Chapter 4

Designing FPGAs with HDL

Xilinx FPGAs provide the benefits of custom CMOS VLSI and allow you to avoid the initial cost, time delay, and risk of conventional masked gate array devices. In addition to the logic in the CLBs and IOBs, the XC4000 family and XC5200 family FPGAs contain system-oriented features such as the following.

- Global low-skew clock or signal distribution network
- Wide edge decoders (XC4000 family only)
- On-chip RAM and ROM (XC4000 family and Spartan)
- IEEE 1149.1 — compatible boundary scan logic support
- Flexible I/O with Adjustable Slew-rate Control and Pull-up/ Pull-down Resistors
- 12-mA sink current per output and 24-mA sink per output pair
- Dedicated high-speed carry-propagation circuit

You can use these device characteristics to improve resource utilization and enhance the speed of critical paths in your HDL designs. The examples in this chapter are provided to help you incorporate these system features into your HDL designs.

This chapter includes the following sections.

- "Using Global Low-skew Clock Buffers" section
- "Using Dedicated Global Set/Reset Resource" section
- "Encoding State Machines" section
- "Using Dedicated I/O Decoders" section
- "Instantiating LogiBLOX Modules" section
- "Implementing Memory" section

- "Implementing Boundary Scan (JTAG 1149.1)" section
- "Implementing Logic with IOBs" section
- "Implementing Multiplexers with Tristate Buffers" section
- "Using Pipelining" section
- "Design Hierarchy" section

Using Global Low-skew Clock Buffers

For designs with global signals, use global clock buffers to take advantage of the low-skew, high-drive capabilities of the dedicated global buffer tree of the target device. When you use the Insert Pads command, the FPGA Compiler automatically inserts a BUFG generic clock buffer whenever an input signal drives a clock signal. The Xilinx implementation software automatically selects the clock buffer that is appropriate for your specified design architecture. If you want to use a specific global buffer, you must instantiate it.

You can instantiate an architecture-specific buffer if you understand the architecture and want to specify how the resources should be used. Each XC4000E/L and Spartan device contains four primary and four secondary global buffers that share the same routing resources. XC4000EX/XL/XV devices have sixteen global buffers; each buffer has its own routing resources. XC5200 devices have four dedicated global buffers in each corner of the device.

XC4000 EX/XL/XV devices have two different types of global buffer, Global Low-Skew Buffers (BUFGLS) and Global Early Buffers (BUFGE). Global Low-Skew buffers are standard global buffers that should be used for most internal clocking or high fanout signals that must drive a large portion of the device. There are eight BUFGLS buffers available, two in each corner of the device. The Global Early buffers are designed to provide faster clock access, but CLB access is limited to one quadrant of the device. I/O access is also limited. Similarly, there are eight BUFGEs, two in each corner of the device.

Because Global early and Global Low-Skew Buffers share a single pad, a single IPAD can drive a BUFGE, BUFGLS or both in parallel. The parallel configuration is especially useful for clocking the fast capture latches of the device. Since the Global Early and Global Low-Skew Buffers share a common input, they cannot be driven by two unique signals.

You can use the following criteria to help select the appropriate global buffer for a given design path.

- The simplest option is to use a Global Low-Skew Buffer.
- If you want a faster clock path, use a BUFG. Initially, the software will try to use a Global Low-Skew Buffer. If timing requirements are not met, a BUFGE is automatically used if possible.
- If a single quadrant of the chip is sufficient for the clocked logic, and timing requires a faster clock than the Global Low-Skew buffer, use a Global Early Buffer.

Note: For more information on using the XC4000 EX/XL/XV device family global buffers, refer to the online Xilinx Data Book or the Xilinx web site at http://www.xilinx.com.

For XC4000E/L and Spartan devices, you can use secondary global buffers (BUFGS) to buffer high-fanout, low-skew signals that are sourced from inside the FPGA. To access the secondary global clock buffer for an internal signal, instantiate the BUFGS_F cell. You can use primary global buffers (BUFGP) to distribute signals applied to the FPGA from an external source. Internal signals can be globally distributed with a primary global buffer, however, the signals must be driven by an external pin.

XC4000E devices have four primary (BUFGP) and four secondary (BUFGS) global clock buffers that share four global routing lines, as shown in the following figure.

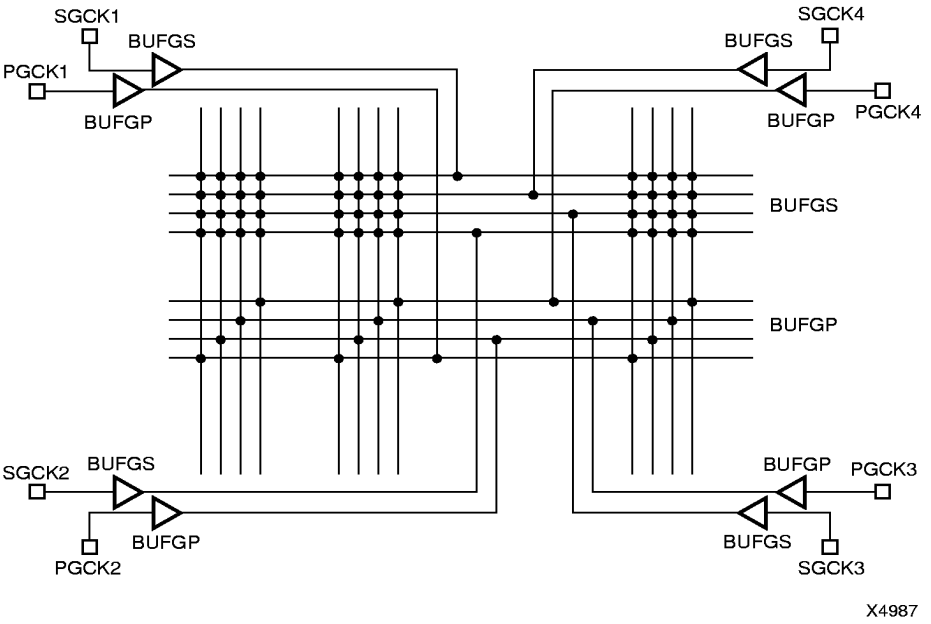

Figure 4-1 Global Buffer Routing Resources (XC4000E, Spartan)

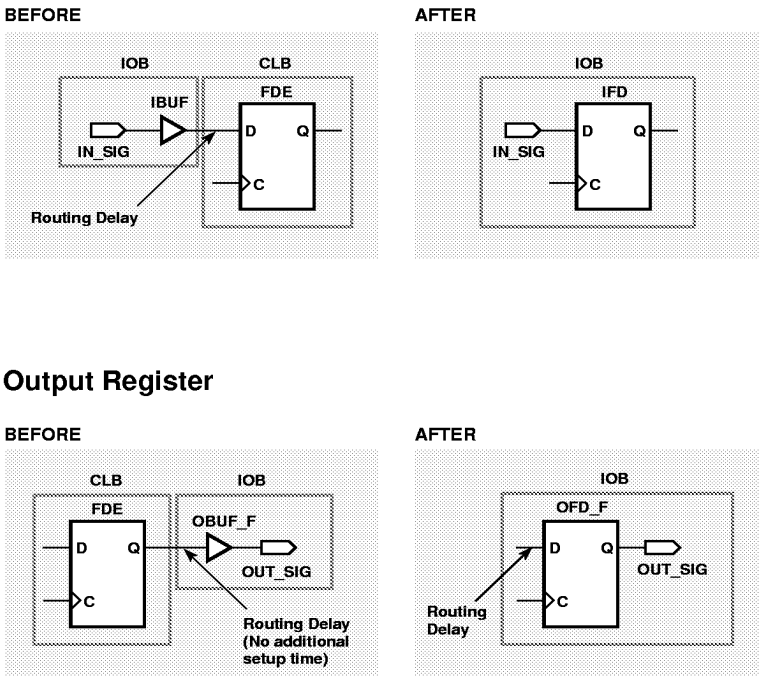

Figure 4-2 Global Buffer Routing Resources

These global routing resources are only available for the eight global buffers. The eight global nets run horizontally across the middle of the device and can be connected to one of the four vertical longlines that distribute signals to the CLBs in a column. Because of this arrangement only four of the eight global signals are available to the CLBs in a column. These routing resources are “free” resources because they are outside of the normal routing channels. Use these resources whenever possible. You may want to use the secondary buffers first because they have more flexible routing capabilities.

You should use the global buffer routing resources primarily for high-fanout clocks that require low skew, however, you can use them to drive certain CLB pins, as shown in the following figure. In addition,

you can use these routing resources to drive high-fanout clock enables, clear lines, and the clock pins (K) of CLBs and IOBs.

In the figure shown, the C pins drive the input to the H function generator, Direct Data-in, Preset, Clear, or Clock Enable pins. The F and G pins are the inputs to the F and G function generators, respectively.

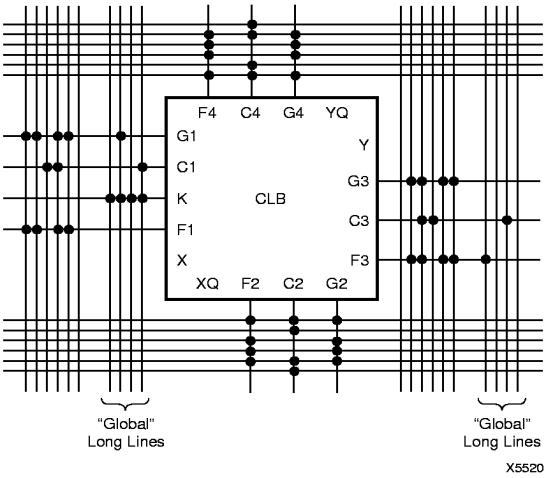

Figure 4-3 Global Longlines Resource CLB Connections

If your design does not contain four high-fanout clocks, use these routing resources for signals with the next highest fanout. To reduce routing congestion, use the global buffers to route high-fanout signals. These high-fanout signals include clock enables and reset signals (*not* global reset signals). Use global buffer routing resources to reduce routing congestion; enable routing of an otherwise unroutable design; and ensure that routing resources are available for critical nets.

Xilinx recommends that you assign up to four secondary global clock buffers to the four signals in your design with the highest fanout (such as clock nets, clock enables, and reset signals). Clock signals that require low skew have priority over low-fanout non-clock signals. You can source the signals with an input buffer or a gate internal to the design. Generate internally sourced clock signals with a register to avoid unwanted glitches. The synthesis tool can insert global clock buffers or you can instantiate them in your HDL code.

Note: Use Global Set/Reset resources when applicable. Refer to the "Using Dedicated Global Set/Reset Resource" section in this chapter for more information.

Inserting Clock Buffers

Note: Refer to the *Synopsys (XSI) Interface/Tutorial Guide* for more information on inserting I/O buffers and clock buffers.

Synopsys tools automatically insert a secondary global clock buffer on all input ports that drive a register's clock pin or a gated clock signal. To disable the automatic insertion of clock buffers and specify the ports that should have a clock buffer, perform the following steps.

1. In the Synopsys Compiler, ports that drive gated clocks or a register's clock pin are assigned a clock attribute. Remove this attribute from ports tagged with the clock attribute by typing:

   ```
   set_pad_type -no_clock "*"
   ```

2. Assign a clock attribute to the input ports that should have a BUFGS as follows:

   ```
   set_pad_type -clock {input ports}
   ```

3. Enter the following commands:

   ```
   set_port_is_pad "*"
   insert_pads
   ```

   The Insert Pads command causes the FPGA Compiler to automatically insert a clock buffer to ports tagged with a clock attribute.

To insert a global buffer other than a BUFGS, such as a BUFGP, use the following commands.

1. Use the following command on all ports with inferred buffers.

   ```
   set_port_is_pad "*"
   ```

2. Specify the buffer as follows.

   ```
   set_pad_type -exact BUFGP_F {port_list}
   ```

   You can replace BUFGP_F with another supported buffer for the device you are targeting. Refer to the *Synopsys (XSI) Interface/ Tutorial Guide* for a list of supported buffers. Port_list specifies the port(s) for the buffer.

3. Generate the buffers for the device as follows.

```
insert_pads
```

Instantiating Global Clock Buffers

You can instantiate global buffers in your code as described in this section.

Instantiating Buffers Driven from a Port

You can instantiate global buffers and connect them to high-fanout ports in your code rather than inferring them from a Synopsys script. If you do instantiate global buffers that are connected to a port, check your Synopsys script to make sure the Set Port Is Pad command is not specified for the buffer.

```
set_port_is_pad {list_of_all_ports_except_instantiated_buffer_port}
insert_pads
```

or

```
set_port_is_pad "*"
remove_attribute {instantiated_buffer_port} port_is_pad
insert_pads
```

Instantiating Buffers Driven from Internal Logic

You must instantiate a global buffer in your code in order to use the dedicated routing resource if a high-fanout signal is sourced from internal flip-flops or logic (such as a clock divider or multiplexed clock), or if a clock is driven from the internal oscillator or non-dedicated I/O pin. The following VHDL and Verilog examples are examples of instantiating a BUFGS for an internal multiplexed clock circuit. A Set Dont Touch attribute is added to the instantiated component.

- VHDL

```
-----------------------------------------------
-- CLOCK_MUX.VHD                             --
-- This is an example of an instantiation of --
-- global buffer (BUFGS) from an internally  --
-- driven signal, a multiplexed clock.       --
-- July 1997                                 --
-----------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity clock_mux is
port (DATA, SEL:         in  STD_LOGIC;
SLOW_CLOCK, FAST_CLOCK: in  STD_LOGIC;
    DOUT:                   out STD_LOGIC);
end clock_mux;

architecture XILINX of clock_mux is

signal CLOCK:      STD_LOGIC;
signal CLOCK_GBUF: STD_LOGIC;

component BUFGS
    port (I: in  STD_LOGIC;
          O: out STD_LOGIC);
end component;

begin

Clock_MUX: process (SEL)
    begin
        if (SEL = '1') then
                CLOCK <= FAST_CLOCK;
        else
            CLOCK <= SLOW_CLOCK;
        end if;
    end process;

GBUF_FOR_MUX_CLOCK: BUFGS
    port map (I => CLOCK,
              O => CLOCK_GBUF);

Data_Path: process (CLOCK_GBUF, DATA)
    begin
```

```
        if (CLOCK_GBUF'event and CLOCK_GBUF='1') then
            DOUT <= DATA;
        end if;
    end process;

end XILINX;
```

- Verilog

```
     // CLOCK_MUX.V                                  //
    // This is an example of an instantiation of //
   // global buffer (BUFGS) from an internally  //
  // driven signal, a multipled clock.         //
 // September 1997                            //
////////////////////////////////////////////////

module clock_mux (DATA, SEL, SLOW_CLOCK, FAST_CLOCK, DOUT);

    input  DATA, SEL;
    input  SLOW_CLOCK, FAST_CLOCK;
    output DOUT;

    reg   CLOCK;
    wire   CLOCK_GBUF;
    reg    DOUT;

    always @ (SEL)
    begin
        if (SEL == 1'b1)
            CLOCK <= FAST_CLOCK;
        else
            CLOCK <= SLOW_CLOCK;
    end

    BUFGS GBUF_FOR_MUX_CLOCK (.O(CLOCK_GBUF), .I(CLOCK));

    always @ (posedge CLOCK_GBUF)
        DOUT = DATA;

endmodule
```

Using Dedicated Global Set/Reset Resource

XC4000 and Spartan devices have a dedicated Global Set/Reset (GSR) net that you can use to initialize all CLBs and IOBs. When the

GSR is asserted, *every* flip-flop in the FPGA is simultaneously preset or cleared. You can access the GSR net from the GSR pin on the STARTUP block or the GSRIN pin of the STARTBUF (VHDL).

Since the GSR net has dedicated routing resources that connect to the Preset or Clear pin of the flip-flops, you do not need to use general purpose routing or global buffer resources to connect to these pins. If your design has a Preset or Clear signal that effects every flip-flop in your design, use the GSR net to increase design performance and reduce routing congestion.

The XC5200 family has a dedicated Global Reset (GR) net that resets all device registers. As in the XC4000 and Spartan devices, the STARTUP or STARTBUF (VHDL) block must be instantiated in your code in order to access this resource. The XC3000A devices also have dedicated Global Reset (GR) that is connected to a dedicated device pin (see device pinout). Since this resource is always active, you do not need to do anything to activate this feature.

