HDL Coding Guidelines

Coding style has a considerable impact on how an FPGA design is implemented and ultimately how it performs. Although many popular synthesis tools have significantly improved optimization algorithms for FPGAs, it is still the designer's responsibility to generate HDL code that guides the synthesis tools and achieves the best result for a given architecture. This chapter provides VHDL and Verilog HDL design guidelines for both novice and experienced designers.

The synthesis software itself has a significant effect on implementation. The style of the code that you employ in one synthesis tool for one outcome can vary greatly from that in another tool. Synthesis tools optimize HDL code for logic utilization and performance, but they do so in a way that might not be close to your intended design. Knowing the effects of these synthesis tools, as well as knowing the most efficient HDL code for your design requirements, are both important.

General HDL Practices

Coding for Hierarchical Synthesis

In order to manage the complexity of modern designs, coding a design using hierarchical approach rather than just one single module is necessary. Such a hierarchical coded design can be synthesized all at one time, or have each module synthesized separately and then combined together. When synthesized all at one time, such a design can either be synthesized as a flat module or as manly hierarchical modules. Each methodology has its advantages and disadvantages. Since designs in smaller blocks are easier to keep track of, applying a hierarchical structure to large and complex FPGA designs is preferable. Hierarchical coding methodology also allows a group of

engineers to work on one design at the same time. It speeds up design compilation, makes changing the implementation of key blocks easier, and reduces the design period by allowing the re-use of design modules for current and future designs. In addition, it produces designs that are easier to understand.

However, if the design mapping into the FPGA is not optimal across hierarchical boundaries, it can lead to higher device utilization and lower design performance. You can overcome this disadvantage with careful design consideration when choosing the design hierarchy. This section describes coding techniques that will create good results when doing hierarchical synthesis, and as a result, synthesis is not able to optimize across the module boundaries. They will not affect the results if the design is synthesized all as a flat module as synthesis is able to optimize across the boundaries of the modules.

Although hierarchical synthesis takes more HDL coding planning and effort, because modules can be synthesized separately, the modules can be synthesized with different goals / tool settings. The HDL coding techniques help avoid the drawback of hierarchical synthesis (vs. flat synthesis) - reduced quality of results as synthesis cannot optimize across module boundaries

Here are some tips for building optimal hierarchical structures:

- The top level should only contain instantiation statements to call all major blocks.
- Any I/O instantiations should be at the top level.
- Any signals going into or out of the devices should be declared as input, output, or bidirectional pins at the top level.
- The tri-state statement for all bidirectional ports should be written at the top-level module. For example, the following Verilog HDL statement should only be in the top level and not in sub-modules:

```
“ouput_signal = en ? data : 16’bz”
```

Design Partitioning

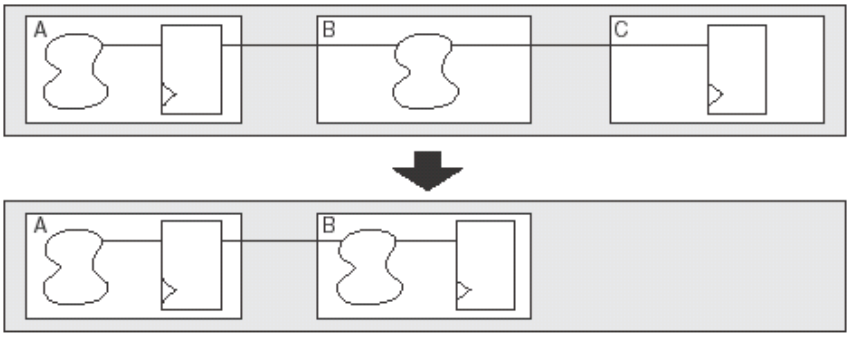
By effectively partitioning the design, you can reduce overall run time and improve synthesis results. Here are some recommendations for design partitioning. In the following descriptions, sub-blocks and blocks refer to either VHDL design units or Verilog HDL modules.

Maintain Synchronous Sub-Blocks by Registering All Outputs

Arrange the design boundary so that the outputs in each block are registered. Registering outputs helps the synthesis tool implement the combinatorial logic and registers in the same logic block. Registering outputs also makes the application of timing constraints easier since it eliminates possible problems with logic optimization across design boundaries. Using a single clock for each synchronous block significantly reduces the timing consideration in the block. It leaves the adjustment of the clock relationships of the whole design at the top level of the hierarchy. Figure 1 shows an example of synchronous

blocks with registered outputs.

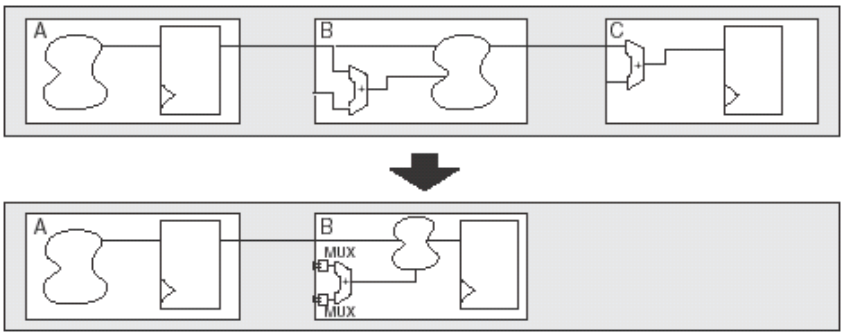
Figure 1: Synchronous Blocks with Registered Outputs



Keep Related Logic Together in the Same Block

Keeping related logic and sharable resources in the same block allows the sharing of common combinatorial terms and arithmetic functions within the block. It also allows the synthesis tools to optimize the entire critical path in a single operation. Since synthesis tools can only effectively handle optimization of certain amounts of logic, optimization of critical paths pending across the boundaries might not be optimal. The example in Figure 2 merges sharable resource in the same block.

Figure 2: Merging Sharable Resource in the Same Block



Separate Logic with Different Optimization Goals

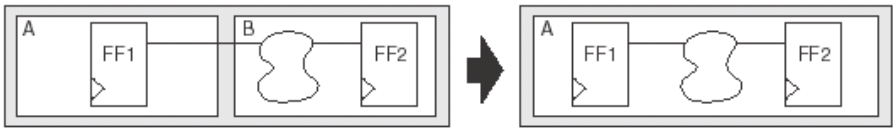
Separating critical paths from non-critical paths might achieve efficient synthesis results. At the beginning of the project, you should consider the design in terms of performance requirements and resource requirements. If a block contains two portions, one that needs to be optimized for area and a second that needs to be optimized for speed, they should be separated into two blocks. By doing this, you can apply different optimization strategies for each module without the two modules being limited by one another.

Keep Logic with the Same Relaxation Constraints in the Same Block

When a portion of the design does not require high performance, you can apply this portion with relaxed timing constraints to achieve high utilization of a device area. Relaxation constraints help to reduce overall run time. They

can also help to efficiently save resources, which can be used on critical paths. Figure 3 shows an example of grouping logic with the same relaxation constraint in one block.

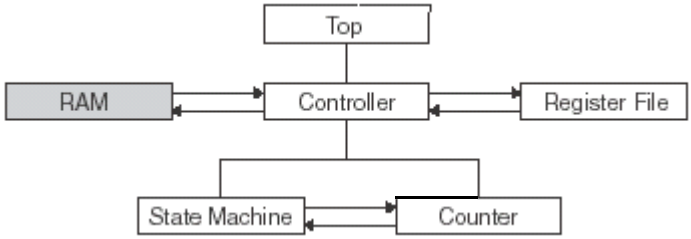
Figure 3: Logic with the Same Relaxation Constraint



Keep Instantiated Code in Separate Blocks

Leave the RAM block in the hierarchy in a separate block, as shown in Figure 4. This coding style facilitates the integration of the Diamond IPexpress tool into the synthesis process.

