Coding Style Guidelines

Coding Style Guidelines

Introduction

This document was created to provide Xilinx users with a guideline for producing fast, reliable, and reusable HDL code.

.

Table of Contents

Top-Down Design   Section 1

HDL coding should start with a top-down design approach. Use a top-level block diagram to communicate to designers the naming required for signals and hierarchical levels. Signal naming is especially important during the debug stage. Consistent naming of signals, from top to bottom, will ensure that project manager A can easily recognize the signals written by designer B.

Behavioral and Structural Code

When creating synthesizable code (RTL), you should write two types of code: behavioral RTL (leaf-level logic inference, sub-blocks) and structural code (blocks) -- each exclusively in its own architecture. A simple example of behavioral RTL versus structural code is shown in Figure 13-1and Figure 13-2, respectively.

```
entity mux2to1 is
   port (
       a     : in  std_logic_vector(1 downto 0);
       sel   : in  std_logic;
       muxed : out std_logic);
end mux2to1;

architecture rtl of mux2to1 is
begin

    muxed <= a(1) when sel = '1' else a(0);

end rtl;
```

Figure 13-1 Behavioral Code

```
entity mux4to1 is
   port (
      input : in  std_logic_vector(3 downto 0);
      sel   : in  std_logic_vector(1 downto 0);
      muxed : out std_logic);
end mux4to1;

architecture structural of mux4to1 is
   signal muxed_mid : std_logic_vector(1 downto 0);
   component mux2to1
      port (
         a     : in  std_logic_vector(1 downto 0);
         sel   : in  std_logic;
         muxed : out std_logic);
   end component;
begin

   mux2to1_1_0: mux2to1
      port map (
         a     => input(1 downto 0),
         sel   => sel(0),
         muxed => muxed_mid(0));
   mux2to1_3_2: mux2to1
      port map (
         a     => input(3 downto 2),
         sel   => sel(0),
         muxed => muxed_mid(1));
   mux2to1_final: mux2to1
      port map (
         a     => muxed_mid,
         sel   => sel(1),
         muxed => muxed);
end structure;
```

Figure 13-2 Structural Code

Rules

- Keep leaf-level (behavioral sub-blocks) coding separate from structural coding (blocks).
- Declarations, Instantiations, and Mappings. It is important to use a consistent, universal style for such things as entity declarations, component declarations, port mappings, functions, and procedures.

Declarations, Instantiations, and Mappings

It is important to use a consistent, universal style for such things as entity declarations, component declarations, port mappings, functions, and procedures.

Rules

- For declarations, instantiations, and mappings use one line for each signal. The exception is for relatively small components, functions, and procedures.
- Always use named association.

The combination of these two rules will help eliminate common coding mistakes. Therefore, this combination will greatly enhance the ease of debugging a design at every stage of verification. A simple example is shown Figure 13-3. Obeying these rules will also increase the readability, and therefore the reusability.

```
architecture structural of mux4to1 is

    . . .
begin

    mux2to1_1_0: mux2to1
        port map (
            a     => input(1 downto 0),
            sel   => sel(0),
            muxed => muxed_mid(0));
```

Figure 13-3 One Line Per Signal / Named Association

Comments

Liberal comments are mandatory to maintain reusable code. Although VHDL is sometimes considered to be self-documenting code, it requires liberal comments to clarify intent, as any VHDL user can verify.

Rules

Three primary levels of commenting:

- Comments should include a header template for each entity-architecture pair and for each package- and package-body pair. See the example in Figure 13-4. The purpose should include a brief description of the functionality of each lower block instantiated within it.
- Use comment headers for processes, functions, and procedures, as shown Figure 13-5. This should be a description of the purpose of that block of code.
- Use comments internal to processes, functions, and procedures to describe what a particular statement is accomplishing. While the other two levels of commenting should always be included, this level is left to the designer to decipher what is required to convey intent. Inline comments are shown in Figure 13-6.

```
-------------------------------------------------------------------------------------------------------------------------
-- Author: John Q. Smith                                    Copyright Xilinx, 2001
--                                                          Xilinx FPGA - VirtexII
-- Begin Date: 1/10/01
-- Revision History      Date           Author              Comments
--                       1/10/01        John Smith          Created
--                       1/14/01        John Smith          changed entity port address & data to addr & dat
-------------------------------------------------------------------------------------------------------------------------
--  Purpose:
-- This entity/architecture pair is a block level with 4 sub-blocks.  This is the processor control interface for the
-- block level <block_level_A>.  So on, and so forth…
-------------------------------------------------------------------------------------------------------------------------
```

Figure 13-4 Header Template

```
---------------------------------------------------------------------------------------
--  demux_proc:  this process dumultiplexes the inputs and registers the
-- demultiplexed signals
---------------------------------------------------------------------------------------
demux_proc : process(clk, reset)
begin …
```