For XC4000, Spartan, and XC5200 devices, the Global Set/Reset (GSR or GR) signal is, by default, set to active high (globally resets device when logic equals 1). For an active low reset, you can instantiate an inverter in your code to invert the global reset signal. The inverter is absorbed by the STARTUP block and does not use any device resources (function generators). Even though the inverted signal may be behaviorally described in your code, Xilinx recommends instantiating the inverter to prevent the mapping of the inverter into a CLB function generator, and subsequent delays to the reset signal and unnecessary use of device resources Also make sure you put a Don't Touch attribute on the instantiated inverter before compiling your design. If you do not add this attribute, the inverter may get mapped into a CLB function generator.

Note: For more information on simulating the Global Set/Reset, see the "Simulating Your Design" chapter.

Startup State

The GSR pin on the STARTUP block or the GSRIN pin on the STARTBUF block drives the GSR net and connects to each flip-flop's Preset and Clear pin. When you connect a signal from a pad to the STARTUP block's GSR pin, the GSR net is activated. Since the GSR net is built into the silicon it does not appear in the pre-routed netlist file. When the GSR signal is asserted High (the default), all flip-flops

and latches are set to the state they were in at the end of configuration. When you simulate the routed design, the gate simulator translation program correctly models the GSR function.

Note: For the XC3000 family and the XC5200 family, all flip-flops and latches are reset to zero after configuration.

Preset vs. Clear (XC4000, Spartan)

The XC4000 family flip-flops are configured as either preset (asynchronous set) or clear (asynchronous reset). Automatic assertion of the GSR net presets or clears each flip-flop. You can assert the GSR pin at any time to produce this global effect. You can also preset or clear individual flip-flops with the flip-flop's dedicated Preset or Clear pin. When a Preset or Clear pin on a flip-flop is connected to an active signal, the state of that signal controls the startup state of the flip-flop. For example, if you connect an active signal to the Preset pin, the flip-flop starts up in the preset state. If you do not connect the Clear or Preset pin, the default startup state is a clear state. To change the default to preset, assign an INIT=S attribute to the flip-flop from the Synopsys run script, as follows.

```
set_attribute "cell" fpga_xilinx_init_state -type
string "S"
```

I/O flip-flops and latches do not have individual Preset or Clear pins. The default value of these flip-flops and latches is clear. To change the default value to preset, assign an INIT=S attribute.

Note: Refer to the *Synopsys (XSI) Interface/Tutorial Guide* for information on changing the initial state of registers that do not use the Preset or Clear pins.

Increasing Performance with the GSR/GR Net

Many designs contain a net that initializes most of the flip-flops in the design. If this signal can initialize *all* the flip-flops, you can use the GSR/GR net. You should always include a net that initializes your design to a known state.

To ensure that your HDL simulation results at the RTL level match the synthesis results, write your code so that every flip-flop and latch is preset or cleared when the GSR signal is asserted. The Synthesis tool cannot infer the GSR/GR net from HDL code. To utilize the GSR

net, you must instantiate the STARTUP or STARTBUF block (VHDL), as shown in the "No_GSR Implemented with Gates" figure.

Design Example without Dedicated GSR/GR Resource

In the following VHDL and Verilog designs, the RESET signal initializes all the registers in the design; however, it does not use the dedicated global resources. The RESET signal is routed using regular routing resources. These designs include two 4-bit counters. One counter counts up and is reset to all zeros on assertion of RESET and the other counter counts down and is reset to all ones on assertion of RESET. The "No_GSR Implemented with Gates" figure shows the No_GSR design implemented with gates.

- VHDL - No GSR

```
-- NO_GSR Example
-- The signal RESET initializes all registers
-- May 1997

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

entity no_gsr is
port (CLOCK: in STD_LOGIC;
      RESET: in STD_LOGIC;
      UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
      DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end no_gsr;

architecture SIMPLE of no_gsr is

signal UP_CNT: STD_LOGIC_VECTOR (3 downto 0);
signal DN_CNT: STD_LOGIC_VECTOR (3 downto 0);

begin
    UP_COUNTER: process (CLOCK, RESET)
    begin
        if (RESET = '1') then
            UP_CNT <= "0000";
        elsif (CLOCK'event and CLOCK = '1') then
            UP_CNT <= UP_CNT + 1;
        end if;
```

```
    end process;

    DN_COUNTER: process (CLOCK, RESET)
    begin
        if (RESET = '1') then
            DN_CNT <= "1111";
        elsif (CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;

    UPCNT <= UP_CNT;
    DNCNT <= DN_CNT;

end SIMPLE;
```

- VHDL - No GR

```
-- NO_GR.VHD Example
-- The signal RESET initializes all registers
-- Without the use of the dedicated Global Reset routing
-- December 1997

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

entity no_gr is
port (CLOCK: in STD_LOGIC;
      RESET: in STD_LOGIC;
      UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
      DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end no_gr;

architecture XILINX of no_gr is

signal UP_CNT: STD_LOGIC_VECTOR (3 downto 0);
signal DN_CNT: STD_LOGIC_VECTOR (3 downto 0);

begin
    UP_COUNTER: process (CLOCK, RESET)
    begin
        if (RESET = '1') then
            UP_CNT <= "0000";
        elsif (CLOCK'event and CLOCK = '1') then
```

```
            UP_CNT <= UP_CNT + 1;
        end if;
    end process;

    DN_COUNTER: process (CLOCK, RESET)
    begin
        if (RESET = '1') then
            DN_CNT <= "1111";
        elsif (CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;

    UPCNT <= UP_CNT;
    DNCNT <= DN_CNT;

end XILINX;
```

- Verilog - No GSR

```
/* NO_GSR Example
 * Synopsys HDL Synthesis Design Guide for FPGAs
 * The signal RESET initializes all registers
 * December 1997 */

module no_gsr ( CLOCK, RESET, UPCNT, DNCNT);

input CLOCK, RESET;
output [3:0] UPCNT;
output [3:0] DNCNT;

reg [3:0] UPCNT;
reg [3:0] DNCNT;

always @ (posedge CLOCK or posedge RESET) begin
    if (RESET) begin
        UPCNT = 4'b0000;
        DNCNT = 4'b1111;
    end else begin
        UPCNT = UPCNT + 1'b1;
        DNCNT = DNCNT - 1'b1;
    end
end
endmodule
```

- Verilog - No GR

```
/* NO_GR.V Example
 * The signal RESET initializes all registers
 * Aug 1997 */

module no_gr ( CLOCK, RESET, UPCNT, DNCNT);

input CLOCK, RESET;
output [3:0] UPCNT;
output [3:0] DNCNT;

reg [3:0] UPCNT;
reg [3:0] DNCNT;

always @ (posedge CLOCK or posedge RESET) begin
    if (RESET) begin
        UPCNT = 4'b0000;
        DNCNT = 4'b1111;
    end else begin
        UPCNT = UPCNT + 1'b1;
        DNCNT = DNCNT - 1'b1;
    end
end

endmodule
```

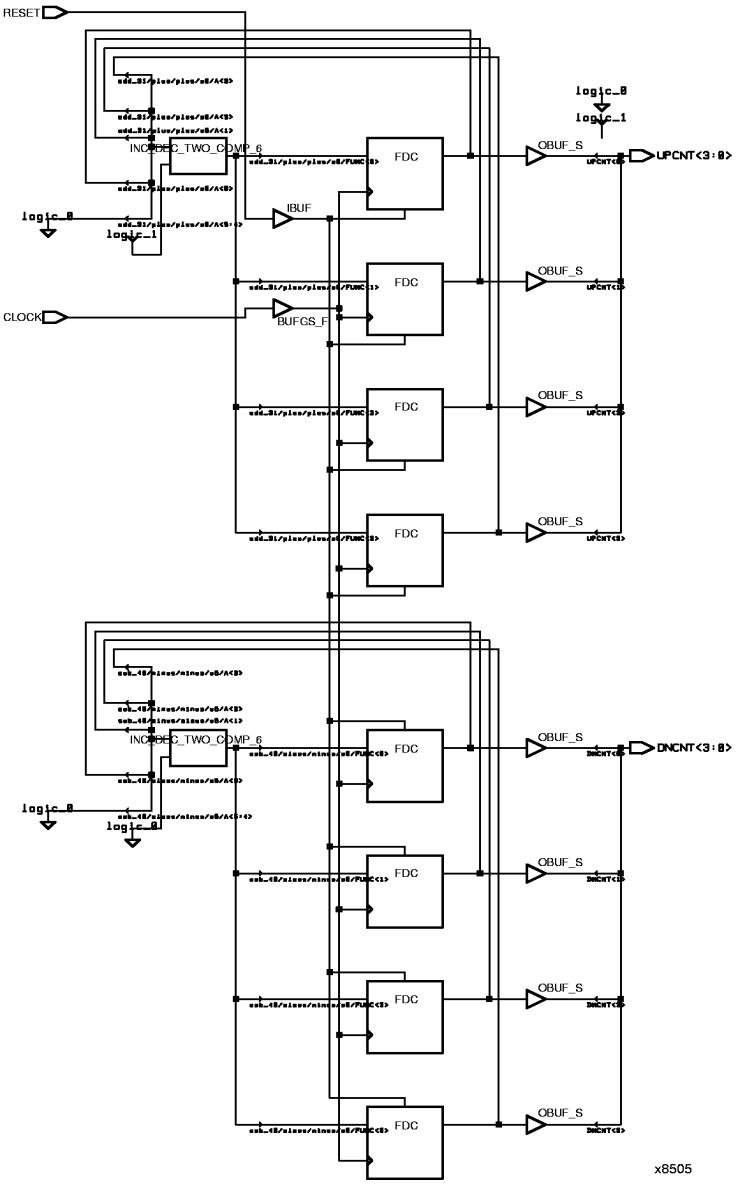

Figure 4-4 No_GSR Implemented with Gates

Design Example with Dedicated GSR/GR Resource

To reduce routing congestion and improve the overall performance of the reset net in the No_GSR and No_GR designs, use the dedicated GSR or GR net instead of the general purpose routing. Instantiate the STARTUP or STARTBUF block in your design and use the GSR or GR pin on the STARTUP block (or the GSRIN pin on the STARTBUF block) to access the global reset net. The modified designs (Use_GSR and Use_GR) are included at the end of this section. The Use_GSR design implemented with gates is shown in "Use_GSR Implemented with Gates" figure.

In XC4000 and Spartan designs, on assertion of the GSR net, flip-flops return to a clear (or Low) state by default. You can override this default by describing an asynchronous preset in your code, or by adding the INIT=S attribute to the flip-flop (described later in this section) from the Synopsys run script.

In XC5200 family designs, the GR resets all flip-flops in the device to a logic zero. If a flip-flop is described as asynchronous preset to a logic 1, Synopsys automatically infers a flip-flop with a synchronous preset, and the M1 software puts an inverter on the input and output of the device to simulate a preset.

The Use_GSR and Use_GR designs explicitly state that the down-counter resets to all ones, therefore, asserting the reset net causes this counter to reset to a default of all zeros. You can use the INIT = S attribute to prevent this reset to zeros.

- Attach the INIT=S attribute to the down-counter flip-flops as follows:

  `set_attribute` *cell name* **`fpga_xilinx_init_state -type string "S"`**

Note: The "\" character represents a continuation marker.

This command allows you to override the default clear (or Low) state when your code does not specify a preset condition. However, since attributes are assigned outside the HDL code, the code no longer accurately represents the behavior of the design.

Note: Refer to the *Synopsys (XSI) Interface/Tutorial Guide* for more information on assigning attributes.

The STARTUP block must not be optimized during the synthesis process. Add a Don't Touch attribute to the STARTUP or STARTBUF block before compiling the design as follows:

set_dont_touch *cell_instance_name*

- VHDL - Use_GSR (XC4000 family)

```
-- USE_GSR.VHD Example
-- The signal RESET is connected to the GSRIN pin of
-- the STARTBUF block
-- May 1997

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

entity use_gsr is
port ( CLOCK: in STD_LOGIC;
       RESET: in STD_LOGIC;
       UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
       DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end use_gsr;

architecture XILINX of use_gsr is

component STARTBUF
    port (GSRIN: in STD_LOGIC);
          GSROUT: out STD_LOGIC);
end component;

signal RESET_INT: STD_LOGIC;
signal UP_CNT: STD_LOGIC_VECTOR (3 downto 0);
signal DN_CNT: STD_LOGIC_VECTOR (3 downto 0);

begin

    U1: STARTBUF port map(GSRIN=>RESET, GSROUT=>RESET_INT);

    UP_COUNTER: process(CLOCK, RESET_INT)
    begin
        if (RESET_INT = '1') then
            UP_CNT <= "0000";
        elsif CLOCK'event and CLOCK = '1') then
```

```
            UP_CNT <= UP_CNT - 1;
        end if;
    end process;

    DN_COUNTER: (CLOCK, RESET_INT)
    begin
        if (RESET_INT = '1') then
            DN_CNT <= "1111";
        elsif CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;

    UPCNT <= UP_CNT;
    DNCNT <= DN_CNT;

end XILINX;
```

- VHDL - Use_GR

```
-----------------------------------------------
-- USE_GR.VHD Version 1.0                     --
-- Xilinx HDL Synthesis Design Guide          --
-- The signal RESET initializes all registers --
-- Using the global reset resources since     --
-- STARTBUF block was added                   --
-- December 1997                              --
-----------------------------------------------

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

entity use_gr is
port ( CLOCK: in STD_LOGIC;
       RESET: in STD_LOGIC;
       UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
       DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end use_gr;

architecture XILINX of use_gr is
```

```
component STARTBUF
    port (GSRIN:  in  STD_LOGIC;
          GSROUT: out STD_LOGIC);
end component;

signal RESET_INT: STD_LOGIC;
signal UP_CNT:    STD_LOGIC_VECTOR (3 downto 0);
signal DN_CNT:    STD_LOGIC_VECTOR (3 downto 0);

begin

    U1: STARTBUF port map(GSRIN=>RESET, GSROUT=>RESET_INT);

    UP_COUNTER: process(CLOCK, RESET_INT)
    begin
        if (RESET_INT = '1') then
            UP_CNT <= "0000";
        elsif (CLOCK'event and CLOCK = '1') then
            UP_CNT <= UP_CNT + 1;
        end if;
    end process;

    DN_COUNTER: process(CLOCK, RESET_INT)
    begin
        if (RESET_INT = '1') then
            DN_CNT <= "1111";
        elsif (CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;

    UPCNT <= UP_CNT;
    DNCNT <= DN_CNT;

end XILINX;
```

- Verilog - Use_GSR

```
//////////////////////////////////////////////
// USE_GSR.V Version 1.0                    //
// Xilinx HDL Synthesis Design Guide        //
// The signal RESET initializes all registers //
// Using the global reset resources (STARTUP) //
// December 1997                            //
//////////////////////////////////////////////
```

```
module use_gsr ( CLOCK, RESET, UPCNT, DNCNT);

input CLOCK, RESET;
output [3:0] UPCNT;
output [3:0] DNCNT;

reg [3:0] UPCNT;
reg [3:0] DNCNT;

STARTUP U1 (.GSR(RESET));

always @ (posedge CLOCK or posedge RESET) begin
    if (RESET) begin
        UPCNT = 4'b0000;
        DNCNT = 4'b1111;
    end else begin
        UPCNT = UPCNT + 1'b1;
        DNCNT = DNCNT - 1'b1;
    end
end

endmodule
```

- Verilog - Use_GR

```
       ////////////////////////////////////////////
      // USE_GR.V Version 1.0                     //
     // Xilinx HDL Synthesis Design Guide        //
    // The signal RESET initializes all registers //
   // Using the global reset resources since     //
  // STARTUP block instantiation was added      //
 // December 1997                              //
////////////////////////////////////////////

module use_gr ( CLOCK, RESET, UPCNT, DNCNT);

input CLOCK, RESET;
output [3:0] UPCNT;
output [3:0] DNCNT;

reg [3:0] UPCNT;
reg [3:0] DNCNT;

STARTUP U1 (.GR(RESET));
```

```
always @ (posedge CLOCK or posedge RESET) begin
    if (RESET) begin
        UPCNT = 4'b0000;
        DNCNT = 4'b1111;
    end else begin
        UPCNT = UPCNT + 1'b1;
        DNCNT = DNCNT - 1'b1;
    end
end

endmodule
```