Figure 4: Separate RAM Block



Balancing Block Size

Although a smaller block methodology allows more control, it might not produce the most efficient design. The smaller the block, the fewer resources the synthesis tool has to apply “resource sharing” algorithms. On the other hand, the larger the block, the more the synthesis tool has to process, and this could cause changes that affect more logic than necessary in an incremental or multi-block design flow. The general recommendation is to limit the block size to functions that either make sense logically or in a manner that lends itself to re-usability, the exception being where the fit and/or timing are really tight and more control over placement of specific elements is required.

Design Registering

Pipelining can improve design performance by restructuring a long data path with several levels of logic and breaking it up over multiple clock cycles. This method allows a faster clock cycle by relaxing the clock-to-output and setup time requirements between the registers. It is usually an advantageous structure for creating faster data paths in register-rich FPGA devices. Knowledge of the FPGA architecture helps in planning pipelines at the beginning of the design cycle. When the pipelining technique is applied, special care must be taken for the rest of the design to account for the additional data path latency. The following illustrates the same data path

before (Figure 5) and after pipelining (Figure 6).

Figure 5: Before Pipelining

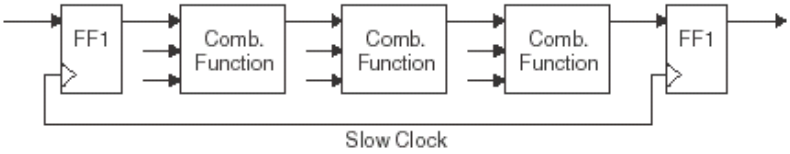
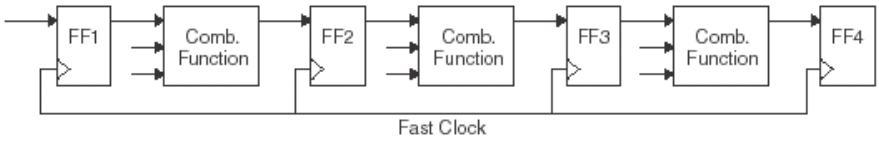


Figure 6: After Pipelining



Before pipelining, the clock speed is determined by three components:

- the clock-to-out time of the source register
- the logic delay through three levels of combinatorial logic and the associated routing delays
- the setup time of the destination register

After pipelining, the clock speed is significantly improved by reducing the delay of three logic levels to one logic level and the associated routing delays, even though the rest of the timing requirements remain the same. However, the overall latency from start to finish through this path is increased. This is a design trade-off you must consider when looking to employ pipelining in your design.

Adding pipeline stages can be done manually in code. Although synthesis tools are capable of moving logic around the pipeline stages for re-timing, in general, doing it manually in code makes it easier to simulate because the behavioral code directly matches the simulation results.

It is always important to check the placement and routing timing report to ensure that the pipelined design provides the desired performance. For lower power do not use more pipelining than is necessary to achieve this.

Figure 7: No Pipelining Results in Three Levels of Logic

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity non_pipelined is
port (clk : in std_logic;
            data_in : in std_logic_vector(8 downto 0);
            data_out : out std_logic
     );
end non_pipelined;

architecture behavioral of non_pipelined is

signal data : std_logic_vector(8 downto 0);
signal result : std_logic;

begin

-- process for registering data_in
process(clk)
begin
    if(rising_edge(clk)) then
        data <= data_in;
    end if;
end process;

--process for AND gate equation.
process(clk)
begin
    if(rising_edge(clk)) then
    -- AND gate is done in a single stage.
      data_out <= data(0) and data(1) and data(2) and
      data(3) and data(4)and data(5) and data(6) and
      data(7) and data(8);
    end if;
end process;

end behavioral;
```

Figure 7: No Pipelining Results in Three Levels of Logic (Continued)

Verilog HDL

```
module non_pipelined(
    input  clk,
   input  [8:0] data_in,
   output reg data_out
  );

reg [8:0] data;

always @ (posedge clk)
  begin
    data     <= data_in;  // process for registering data_in
    data_out <= (& data); // AND gate is done in a single
stage.
  end

endmodule
```

Figure 8: With Pipelining Results in One Level of Logic

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity pipelined is
port (clk : in std_logic;
    data_in : in std_logic_vector(8 downto 0);
    data_out : out std_logic
     );
end pipelined;

architecture behavioral of pipelined is

signal data: std_logic_vector(8 downto 0);
signal temp1,temp2 : std_logic;

begin

  -- process for registering data_in
process(clk)
begin
    if(rising_edge(clk)) then
        data <= data_in;
    end if;
end process;

--process for AND gate equation.
process(clk)
begin
    if(rising_edge(clk)) then
       temp1    <= (data(0) and data(1) and data(2) and
data(3));
       temp2    <= (data(4) and data(5) and data(6) and
data(7));
       data_out <= (temp1 and temp2 and data(8));
    end if;
end process;

end behavioral;
```

Figure 8: With Pipelining Results in One Level of Logic (Continued)

Verilog HDL

```
module pipelined (
    input clk,
   input [8:0]data_in,
   output reg data_out
  );

reg [8:0] data;
reg temp1,temp2;

always @ (posedge clk)
  begin
    data     <= data_in;                       // input register
    temp1    <= (& data[3:0]);                 // store
temporary result
    temp2    <= (& data[7:4]);                 // store
temporary result
    data_out <= (temp1 && temp2 && data[8]);  // output
register
  end

endmodule
```

Comparing If-Then-Else and Case Statements

Case and if-then-else statements are common for sequential logic in HDL designs. The if-then-else statement generally generates priority-encoded logic, whereas the case statement implements balanced logic. An if-then-else statement can contain a set of different expressions, but a case statement is evaluated against a common controlling expression. Both statements give the same functional implementation if the decode conditions are mutually exclusive, as shown in Figure 9 and Figure 10.

Figure 9: Case Statements with Mutually Exclusive Conditions

VHDL

```
process (s, x, y, z)
begin
  Out1 <= '0';
  Out2 <= '0';
  Out3 <= '0';

  case (s) is
    when "00" => Out1 <= x;
    when "01" => Out2 <= y;
    when "10" => Out3 <= z;
    when others => Out1 <= '0'; Out2 <= '0';
      Out3 <= '0';
  end case;
end process;
```

Figure 9: Case Statements with Mutually Exclusive Conditions

Verilog HDL

```
module case_example (
  input [1:0] s,
  input x, y, z,
  output reg Out1, Out2, Out3
);


always @ (s or x or y or z)
  begin
    Out1 <= 1'b0;
    Out2 <= 1'b0;
    Out3 <= 1'b0;

    case (s)
      2'b00 : Out1 <= x;
      2'b01 : Out2 <= y;
      2'b10 : Out3 <= z;
      default:
        begin
          Out1 <= 1'b0;
          Out2 <= 1'b0;
          Out3 <= 1'b0;
        end
    endcase
  end

endmodule
```

Figure 10: If-Then-Else Statements with Mutually Exclusive Conditions

VHDL

```
process (s, x, y, z)
begin
  Out1 <= '0';
  Out2 <= '0';
  Out3 <= '0';

  if s = "00" then Out1 <= x;
  elsif s = "01" then Out2 <= y;
  elsif s = "10" then Out3 <= z;
  else Out1 <= '0'; Out2 <= '0'; Out3 <= '0';
  end if;
end process;
```

Figure 10: If-Then-Else Statements with Mutually Exclusive Conditions

Verilog HDL

```
module case_example (
  input [1:0] s,
  input x, y, z,

  output reg Out1, Out2, Out3
);


always @ (s or x or y or z)
  begin
    Out1 <= 1'b0;
    Out2 <= 1'b0;
    Out3 <= 1'b0;

    if (s == 2'b00)
      Out1 <= x;
    else if (s == 2'b01)
      Out2 <= y;
    else if (s == 2'b10)
      Out3 <= z;
    else
        begin
          Out1 <= 1'b0;
          Out2 <= 1'b0;
          Out3 <= 1'b0;
        end
  end
endmodule
```

However, the use of the if-then-else construct could make the design more complex than necessary, because extra logic is needed to build a priority tree.

