greater Than 31
Greater Than Equal To 31

H
HDL Design Flow
   Design Creation/Verification 2
   Design Implementation 3
   Programming 4
   System Verification 4
HDL Source Entry 2
HLD Web Site 97

I
If-Then-Else Statement 17
Input-Output Buffer 44–48
   Bi-Directional 47
   Tri-State 45
Instantiating
   CKLBUF Driver 73
   CLKINT Driver 73
   Counters 75
   Dual Coding 77
   FIFO 92
   QCKLBUF Driver 75
   QCLKINT Driver 75
   RAM 84
   Registered I/Os 71
Internal Tri-State Mapping 62
Internally Generated Clock 73, 75

K
Keywords
   Verilog xii
   VHDL xi

L
Latch 5
   Master 71
   Slave 71
Less Than 31
Less Than Equal To 31
Level-Sensitive Memory Device 5
Load Reduction 57
Logic Level Reduction 51
Loops 55

M
Merging Logic Modules 56
Module Block Partitioning 60
Multiplexor 19, 61
   Case X 23
   Four to One 20
   Mapping Internal Tri-State to 62
   Moving Operators Outside Loops 55
   Twelve to One 21
   Multiplication 29

N
Naming Conventions
   Verilog xii
   VHDL xi
Netlist Generation
   EDIF 3
   Gate-Level 2
   Structural 3
Index

0
One Hot State Machine 36
Operator
  Arithmetic 29
  Equality 32
  Inside Loops 55
  Relational 31
  Removing from Loops 55
  Shift 33

P
Parallel Encoding 19
Parallel Operation 61
Parameters 49–50
Partitioning a Design 60
Performance Driven Coding 51–60
Place and Route 3
Priority Encoding 17
Product Support 95–97
  Action Facts 96
  Customer Applications Center 96
  Customer Service 95
  Electronic Mail 97
  FTP Site 97
  HLD Web Site 97
  Technical Support 96, 97
  Toll-Free Line 95
  Web Site 96
  Programming a Device 4

Q
QCLKBUF 75
QCLKINT 75
Quadrant Clock 75
  Limitations 75

R
RAM 84
Reducing Fanout 57
Reducing Logic
  on a Critical Path 51
  Usage 53
Register 64
  See Also Flip-Flop
  Asynchronous Preset 68
  Asynchronous Preset and Clear 71
  Clock Enabled 66
  Duplication 57
  Functionally Equivalent Asynchronous Preset 68
  Placed at Hierarchical Boundaries 60
  Recommended Usage 64–71
  Synchronous Clear or Preset 65
  Register-Based
    FIFO 86
    SRAM 80–83
      Dual Port 82
      Single Port 80
    Registered I/O 71
    BREPTH 71
    Related Manuals xiii
    Relational Operator 31
    Removing Operators from Loops 55
    Reset Signals 73, 75
    Resource Sharing 53

S
Sequential Device 5–13
  D-Latch 5
  Flip-Flop 5
  Sharing Resources 53
  Resource Sharing 53
  Shift Operator 33
Shift Register 30
Simulation
   Behavioral 2
   Structural 3
   Timing 4
Single Port SRAM 80
Size 49
SRAM 80–85
   ACTgen Implementation 84
   Dual Port 82
   Register Based 80
   Single Port 80
   Structural Implementation 84
Static Timing Analysis 3
Structural Netlist Generation 3
Structural Simulation 3
Subtraction 29
Synthesis 2
   Reducing Duration of 60
System Verification
   Action Probe 4
   Silicon Explorer 4

T
Technical Support 96
Technology Independent Coding 5–50
Technology Specific Coding 61–93
Timing Analysis 3
Timing Constraints 51
Timing Simulation 4
Toll-Free Line 95
Tri-State Mapping 62
True/False Operands 32

U
Unit Delays 2

V
Verilog
   Naming Conventions xii
   Reserved Words xii
VHDL
   Naming Conventions xi
   Reserved Words xi

W
Web Based Technical Support 97
Width 49