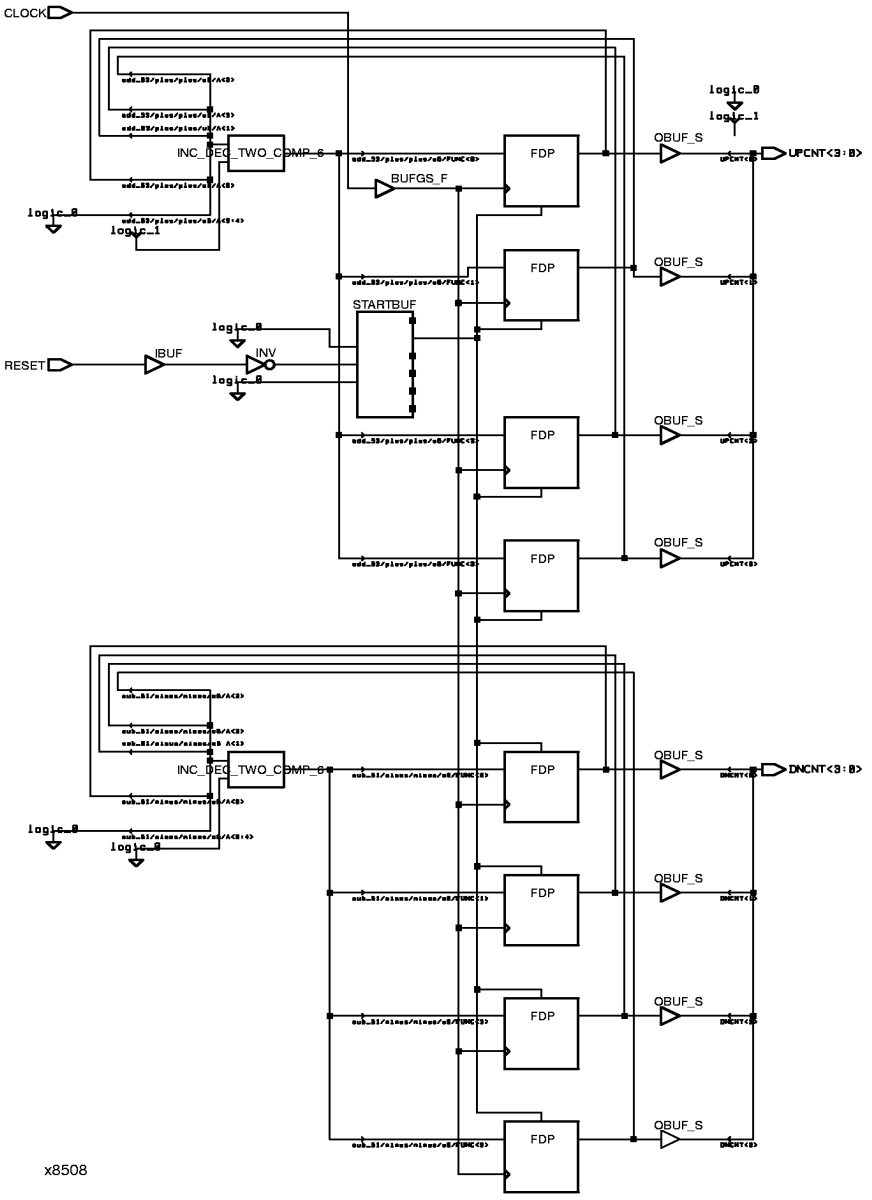

Figure 4-5 Use_GSR Implemented with Gates

Design Example with Active Low GSR/GR Signal

The Active_Low_GSR design is identical to the Use_GSR design except an INV is instantiated and connected between the RESET port and the STARTUP block. Also, a Set_dont_touch attribute is added to the Synopsys script for both the INV and STARTUP or STARTBUF (VHDL) symbols. By instantiating the inverter, the global set/reset signal is now active low (logic level 0 resets all FPGA flip-flops). The inverter is absorbed into the STARTUP block in the device and no CLB resources are used to invert the signal. VHDL and Verilog Active_Low_GSR designs are shown following.

- VHDL - Active_Low_GSR

```
------------------------------------------------------
-- ACTIVE_LOW_GSR.VHD Version 1.0                   --
-- Xilinx HDL Synthesis Design Guide                --
-- The signal RESET is inverted before being        --
-- connected to the GSRIN pin of the STARTBUF       --
-- The inverter will be absorbed by the STARTBUF    --
-- September 1997                                   --
------------------------------------------------------

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

entity active_low_gsr is
    port ( CLOCK: in STD_LOGIC;
           RESET: in STD_LOGIC;
           UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
           DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end active_low_gsr;

architecture XILINX of active_low_gsr is

    component INV
        port (I: in  STD_LOGIC;
              O: out STD_LOGIC);
    end component;

    component STARTBUF
        port (GSRIN:  in  STD_LOGIC;
              GSROUT: out STD_LOGIC);
```

```
    end component;

    signal RESET_NOT:     STD_LOGIC;
    signal RESET_NOT_INT: STD_LOGIC;
    signal UP_CNT:        STD_LOGIC_VECTOR (3 downto 0);
    signal DN_CNT:        STD_LOGIC_VECTOR (3 downto 0);

    begin

    U1: INV port map(I => RESET, O => RESET_NOT);

    U2: STARTBUF port map(GSRIN=>RESET_NOT,
                          GSROUT=>RESET_NOT_INT);

    UP_COUNTER: process(CLOCK, RESET_NOT_INT)
    begin
        if (RESET_NOT_INT = '1') then
            UP_CNT <= "0000";
        elsif (CLOCK'event and CLOCK = '1') then
            UP_CNT <= UP_CNT + 1;
        end if;
    end process;

    DN_COUNTER: process(CLOCK, RESET_NOT_INT)
    begin
        if (RESET_NOT_INT = '1') then
            DN_CNT <= "1111";
        elsif (CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;

    UPCNT <= UP_CNT;
    DNCNT <= DN_CNT;

end XILINX;
```

- Verilog - Active_low_GSR

```
     ////////////////////////////////////////////////
    // ACTIVE_LOW_GSR.V Version 1.0                 //
   // Xilinx HDL Synthesis Design Giude            //
  // The signal RESET is inverted before being     //
 // connected to the GSR pin of the STARTUP block  //
// The inverter will be absorbed by STARTUP in M1 //
```

```
   // Inverter is instantiated to avoid being mapped //
  // into a LUT by Synopsys                          //
 // September 1997                                  //
////////////////////////////////////////////////////

module active_low_gsr ( CLOCK, RESET, UPCNT, DNCNT);

   input        CLOCK, RESET;
   output [3:0] UPCNT;
   output [3:0] DNCNT;

   wire       RESET_NOT;
   reg  [3:0] UPCNT;
   reg  [3:0] DNCNT;

   INV U1 (.O(RESET_NOT), .I(RESET));

   STARTUP U2 (.GSR(RESET_NOT));

   always @ (posedge CLOCK or posedge RESET_NOT)
   begin
      if (RESET_NOT)
      begin
         UPCNT = 4'b0000;
         DNCNT = 4'b1111;
      end
      else
      begin
         UPCNT = UPCNT + 1'b1;
         DNCNT = DNCNT - 1'b1;
      end
   end

endmodule
```

Encoding State Machines

The traditional methods used to generate state machine logic result in highly-encoded states. State machines with highly-encoded state variables typically have a minimum number of flip-flops and wide combinatorial functions. These characteristics are acceptable for PAL and gate array architectures. However, because FPGAs have many flip-flops and narrow function generators, highly-encoded state variables can result in inefficient implementation in terms of speed and density.

One-hot encoding allows you to create state machine implementations that are more efficient for FPGA architectures. You can create state machines with one flip-flop per state and decreased width of combinatorial logic. One-hot encoding is usually the preferred method for large FPGA-based state machine implementation. For small state machines (fewer than 8 states), binary encoding may be more efficient. To improve design performance, you can divide large (greater than 32 states) state machines into several small state machines and use the appropriate encoding style for each.

Three design examples are provided in this section to illustrate the three coding methods (binary, enumerated type, and one-hot) you can use to create state machines. All three examples contain an identical Case statement. To conserve space, the complete Case statement is only included in the binary encoded state machine example; refer to this example when reviewing the enumerated type and one-hot examples.

Note: The bold text in each of the three examples indicates the portion of the code that varies depending on the method used to encode the state machine.

Using Binary Encoding

The state machine bubble diagram in the following figure shows the operation of a seven-state machine that reacts to inputs A through E as well as previous-state conditions. The binary encoded method of coding this state machine is shown in the VHDL and Verilog examples that follow. These design examples show you how to take a design that has been previously encoded (for example, binary encoded) and synthesize it to the appropriate decoding logic and registers. These designs use three flip-flops to implement seven states.

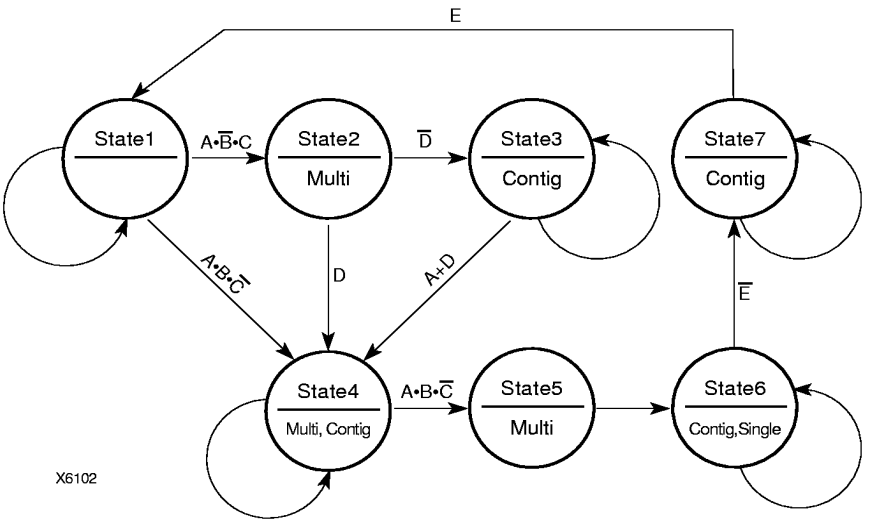

Figure 4-6 State Machine Bubble Diagram

VHDL - Binary Encoded State Machine Example

```
-------------------------------------------------
-- BINARY.VHD Version 1.0                       --
-- Example of a binary encoded state machine    --
-- Xilinx HDL Synthesis Design Guide for FPGAs --
-- May 1997                                     --
-------------------------------------------------
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity binary is
    port (CLOCK, RESET : in STD_LOGIC;
          A, B, C, D, E: in BOOLEAN;
          SINGLE, MULTI, CONTIG: out STD_LOGIC);
end binary;

architecture BEHV of binary is

type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of STATE_TYPE:type is "001 010 011 100 101 110
111";
```

```
signal CS, NS: STATE_TYPE;

begin

    SYNC_PROC: process (CLOCK, RESET)
    begin
        if (RESET='1') then
           CS <= S1;
        elsif (CLOCK'event and CLOCK = '1') then
           CS <= NS;
        end if;
    end process; --End REG_PROC

    COMB_PROC: process (CS, A, B, C, D, E)
    begin
        case CS is
           when S1 =>
               MULTI  <= '0';
               CONTIG <= '0';
               SINGLE <= '0';
            if (A and not B and C) then
                NS <= S2;
            elsif (A and B and not C) then
                NS <= S4;
            else
                NS <= S1;
            end if;
           when S2 =>
               MULTI  <= '1';
               CONTIG <= '0';
               SINGLE <= '0';
               if (not D) then
                   NS <= S3;
               else
                   NS <= S4;
               end if;
           when S3 =>
                MULTI  <= '0';
                CONTIG <= '1';
                SINGLE <= '0';
                if (A or D) then
                   NS <= S4;
                else
                   NS <= S3;
```

```
            end if;
         when S4 =>
            MULTI  <= '1';
            CONTIG <= '1';
            SINGLE <= '0';
            if (A and B and not C) then
               NS <= S5;
            else
               NS <= S4;
            end if;
         when S5 =>
            MULTI  <= '1';
            CONTIG <= '0';
            SINGLE <= '0';
            NS <= S6;
         when S6 =>
            MULTI  <= '0';
            CONTIG <= '1';
            SINGLE <= '1';
            if (not E) then
                NS <= S7;
            else
                NS <= S6;
            end if;
         when S7 =>
            MULTI  <= '0';
            CONTIG <= '1';
            SINGLE <= '0';
            if (E) then
                NS <= S1;
            else
                NS <= S7;
            end if;
      end case;
   end process; -- End COMB_PROC

end BEHV;
```

Verilog - Binary Encoded State Machine Example

```
    ////////////////////////////////////////////////
   // BINARY.V Version 1.0                         //
  // Example of a binary encoded state machine   //
 // Xilinx HDL Synthesis Design Guide for FPGAs //
// May 1997                                    //
////////////////////////////////////////////////
```

```
module binary (CLOCK, RESET, A, B, C, D, E,
               SINGLE, MULTI, CONTIG);

input    CLOCK, RESET;
input    A, B, C, D, E;
output   SINGLE, MULTI, CONTIG;

reg      SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [2:0] //synopsys enum STATE_TYPE
    S1 = 3'b001,
    S2 = 3'b010,
    S3 = 3'b011,
    S4 = 3'b100,
    S5 = 3'b101,
    S6 = 3'b110,
    S7 = 3'b111;

// Declare current state and next state variables
reg [2:0] /* synopsys enum STATE_TYPE */ CS;
reg [2:0] /* synopsys enum STATE_TYPE */ NS;

// synopsys state_vector CS

    always @ (posedge CLOCK or posedge RESET)
    begin
        if (RESET == 1'b1)
            CS = S1;
        else
            CS = NS;
    end

    always @ (CS or A or B or C or D or D or E)
    begin
    case (CS) //synopsys full_case
            S1 :
            begin
                MULTI  = 1'b0;
                CONTIG = 1'b0;
                SINGLE = 1'b0;
            if (A && ~B && C)
                NS = S2;
            else if (A && B && ~C)
```

```
     NS = S4;
 else
     NS = S1;
end
S2 :
begin
     MULTI  = 1'b1;
     CONTIG = 1'b0;
     SINGLE = 1'b0;
     if (!D)
         NS = S3;
     else
         NS = S4;
end
S3 :
begin
     MULTI  = 1'b0;
     CONTIG = 1'b1;
     SINGLE = 1'b0;
     if (A || D)
         NS = S4;
     else
        NS = S3;
end
S4 :
begin
     MULTI  = 1'b1;
     CONTIG = 1'b1;
     SINGLE = 1'b0;
     if (A && B && ~C)
         NS = S5;
     else
         NS = S4;
end
S5 :
begin
     MULTI  = 1'b1;
     CONTIG = 1'b0;
     SINGLE = 1'b0;
       NS = S6;
end
S6 :
begin
     MULTI  = 1'b0;
     CONTIG = 1'b1;
```

```
                SINGLE = 1'b1;
                if (!E)
                    NS = S7;
                else
                    NS = S6;
            end
            S7 :
            begin
                MULTI  = 1'b0;
                CONTIG = 1'b1;
                SINGLE = 1'b0;
                if (E)
                    NS = S1;
                else
                    NS = S7;
            end
        endcase
  end

endmodule
```

Using Enumerated Type Encoding

The recommended encoding style for state machines depends on which synthesis tool you are using. You can explicitly declare state vectors or you can allow the Synopsys tool to determine the vectors. Synopsys recommends that you use enumerated type encoding to specify the states and use the Finite State Machine (FSM) extraction commands to extract and encode the state machine as well as to perform state minimization and optimization algorithms. The enumerated type method of encoding the seven-state machine is shown in the following VHDL and Verilog examples. The encoding style is not defined in the code, but can be specified later with the FSM extraction commands. Alternatively, you can allow the Synopsys compiler to select the encoding style that results in the lowest gate count when the design is synthesized.

Note: Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code.