Consider the examples in Figure 11 and Figure 12.

Figure 11: Example A – If-Then-Else Statement with Lower Logic Requirement

VHDL

```
process (s1, s2, s3, x, y, z)
begin
  Out1 <= '0';
  Out2 <= '0';
  Out3 <= '0';

  if s1 = '1' then Out1 <= x;
  elsif s2 = '1' then Out2 <= y;
  elsif s3 = '1' then Out3 <= z;
  end if;

end process;
```

Figure 11: Example A – If-Then-Else Statement with Lower Logic Requirement (Continued)

Verilog

```
module case_example (
  input s1, s2, s3, x, y, z,
  output reg Out1, Out2, Out3
);


always @ (s1 or s2 or s3 or x or y or z)
  begin
    Out1 <= 1'b0;
    Out2 <= 1'b0;
    Out3 <= 1'b0;

    if (s1)
      Out1 <= x;
    else if (s2)
      Out2 <= y;
    else if (s3)
      Out3 <= z;
  end
endmodule
```

Figure 12: Example B – Simplified O3 Equations

VHDL

```
process (s1, s2, s3, x, y, z)
begin
  Out1 <= '0';
  Out2 <= '0';
  Out3 <= '0';

  if s1 = '1' then Out1 <= x;
  end if;
  if s2 = '1' then Out2 <= y;
  end if;
  if s3 = '1' then Out3 <= z;
  end if;

end process;
```

Figure 12: Example B – Simplified O3 Equations (Continued)

Verilog

```
module case_example (
  input s1, s2, s3, x, y, z,
  output reg Out1, Out2, Out3
);

always @ (s1 or s2 or s3 or x or y or z)
  begin
    Out1 <= 1'b0;
    Out2 <= 1'b0;
    Out3 <= 1'b0;

  if (s1)
    Out1 <= x;
  if (s2)
    Out2 <= y;
  if (s3)
    Out3 <= z;

  end
endmodule
```

If the decode conditions are not mutually exclusive, the if-then-else construct causes the last output to be dependent on all the control signals. The equation for O3 output in example A is:

```
O3 <= z and (s3) and (not (s1 and s2));
```

When the same code can be written as in example B, most of synthesis tools remove the priority tree and decode the output as:

```
O3 <= z and s3;
```

This reduces the logic requirement for the state machine decoder. If each output is indeed dependent of all of the inputs, it is better to use a case statement, since case statements provide equal branches for each output.

Avoiding Unintentional Latches

While latches can be suitable (or even preferred) in an ASIC where you control the gates, in an FPGA they have to be mapped to the technology available, because FPGA fabric usually does not include latches for simplifying the design flow.

FPGA users should avoid using latches. If a design does have latches and the target FPGA does not have latches, the synthesis tools have to build them out of muxes with feedback loops—which will cause design area increase, performance degradation, and problems with static timing analysis by introducing combinatorial feedback loops that create asynchronous timing problems.

Synthesis tools infer latches from incomplete conditional expressions, such as an if-then-else statement without an else clause. To avoid unintentional latches, specify all conditions explicitly or specify a default assignment. Unintentional latches can be avoided by using clocked registers or by employing any of the following coding techniques:

- Complete Assignment for all input conditions using conditional statements (Figure 13)
- Complete Assignment for all input conditions using case statements (Figure 14)
- Complete Assignment for all input conditions using if and else statements (Figure 15)

Figure 13: Complete Assignment for All Input Conditions Using Conditional Statements

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity complete_assignment_conditional is
port (
    sel, sel_2, sel_3, a, b : in  std_logic;
    f, g                    : out std_logic
  );
end complete_assignment_conditional;

architecture Behavioral of complete_assignment_conditional is

signal sel_wire : std_logic_vector(2 downto 0);

begin

-- concatenate inputs for convenience
sel_wire <= (sel_3 & sel_2 & sel);

f <= a when (sel_wire(0) = '0') else b;

g <=   (a and b)       when (sel_wire = "000") else
       (not (a and b)) when (sel_wire = "001") else
       (a and b)       when (sel_wire = "010") else
       (a and b)       when (sel_wire = "011") else
       (not b)         when (sel_wire = "100") else
       (a xor b)       when (sel_wire = "101") else
       (not a)         when (sel_wire = "110") else
       (not a)         when (sel_wire = "111");

end Behavioral;
```

Figure 13: Complete Assignment for All Input Conditions Using Conditional Statements (Continued)

Verilog HDL

```
module complete_assignment_conditional (
    input sel, sel_2, sel_3,
    input a,b,
    output f,g
  );

wire [2:0] sel_wire;

assign sel_wire = {sel_3, sel_2, sel};  // concatenate inputs
for convenience

assign f = (sel_wire[0] ? a : b);

assign g = (sel_wire == 3'b000 ? (a & b)  :
           (sel_wire == 3'b001 ? !(a & b) :
           (sel_wire == 3'b010 ? (a & b)  :
           (sel_wire == 3'b011 ? (a & b)  :
           (sel_wire == 3'b100 ? !b       :
           (sel_wire == 3'b101 ? (a ^ b)  :
           (sel_wire == 3'b110 ? (!a      :
         (sel_wire == 3'b111 ? (!a     : !a)))))))));
endmodule
```

Figure 14: Assigning Outputs for All Input Conditions Using Case Statements

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity complete_assignment_case is
port (
    sel, sel_2, sel_3, a, b : in  std_logic;
    f, g                    : out std_logic
  );
end complete_assignment_case;

architecture behavioral of complete_assignment_case is

signal sel_wire : std_logic_vector(2 downto 0);

begin

sel_wire <= (sel_3 & sel_2 & sel);

process (sel_wire, a, b)
  begin
    case (sel_wire) is

      when "000" =>
          g <= (a and b);
          f <=  b;

      when "001" =>
          g <= not(a and b);
          f <= a;

      when "010" =>
          g <= (a and b);
          f <=  b;

      when "011" =>
          g <= (a and b);
          f <= a;

      when "100" =>
          g <= not b;
          f <=  b;

      when "101" =>
          g <= (a xor b);
          f <=  a;

      when "110" =>
          g <= not a;
          f <=  b;

      when "111" =>
          g <= not a;
          f <=  a;
```

Figure 14: Assigning Outputs for All Input Conditions Using Case Statements (Continued)

```
-- vhdl requires others case even though all cases are
accounted for
      when others =>
          g <= not a;
          f <=  a;

    end case;
end process;

end behavioral;
```

Verilog HDL

```
module complete_assignment_assign (
    input sel, sel_2, sel_3,
    input a,b,
    output reg f,g
  );

wire [2:0] sel_wire;

assign sel_wire = {sel_3, sel_2, sel};  // concatenate
inputs for convenience

always @ (sel_wire or a or b)
  begin
    case (sel_wire)
      3'b000:
        begin
          g <= (a & b);
          f <=  b;
        end

3'b001:
        begin
          g <= !(a & b);
          f <= a;
        end

      3'b010:
        begin
          g <= (a & b);
          f <=  b;
        end
```

Figure 14: Assigning Outputs for All Input Conditions Using Case Statements (Continued)

```
 3'b011:
        begin
          g <= (a & b);
          f <= a;
        end

      3'b100:
        begin
          g <= !b;
          f <=  b;
        end

      3'b101:
        begin
          g <= (a ^ b);
          f <=  a;
        end

      3'b110:
        begin
          g <= !a;
          f <=  b;
        end

      3'b111:
        begin
          g <= !a;
          f <=  a;
        end

    endcase
  end
// no default case needed because all cases are accounted
for

endmodule
```

Figure 15: Complete Assignment for All Input Conditions Using If and Else Statements