Figure 13-5 Process, Function, and Procedure Header

```
-------------------------------------------------------------------------------------
--  demux_proc:  this process dumultiplexes the inputs and registers the
-- demultiplexed signals
-------------------------------------------------------------------------------------
demux_proc : process(clk, reset)
begin …
if reset = '1' then
   demux <= (others => '0');
elsif rising_edge(clk) then
  -- demultiplex input onto the signal demux
   case (sel) is
   when '0' =>
     demux(0) <= input;
   when '1' =>
     demux(1) <= input;
   when others =>
     demux <= (others => '0');
   end case;
end if;
end process;
```

Figure 13-6 Inline Comments

Indentation

Proper indentation ensures readability and reuse. Therefore, a consistent style is warranted. Many text editors are VHDL-aware, automatically indenting for "blocks" of code, providing consistent indentation. Emacs and CodeWright are two of the most common editors that have this capability. Figure 13-7 shows an example of proper indentation. Proper indentation greatly simplifies reading the code. If it is easier to read, it is less likely that there will be coding mistakes by the designer.

Rules

- Use a VHDL-aware text editor that provides a consistent indentation style.
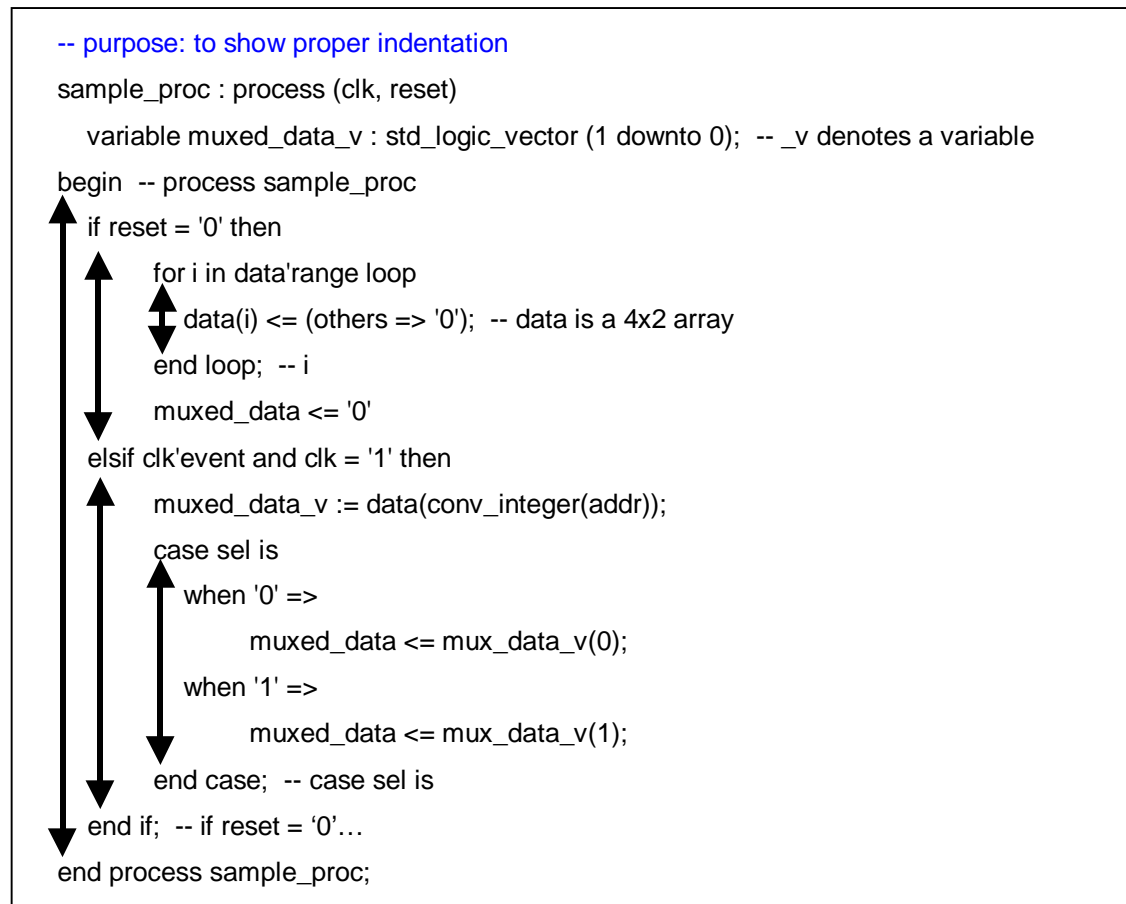
```
-- purpose: to show proper indentation
sample_proc : process (clk, reset)
    variable muxed_data_v : std_logic_vector (1 downto 0);  -- _v denotes a variable
begin  -- process sample_proc
    if reset = '0' then
        for i in data'range loop
            data(i) <= (others => '0');  -- data is a 4x2 array
        end loop;  -- i
        muxed_data <= '0'
    elsif clk'event and clk = '1' then
        muxed_data_v := data(conv_integer(addr));
        case sel is
            when '0' =>
                muxed_data <= mux_data_v(0);
            when '1' =>
                muxed_data <= mux_data_v(1);
        end case;  -- case sel is
    end if;  -- if reset = '0'…
end process sample_proc;
```