VHDL- Enumerated Type Encoded State Machine Example

```
Library IEEE;
use IEEE.std_logic_1164.all;

entity enum is
    port (CLOCK, RESET : in STD_LOGIC;
          A, B, C, D, E: in BOOLEAN;
          SINGLE, MULTI, CONTIG: out STD_LOGIC);
end enum;

architecture BEHV of enum is

type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);

signal CS, NS: STATE_TYPE;

begin

    SYNC_PROC: process (CLOCK, RESET)
    begin
             if (RESET='1') then
                 CS <= S1;
             elsif (CLOCK'event and CLOCK = '1') then
             CS <= NS;
             end if;
    end process; --End SYNC_PROC

    COMB_PROC: process (CS, A, B, C, D, E)
    begin
            case CS is
            when S1 =>
                       MULTI  <= '0';
                       CONTIG <= '0';
                       SINGLE <= '0';
   .
   .
   .
```

Verilog - Enumerated Type Encoded State Machine Example

```
     ////////////////////////////////////////////////
    // ENUM.V Version 1.0                          //
   // Example of an enumerated encoded state machine //
  // Xilinx HDL Synthesis Design Guide for FPGAs    //
 // May 1997                                      //
////////////////////////////////////////////////

module enum (CLOCK, RESET, A, B, C, D, E,
              SINGLE, MULTI, CONTIG);

input  CLOCK, RESET;
input  A, B, C, D, E;
output SINGLE, MULTI, CONTIG;

reg    SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [2:0] //synopsys enum STATE_TYPE
    S1 = 3'b000,
    S2 = 3'b001,
    S3 = 3'b010,
    S4 = 3'b011,
    S5 = 3'b100,
    S6 = 3'b101,
    S7 = 3'b110;

// Declare current state and next state variables
reg [2:0] /* synopsys enum STATE_TYPE */ CS;
reg [2:0] /* synopsys enum STATE_TYPE */ NS;

// synopsys state_vector CS

    always @ (posedge CLOCK or posedge RESET)
    begin
        if (RESET == 1'b1)
            CS = S1;
        else
            CS = NS;
    end

    always @ (CS or A or B or C or D or D or E)
    begin
```

```
       case (CS) //synopsys full_case
            S1 :
            begin
            MULTI  = 1'b0;
            CONTIG = 1'b0;
            SINGLE = 1'b0;
            if (A && ~B && C)
                NS = S2;
            else if (A && B && ~C)
                NS = S4;
            else
                NS = S1;
            end
   .
   .
   .
```

Using One-Hot Encoding

The following examples show a one-hot encoded state machine. Use this method to control the state vector specification or when you want to specify the names of the state registers. These examples use one flip-flop for each of the seven states.

Note: Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code. See the "Accelerate FPGA Macros with One-Hot Approach" appendix for a detailed description of one-hot encoding and its applications.

VHDL - One-hot Encoded State Machine Example

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


entity one_hot is
    port (CLOCK, RESET : in STD_LOGIC;
          A, B, C, D, E: in BOOLEAN;
          SINGLE, MULTI, CONTIG: out STD_LOGIC);
end one_hot;

architecture BEHV of one_hot is
```

```
type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of STATE_TYPE: type is "0000001 0000010 0000100
0001000 0010000 0100000 1000000 ";

signal CS, NS: STATE_TYPE;

begin

    SYNC_PROC: process (CLOCK, RESET)
    begin
            if (RESET='1') then
                CS <= S1;
            elsif (CLOCK'event and CLOCK = '1') then
            CS <= NS;
            end if;
    end process; --End SYNC_PROC

    COMB_PROC: process (CS, A, B, C, D, E)
    begin
            case CS is
            when S1 =>
                      MULTI  <= '0';
                      CONTIG <= '0';
                      SINGLE <= '0';
            if (A and not B and C) then
                    NS <= S2;
            elsif (A and B and not C) then
                     NS <= S4;
            else
                     NS <= S1;
            end if;
             .
             .
             .
```

Verilog - One-hot Encoded State Machine Example

```
     ////////////////////////////////////////////////
    // ONE_HOT.V Version 1.0                       //
   // Example of a one-hot encoded state machine  //
  // Xilinx HDL Synthesis Design Guide for FPGAs //
 // May 1997                                    //
////////////////////////////////////////////////

module one_hot (CLOCK, RESET, A, B, C, D, E,
                SINGLE, MULTI, CONTIG);

input   CLOCK, RESET;
input   A, B, C, D, E;
output  SINGLE, MULTI, CONTIG;

reg SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [6:0] //synopsys enum STATE_TYPE
    S1 = 7'b0000001,
    S2 = 7'b0000010,
    S3 = 7'b0000100,
    S4 = 7'b0001000,
    S5 = 7'b0010000,
    S6 = 7'b0100000,
    S7 = 7'b1000000;

// Declare current state and next state variables
reg [2:0] /* synopsys enum STATE_TYPE */ CS;
reg [2:0] /* synopsys enum STATE_TYPE */ NS;

// synopsys state_vector CS

    always @ (posedge CLOCK or posedge RESET)
    begin
        if (RESET == 1'b1)
           CS = S1;
        else
           CS = NS;
    end

    always @ (CS or A or B or C or D or D or E)
    begin
        case (CS) //synopsys full_case
            S1 :
```

```
                begin
                MULTI  = 1'b0;
                CONTIG = 1'b0;
                SINGLE = 1'b0;
                if (A && ~B && C)
                    NS = S2;
                else if (A && B && ~C)
                    NS = S4;
                else
                    NS = S1;
    end
            .
            .
            .
```

Summary of Encoding Styles

In the three previous examples, the state machine's possible states are defined by an enumeration type. Use the following syntax to define an enumeration type.

type *type_name* is (*enumeration_literal* {, *enumeration_literal*});

After you have defined an enumeration type, declare the signal representing the states as the enumeration type as follows:

type *STATE_TYPE* is (S1, S2, S3, S4, S5, S6, S7);

signal CS, NS: *STATE_TYPE*;

The state machine described in the three previous examples has seven states. The possible values of the signals CS (Current_State) and NS (Next_State) are S1, S2, ... , S6, S7.

To select an encoding style for a state machine, specify the state vectors. Alternatively, you can specify the encoding style when the state machine is compiled. Xilinx recommends that you specify an encoding style. If you do not specify a style, the Synopsys Compiler selects a style that minimizes the gate count. For the state machine shown in the three previous examples, the compiler selected the binary encoded style: S1="000", S2="001", S3="010", S4="011", S5="100", S6="101", and S7="110".

You can use the FSM extraction tool to change the encoding style of a state machine. For example, use this tool to convert a binary-encoded state machine to a one-hot encoded state machine. The Synopsys enum.script file contains the commands you need to convert an

enumerated types encoded state machine to a one-hot encoded state machine.

Note: Refer to the Synopsys documentation for instructions on how to extract the state machine and change the encoding style.

Comparing Synthesis Results for Encoding Styles

The following table summarizes the synthesis results from the different methods used to encode the state machine in the three previous VHDL and Verilog state machine examples. The results are for an XC4005EPC84-2 device

Note: The Timing Analyzer was used to obtain the timing results in this table.

Table 4-1 State Machine Encoding Styles Comparison (XC4005E-2)

Comparison	One-Hot	Binary	Enum (One-hot)
Occupied CLBs	6	9	6
CLB Flip-flops	6	3	7
PadToSetup	9.4 ns (3[a])	13.4 ns (4)	9.6 ns (3)
ClockToPad	15.1 ns (3)	15.1 ns (3)	14.9 ns (3)
ClockToSetup	13.0 ns (4)	13.9 ns (4)	10.1 ns (3)

a.The number in parentheses represents the CLB block level delay.

The binary-encoded state machine has the longest ClockToSetup delay. Generally, the FSM extraction tool provides the best results because the Synopsys Compiler reduces any redundant states and optimizes the state machine after the extraction.

Initializing the State Machine

When you use one-hot encoding, add the following lines of code to your design to ensure that the FPGA is initialized to a Set state.

- VHDL

```
SYNC_PROC: process (CLOCK, RESET)
begin
```

```
    if (RESET='1') then
        CS <= s1;
```

- Verilog

```
always @ (posedge CLOCK or posedge RESET)
begin
  if (RESET == 1'b 1)
      CS = S1;
```

Alternatively, you can assign an INIT=S attribute to the initial state register to specify the initial state.

```
set_attribute "CS_reg<0>"\
fpga_xilinx_init_state -type string "S"
```

Note: The "\" character in this command represents a continuation marker.

In the Binary Encode State Machine example, the RESET signal forces the S1 flip-flop to be preset (initialized to 1) while the other flip-flops are cleared (initialized to 0).

Using Dedicated I/O Decoders

The periphery of XC4000 family devices has four wide decoder circuits at each edge. The inputs to each decoder are any of the IOB signals on that edge plus one local interconnect per CLB row or column. Each decoder generates a High output (using a pull-up resistor) when the AND condition of the selected inputs or their complements is true. The decoder outputs drive CLB inputs so they can be combined with other logic or can be routed directly to the chip outputs.

To implement XC4000 family edge decoders in HDL, you must instantiate edge decoder primitives. The primitive names you can use vary with the synthesis tool you are using. Using the Synopsys tools, you can instantiate the following primitives: DECODE1_IO, DECODE1_INT, DECODE4, DECODE8, and DECODE16. These primitives are implemented using the dedicated I/O edge decoders. The XC4000 family wide decoder outputs are effectively open-drain and require a pull-up resistor to take the output High when the specified pattern is detected on the decoder inputs. To attach the pull-up resistor to the output signal, you must instantiate a PULLUP component.

The following VHDL example shows how to use the I/O edge decoders by instantiating the decode primitives from the XSI library. Each decoder output is a function of ADR (IOB inputs) and CLB_INT (local interconnects). The AND function of each DECODE output and Chip Select (CS) serves as the source of a flip-flop Clock Enable pin. The four edge decoders in this design are placed on the same device edge. The "Schematic Block Representation of I/O Decoder" figure shows the schematic block diagram representation of this I/O decoder design.

VHDL - Using Dedicated I/O Decoders Example

```
--Edge Decoder
--A XC4000 LCA has special decoder circuits at each edge. These decoders
--are open-drained wired-AND gates. When one or more of the inputs (I) are
--Low output(O) is Low. When all of the inputs are High, the output is --
--High.A pull-up resistor must be connected to the output node to achieve
--a true logic High.

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity io_decoder is
  port (ADR: in std_logic_vector (4 downto 0);
        CS: in std_logic;
        DATA: in std_logic_vector (3 downto 0);
        CLOCK: in std_logic;
        QOUT: out std_logic_vector (3 downto 0));
end io_decoder;

architecture STRUCTURE of io_decoder is

COMPONENT DECODE1_IO
  PORT ( I: IN std_logic;
         O: OUT std_logic );
END COMPONENT;

COMPONENT DECODE1_INT
  PORT ( I: IN std_logic;
         O: OUT std_logic );
END COMPONENT;

COMPONENT DECODE4
  PORT ( A3, A2, A1, A0: IN std_logic;
```

```
         O: OUT std_logic );
END COMPONENT;

COMPONENT PULLUP
  PORT ( O: OUT std_logic );
END COMPONENT;

---- Internal Signal Declarations ----------------------
signal DECODE, CLKEN, CLB_INT: std_logic_vector (3 downto 0);
signal ADR_INV, CLB_INV: std_logic_vector (3 downto 0);
begin

ADR_INV <= not ADR (3 downto 0);
CLB_INV <= not CLB_INT;

----- Instantiation of Edge Decoder: Output "DECODE(0)" ---------------
    A0: DECODE4 port map (ADR(3), ADR(2), ADR(1), ADR_INV(0), DECODE(0));

    A1: DECODE1_IO port map (ADR(4), DECODE(0));

    A2: DECODE1_INT port map (CLB_INV(0), DECODE(0));

    A3: DECODE1_INT port map (CLB_INT(1), DECODE(0));

    A4: DECODE1_INT port map (CLB_INT(2), DECODE(0));

    A5: DECODE1_INT port map (CLB_INT(3), DECODE(0));

    A6: PULLUP port map (DECODE(0));

----- Instantiation of Edge Decoder: Output "DECODE(1)" ---------------
    B0: DECODE4 port map (ADR(3), ADR(2), ADR_INV(1), ADR(0), DECODE(1));

    B1: DECODE1_IO port map (ADR(4), DECODE(1));

    B2: DECODE1_INT port map (CLB_INT(0), DECODE(1));

    B3: DECODE1_INT port map (CLB_INV(1), DECODE(1));

    B4: DECODE1_INT port map (CLB_INT(2), DECODE(1));

    B5: DECODE1_INT port map (CLB_INT(3), DECODE(1));

    B6: PULLUP port map (DECODE(1));
```

```
----- Instantiation of Edge Decoder: Output "DECODE(2)" ---------------
    C0: DECODE4 port map (ADR(3), ADR_INV(2), ADR(1), ADR(0), DECODE(2));

    C1: DECODE1_IO port map (ADR(4), DECODE(2));

    C2: DECODE1_INT port map (CLB_INT(0), DECODE(2));

    C3: DECODE1_INT port map (CLB_INT(1), DECODE(2));

    C4: DECODE1_INT port map (CLB_INV(2), DECODE(2));

    C5: DECODE1_INT port map (CLB_INT(3), DECODE(2));

    C6: PULLUP port map (DECODE(2));

----- Instantiation of Edge Decoder: Output "DECODE(3)" ---------------
    D0: DECODE4 port map (ADR_INV(3), ADR(2), ADR(1), ADR(0), DECODE(3));

    D1: DECODE1_IO port map (ADR(4), DECODE(3));

    D2: DECODE1_INT port map (CLB_INT(0), DECODE(3));

    D3: DECODE1_INT port map (CLB_INT(1), DECODE(3));

    D4: DECODE1_INT port map (CLB_INT(2), DECODE(3));

    D5: DECODE1_INT port map (CLB_INV(3), DECODE(3));

    D6: PULLUP port map (DECODE(3));

-----CLKEN is the AND function of CS & DECODE--------

CLKEN(0) <= CS and DECODE(0);
CLKEN(1) <= CS and DECODE(1);
CLKEN(2) <= CS and DECODE(2);
CLKEN(3) <= CS and DECODE(3);

--------Internal 4-bit counter --------------
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
                CLB_INT <= CLB_INT + 1;
        end if;
    end process;
```

```
-------"QOUT(0)" Data Register Enabled by "CLKEN(0)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
          if (CLKEN(0) = '1') then
              QOUT(0) <= DATA(0);
          end if;
        end if;
    end process;

-------"QOUT(1)" Data Register Enabled by "CLKEN(1)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
          if (CLKEN(1) = '1') then
              QOUT(1) <= DATA(1);
          end if;
        end if;
    end process;

-------"QOUT(2)" Data Register Enabled by "CLKEN(2)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
          if (CLKEN(2) = '1') then
              QOUT(2) <= DATA(2);
          end if;
        end if;
    end process;