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity complete_assignment_if_else is
port (
    sel, sel_2, sel_3, a, b : in  std_logic;
    f, g                    : out std_logic
  );
end complete_assignment_if_else;

architecture behavioral of complete_assignment_if_else is

begin
process (sel, sel_2, sel_3, a, b)
  begin
    if (sel = '1') then
      f <= a;
      if (sel_2 = '1') then
        g <= not a;
      else
        if (sel_3 = '1') then
          g <= (a xor b);
        else
          g <= not b;
        end if;
      end if;
    else
      f <= b;
      if (sel_2 = '1') then
        g <= (a and b);
      else
        if (sel_3 = '1') then
          g <= not (a and b);
        else
          g <= (a and b);
        end if;
      end if;
    end if;
end process;

end behavioral;
```

Figure 15: Complete Assignment for All Input Conditions Using If and Else Statements (Continued)

Verilog HDL

```
module complete_assignment_if_else (
    input sel, sel_2, sel_3,
    input a,b,
    output reg f,g
  );

always @ (sel or sel_2 or sel_3 or a or b)
  begin
    if (sel == 1)
      begin
        f = a;
        if (sel_2 == 1)
          g = ~ a;
        else
          begin
            if (sel_3 == 1)
              g = a ^ b;
            else
              g = ~ b;
          end
      end
    else
      begin
        f = b;
        if (sel_2 == 1)
          g = (a & b);
        else
          if (sel_3 == 1)
            g = ~(a & b);
        else
            g = a & b;
      end
    end
endmodule
```

Another way to avoid unintentional latches is to check the synthesis tool outputs. Most of the synthesis tools give warnings whenever there are latches in the design. Checking the warning list after synthesis saves a tremendous amount of effort in trying to determine why a design is so large later in the place-and-route stage.

Register Control Signals

The general-purpose latches and flip-flops in the PFU are used in a variety of configurations, depending on the device family.

For example, in the LatticeEC family of devices, you can apply clock, clock-enable, and LSR control to the registers on a slice basis. Each slice contains two LUT4 lookup tables feeding two registers (programmed to be in flip-flop or latch mode) and some associated logic that allows the LUTs to be combined to perform functions, such as LUT5, LUT6, LUT7, and LUT8. Control logic performs set/reset functions (programmable as synchronous/asynchronous), clock-select, chip-select, and wider RAM/ROM functions.

When writing design codes in HDL, keep the architecture in mind to avoid wasting resources in the device. Here are several points for consideration:

- If the register number is not a multiple of 2 or 4 (dependent on device family), try to code the registers in such a way that all registers share the same clock, and in a way that all registers share the same control signals.
- Lattice Semiconductor FPGA devices have multiple dedicated clock enable signals per PFU. Try to code the asynchronous clocks as clock enables, so that PFU clock signals can be released to use global low-skew clocks.
- Try to code the registers with local synchronous set/reset and global asynchronous set/reset.

For more detailed architecture information, refer to the Lattice Semiconductor FPGA data sheets.

Global Reset (GSR) and Local Resets (LSR)

Lattice FPGAs contains a GSR (Global Set Reset) resource. The GSR hardware resource in Lattice FPGAs provides a convenient mechanism to allow design components to be reset without using any general routing resources. How the design is coded can impact how much the GSR resource can be exploited. During power up, the device is configured with its bitstream, a reset is issued across the entire device to put it into a known state, and then the device begins to operate. This reset event is called Power Up Reset, and it is distributed across the device using the GSR resource.

Note: A PUR component is provided to allow simulation test benches to simulate this pulse, but this component is never used as part of the design.

There are two primary ways to take advantage of the GSR hardware resource in your design: use the GSR to reset all components on your FPGA or to use the GSR to eliminate any routing resources needed for one reset in a multiple reset design. If there is only one reset signal for the entire design, you would want to use the GSR in the first way, to reset all components on the FPGA. When using the GSR to eliminate any routing resources needed for one reset in a multiple reset design, typically the GSR would be used for the reset with the highest fan-out.

The GSR can only be used for asynchronous active low resets due to the underlying hardware. The software will take this into account automatically and will not implement a synchronous reset using GSR when the Inferred GSR or User Specified Inferred GSR modes are used.

GSR Usage Cases

There are three cases with respect to initialization set/resets: Inferred GSR, Global GSR, and LSR (No GSR). The three GSR usage cases are defined as follows:

- Inferred GSR – In this usage case, the software automatically determines which reset signal has the highest fan-out (for either single or multiple reset designs) and uses the GSR resource as the routing for that reset signal. This usage case is the default condition if there is no user-instantiated GSR component in the design. This usage case is the best choice for most applications. The software determines the reset with the most loads and uses the GSR resource for that signal, which provides the largest reduction in needed routing resources. The Inferred GSR usage case can be used whether the design has a single or multiple resets. The user can also optionally specify, through a preference, which reset signal is to be implemented using the GSR resource.
- Global GSR – This usage case delivers a reset pulse (over GSR) to all elements in the design, even if an element is not connected in the HDL to the signal driving this reset pulse.
- LSR (No GSR) – LSR (local set/reset) specifies that no GSR is to be used, which means that all resets will use local routing resources instead of using the GSR resource.

In the Inferred GSR case, the software will only connect elements with an asynchronous reset to GSR. Elements requiring synchronous reset will use only local routing.

Inferred GSR

The Inferred GSR usage case is the simplest to use. If everything is left to default software settings and no GSR component is instantiated in the design, then the software will implement the reset signal with the highest fan-out of active low asynchronous elements on the GSR resource. Inferred GSR is the recommended usage case unless any of the following conditions exist:

- If you need to implement a specific reset signal on GSR.

- If you need to implement a global reset and want to avoid coding it throughout the HDL.
- If you need to completely disable GSR.

To use the Inferred GSR usage case, there are no design changes necessary. You simply implement the design with default settings. To specify which net to use, instead of having the software automatically determine this, you can use the GSR_NET preference.

Global GSR

The Global GSR usage case is intended for all elements in a design to be reset using the GSR resource. This usage is a good fit for a design with a single reset. It can also be used with multiple resets in the design, but this can produce unexpected functionality and is not recommended. See the topic “How to Use the Global Set/Reset (GSR) Signal” in the Lattice Diamond online Help.

To use the Global GSR usage case, a GSR component must be instantiated in the design and connected to the signal that is targeted as the reset signal, usually a primary input. If a GSR component is not instantiated in the design, the software will not treat the design as a Global GSR usage case. The GSR component must be instantiated into the design itself, not into the test bench for the design

LSR (No GSR)

The LSR (local set/reset) usage case always uses local routing for the reset signals and does not use the GSR resource. This is the recommended usage case if there is a requirement to do timing analysis on the reset signals or if synchronous reset is being used throughout the design. To use the LSR usage case, there must be no GSR instantiated in the design, no GSR_NET preference specified, and the software settings used must not infer any GSR resource.

Clock Enable

Figure 16 shows an example of gated clocking. Gating clocks is not encouraged in digital designs because it can cause timing issues, such as unexpected clock skews. The structure of the PFU makes the gating clock even more undesirable, because it uses up all the clock resources in one PFU and sometimes wastes the flip-flop and latch resources in the PFU..

Figure 16: Asynchronous: Gated Clocking (Not Recommended)

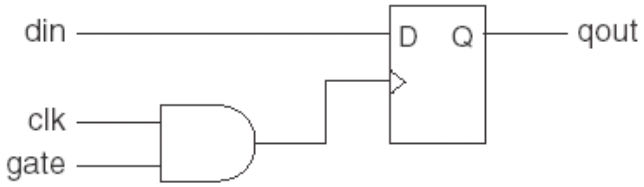
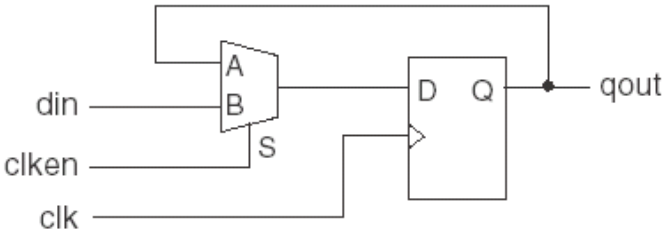


Figure 17 shows a better alternative. By using the clock enable in the PFU, you can achieve the same functionality without worrying about timing issues, since only one signal is controlling the clock. Since only one clock is used in the PFU, all related logic can be implemented in one block to achieve better performance. Lastly, lower power can be achieved by using the clock enable to keep registers from switching when not needed.

Figure 17: Synchronous: Clock Enabling (Recommended)



Samples of the VHDL and Verilog HDL code for clock enable are shown in Figure 18.

Figure 18: Clock Enable Coding

VHDL

```
Clock_Enable: process (clk, clken, din)
begin
  if (clk'event and clk = '1') then
    if (clken = '1') then
      qout <= din;
    end if;
  end if;
end process Clock_Enable;
```

Verilog HDL

```
always @(posedge clk)
  qout <= clken ? din : qout;
```

The following are guidelines for coding the clock enable in Lattice Semiconductor FPGAs:

- Clock enable is only supported by flip-flops, not latches.
- Flip-flop pairs inside a slice block share the same clock enable.
- All flip-flops have a positive clock enable input.
- The clock-enable input has higher priority than the synchronous set/reset by default.

Regarding the priority, you can program the synchronous LSR to have a higher priority than the clock enable by instantiating the library element in the source code. For example, the library element FD1P3IX is a flip-flop that allows the synchronous clear to override the clock enable. You can also specify the priority of generic coding by setting the priority of the control signals differently.

The examples in Figure 19 and Figure 20 demonstrate coding methodologies to help the synthesis tools set the priorities of the clock enable and the synchronous LSR.

Figure 19: Clock Enable over Synchronous LSR

VHDL

```
COUNT8: process (CLK, GRST)
begin
  if (GRST = '1') then
    cnt <= (others => '0');
  elsif (CLK'event and CLK = '1') then
    if (CKEN = '1') then
      cnt <= cnt + 1;
    elsif (LRST = '1') then
      cnt <= (others => '0');
    endif;
  endif;
end process COUNT8;
```

Verilog HDL

```
always @(posedge CLK or posedge GRST)
begin
  if (GRST)
    cnt = 4'b0;
  else if (CKEN)
    cnt = cnt + 1'b1;
  else if (LRST)
    cnt = 4'b0;
end
```

Figure 20: Synchronous LSR over Clock Enable

VHDL

```
COUNT8: process (CLK, GRST)
begin
  if (GRST = '1') then
    cnt <= (others => '0');
  elsif (CLK'event and CLK = '1') then
    if (LRST = '1') then
      cnt <= (others => '0');
    elsif (CKEN = '1') then
      cnt <= cnt + 1;
    endif;
  endif;
end process COUNT8;
```

Verilog HDL

```
always @(posedge CLK or posedge GRST)
begin
  if (GRST)
    cnt = 4'b0;
  else if (LRST)
    cnt = 4'b0;
  else if (CKEN)
    cnt = cnt + 1'b1;
end
```

Multiplexers

The flexible configurations of LUTs within slice blocks can realize any 4-, 5-, 6-, 7-, or 8-input logic function like 2-to-1, 3-to-1, 4-to-1, or 5-to-1 multiplexers.

You can efficiently create larger multiplexers by programming multiple 4-input LUTs. Synthesis tools can automatically infer Lattice Semiconductor FPGA optimized multiplexer library elements according to the behavioral description in the HDL source code. This provides the flexibility to the mapper and place-and-route tools to configure the LUT mode and connections in an optimal fashion.

Figure 21: 16:1 Multiplexer

VHDL

```
process (sel, din)
begin
  if    (sel = "0000") then muxout <= din(0);
  elsif (sel = "0001") then muxout <= din(1);
  elsif (sel = "0010") then muxout <= din(2);
  elsif (sel = "0011") then muxout <= din(3);
  elsif (sel = "0100") then muxout <= din(4);
  elsif (sel = "0101") then muxout <= din(5);
  elsif (sel = "0110") then muxout <= din(6);
  elsif (sel = "0111") then muxout <= din(7);
  elsif (sel = "1000") then muxout <= din(8);
  elsif (sel = "1001") then muxout <= din(9);
  elsif (sel = "1010") then muxout <= din(10);
  elsif (sel = "1011") then muxout <= din(11);
  elsif (sel = "1100") then muxout <= din(12);
  elsif (sel = "1101") then muxout <= din(13);
  elsif (sel = "1110") then muxout <= din(14);
  elsif (sel = "1111") then muxout <= din(15);
  else muxout <= '0';
  end if;
end process;
```

Figure 21: 16:1 Multiplexer (Continued)

```
-- or doing it outside a process guarantees that your results
are not dependant on
-- the sensitivity list:

WITH sel SELECT
  muxout <= din(0)  when "0000",
            din(1)  when "0001",
            din(2)  when "0010",
            din(3)  when "0011",
            din(4)  when "0100",
            din(5)  when "0101",
            din(6)  when "0110",
            din(7)  when "0111",
            din(8)  when "1000",
            din(9)  when "1001",
            din(10) when "1010",
            din(11) when "1011",
            din(12) when "1100",
            din(13) when "1101",
            din(14) when "1110",
            din(15) when "1111",
            '0'     when others;
```

Verilog HDL

```
module multiplexer (
    input [3:0] sel,
    input [15:0] din,
    output reg muxout
  );

always @ (sel or din)
  begin
    case (sel)
      4'b0000 :  muxout <= din[0];
      4'b0001 :  muxout <= din[1];
      4'b0010 :  muxout <= din[2];
      4'b0011 :  muxout <= din[3];
      4'b0100 :  muxout <= din[4];
      4'b0101 :  muxout <= din[5];
      4'b0110 :  muxout <= din[6];
      4'b0111 :  muxout <= din[7];
      4'b1000 :  muxout <= din[8];
      4'b1001 :  muxout <= din[9];
      4'b1010 :  muxout <= din[10];
      4'b1011 :  muxout <= din[11];
      4'b1100 :  muxout <= din[12];
      4'b1101 :  muxout <= din[13];
      4'b1110 :  muxout <= din[14];
      4'b1111 :  muxout <= din[15];
      default :  muxout <= 1'b0;
    endcase
  end

endmodule
```

Bidirectional Buffers

The use of bidirectional buffers instead of unidirectional (dedicated inputs or outputs) buffers allows for fewer device pins and, consequently, smaller device packages to be used, which reduces cost. In addition, the ability to disable outputs from toggling, when not needed, reduces power consumption.

You can instantiate bidirectional buffers in the same manner as regular I/O buffers or infer them from the HDL source, as shown in Figure 22. For the most control, disable automatic I/O insertion in your synthesis tool and then manually instantiate the I/O pads for specific pins, as needed.

Figure 22: HDL for Bidirectional Buffer

VHDL

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

entity bireg is port (
  datain : in std_logic_vector (7 downto 0);
  clk,en_o : in std_logic;
  Qo1 : out std_logic_vector (7 downto 0);
  Qio : inout std_logic_vector (7 downto 0));
end bireg;
architecture beh of bireg is
  signal Q_reg : std_logic_vector (7 downto 0);
  signal Qio_int : std_logic_vector (7 downto 0);
begin
  process(clk,datain) begin
    if clk'event and clk = '1' then
      Q_reg <= datain;
    end if;
  end process;
process (Q_reg,en_o) begin
  if en_o = '1' then
      Qio_int <= Q_reg ;
    else
      Qio_int <= (others=>'Z');
    end if;
  end process;
  Qio <= Qio_int;
  Qo1 <= Qio;
end;
```

Figure 22: HDL for Bidirectional Buffer (Continued)

Verilog HDL

```
module bireg (datain, clk, en_o, Qo1, Qio);
  input [7:0] datain;
  input clk, en_o;
  output [7:0] Qo1;
  inout [7:0] Qio;
  reg [7:0] Q_reg;
  reg [7:0] Qio_int;
  wire [7:0] Qo1;
  wire [7:0] Qio;
  always @(posedge clk)
  begin
    Q_reg = datain;
  end
  always @(en_o or Q_reg)
  begin
    if (en_o)
      Qio_int <= Q_reg;
    else
      Qio_int <= 8'hz;
  end
  assign Qio = Qio_int;
  assign Qo1 = Qio;
endmodule
```

Cross Clock Domains

When passing data from one clock domain to another, special care must be taken to ensure that metastability issues do not arise as a result of set-up and hold timing violations. The general recommendation to address this depends on whether just a single signal or a data bus will be passed between clock domains. For single signals, bring the data into the second clock domain using a double register structure, as shown in Figure 23. This will keep any metastability from propagating into the second clock domain.