-------"QOUT(3)" Data Register Enabled by "CLKEN(3)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
          if (CLKEN(3) = '1') then
              QOUT(3) <= DATA(3);
          end if;
        end if;
    end process;

end STRUCTURE;
```

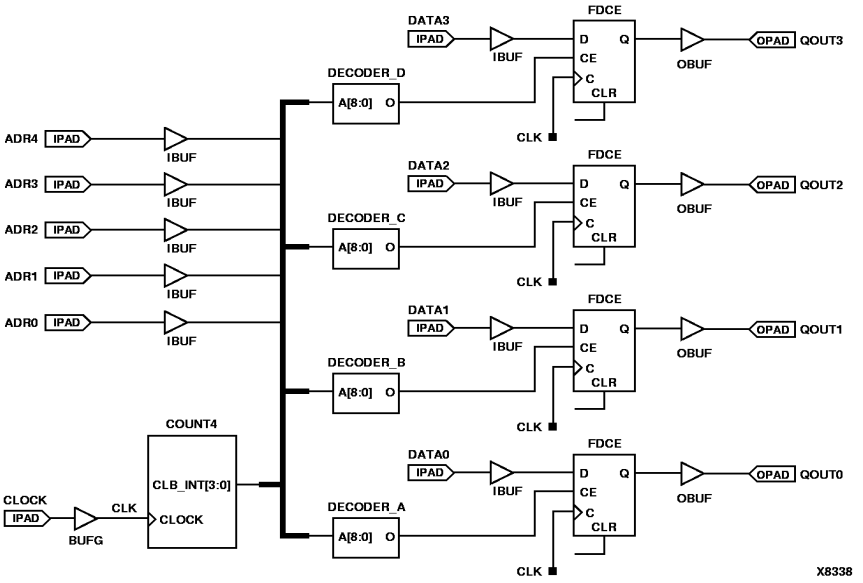

Figure 4-7 Schematic Block Representation of I/O Decoder

Note: In the previous figure, the pull-up resistors are inside the Decoder blocks.

Instantiating LogiBLOX Modules

Note: Refer to the *LogiBLOX Reference/User Guide* for detailed instructions on using LogiBLOX.

Synopsys can infer arithmetic DesignWare modules from VHDL or Verilog code for these operators: +, –, <, <=, >, >=, =, +1, –1. These adders, subtracters, comparators, incrementers, and decrementers use FPGA dedicated device resources, such as carry logic, to improve the speed and area of designs. For bus widths greater than four, DesignWare modules are generally faster unless multiple instances of the same function are compiled together. For more information on the DesignWare libraries, refer to the *Synopsys (XSI) Interface/Tutorial Guide*.

If you want to use a module that is not in the DesignWare libraries, or if you want better performance or area, you can use LogiBLOX to

create components that can be instantiated in your code. A simulation model is also created so that RTL simulation can be performed before your design is compiled.

LogiBLOX is a graphical tool that allows you to select from several arithmetic, logic, I/O, sequential, and data storage modules for inclusion in your HDL design. Use LogiBLOX to instantiate the modules listed in the following table.

Table 4-2 LogiBLOX Modules

Module	Description
Arithmetic	
Accumulator	Adds data to or subtracts it from the current value stored in the accumulator register
Adder/Subtracter	Adds or subtracts two data inputs and a carry input
Comparator	Compares the magnitude or equality of two values
Counter	Generates a sequence of count values
Logic	
Constant	Forces a constant value onto a bus
Decoder	Routes input data to 1-of-n lines on the output port
Multiplexer	Type 1, Type 2 - Routes input data on 1-of-n lines to the output port
Simple Gates	Type 1, Type 2, Type 3 - Implements the AND, INVERT, NAND, NOR, OR, XNOR, and XOR logic functions
Tristate	Creates a tri-stated internal data bus
I/O	
Bi-directional Input/ Output	Connects internal and external pin signals
Pad	Simulates an input/output pad
Sequential	
Clock Divider	Generates a period that is a multiple of the clock input period
Counter	Generates a sequence of count values
Shift Register	Shifts the input data to the left of right

Table 4-2 LogiBLOX Modules

Module	Description
Storage	
Data Register	Captures the input data on active clock transitions
Memory: ROM, RAM, SYNC_RAM, DP_RAM	Stores information and makes it readable

Using LogiBLOX in HDL Designs

1. Before using LogiBLOX, verify the following.
   - Xilinx software is correctly installed
   - Environment variables are set correctly
   - Your display environment variable is set to your machine's display
2. To run LogiBLOX, enter the following command.

   ```
   lbgui
   ```

   The LogiBLOX Setup Window appears after the LogiBLOX module generator is loaded. This window allows you to name and customize the module you want to create.
3. Select the Vendor tab in the Setup Window. Select Synopsys in the Vendor Name field to specify the correct bus notation for connecting your module.

   Select the Project Directory tab.Enter the directory location of your Synopsys project in the LogiBLOX Project Directory field.

   Select the Device Family tab. Select the target device for your design in the Device Family field.

   Select the Options tab and select the applicable options for your design as follows.

   - Simulation Netlist

     This option allows you to create simulation netlists of the selected LogiBLOX module in different formats. You can choose one or more of the outputs listed in the following table.

Table 4-3 Simulation Netlist Options

Option	Description
Behavioral VHDL netlist	Generates a simulation netlist in behavioral VHDL; output file has a .vhd extension.
Gate level EDIF netlist	Generates a simulation netlist in EDIF format; output file has an .edn extension.
Structural Verilog netlist	Generates a simulation netlist in structural Verilog; output file has a .v extension.

- Component Declaration

  This option creates instantiation templates in different formats that can be copied into your design. You can select none, one, or both of the following options.

Table 4-4 Component Declaration Options

Option	Description
VHDL template	Generates a LogiBLOX VHDL component declaration/instantiation template that is copied into your VHDL design when a LogiBLOX module is instantiated. The output file has a .vhi extension.
Verilog template	Generates a LogiBLOX Verilog module definition/instantiation template that is copied into your Verilog design when a LogiBLOX module is instantiated. The output file has a .vei extension.

- Implementation Netlist

  Select NGO File to generate an implementation netlist in Xilinx NGD binary format. You must select this option when instantiating LogiBLOX symbols in an HDL design. The

output file has an .ngo extension and can be used as input to NGDBuild.

- LogiBLOX DRC

  Select the Stop Process on Warning option to stop module processing if any warning messages are encountered during the design process.

For example, if you have a Verilog design, and you are simulating with Verilog-XL, select Structural Verilog netlist, Verilog template, NGO File, and Stop Process on Warning. For a VHDL design and simulating with Synopsys VSS, select Behavioral VHDL, VHDL template, NGO File, and Stop Process on Warning.

Select OK.

4. Enter a name in the Module Name field in the Module Selector Window.

   Select a base module type from the Module Type field.

   Select a bus width from the Bus Width field.

   Customize your module by selecting pins and specifying attributes.

   After you have completed module specification, select OK.

   This initiates the generation of a component instantiation declaration, a behavioral model, and an implementation netlist.

5. Copy the module declaration/instantiation into your design. The template file created by LogiBLOX is *module_name*.vhi (VHDL) or *module_name*.vei (Verilog), and is saved in the project directory as specified in the LogiBLOX setup.

6. Complete the signal connections of the instantiated module to the rest of your design.

Note: For more information on simulation, refer to the "Simulating Your Design" chapter.

7. Create a Synopsys implementation script. Add a Set_dont_touch attribute to the instantiated LogiBLOX module, and compile your design.

Note: Logiblox_instance_name is the name of the instantiated module in your design. Logiblox_name is the LogiBLOX component that corresponds to the .ngo file created by LogiBLOX.

set_dont_touch *logiblox_instance_name*

compile

Also, if you have a Verilog design, use the Remove Design command before writing the .sxnf netlist.

Note: If you do not use the Remove Design command, Synopsys may write an empty .sxnf file. If this occurs, the Xilinx software will trim this module/component and all connected logic.

remove_design *logiblox_name*

write **-format xnf** **-hierarchy** **-output** *design.sxnf*

8. Compile your design and create a .sxnf file. You can safely ignore the following error messages.

   `Warning: Can't find the design in the library` *WORK.* `(LBR-1)`

   `Warning: Unable to resolve reference` *LogiBLOX_name* `in` *design_name*`. (LINK-5)`

   `Warning: Design` *design_name* `has` *1* `unresolved references. For more detailed information, use the "link" command. (UID-341)`

9. Implement your design with the Xilinx tools. Verify that the .ngo file created by LogiBLOX is in the same project directory as the Synopsys netlist.

   You may get the following warnings during the NGDBuild and mapping steps. These messages are issued if the Xilinx software can not locate the corresponding .ngo file created by LogiBLOX.

   `Warning: basnu - logical block` *LogiBLOX_instance_name* `of type` *LogiBLOX_name* `is unexpanded. Logical Design DRC complete with 1 warning(s).`

   If you get this message, you will get the following message during mapping.

   `ERROR:basnu - logical block` *LogiBLOX_instance_name* `of type` *LogiBLOX_name* `is unexpanded. Errors detected in general drc.`

If you get these messages, first verify that the .ngo file created by LogiBLOX is in the project directory. If the file is there, verify that the module is properly instantiated in the code.

10. To simulate your post-layout design, convert your design to a timing netlist and use the back-annotation flow applicable to Synopsys.

Note: For more information on simulation, refer to the "Simulating Your Design" chapter.

Implementing Memory

XC4000E/EX/XL and Spartan FPGAs provide distributed on-chip RAM or ROM. CLB function generators can be configured as ROM (ROM16X1, ROM32X1); level-sensitive RAM (RAM16X1, RAM 32X1); edge-triggered, single-port (RAM16X1S, RAM32X1S); or dual-port (RAM16x1D) RAM. The edge-triggered capability simplifies system timing and provides better performance for RAM-based designs. This distributed RAM can be used for status registers, index registers, counter storage, constant coefficient multipliers, distributed shift registers, LIFO stacks, latching, or any data storage operation. The dual-port RAM simplifies FIFO designs.

Note: For more information on XC4000 family RAM, refer to the Xilinx Web site or the current release of the Xilinx Data Book.

Implementing XC4000 and Spartan ROMs

ROMs can be implemented in Synopsys as follows.

- Use RTL descriptions of ROMs
- Instantiate 16x1 and 32x1 ROM primitives
- Use LogiBLOX to implement any other ROM size

VHDL and a Verilog examples of an RTL description of a ROM follow.

VHDL - RTL Description of a ROM

```
--
--  Behavioral 16x4 ROM Example
--           rom_rtl.vhd
--
```

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

entity rom_rtl is
     port (ADDR: in INTEGER range 0 to 15;
           DATA: out STD_LOGIC_VECTOR (3 downto 0));
end rom_rtl;

architecture XILINX of rom_rtl is

    subtype ROM_WORD is STD_LOGIC_VECTOR (3 downto 0);
    type ROM_TABLE is array (0 to 15) of ROM_WORD;
    constant ROM: ROM_TABLE := ROM_TABLE'(
        ROM_WORD'("0000"),
        ROM_WORD'("0001"),
        ROM_WORD'("0010"),
        ROM_WORD'("0100"),
        ROM_WORD'("1000"),
        ROM_WORD'("1100"),
        ROM_WORD'("1010"),
        ROM_WORD'("1001"),
        ROM_WORD'("1001"),
        ROM_WORD'("1010"),
        ROM_WORD'("1100"),
        ROM_WORD'("1001"),
        ROM_WORD'("1001"),
        ROM_WORD'("1101"),
        ROM_WORD'("1011"),
        ROM_WORD'("1111"));

    begin
        DATA <= ROM(ADDR);  -- Read from the ROM

end XILINX;
```

Verilog - RTL Description of a ROM

```
/*
 * ROM_RTL.V
 * Behavioral Example of 16x4 ROM
*/

module rom_rtl(ADDR, DATA) ;
input [3:0] ADDR ;
output [3:0] DATA ;

reg [3:0] DATA ;

// A memory is implemented
// using a case statement

always @(ADDR)
begin
   case (ADDR)
      4'b0000 : DATA = 4'b0000 ;
      4'b0001 : DATA = 4'b0001 ;
      4'b0010 : DATA = 4'b0010 ;
      4'b0011 : DATA = 4'b0100 ;
      4'b0100 : DATA = 4'b1000 ;
      4'b0101 : DATA = 4'b1000 ;
      4'b0110 : DATA = 4'b1100 ;
      4'b0111 : DATA = 4'b1010 ;
      4'b1000 : DATA = 4'b1001 ;
      4'b1001 : DATA = 4'b1001 ;
      4'b1010 : DATA = 4'b1010 ;
      4'b1011 : DATA = 4'b1100 ;
      4'b1100 : DATA = 4'b1001 ;
      4'b1101 : DATA = 4'b1001 ;
      4'b1110 : DATA = 4'b1101 ;
      4'b1111 : DATA = 4'b1111 ;
   endcase
end

endmodule
```

When using an RTL description of a ROM, Synopsys creates ROMs from random logic gates that are implemented using function generators.

Another method for implementing ROMs in your Synopsys design is to instantiate the 16x1 or 32x1 ROM primitives. To define the ROM value use the Set_attribute command as follows.

```
set_attribute instance_name xnf_init rom_value –type
string
```

This attribute allows Synopsys to write the ROM contents to the netlist file so the Xilinx tools can initialize the ROM. Rom_value should be specified in hexadecimal values. See the VHDL and Verilog RAM examples in the following section for examples of this attribute using a RAM primitive.

Implementing XC4000 Family RAMs

Note: Do not use RTL descriptions of RAMs in your code because they do not compile efficiently and can cause combinatorial loops.

You can implement RAMs in your HDL code as follows.

- Instantiate 16x1 and 32x1 RAM primitives (RAM16X1, RAM32X1, RAM16X1S, RAM32X1S, RAM16X1D)
- Use LogiBLOX to implement any other RAM size

When implementing RAM in XC4000 and Spartan designs, Xilinx recommends using the synchronous write, edge-triggered RAM (RAM16X1S, RAM32X1S, or RAM16X1D) instead of the asynchronous-write RAM (RAM16X1 or RAM32X1) to simplify write timing and increase RAM performance.

Examples of an instantiation of edge-triggered RAM primitives are provided in the following VHDL and Verilog designs. As with ROMs, initial RAM values can be specified from within a Synopsys Script as follows.

```
set_attribute instance_name xnf_init ram_value –type
string
```

Ram_value is specified in hexadecimal values.

VHDL - Instantiating RAM

```
-----------------------------------------
-- RAM_PRIMITIVE.VHD                   --
-- Example of instantiating 4          --
-- 16x1 synchronous RAMs               --
-- HDL Synthesis Design Guide for FPGAs --
-- May 1997                            --
-----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;


entity ram_primitive is

   port ( DATA_IN, ADDR  : in STD_LOGIC_VECTOR(3 downto 0);
          WE, CLOCK      : in STD_LOGIC;
          DATA_OUT       : out STD_LOGIC_VECTOR(3 downto 0));

end ram_primitive;


architecture STRUCTURAL_RAM of ram_primitive is

   component RAM16X1S
       port (D, A3, A2, A1, A0, WE, WCLK : in STD_LOGIC;
             O : out STD_LOGIC);
   end component;

begin

   RAM0 :  RAM16X1S port map (O => DATA_OUT(0), D => DATA_IN(0),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);

   RAM1 :  RAM16X1S port map (O => DATA_OUT(1), D => DATA_IN(1),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);

   RAM2 :  RAM16X1S port map (O => DATA_OUT(2), D => DATA_IN(2),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);
```

```
   RAM3 :  RAM16X1S port map (O => DATA_OUT(3), D => DATA_IN(3),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);

end STRUCTURAL_RAM;
```

Verilog - Instantiating RAM

```
      ////////////////////////////////////////
     // RAM_PRIMITIVE.V                     //
    // Example of instantiating 4          //
   // 16x1 Synchronous RAMs                //
  // HDL Synthesis Design Guide for FPGAs //
 // August 1997                          //
////////////////////////////////////////


module ram_primitive (DATA_IN, ADDR, WE, CLOCK, DATA_OUT);

input  [3:0] DATA_IN, ADDR;
input        WE, CLOCK;
output [3:0] DATA_OUT;

RAM16X1S RAM0 (.O(DATA_OUT[0]), .D(DATA_IN[0]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

RAM16X1S RAM1 (.O(DATA_OUT[1]), .D(DATA_IN[1]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

RAM16X1S RAM2 (.O(DATA_OUT[2]), .D(DATA_IN[2]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

RAM16X1S RAM3 (.O(DATA_OUT[3]), .D(DATA_IN[3]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

endmodule
```

Using LogiBLOX to Implement Memory

Use LogiBLOX to create a memory module larger than 32X1 (16X1 for Dual Port). Implementing memory with LogiBLOX is similar to implementing any module with LogiBLOX except for defining the Memory initialization file. Use the following steps to create a memory module.

Note: Refer to the "Using LogiBLOX in HDL Designs" section for more information on using LogiBLOX.

1. Before using LogiBLOX, verify the following.
   - Xilinx software is correctly installed
   - Environment variables are set correctly
   - Your display environment variable is set to your machine's display
2. To run LogiBLOX, enter the following command.

   ```
   lbgui
   ```

   The LogiBLOX Setup Window appears after the LogiBLOX module generator is loaded. This window allows you to name and customize the module you want to create.
3. Select the Vendor tab in the Setup Window. Select Synopsys in the Vendor Name field to specify the correct bus notation for connecting your module.

   Select the Project Directory tab. Enter the directory location of your Synopsys project in the LogiBLOX Project Directory field.

   Select the Device Family tab. Select the target device for your design in the Device Family field.

   Select the Options tab and select the applicable options for your design.