For data buses in difference clock domains, asynchronous FIFOs are recommended, which will allow the data to be written with one clock and read with another, thereby avoiding set-up and hold timing violations and any resulting metastability. As previously mentioned in the Distributed and Block Memory section, IPexpress should be used, since the FIFO will be built using embedded block RAM. The control logic is coded using the write enable, write data, full, and almost flags for the write clock domain, and the read data, read enable, empty, and almost empty in the read clock domain in conjunction with a state machine.

Figure 23: Using a Double Register Structure for Passing a Single Signal into another Clock Domain

VHDL

```
process (clk_a)
  begin
    if rising_edge (clk_a) then
      a_reg  <= signal_a;
    end if;
  end process;

process (clk_b)
  begin
    if rising_edge (clk_b) then
      b_reg    <= a_reg;
      signal_b <= b_reg;
    end if;
  end process;
```

Verilog HDL

```
always @ (posedge clk_a)
  begin
    a_reg <= signal_a;
  end

always @ (posedge clk_b)
  begin
    b_reg    <= a_reg;
    signal_b <= b_reg;
  end
```

HDL Coding for Distributed and Block Memory

Although an RTL description of RAM is portable and the coding is straightforward, because the structure of RAM blocks in every architecture are unique, synthesis tools may not generate optimized RAM implementations and could generate inefficient netlists. For Lattice Semiconductor FPGA devices, it is generally recommended that RAM blocks be generated through IPexpress in Diamond.

When implementing large memories, use the embedded block RAM (EBR) components found in every Lattice Semiconductor FPGA device and be sure the use of Output Registers is enabled (default for IPexpress) for improved timing performance. When implementing small memories, use the resources in the PFU. Using Diamond IPexpress, you can target a memory module to the PFU-based distributed memory or to the sysMEM EBR block.

Lattice Semiconductor FPGAs support many different memory types, including synchronous dual-port RAM, synchronous single-port RAM, synchronous FIFO, and synchronous ROM. For more information on supported memory types per FPGA architecture, consult the Lattice Semiconductor FPGA data sheets.

Resource Sharing

Resource sharing is generally the default for most synthesis tools, including Synopsis Synplify and Mentor Graphics Precision RTL Synthesis, because it usually produces efficient implementations by conserving resources when possible. However, it might do this at the expense of timing performance by creating longer routes and adding to routing congestion. If global application

of resource sharing is causing timing problems, turn it off globally, and then implement only as needed to conserve area using attributes on an individual basis on lower level modules/architectures. How this is done varies, so refer to the documentation for your synthesis tool for details. The example in Figure 24 is for Synplify Pro.

Figure 24: Synplify Pro Attributes to Control Resource Sharing

VHDL

```
VHDL architecture rtl of lower is
attribute syn_sharing : string;
attribute syn_sharing of rtl : architecture is "off";
```

Verilog HDL

```
Verilog module lower(out, in, clk_in)
/* synthesis syn_sharing = "on" */;
```

Finite State Machine Guidelines

A finite state machine is a hardware component that advances from the current state to the next state at the clock edge. This section discusses methods and strategies for state machine encoding.

State Encoding Methods for State Machines

There are several ways to encode a state machine, including sequential (binary), gray code, and one-hot encoding. State machines with sequentially encoded states have minimal numbers of flip-flops and wide combinatorial functions. However, most FPGAs have many flip-flops and relatively narrow combinatorial function generators. Sequential or gray-code encoding schemes can result in inefficient implementation in terms of speed and density for FPGAs. On the other hand, a one-hot encoded state machine represents each state with one flip-flop. As a result, it decreases the width of combinatorial logic, which matches well with register rich FPGA architectures.

In general, for FPGA Architectures, sequential or gray code encoding results in a smaller area implementation and one-hot encoding results in a faster implementation. For small state machines, less than 5 states, sequential is typically the default because it only requires two bits so decoding the state is fairly minimal. For larger state machines, 5 states or greater, one-hot is the default because even though each state requires a register, the states do not need to be decoded allowing for better performance.

There are various reasons you may choose to invoke a particular state machine encoding scheme for a design. For example, you may choose to use sequential encoding on a large state machine because performance in not critical and you need to use a lot of register elsewhere in your design. You can

hard code the states in the source code by specifying a numerical value for each state. This approach ensures the correct encoding of the state machine but is more restrictive in the coding style. Alternatively, the enumerated coding style leaves the flexibility of state machine encoding to the synthesis tools. Most synthesis tools allow you to define encoding styles either through attributes in the source code or through the tool's user interface. Each synthesis tool has its own synthesis attributes and syntax for choosing the encoding styles. Refer to your synthesis tool's documentation for details about attributes syntax and values.

Here is a VHDL example of enumeration states:

```
type STATE_TYPE is (S0,S1,S2,S3,S4);
signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
```

The following is an example of Synplify VHDL synthesis attributes:

```
attribute syn_encoding : string;
attribute syn_encoding of <signal_name> : type is "value ";
-- The syn_encoding attribute has 4 values:
-- sequential, onehot, gray and safe.
```

The following is an example of Precision RTL Synthesis VHDL synthesis attributes:

```
-- Declare TYPE_ENCODING_STYLE attribute
-- Not needed if the exemplar_1164 package is used
type encoding_style is (BINARY, ONEHOT, GRAY, RANDOM, AUTO);
attribute TYPE_ENCODING_STYLE : encoding style;
...
attribute TYPE_ENCODING_STYLE of <typename> : type is ONEHOT;
```

In Verilog HDL, you must provide explicit state values for states by using a bit pattern, such as 3'b001, or by defining a parameter and using it as the case item. The latter method is preferable. The following is an example using parameter for state values:

```
Parameter state1 = 2'h1, state2 = 2'h2;
...
current_state = state2; setting current state to 2'h2
```

The attributes in the source code override the default encoding style assigned during synthesis. Since Verilog HDL does not have predefined attributes for synthesis, attributes are usually attached to the appropriate objects in the source code as comments. The attributes and their values are case-sensitive and usually appear in lower case. The following example uses attributes in the Synplify Verilog HDL source code to specify state machine encoding style:

```
Reg[2:0] state; /* synthesis syn_encoding = "value" */;
// The syn_encoding attribute has 4 values:
// sequential, onehot, gray and safe.
```

In Precision RTL Synthesis, it is also recommended that you define a Verilog HDL parameter and use it as the case item. The setup_design_encoding command in Precision RTL Synthesis is used to specify the encoding style.

Typically, synthesis tools select the optimal encoding style that takes into account the target device architecture and size of the decode logic. You can

always apply synthesis attributes to override the default encoding style if necessary.

State Machine Coding Guidelines

As mentioned earlier, the preferred scheme for FPGA architectures is one-hot encoding. This section discusses some common issues that you may encounter when constructing state machines, such as initialization and state coverage and special case statements in Verilog HDL.

General State Machine Description

Generally, there are two approaches to describing a state machine. One approach is to use one process or block to handle both state transitions and state outputs. The other is to separate the state transition and the state outputs into two different processes or blocks. The latter approach is more straightforward, because it separates the synchronous state registers from the decoding logic that is used in the computation of the next state and the outputs. This not only makes the code easier to read and modify but makes the documentation more efficient. If the outputs of the state machine are combinatorial signals, the second approach is almost always necessary because it prevents the accidental registering of the state machine outputs.