   Select OK.
4. Enter a name in the Module Name field in the Module Selector Window.

   Select the Memories module type from the Module Type field to specify that you are creating a memory module.

Select a width (any value from 1 to 64 bits) for the memory from the Data Bus Width field.

In the Details field, select the type of memory you are creating (ROM, RAM, SYNC_RAM, or DP_RAM).

Enter a value in the Memory Depth field for your memory module.

Note: Xilinx recommends (this is not a requirement) that you select a memory depth value that is a multiple of 16 since this is the memory size of one lookup table.

5. If you want the memory module initialized to all zeros on power up, you do not need to create a memory file (Mem File). However, if you want the contents of the memory initialized to a value other than zero, you must create and edit a memory file. Enter a memory file name in the Mem File field and click on the Edit button. Continue with the following steps.

   a) A memory template file in a text editor is displayed. This file does not contain valid data, and must be edited before you can use it. The data values specified in the memory file Data Section define the contents of the memory. Data values are specified sequentially, beginning with the lowest address in the memory, as defined.

   b) Specify the address of a data value. The default radix of the data values is 16. If more than one radix definition is listed in the memory file header section, the last definition is the radix used in the Data Section.

      The following definition defines a 16-word memory with the contents 6, 4, 5, 5, 2, 7, 5, 3, 5, 5, 5, 5, 5, 5, 5, 5, starting at address 0. Note that the contents of locations 2, 3, 6, and 8 through 15 are defined via the default definition. Two starting addresses, 4 and 7, are given.

```
depth 16
default 5
data 6,4,
4: 2, 7,
7: 3
```

   c) After you have finished specifying the data for the memory module, save the file and exit the editor.

6. Click the OK button. Selecting OK initiates the generation of a component instantiation declaration, a behavioral model, and an implementation netlist.
7. Copy the HDL module declaration/instantiation into your HDL design. The template file created by LogiBLOX is *module_name*.vhi for VHDL and *module_name*.vei for Verilog, and is saved in the project directory as specified in the LogiBLOX setup.
8. Complete the signal connections of the instantiated LogiBLOX memory module to the rest of your HDL design, and complete initial design coding.
9. Perform a behavioral simulation on your design. For more information on behavioral simulation, refer to the "Simulating Your Design" chapter.
10. Create a Synopsys implementation script. Add a Set_dont_touch attribute to the instantiated LogiBLOX memory module, and compile your design.

Note: Logiblox_instance_name is the name of the instantiated module in your design. Logiblox_name is the LogiBLOX component that corresponds to the .ngo file created by LogiBLOX.

```
set_dont_touch logiblox_instance_name
compile
```

Also, if you have a Verilog design, use the Remove Design command before writing the .sxnf netlist.

Note: If you do not use the Remove Design command, Synopsys may write an empty .sxnf file. If this occurs, the Xilinx software will trim this module/component and all connected logic.

```
remove_design logiblox_name
write -format xnf -hierarchy -output design.sxnf
```

11. Compile your design and create a .sxnf file. You can safely ignore the following warning messages.

```
Warning: Can't find the design in the library WORK.
(LBR-1)
Warning: Unable to resolve reference LogiBLOX_name in
design_name. (LINK-5)
```

```
Warning: Design design_name has 1 unresolved references.
For more detailed information, use the "link" command.
(UID-341)
```

12. Implement your design with the Xilinx tools. Verify that the .ngo file created by LogiBLOX is in the same project directory as the Synopsys netlist.

    You may get the following warnings during the NGDBuild and mapping steps. These messages are issued if the Xilinx software can not locate the corresponding .ngo file created by LogiBLOX.

    ```
    Warning: basnu - logical block LogiBLOX_instance_name of
    type LogiBLOX_name is unexpanded. Logical Design DRC
    complete with 1 warning(s).
    ```

    If you get this message, you will get the following message during mapping.

    ```
    ERROR:basnu - logical block LogiBLOX_instance_name of
    type LogiBLOX_name is unexpanded. Errors detected in
    general drc.
    ```

    If you get these messages, first verify that the .ngo file created by LogiBLOX is in the project directory. If the file is there, verify that the module is properly instantiated in the code.

13. To simulate your post-layout design, convert your design to a timing netlist and use the back-annotation flow applicable to Synopsys.

Note: For more information on simulation, refer to the "Simulating Your Design" chapter.

Implementing Boundary Scan (JTAG 1149.1)

Note: Refer to the *Development System User Guide* for a detailed description of the XC4000/XC5200 boundary scan capabilities.

XC4000, Spartan, and XC5200 FPGAs contain boundary scan facilities that are compatible with IEEE Standard 1149.1. Xilinx devices support external (I/O and interconnect) testing and have limited support for internal self-test.

You can access the built-in boundary scan logic between power-up and the start of configuration. Optionally, the built-in logic is available after configuration if you specify boundary scan in your design.

During configuration, a reduced boundary scan capability (sample/preload and bypass instructions) is available.

In a configured FPGA device, the boundary scan logic is enabled or disabled by a specific set of bits in the configuration bitstream. To access the boundary scan logic after configuration in HDL designs, you must instantiate the boundary scan symbol, BSCAN, and the boundary scan I/O pins, TDI, TMS, TCK, and TDO.

The XC5200 BSCAN symbol contains three additional pins: RESET, UPDATE, and SHIFT, which are not available for XC4000 and Spartan. These pins represent the decoding of the corresponding state of the boundary scan internal state machine. If this function is not used, you can leave these pins unconnected in your HDL design.

Note: Do not use the FPGA Compiler boundary scan commands such as set_jtag_implementation, set_jtag_instruction, and set_jtag_port with FPGA devices.

Instantiating the Boundary Scan Symbol

To incorporate the boundary scan capability in a configured FPGA using Synopsys tools, you must manually instantiate boundary scan library primitives at the source code level. These primitives include TDI, TMS, TCK, TDO, and BSCAN. The following VHDL and Verilog examples show how to instantiate the boundary scan symbol, BSCAN, into your HDL code. Note that the boundary scan I/O pins are not declared as ports in the HDL code. The schematic for this design is shown in the "Bnd_scan Schematic" figure.

You *must* assign a Synopsys Set Don't Touch attribute to the net connected to the TDO pad before you use the Insert Pads and Compile commands. Otherwise, the TDO pad is removed by the compiler. In addition, you do not need IBUFs or OBUFs for the TDI, TMS, TCK, and TDO pads. These special pads connect directly to the Xilinx boundary scan module.

VHDL - Boundary Scan

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

entity bnd_scan is
    port (TDI_P, TMS_P, TCK_P : in STD_LOGIC;
```

```
          LOAD_P, CE_P, CLOCK_P, RESET_P: in STD_LOGIC;
          DATA_P: in STD_LOGIC_VECTOR(3 downto 0);
          TDO_P: out STD_LOGIC;
          COUT_P: out STD_LOGIC_VECTOR(3 downto 0));
end bnd_scan;

architecture XILINX of bnd_scan is

    component BSCAN
        port (TDI, TMS, TCK :in STD_LOGIC;
              TDO: out STD_LOGIC);
    end component;

    component TDI
        port (I: in STD_LOGIC);
    end component;

    component TMS
        port (I: in STD_LOGIC);
    end component;

    component TCK
        port (I: in STD_LOGIC);
    end component;

    component TDO
        port (O: out STD_LOGIC);
    end component;

    component count4
        port (LOAD, CE, CLOCK, RST: in STD_LOGIC;
              DATA: in STD_LOGIC_VECTOR (3 downto 0);
              COUT: out STD_LOGIC_VECTOR (3 downto 0));
    end component;

    -- Defining signals to connect BSCAN to Pins --
    signal TCK_NET  : STD_LOGIC;
    signal TDI_NET  : STD_LOGIC;
    signal TMS_NET  : STD_LOGIC;
    signal TDO_NET  : STD_LOGIC;

begin

    U1: BSCAN port map (TDO => TDO_NET,
                        TDI => TDI_NET,
```

```
                        TMS => TMS_NET,
                        TCK => TCK_NET);

    U2: TDI port map (I =>TDI_NET);

    U3: TCK port map (I =>TCK_NET);

    U4: TMS port map (I =>TMS_NET);

    U5: TDO port map (O =>TDO_NET);

    U6: count4 port map (LOAD  => LOAD_P,
                         CE    => CE_P,
                         CLOCK => CLOCK_P,
                         RST   => RESET_P,
                         DATA  => DATA_P,
                         COUT  => COUT_P);

end XILINX;
```

Verilog - Boundary Scan

```
      ////////////////////////////////////////////////////
     // BND_SCAN.V                                       //
    // Example of instantiating the BSCAN symbol in     //
   // activating the Boundary Scan circuitry           //
  // Count4 is an instantiated .v file of a counter   //
 // September 1997                                   //
////////////////////////////////////////////////////

module bnd_scan (LOAD_P, CLOCK_P, CE_P, RESET_P,
 DATA_P, COUT_P);

    input          LOAD_P, CLOCK_P, CE_P, RESET_P;
    input  [3:0] DATA_P;
    output [3:0] COUT_P;

    wire          TDI_NET, TMS_NET, TCK_NE, TDO_NET;

    BSCAN U1 (.TDO(TDO_NET), .TDI(TDI_NET), .TMS(TMS_NET), .TCK(TCK_NET));

    TDI U2 (.I(TDI_NET));

    TCK U3 (.I(TCK_NET));

    TMS U4 (.I(TMS_NET));
```

```
    TDO U5 (.O(TDO_NET));

    count4 U6 (.LOAD(LOAD_P), .CLOCK(CLOCK_P), .CE(CE_P),
               .RST(RESET_P), .DATA(DATA_P), .COUT(COUT_P));

endmodule
```

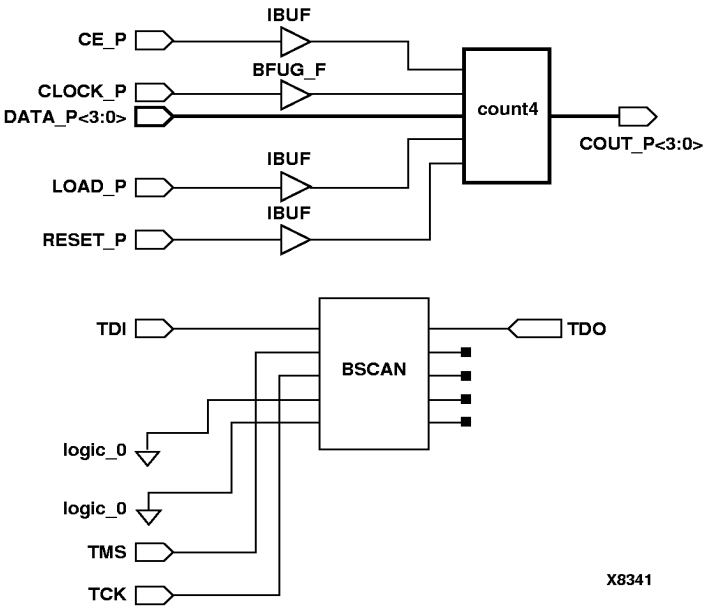

Figure 4-8 Bnd_scan Schematic

Implementing Logic with IOBs

You can move logic that is normally implemented with CLBs to IOBs. By moving logic from CLBs to IOBs, additional logic can be implemented in the available CLBs. Using IOBs also improves design performance by increasing the number of available routing resources.

The XC4000 and Spartan devices have different IOB functions. The following sections provide a general description of the IOB function in XC4000E/EX/XL/XV and Spartan devices. A description of how to manually implement additional I/O features is also provided.

XC4000E/EX/XL/XV and Spartan IOBs

You can configure XC4000E/EX/XL/XV and Spartan IOBs as input, output, or bidirectional signals. You can also specify pull-up or pull-down resistors, independent of the pin usage.

These various buffer and I/O structures can be inferred from commands executed in a script or the Design Analyzer. The Set Port Is Pad command in conjunction with the Insert Pads command allows Synopsys to create the appropriate buffer structure according to the direction of the specified port in the HDL code. Attributes can also be added to these commands to further control pull-up, pull-down, and clock buffer insertion, as well as slew-rate control.

Inputs

The buffered input signal that drives the data input of a storage element can be configured as either a flip-flop or a latch. Additionally, the buffered signal can be used in conjunction with the input flip-flop or latch, or without the register.

To avoid external hold-time requirements, IOB input flip-flops and latches have a delay block between the external pin and the D input. You can remove this default delay by instantiating a flip-flop or latch with a NODELAY attribute. The NODELAY attribute decreases the setup-time requirement and introduces a small hold time.

If an IOB or register is instantiated in your HDL code, do not use the Set Port Is Pad command on that port. Doing so may automatically infer a buffer on that port and create an invalid double-buffer structure.

Note: Registers that connect to an input or output pad and require a Direct Clear or Preset pin are not implemented by the FPGA or Design Compiler in the IOB.

Outputs

The output signal that drives the programmable tristate output buffer can be a registered or a direct output. The register is a positive-edge triggered flip-flop and the clock polarity can be inverted inside the IOB. (Xilinx software automatically optimizes any inverters into the IOB.) The XC4000 and Spartan output buffers can sink 12 mA. Two adjacent outputs can be inter-connected externally to sink up to 24mA.

Note: The FPGA Compiler and Design Compiler can optimize flip-flops attached to output pads into the IOB. However, these compilers cannot optimize flip-flops into an IOB configured as a bidirectional pad.

Slew Rate

Add the following to your Synopsys script to control slew rate; add after the Set Port Is Pad command, and before the Insert Pads command.

```
set_pad_type -slewrate HIGH {output_ports}
set_pad_type -slewrate LOW {output_ports}
```

Note: Synopsys High slew control corresponds to Xilinx Slow slew rate. Synopsys Low slew control corresponds to Xilinx Fast slew rate. If a slew rate is not specified, the default is High or Slow.

You can also specify slew rate by instantiating the appropriate OBUF primitive. To specify a fast slew rate, instantiate a fast output primitive (such as, OBUF_F and OBUFT_F). The default output buffer has a slow slew rate (such as, OBUF_S and OBUFT_S), and you do not need to instantiate it in your HDL code. If you do instantiate an I/O buffer or register, make sure that the Set Port Is Pad is not performed on this port to prevent creation of a double I/O buffer.

Pull-ups and Pull-downs

XC4000 and Spartan devices have programmable pull-up and pull-down resistors available in the I/O regardless of whether it is configured as an input, output, or bi-directional I/O. By default, all unused IOBs are configured as an input with a pull-up resistor. The value of the pull-ups and pull-downs vary depending on operating conditions and device process variances but should be approximately 50 K Ohms to 100 K Ohms. If a more precise value is required, use an external resistor.

To specify these internal pull-up or pull-down I/O resistors, add the following to your Synopsys script rather than instantiating them in your HDL code. Add the following to your script after executing the Set Port Is Pad command and before executing the Insert Pads command.

set_pad_type **-pullup** {*port_names*}

set_pad_type **-pulldown** {*port_names*}

XC4000EX/XL/XV Output Multiplexer/2-Input Function Generator

A function added to XC4000EX/XL/XV families is a two input multiplexer connected to the IOB output allowing the output clock to select either the output data or the IOB clock enable as the output pad. This allows you to share output pins between two signals effectively doubling the number of device outputs without requiring a larger device and/or package. Additionally, this multiplexer can be configured as a two-input function generator allowing you to implement any 2-input logic function in the IOB thus freeing up additional logic resources in the device and allowing for very fast pin-to-pin paths.

To use the output multiplexer (OMUX), you must instantiate it in your code. See the following VHDL and Verilog examples. Instantiation of the other types of two-input output primitives, (OAND2, OOR2, OXOR2, etc.) are similar to these examples.

Note: Since the OMUX uses the IOB output clock and clock enable routing structures, the output flip-flop (OFD) can not be used within the same IOB. The input flip-flop (IFD) can be used if the clock enable is not used.

- VHDL - Output Multiplexer

```
----------------------------------------
-- OMUX_EXAMPLE.VHD                    --
-- Example of OMUX instantiation       --
-- For an XC4000EX/XL/XV device        --
-- HDL Synthesis Design Guide for FPGAs --
-- August 1997                         --
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity  omux_example is

    port (DATA_IN: in STD_LOGIC_VECTOR (1 downto 0);
          SEL: in STD_LOGIC;
          DATA_OUT: out STD_LOGIC);
```

```
end omux_example;

architecture XILINX of omux_example is

    component OMUX2
        port (D0, D1, S0 : in  STD_LOGIC;
              O :          out STD_LOGIC);
    end component;

begin

     DUEL_OUT: OMUX2 port map (O=>DATA_OUT, D0=>DATA_IN(0),
                               D1=>DATA_IN(1), S0=>SEL);

end XILINX;
```

- Verilog - Output Multiplexer

```
      ////////////////////////////////////////
     // OMUX_EXAMPLE.V                      //
    // Example of instantiating an OMUX2    //
   // in an XC4000EX/XL IOB                //
  // HDL Synthesis Design Guide for FPGAs //
 // August 1997                          //
////////////////////////////////////////

module omux_example (DATA_IN, SEL, DATA_OUT) ;

input  [1:0] DATA_IN ;
input        SEL ;
output       DATA_OUT ;

OMUX2 DUEL_OUT (.O(DATA_OUT), .D0(DATA_IN[0]),
                .D1(DATA_IN[1]), .S0(SEL));

endmodule
```

XC5200 IOBs

XC5200 IOBs consist of an input buffer and an output buffer that can be configured as an input, output, or bi-directional I/O. The structure of the XC5200 is similar to the XC4000 IOB except the XC5200 does not contain a register/latch. The XC5200 IOB has a programmable pull-up or pull-down resistor, and two slew rate control modes (Fast and Slow) to minimize bus transients. The input buffer can be

globally configured to TTL or CMOS levels, and the output buffer can sink or source 8.0 mA.

I/O buffer structures (as with the XC4000 IOBs) can be inferred from a Synopsys script with the Set Port Is Pad command in conjunction with the Insert Pads command. Controlling pull-up and pull-down insertion and slew rate control are performed as previously described for the XC4000 IOB.

The XC5200 IOB also contains a delay element so that an input signal that is directly registered or latched can have a guaranteed zero hold time at the expense of a longer setup time. You can disable this (equivalent to NODELAY in XC4000) by instantiating an IBUF_F buffer for that input port. This only needs to be done for ports that connect directly to the D input of a register in which a hold time can be tolerated.

Bi-directional I/O

You can create bi-directional I/O with one or a combination of the following methods.

- Behaviorally describe the I/O path
- Structurally instantiate appropriate IOB primitives
- Create the I/O using LogiBLOX

Xilinx FPGA IOBs consist of a direct input path into the FPGA through an input buffer (IBUF) and an output path to the FPGA pad through a tri-stated buffer (OBUFT). The input path can be registered or latched; the output path can be registered. If you instantiate or behaviorally describe the I/O, you must describe this bi-directional path in two steps. First, describe an input path from the declared INOUT port to a logic function or register. Second, describe an output path from an internal signal or function in your code to a tri-stated output with a tri-state control signal that can be mapped to an OBUFT.

You should always describe the I/O path at the top level of your code. If the I/O path is described in a lower level module, the FPGA Compiler may incorrectly create the I/O structure.

Inferring Bi-directional I/O

This section includes VHDL and Verilog examples that show how to infer a bi-directional I/O. In these examples, the input path is latched by a CLB latch that is gated by the active high READ_WRITE signal.

The output consists of two latched outputs with an AND and OR, and connected to a described tri-state buffer. The active low READ_WRITE signal enables the tri-state gate.

- VHDL - Inferring a Bi-directional Pin

```
------------------------------------------------------
--  BIDIR_INFER.VHD                                 --
--  Example of inferring a Bi-directional pin        --
--  August 1997                                     --
------------------------------------------------------

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_infer is

    port (DATA :       inout STD_LOGIC_VECTOR(1 downto 0);
          READ_WRITE : in    STD_LOGIC);

end bidir_infer;

architecture XILINX of bidir_infer is

    signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);

    begin

    process(READ_WRITE)
    begin

        if (READ_WRITE = '1') then
            LATCH_OUT <= DATA;
        end if;

    end process;

    process(READ_WRITE)
        begin
```

```
        if (READ_WRITE = '0') then
            DATA(0) <= LATCH_OUT(0) and LATCH_OUT(1);
            DATA(1) <= LATCH_OUT(0) or LATCH_OUT(1);
        else
             DATA(0) <= 'Z';
             DATA(1) <= 'Z';
        end if;

    end process;

end XILINX;
```

- Verilog - Inferring a Bi-directional Pin

```
      ////////////////////////////////////////////////////////////////////
     // BIDIR_INFER.V                                                    //
    // This is an example of an inference of a bi-directional signal. //
   // Note: Logic description of port should always be on top-level  //
  //        code when using Synopsys Compiler and verilog.         //
 // August 1997                                                    //
////////////////////////////////////////////////////////////////////

module bidir_infer (DATA, READ_WRITE);

    input       READ_WRITE ;
    inout [1:0] DATA ;

    reg   [1:0] LATCH_OUT ;

    always @ (READ_WRITE)
        begin
        if (READ_WRITE == 1'b1)
            LATCH_OUT <= DATA;
        end

    assign DATA[0] = READ_WRITE ?  1'bZ : (LATCH_OUT[0] & LATCH_OUT[1]);
    assign DATA[1] = READ_WRITE ? 1'bZ : (LATCH_OUT[0] | LATCH_OUT[1]);

endmodule
```

Instantiating Bi-directional I/O

Instantiating the bi-directional I/O gives the you more control over the implementation of the circuit; however, as a result, your code is more architecture-specific and usually more verbose. The VHDL and

Verilog examples in this section are identical to the examples in the "Inferring Bi-directional I/O" section; however, since there is more control over the implementation, an input latch is specified rather than the CLB latch inferred in the previous examples. The following examples are a more efficient implementation of the same circuit.

When instantiating I/O primitives, do not specify the Set Port Is Pad command on the instantiated ports to prevent the I/O buffers from being inferred by Synopsys. This precaution also prevents the creation of an illegal structure.

- VHDL - Instantiation of a Bi-directional Pin

```
-----------------------------------
--  BIDIR_INSTANTIATE.VHD        --
--  Example of  an instantiation --
--  of a Bi-directional pin      --
--  August 1997                  --
-----------------------------------

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_instantiate is

    port (DATA :       inout STD_LOGIC_VECTOR(1 downto 0);
          READ_WRITE : in    STD_LOGIC);

end bidir_instantiate;

architecture XILINX of bidir_instantiate is

    signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
    signal DATA_OUT :  STD_LOGIC_VECTOR(1 downto 0);
    signal GATE :      STD_LOGIC;

    component ILD_1
        port (D, G : in  STD_LOGIC;
              Q    : out STD_LOGIC);
    end component;

    component OBUFT_S
        port (I, T : in  STD_LOGIC;
              O :    out STD_LOGIC);
    end component;
```

```
    begin

    DATA_OUT(0) <= LATCH_OUT(0) and LATCH_OUT(1);
    DATA_OUT(1) <= LATCH_OUT(0) or LATCH_OUT(1);

    GATE <= not READ_WRITE;

    INPUT_PATH_0 : ILD_1
        port map (D => DATA(0), G => GATE,
                  Q => LATCH_OUT(0));

    INPUT_PATH_1 : ILD_1
        port map (D => DATA(1), G => GATE,
                  Q => LATCH_OUT(1));

    OUPUT_PATH_0 : OBUFT_S
        port map (I => DATA_OUT(0), T => READ_WRITE,
                  O => DATA(0));

    OUPUT_PATH_1 : OBUFT_S
        port map (I => DATA_OUT(1), T => READ_WRITE,
                  O => DATA(1));

end XILINX;
```

- Verilog - Instantiation of a Bi-directional Pin

```
     ////////////////////////////////////////////
    // BIDIR_INSTANTIATE.V                      //
   // This is an example of an instantiation //
  // of a bi-directional port.              //
 // August 1997                            //
////////////////////////////////////////////

module bidir_instantiate (DATA, READ_WRITE);

    input       READ_WRITE ;
    inout [1:0] DATA ;

    reg   [1:0] LATCH_OUT ;
    wire  [1:0] DATA_OUT ;
    wire        GATE ;

    assign GATE = ~READ_WRITE;
```

```
    assign DATA_OUT[0] = LATCH_OUT[0] & LATCH_OUT[1];
    assign DATA_OUT[1] = LATCH_OUT[0] | LATCH_OUT[1];

    // I/O primitive instantiation

    ILD_1 INPUT_PATH_0 (.Q(LATCH_OUT[0]), .D(DATA[0]), .G(GATE));

    ILD_1 INPUT_PATH_1 (.Q(LATCH_OUT[1]), .D(DATA[1]), .G(GATE));

    OBUFT_S OUPUT_PATH_0 (.O(DATA[0]), .I(DATA_OUT[0]), .T(READ_WRITE));

    OBUFT_S OUPUT_PATH_1 (.O(DATA[1]), .I(DATA_OUT[1]), .T(READ_WRITE));

endmodule
```

Using LogiBLOX to Create Bi-directional I/O

You can use LogiBLOX to create I/O structures in an FPGA. LogiBLOX gives you the same control as instantiating I/O primitives, and is usually less verbose. LogiBLOX is especially useful for bused I/O ports.

Note: Refer to the "Using LogiBLOX in HDL Designs" section section, for details on creating, instantiating, and compiling LogiBLOX modules.

Do not use the Set Port Is Pad command on LogiBLOX-created ports. Also, when designing with Verilog, you must issue a Remove Design command before writing out the .sxnf files from Synopsys.

The following VHDL and Verilog examples show how to instantiate bi-directional I/O created with LogiBLOX. These examples produce the same results as the examples in the "Instantiating Bi-directional I/O" section.

- VHDL - Using LogiBLOX to Create a Bi-directional Port

```
----------------------------------------
--  BIDIR_LOGIBLOX.VHD                --
--  Example of using LogiBLOX         --
--  to create a Bi-directional port   --
--  August 1997                       --
----------------------------------------
------------------------------------------------------
-- LogiBLOX BIDI Module "bidir_io_from_lb"
-- Created by LogiBLOX version M1.3.7
--    on Mon Sep  8 13:14:02 1997
-- Attributes
--    MODTYPE = BIDI
--    BUS_WIDTH = 2
--    IN_TYPE = LATCH
--    OUT_TYPE = TRI
------------------------------------------------------

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_logiblox is

    port (DATA :       inout STD_LOGIC_VECTOR(1 downto 0);
          READ_WRITE : in    STD_LOGIC);

end bidir_logiblox;

architecture XILINX of bidir_logiblox is

    signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
    signal DATA_OUT :  STD_LOGIC_VECTOR(1 downto 0);

    ------------------------------------------------
    -- Component Declaration
    ------------------------------------------------
    component bidir_io_from_lb
        PORT( O:     IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
              OE:    IN    STD_LOGIC;
              IGATE: IN    STD_LOGIC;
              IQ:    OUT   STD_LOGIC_VECTOR(1 DOWNTO 0);
              P:     INOUT STD_LOGIC_VECTOR(1 DOWNTO 0));
    end component;
```

```
    begin

    DATA_OUT(0) <= LATCH_OUT(0) and LATCH_OUT(1);
    DATA_OUT(1) <= LATCH_OUT(0) or LATCH_OUT(1);

    -------------------------------------------------
    -- Component Instantiation
    -------------------------------------------------
    BIDIR_BUSSED_PORT : bidir_io_from_lb
        port map (O => DATA_OUT, OE => READ_WRITE,
                  IGATE => READ_WRITE, IQ => LATCH_OUT, P => DATA);

end XILINX;
```

- Verilog - Using LogiBLOX to Create a Bi-directional Port

```
     ////////////////////////////////////////
    // BIDIR_LOGIBLOX.V                    //
   // This is an example of using LogiBLOX  //
  // to create a bi-directional port.      //
 // August 1997                          //
////////////////////////////////////////

//-------------------------------------------------
// LogiBLOX BIDI Module "bidir_io_from_lb"
// Created by LogiBLOX version M1.3.7
//    on Mon Sep  8 17:10:15 1997
// Attributes
//    MODTYPE = BIDI
//    BUS_WIDTH = 2
//    IN_TYPE = LATCH
//    OUT_TYPE = TRI
//-------------------------------------------------

module bidir_logiblox (DATA, READ_WRITE);

    input       READ_WRITE ;
    inout [1:0] DATA ;

    reg   [1:0] LATCH_OUT ;
    wire  [1:0] DATA_OUT ;


    assign DATA_OUT[0] = LATCH_OUT[0] & LATCH_OUT[1];
    assign DATA_OUT[1] = LATCH_OUT[0] | LATCH_OUT[1];
```

```
    // LogiBLOX instantiation

    bidir_io_from_lb  BIDIR_BUSSED_PORT
        ( .O(DATA_OUT), .OE(READ_WRITE), .P(DATA),
          .IQ(LATCH_OUT), .IGATE(READ_WRITE));

endmodule


module bidir_io_from_lb (O, OE, P, IQ, IGATE);
    input  [1:0] O;
    input        OE;
    input        IGATE;
    inout  [1:0] P;
    output [1:0] IQ;
endmodule
```

Specifying Pad Locations

Although Xilinx recommends allowing the software to select pin locations to ensure the best possible pin placement in terms of design timing and routing resources, sometimes you must define the pad locations prior to placement and routing. You can assign pad locations either from a Synopsys script prior to writing out the netlist file or from a User Constraints File (UCF), as follows.

- Assigning Pad Location with a Synopsys Script

  set_attribute *port_name* **"pad location" -type string** *pad_location*

- Assigning Pad Location with a UCF

  NET *port_name* **LOC =** *pad_location*;

Use one or the other method, but not both. Refer to the Xilinx Data Book or the Xilinx Web site for the pad locations for your device and package.

Moving Registers into the IOB

Note: XC5200 devices do not have input and output flip-flops.

IOBs contain an input register or latch and an output register. IOB inputs can be register or latch inputs as well as direct inputs to the device array. Registers without a direct reset or set function can be moved into IOBs. Moving registers or latches into IOBs may reduce

the number of CLBs used and decreases the routing congestion. In addition, moving input registers and latches into the IOB reduces the external setup time, as shown in the following figure.

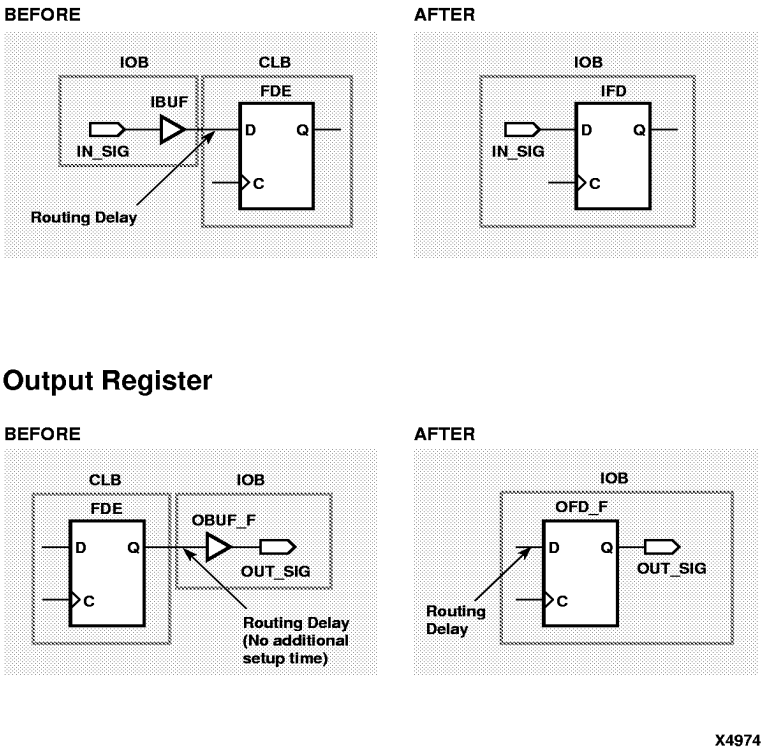

Figure 4-9 Moving Registers into the IOB

Although moving output registers into the IOB may increase the internal setup time, it may reduce the clock-to-output delay, as shown in this figure. The Synopsys Compiler automatically moves registers into IOBs if the Preset, Clear, and Clock Enable pins are *not* used.

Use -pr Option with Map

Use the -pr (pack registers) option when running MAP. The -pr {i | o | b} (input | output | both) option specifies to the MAP program to move registers into IOBs under the following circumstances.

1. The input of the register must be connected to an input port, or the Q pin must be connected to an output port. For the XC4000EX/XL/XV this applies to non-I/O latches, as well as flip-flops.
2. IOBs must have input or output flip-flops. XC5200 devices do not have IOB flip-flops.
3. The flip-flop does not use an asynchronous set or reset signal.
4. In XC4000, Spartan, and XC3000 devices, a flop/latch is not added to an IOB if it has a BLKNM or LOC conflict with the IOB.
5. In XC4000 or Spartan devices, a flop/latch is not added to an IOB if its control signals (clock or clock enable) are not compatible with those already defined in the IOB. This occurs when a flip-flop (latch) is already in the IOB with different clock or clock enable signals, or when the XC4000EX/XL/XV output MUX is used in the same IOB.
6. In XC4000EX/XV devices, if a constant 0 or 1 is driven on the IOPAD, a flip-flop/latch with a CE is not added to the input side of the IOB.

Using Unbonded IOBs (XC4000E/EX/XL/XV and Spartan Only)

In some package/device pairs, not all pads are bonded to a package pin. You can use these unbonded IOBs and the flip-flops inside them in your design by instantiating them in the HDL code. You can implement shift registers with these unbonded IOBs. The VHDL and Verilog examples in this section show how to instantiate unbonded IOB flip-flops in a 4-bit shift register in an XC4000 device.

Note: The Synopsys compilers cannot infer unbonded primitives. Refer to the *Synopsys (XSI) Interface/Tutorial Guide* for a list of library primitives that can be used for instantiations.

VHDL - 4-bit Shift Register Using Unbonded I/O