The examples in Figure 25 and Figure 26 describe a simple state machine in VHDL and Verilog HDL. In the VHDL example, a sequential process is

separated from the combinatorial process. In the Verilog HDL code, two always blocks are used to describe the state machine in a similar way.

Figure 25: VHDL Example for State Machine

```
architecture lattice_fpga of dram_refresh is
type state_typ is (s0, s1, s2, s3, s4);
signal present_state, next_state : state_typ;

begin
  -- process to update the present state
  registers: process (clk, reset)
  begin
    if (reset = '1') then
      present_state <= s0;
    elsif clk'event and clk='1' then
      present_state <= next_state;
    end if;
  end process registers;

  -- process to calculate the next state & outputs
  transitions: process (present_state, refresh, cs)
  begin
    ras <= 'X'; cas <= 'X'; ready <= 'X';
    case present_state is
      when s0 =>
        if (refresh = '1') then
          next_state <= s3;
          ras <= '1'; cas <= '0'; ready <= '0';
        elsif (cs = '1') then
          next_state <= s1;
          ras <= '0'; cas <= '1'; ready <= '0';
        else
          next_state <= s0;
          ras <= '0'; cas <= '1'; ready <= '1';
        end if;
      when s1 =>
        next_state <= s2;
        ras <= '0'; cas <= '0'; ready <= '0';
      when s2 =>
        if (cs = '0') then
          next_state <= s0;
          ras <= '1'; cas <= '1'; ready <= '1';
        else
          next_state <= s2;
          ras <= '0'; cas <= '0'; ready <= '0';
        end if;
      when s3 =>
        next_state <= s4;
        ras <= '1'; cas <= '0'; ready <= '0';
      when s4 =>
        next_state <= s0;
        ras <= '0'; cas <= '0'; ready <= '0';
      when others =>
        next_state <= s0;
        ras <= '0'; cas <= '0'; ready <= '0';
    end case;
  end process transitions;
```

Figure 26: Verilog HDL Example for State Machine

```
parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4;

reg[2:0] present_state, next_state;
reg ras, cas, ready;

// always block to update the present_state
always @(posedge clk or posedge reset)
begin
  if (reset) present_state = s0;
  else present_state = next_state;
end

// always block to calculate the next state & outputs
always @ (present_state or refresh or cs)
begin
  next_state = s0;
  ras = 1'bX; cas = 1'bX; ready = 1'bX;
  case (present_state)
    s0 : if (refresh) begin
        next_state = s3;
        ras = 1'b1; cas = 1'b0; ready = 1'b0;
      end
      else if (cs) begin
        next_state = s1;
        ras = 1'b0; cas = 1'b1; ready = 1'b0;
      end
      else begin
        next_state = s0;
        ras = 1'b0; cas = 1'b1; ready = 1'b1;
      end
    s1 : begin
        next_state = s2;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
    s2 : if (~cs) begin
        next_state = s0;
        ras = 1'b1; cas = 1'b1; ready = 1'b1;
      end
      else begin
        next_state = s2;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
    s3 : begin
        next_state = s4;
        ras = 1'b1; cas = 1'b0; ready = 1'b0;
      end
    s4 : begin
        next_state = s0;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
    default : begin
        next_state = s0;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
  endcase
end
```

Initialization and Default State for Safe State Machine

A state machine must be initialized to a valid state after power-up. You can initialize it at the device level during power-up or by including a reset operation to bring it to a known state. For all Lattice Semiconductor FPGA devices, the global set/reset (GSR) is pulsed at power-up, regardless of the function to employ the GSR defined in the design source code. The examples in Figure 27, show how asynchronous reset can be used to bring the state machine to a valid initialization state.

In the same manner, a state machine should have a default state to ensure that the state machine does not go into an invalid state. This could happen if not all the possible combinations are clearly defined in the design source code. VHDL and Verilog HDL have different syntax for default state declaration. In VHDL, if a case statement is used to construct a state machine, “when others” should be used as the last statement before the end of the statement. If an if-then-else statement is used, “else” should be the last assignment for the state machine. In Verilog HDL, use “default” as the last

assignment for a case statement, and use “else” for the if-then-else statement. Again, see the examples in Figure 27.

Figure 27: Initialization and Default State Example

When Others in VHDL

```
architecture lattice_fpga of FSM1 is
type state_typ is
  (deflt, idle, read, write);
signal next_state : state_typ;

begin
  process (clk, rst)
  begin
    if (rst = '1') then
      next_state <= idle; dout <= '0';
    elsif (clk'event and clk = '1') then
      case next_state is
        when idle =>
          next_state <= read;
          dout <= din(0);
        when read =>
          next_state <= write;
          dout <= din(1);
        when write =>
          next_state <= idle;
          dout <= din(2);
        when others =>
          next_state <= deflt;
          dout <= '0';
      end case;
    end if;
  end process;
```

Default Clause in Verilog HDL

```
// Define state labels explicitly
parameter deflt = 2'bxx;
parameter idle  = 2'b00;
parameter read  = 2'b01;
parameter write = 2'b10;

reg[1:0] next_state;
reg dout;

always @(posedge clk or posedge rst)
  if (rst) begin
    next_state <= idle;
    dout <= 1'b0;
  end
  else begin
    case (next_state)
      idle: begin
        next_state <= read;
        dout <= din[0];
      end
      read: begin
        next_state <= write;
        dout <= din[1];
      end
      write: begin
        next_state <= idle;
        dout <= din[2];
      end
      default: begin
        next_state <= deflt;
        dout <= 1'b0;
      end
  end
```

Full Case and Parallel Case Specification in Verilog HDL

Verilog HDL has additional attributes to define the default state without writing it specifically in the code. The “full_case” attribute is intended to achieve the same results as “default,” and the “parallel_case” attribute makes sure that all the statements in a case statement are mutually exclusive and informs the synthesis tools that only one case can be true at a time. However, both of these can cause a simulation mismatch after synthesis, so it is recommended that they not be used. Instead, it is recommended that all necessary branches be written for if-else and case statements.

Lattice FPGA Hardware Features

SERDES/PCS

Use IPexpress to generate Serializer/Deserializer (SERDES) Physical Coding Sublayer (PCS) logic, which can be configured to support numerous industry-standard, high-speed serial data transfer protocols such as GbE, XAUI, SONET/SDH, PCI Express, SRIO, CPRI, OBSAI, SD-SDI, HD-SDI and 3G-SDI. In addition, the protocol-based logic can be fully or partially bypassed in a number of configurations to allow users flexibility in designing their own high-speed data interface.

Although there are no specific HDL coding recommendations, since IPexpress generates the PCS logic, the handing of the interface—both inside the FPGA and outside the pins at the board level—is the designer's responsibility. If both areas are handled correctly, the SERDES/PCS will perform as specified. With regards to the interface inside the FPGA and within the scope of this chapter, perhaps the most important consideration is that it is a parallel data bus. This means the data must be registered very near the interface. Similar to the previous section on crossing clock domains, an asynchronous FIFO works well for this but the latency must be accounted for.

DSP

For DSP designs in Lattice Semiconductor FPGAs, the use of IPexpress is recommended over relying on inference and the synthesis tool's ability to correctly utilize sysDSP slice resources in the fabric. Another alternative is to directly instantiate the sysDSP slice primitives. The advantage of instantiating primitives is that it provides access to all ports and sets all available parameters available to the user. The disadvantage of this flow is that all this customization requires extra coding by the user. This approach is only recommended if IPexpress does not provide a very specific implementation that your design requires.

Low Power

In addition to the recommendations for achieving lower power that are mentioned elsewhere in this chapter, here are some other guidelines:

- Optimize for area whenever possible to reduce routing lengths. Synthesis tools tend to favor area over performance by default, trading off smaller area for performance. So it is important to understand that changing synthesis directives can also impact power consumption.
- Use IPexpress as much as possible for the most power-efficient (least area and resources) implementation. This is especially true for DSP/ Arithmetic functions to ensure that the sysDSP slices are used instead of regular slice logic.