```
-------------------------------------------------
-- UNBONDED_IO.VHD Version 1.0                  --
-- XC4000 LCA has unbonded IOBs which have      --
-- storage elements that can be used to build   --
-- shift registers.                             --
-- Below is a 4-bit Shift Register using        --
-- Unbonded IOB Flip Flops                      --
-- Xilinx HDL Synthesis Design Guide for FPGAs --
-- May 1997                                     --
-------------------------------------------------

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity unbonded_io is
    port (A, B: in STD_LOGIC;
          CLK: in STD_LOGIC;
           Q_OUT: out STD_LOGIC);
end unbonded_io;

architecture XILINX of unbonded_io is

    component IFD_U  -- Unbonded Input FF with INIT=Reset
         port (Q: out std_logic;
               D, C: in  std_logic);
    end component;

    component IFDI_U -- Unbonded Input FF with INIT=Set
         port (Q: out std_logic;
                 D, C: in  std_logic);
    end component;

    component OFD_U -- Unbonded Output FF with INIT=Reset
         port (Q: out std_logic;
                 D, C: in  std_logic);
    end component;

    component OFDI_U -- Unbonded Output FF with INIT=Set
         port (Q: out std_logic;
                 D, C: in  std_logic);
    end component;

--- Internal Signal Declarations -----
```

```
    signal U_Q : STD_LOGIC_VECTOR (3 downto 0);
    signal U_D : STD_LOGIC;

begin
U_D <= A and B;
Q_OUT <= U_Q(0);

    U3: OFD_U  port map (Q => U_Q(3),
                         D => U_D,
                         C => CLK);

    U2: IFDI_U port map (Q => U_Q(2),
                         D => U_Q(3),
                         C => CLK);

    U1: OFDI_U port map (Q => U_Q(1),
                         D => U_Q(2),
                         C => CLK);

    U0: IFD_U  port map (Q => U_Q(0),
                         D => U_Q(1),
                         C => CLK);

end XILINX;
```

Verilog - 4-bit Shift Register Using Unbonded I/O

```
        ////////////////////////////////////////////////
       // UNBONDED.V                                   //
      // XC4000 family has unbonded IOBs which have    //
     // storage elements that can be used to build    //
    // functions lie shift registers.                 //
   // Below is a 4-bit Shift Register using Unbonded //
  // IOB Flip Flops                                  //
 // HDL Synthesis Design Guide for FPGAs            //
// May 1997                                        //
////////////////////////////////////////////////

module unbonded_io (A, B, CLK, Q_OUT);

input A, B, CLK;
output Q_OUT;

wire[3:0] U_Q;
wire      U_D;
```

```
assign U_D = A & B;
assign Q_OUT = U_Q[0];

    OFD_U U3 (.Q(U_Q[3]), .D(U_D), .C(CLK));

    IFDI_U U2 (.Q(U_Q[2]), .D(U_Q[3]), .C(CLK));

    OFDI_U U1 (.Q(U_Q[1]), .D(U_Q[2]), .C(CLK));

    IFD_U  U0 (.Q(U_Q[0]), .D(U_Q[1]), .C(CLK));

endmodule
```

Implementing Multiplexers with Tristate Buffers

A 4-to-1 multiplexer is efficiently implemented in a single XC4000 or Spartan family CLB. The six input signals (four inputs, two select lines) use the F, G, and H function generators. Multiplexers that are larger than 4-to-1 exceed the capacity of one CLB. For example, a 16-to-1 multiplexer requires five CLBs and has two logic levels. These additional CLBs increase area and delay. Xilinx recommends that you use internal tristate buffers (BUFTs) to implement large multiplexers.

Large multiplexers built with BUFTs have the following advantages.

- Can vary in width with only minimal impact on area and delay
- Can have as many inputs as there are tristate buffers per horizontal longline in the target device
- Have one-hot encoded selector inputs

This last point is illustrated in the following VHDL and Verilog designs of a 5-to-1 multiplexer built with gates. Typically, the gate version of this multiplexer has binary encoded selector inputs and requires three select inputs (SEL<2:0>). The schematic representation of this design is shown in the "5-to-1 MUX Implemented with Gates" figure.

The VHDL and Verilog designs provided at the end of this section show a 5-to-1 multiplexer built with tristate buffers. The tristate buffer version of this multiplexer has one-hot encoded selector inputs and requires five select inputs (SEL<4:0>). The schematic representation of these designs is shown in the "5-to-1 MUX Implemented with BUFTs" figure.

VHDL - Mux Implemented with Gates

```
-- MUX_GATE.VHD
-- 5-to-1 Mux Implemented in Gates
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- May 1997

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

entity mux_gate is
port (SEL: in STD_LOGIC_VECTOR (2 downto 0);
A,B,C,D,E: in STD_LOGIC;
        SIG: out STD_LOGIC);
end mux_gate;

architecture RTL of mux_gate is
begin
    SEL_PROCESS: process (SEL,A,B,C,D,E)
    begin
        case SEL is
           when "000"  => SIG <= A;
           when "001"  => SIG <= B;
           when "010"  => SIG <= C;
           when "011"  => SIG <= D;
           when others => SIG <= E;
        end case;
    end process SEL_PROCESS;
end RTL;
```

Verilog - Mux Implemented with Gates

```
/* MUX_GATE.V
 * Synopsys HDL Synthesis Design Guide for FPGAs
 * May 1997 */

module mux_gate (A,B,C,D,E,SEL,SIG);

input A,B,C,D,E;
input [2:0] SEL;
output SIG;
reg SIG;

    always @ (A or B or C or D or SEL)
```

```
    case (SEL)
        3'b000:
            SIG=A;
        3'b001:
            SIG=B;
        3'b010:
            SIG=C;
        3'b011:
            SIG=D;
        3'b100:
            SIG=E;
default: SIG=A;
     endcase

endmodule
```

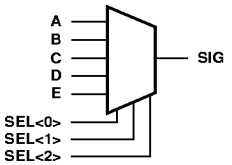

Figure 4-10 5-to-1 MUX Implemented with Gates

VHDL - Mux Implemented with BUFTs

```
-- MUX_TBUF.VHD
-- 5-to-1 Mux Implemented in 3-State Buffers
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- May 1997

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

entity mux_tbuf is
port (SEL: in STD_LOGIC_VECTOR (4 downto 0);
A,B,C,D,E: in STD_LOGIC;
        SIG: out STD_LOGIC);
end mux_tbuf;

architecture RTL of mux_tbuf is
```

```
begin

    SIG <= A when (SEL(0)='0') else 'Z';
    SIG <= B when (SEL(1)='0') else 'Z';
    SIG <= C when (SEL(2)='0') else 'Z';
    SIG <= D when (SEL(3)='0') else 'Z';
    SIG <= E when (SEL(4)='0') else 'Z';

end RTL;
```

Verilog - Mux Implemented with BUFTs

```
/* MUX_TBUF.V
 * Synopsys HDL Synthesis Design Guide for FPGAs
 * May 1997 */

module mux_tbuf (A,B,C,D,E,SEL,SIG);

input A,B,C,D,E;
input [4:0] SEL;
output SIG;
reg SIG;

    always @ (SEL or A)
    begin
        if (SEL[0]==1'b0)
            SIG=A;
        else
            SIG=1'bz;
     end

     always @ (SEL or B)
     begin
         if (SEL[1]==1'b0)
             SIG=B;
         else
             SIG=1'bz;
         end

     always @ (SEL or C)
     begin
         if (SEL[2]==1'b0)
             SIG=C;
         else
             SIG=1'bz;
```

```
    end

    always @ (SEL or D)
    begin
        if (SEL[3]==1'b0)
            SIG=D;
        else
            SIG=1'bz;
        end

    always @ (SEL or E)
    begin
        if (SEL[4]==1'b0)
            SIG=E;
        else
            SIG=1'bz;
        end

endmodule
```

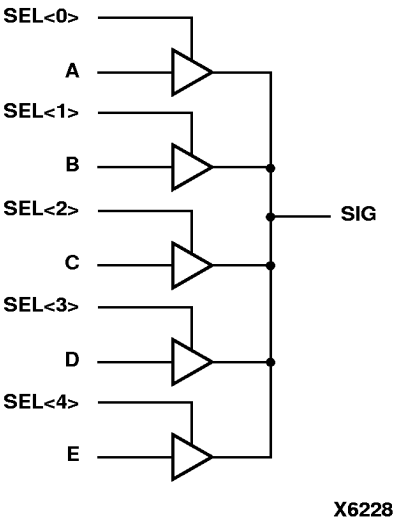

Figure 4-11 5-to-1 MUX Implemented with BUFTs

A comparison of timing and area for a 5-to-1 multiplexer built with gates and tristate buffers in an XC4005EPC84-2 device is provided in the following table. When the multiplexer is implemented with tristate buffers, no CLBs are used and the delay is smaller.

Table 4-5 Timing/Area for 5-to-1 MUX (XC4005EPC84-2)

Timing/Area	Using BUFTs	Using Gates
Timing	15.31 ns (1 block level)	17.56 ns (2 block levels)
Area	0 CLBs, 5 BUFTs	3 CLBs

Using Pipelining

You can use pipelining to dramatically improve device performance. Pipelining increases performance by restructuring long data paths with several levels of logic and breaking it up over multiple clock cycles. This method allows a faster clock cycle and, as a result, an increased data throughput at the expense of added data latency. Because the Xilinx FPGA devices are register-rich, this is usually an advantageous structure for FPGA designs since the pipeline is created at no cost in terms of device resources. Because data is now on a multi-cycle path, special considerations must be used for the rest of your design to account for the added path latency. You must also be careful when defining timing specifications for these paths.

Before Pipelining

In the following example, the clock speed is limited by the clock-to out-time of the source flip-flop; the logic delay through four levels of logic; the routing associated with the four function generators; and the setup time of the destination register.

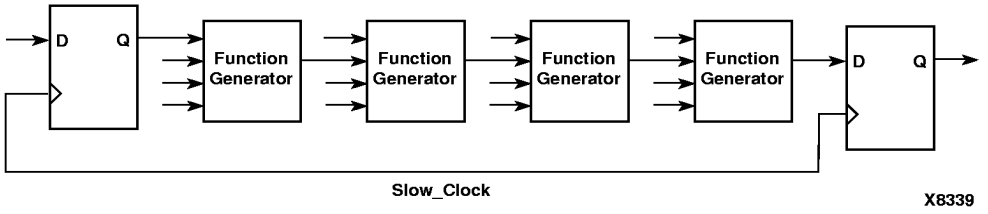

Figure 4-12 Before Pipelining

After Pipelining

This is an example of the same data path in the previous example after pipelining. Since the flip-flop is contained in the same CLB as the function generator, the clock speed is limited by the clock-to-out time of the source flip-flop; the logic delay through one level of logic; one routing delay; and the setup time of the destination register. In this example, the system clock runs faster than in the previous example.

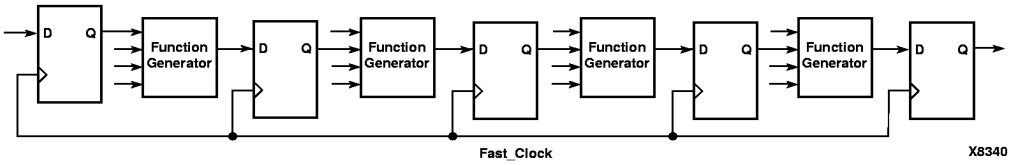

Figure 4-13 After Pipelining

Design Hierarchy

HDL Designs can either be synthesized as a flat module or as many small modules. Each methodology has its advantages and disadvantages, but as higher density FPGAs are created, the advantages of hierarchical designs outweigh any disadvantages.

Advantages to building hierarchical designs are as follows.

- Easier and faster verification/simulation
- Allows several engineers to work on one design at the same time

- Speeds up design compilation
- Reduces design time by allowing design module re-use for this and future designs.
- Allows you to produce design that are easier to understand
- Allows you to efficiently manage the design flow

Disadvantages to building hierarchical designs are as follows.

- Design mapping into the FPGA may not be as optimal across hierarchical boundaries; this can cause lesser device utilization and decreased design performance
- Design file revision control becomes more difficult
- Designs become more verbose

Most of the disadvantages listed above can be overcome with careful design consideration when choosing the design hierarchy.

Using Synopsys with Hierarchical Designs

By effectively partitioning your designs, you can significantly reduce compile time and improve synthesis results. This section provides recommendations for partitioning your designs.

Restrict Shared Resources to Same Hierarchy Level

Resources that can be shared should be on the same level of hierarchy. If these resources are not on the same level of hierarchy, the synthesis tool cannot determine if these resources should be shared.

Compile Multiple Instances Together

You may want to compile multiple occurrences of the same instance together to reduce the gate count. However, to increase design speed, do not compile a module in a critical path with other instances.

Restrict Related Combinatorial Logic to Same Hierarchy Level

Keep related combinatorial logic in the same hierarchical level to allow the synthesis tool to optimize an entire critical path in a single operation. Boolean optimization does not operate across hierarchical boundaries. Therefore, if a critical path is partitioned across boundaries, logic optimization is restricted. In addition, constraining modules is difficult if combinatorial logic is not restricted to the same level of hierarchy.

Separate Speed Critical Paths from Non-critical Paths

To achieve satisfactory synthesis results, locate design modules with different functions at different levels of the hierarchy. Design speed is the first priority of optimization algorithms. To achieve a design that efficiently utilizes device area, remove timing constraints from design modules.

Restrict Combinatorial Logic that Drives a Register to Same Hierarchy Level

To reduce the number of CLBs used, restrict combinatorial logic that drives a register to the same hierarchical block.

Restrict Module Size

Restrict module size to 100 - 200 CLBs. This range varies based on your computer configuration; the time required to complete each optimization run; if the design is worked on by a design team; and the target FPGA routing resources. Although smaller blocks give you more control, you may not always obtain the most efficient design.

Register All Outputs

Arrange your design hierarchy so that registers drive the module output in each hierarchical block. Registering outputs makes your design easier to constrain because you only need to constrain the clock period and the ClockToSetup of the previous module. If you have multiple combinatorial blocks at different levels of the hierarchy, you must manually calculate the delay for each module. Also, registering the outputs of your design hierarchy can eliminate any

possible problems with logic optimization across hierarchical boundaries.

Restrict One Clock to Each Module or to Entire Design

By restricting one clock to each module, you only need to describe the relationship between the clock at the top level of the design hierarchy and each module clock. By restricting one clock to the entire design, you only need to describe the clock at the top level of the design hierarchy.

Note: See the *Synopsys (XSI) Interface/Tutorial Guide* for more information on optimizing logic across hierarchical boundaries and compiling hierarchical designs.