- Eliminate known glitches for power reasons, even if they are not causing functional problems. The addition of extra logic or registers to do this is a good trade-off for eliminating needless power consumption from high-frequency switching due to glitches.
- Stagger I/O toggling and reduce the toggle rate whenever possible.

Coding to Avoid Simulation/Synthesis Mismatches

Certain coding styles can lead to pre-synthesis simulation that differs from post-synthesis gate-level simulations. This problem is caused by HDL models that contain information that cannot be passed to the synthesis tool because of style or pragmas that are ignored by a simulator. Many error-prone coding styles will be detected by running best-known-method (BKM) Check, which is available from the Design menu.

The examples in this section illustrate common mistakes to avoid. Where possible, examples of BKM messages are also provided.

Sensitivity Lists

In VHDL and Verilog HDL, combinational logic is typically modeled using a continuous assignment. Combinational logic can also be modeled when using a Verilog always statement or a VHDL process statement in which the event sensitivity list does not contain any edge events (posedge/negedge or `event). The event sensitivity list does not affect the synthesized netlist. Therefore, it might be necessary to include all the signals read in the event sensitivity list to avoid mismatches between simulation and synthesized logic.

Figure 28 shows a style that leads to a mismatch due to an incomplete sensitivity list. During pre-synthesis simulation, the always statement is only activated when an event occurs on variable a. However, the post-synthesis result will infer a 2-input and gate.

Figure 28: Coding Style that Leads to a Mismatch Due to an Incomplete Sensitivity List

```
module code1b (o, a, b);
  output o;
  input a, b;
  reg o;

  always @(a)
    o = a & b;

endmodule

// Supported, but simulation mismatch may occur.
// To assure the simulation will match the synthesized logic,
// add variable b to the event list so the event list
// reads: always @(a or b).
// Alternatively, Verilog 2001 supports:
// Always @(*)
// O= a & b;
// which will avoid an incomplete sensitivity list, but it’s
still
// recommended to write RTL to avoid an incomplete sensitivity
list.
```

The synthesis-related BKM check reported is:

```
WARNING: (ST-6003) Always Block 'code1b.@( a)' has the
following blocking assignment with driving signals that are not
in the sensitivity list. Possible Simulation/Synthesis
mismatch.
// o = (a & b) ;
```

Not all variables that appear in the right-hand side of an assignment are required to appear in the event sensitivity list. For example, Verilog variables that are assigned values inside the always statement body before being used by other expressions do not have to appear in the sensitivity list.

Blocking/Nonblocking Assignments in Verilog

A subtle Verilog coding style that can lead to unexpected results is the blocking/nonblocking style of variable assignment. The following guidelines are recommended:

- Use blocking assignments in always blocks that are written to generate combinational logic.
- Use nonblocking assignments in always blocks that are written to generate sequential logic.
- Use nonblocking assignments with register models to avoid race conditions.

Execution of blocking assignments can be viewed as a one-step process:

- Evaluate the RHS (right-hand side equation) and update the LHS (left hand side expression) of the blocking assignment without interruption from any other Verilog statement. A blocking assignment "blocks" trailing assignments in the same always block, meaning that it prevents them from occurring until after the current assignment has been completed.

A problem with blocking assignments occurs when the RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block and both equations are scheduled to execute in the same simulation time step, such as on the same clock edge. If blocking assignments are not properly ordered, a race condition can occur. When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

According to the IEEE Verilog Standard for the language itself (not the synthesis standard), the two always blocks can be scheduled in any order. In Figure 29, if the first always block executes first after a reset, both y1 and y2 will take on the value of 1. If the second always block executes first after a reset, both y1 and y2 will take on the value 0. This clearly represents a race condition.

Figure 29: Two Always Blocks Can Be Scheduled in Any Order

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input clk, rst;

  reg y1, y2;

always @(posedge clk or posedge rst)
  if (rst) y1 = 0; // reset
  else y1 = y2;
always @(posedge clk or posedge rst)
  if (rst) y2 = 1; // preset
  else y2 = y1;
endmodule
```

BKM will also report potential problems, given the combination of an edge-based sensitivity list with blocking assignments. For example:

```
// WARNING: (SUNBURST-0001) Always Block 'lfsrb1.@(posedge clk
or negedge pre_n)' has a edge based sensitivity list, but has
the following blocking assignments. Possible Simulation/
Synthesis mismatch.
// q3 = 1'b1 ;
// q2 = 1'b1 ;
// q1 = 1'b1 ;
// q3 = q2 ;
// q2 = n1 ;
// q1 = q3 ;
```

In Verilog, a variable assigned in an always statement cannot be assigned using both a blocking assignment (=) and a non-blocking assignment (<=) in the same always block.

```
always @ (IN1 or IN2 or SEL) begin
  OUT = IN1;
  if (SEL)
    OUT <= 2;
end
```

References

"RTL Coding Styles that Yield Simulation and Synthesis Mismatches"
Don Mills, LCDM Engineering
Clifford E. Cummings, Sunburst Design, Inc.

"Nonblocking Asisgnments in Verilog Synthesis, Coding Styles That Kill!"
Clifford E. Cummings, Sunburst Design, Inc.
http://www.sunburst-design.com/papers

Signal Fan-Out

Signal fan-out refers to the number of inputs that can be connected to an output before the current required by the inputs exceeds the current that can be delivered by the output while maintaining correct logic levels or performance requirements. FPGA logic synthesis will automatically maintain reasonable fan-out levels by replicating drivers or buffering a signal. Because of this behavior, the resulting FPGA route might be slower due to the additional intrinsic delays.

Signal fan-out control is available with logic synthesis to maintain reasonable fan-outs by controlling the degree to which drivers are replicated. You should anticipate the availability of FPGA routing resources that are reserved for high fan-out, low-skew networks like clocks, clock-enables, resets, and others. BKM can be configured, through Tools > Options, to detect high fan-out conditions, as in the following example:

```
WARNING: (ST-5002) Net 'sc_dist_dpram.dec_wre1' violates Max
Fanout Rule with a load of '8' pins.
sc_dist_dpram.v(71,72-71,86): Input:mem_0_0.WRE
sc_dist_dpram.v(81,72-81,86): Input:mem_0_1.WRE
```

```
sc_dist_dpram.v(151,72-151,86): Input:mem_4_0.WRE
sc_dist_dpram.v(161,72-161,86): Input:mem_4_1.WRE
sc_dist_dpram.v(231,72-231,86): Input:mem_8_0.WRE
sc_dist_dpram.v(241,72-241,86): Input:mem_8_1.WRE
sc_dist_dpram.v(311,72-311,86): Input:mem_12_0.WRE
sc_dist_dpram.v(321,72-321,86): Input:mem_12_1.WRE
```

The value set for the fan-out attribute is just a reference, and not an absolute requirement. The software will work as much as possible to meet the value, but might fail in some cases. For example, when using Synplify, **syn_maxfan** and the default fanout guide are suggested guidelines only, but in certain cases they function as hard limits. When they are guidelines, the synthesis tool takes them into account, but does not always respect them absolutely. The synthesis tool does not respect the **syn_maxfan** limit if the limit imposes constraints that interfere with optimization. To ensure that fan-out limits are not overly impacting the meeting of timing constraints, be sure to apply SDC multi-cycle constraints where possible.

Lattice Semiconductor FPGA device architectures are designed to handle high signal fan-outs. When you use clock resources, there are no hindrances on fan-outs. However, synthesis tools tend to replicate logic to reduce fan-out during logic synthesis. For example, if the code implies clock enable and is synthesized with speed constraints, the synthesis tool might replicate the clock-enable logic. This type of logic replication occupies more resources in the devices and makes performance checking more difficult; it also results in additional power consumption. Control the logic replication in the synthesis process by using attributes for a high-fan-out limit.

If logic replication is causing problems, try turning it off globally by setting the fan-out limit very high. Higher fan-out limit means less replicated logic and fewer buffers inserted during synthesis, and a consequently smaller area. Then manually replicate logic only where needed to make timing.