Figure 13-7 Proper Indentation

Naming Conventions

Naming conventions maintain a consistent style, which facilitates design reuse. If all designers use the same conventions, designer A can easily understand and use designer B's VHDL code.

Entities, Architectures, Procedures, and Functions

Rules

- Use all lowercase names with underscores, for readability and name delimiting.
- Each entity should have a unique name that describes that block.
- Architectures do not need unique names because they are individually bound to a specific entity that has a unique name. Names for architectures should be *rtl*, to indicate a leaf-level sub-block, and *structural,* to indicate a block with no leaf-level logic – with only sub-blocks.
    - For entities with more than one architecture, use *"rtl_xilinx"* (or *"structural_xilinx"*) for a Xilinx-specific architecture and *"rtl_asic"* (or *"structural_asic"*) for an ASIC-specific architecture.

Signal Naming Conventions

For a design implemented in VHDL, an up-front specification of signal naming conventions should help you reduce the amount of non-conformity. The primary motivating factor is enhanced readability during the verification of the design. General signal naming conventions are listed below.

General Signal Naming Guidelines

- Use addr for addresses. This might include sys_addr, up_addr, etc.
- Use clk for clock. This might include clk_div2 (clock divided by 2), clk_x2 (clk multiplied by 2), etc.
- Use reset or rst for synchronous reset.
- Use areset or arst for asynchronous reset.
- Use areset_l for active-low asynchronous reset.
- Use rw_l for read/write (write is active low).

Rules

The following rules specify the suggested nomenclature for other widely used signals
- Use <signal_name>_io for bi-directional signals.
- Use a _l suffix for active low signals <signal_name>_l.

- Do not use _in and _out suffixes for port signal names.
  - Use of *in* and *out* is very confusing in text, especially at hierarchical boundaries. Therefore, the use of _in and _out should be strictly monitored. If they must be used, be sure that _in indicates input, and, likewise, that _out is an output to the correct level of hierarchy. Figure 13-8 shows an example entity and the instantiation of that entity in a higher block. Here, *data_in* is connected to *data_out*, making the code confusing.

```
entity in_out is
port (    data_in : in std_logic_vector (31 downto 0);
         data_out : out std_logic_vector(31 downto 0));
end entity in_out;
...

in_out_inst: in_out
port map ( data_in => ram_data_out,
           data_out => ram_data_in);
```

Figure 13-8 Confusing _in and _out suffixes

- Use _i to denote local signal names that are *internal* representations of an output port. This nomenclature is used to easily identify the internal signal that will eventually be used as an output port.
  - The counter in Figure 13-9 provides a simple example of an output port that cannot be read. The output port *count* cannot be incremented because it would require *count* to be read. The problem is solved in the example by incrementing the local internal signal *count_i*. Some designers try to overcome this problem by using the port as an *inout*; however, not all synthesis compilers will allow this unless it is three-stated. Declaring the signal to be of type *buffer* is another common trap. This complicates the code because all signals to which it connects also must be of type *buffer*. Not all synthesis vendors support the data-type *buffer*. In addition, data-type *buffer* does not have all of the required defined functions to perform common arithmetic operations.

```
count <= count_i;
process (clk, reset)
begin
  if reset = '1' then
     count_i <= (others => '0');
  elsif rising_edge(clk) then
     count_i <= count_i + 1;
  end if;
end process;
```

Figure 13-9 Internal Signals Representing Output Ports

- Use _v to indicate a variable. Variables can be very useful if used correctly. The _v will serve as a reminder to the designer as to the intent and use of that signal.

- Use *<signal_name>_p0, <signal_name>_p1,* and so forth, to represent a pipelined version of the signal *<signal_name>* when *<signal_name>* comes after the pipelining. Use *<signal_name>_q0, <signal_name>_q1,* and so forth, to represent a pipelined version of the *<signal_name>* when *<signal_name>* comes before the pipeline. See Figure 13-19 in section 4 for an example of how to use this pipelined signal naming convention.
- Append a suffix to signals that use a clock enable and will be part of a clock-enabled path (i.e., multi-cycle path). For example, if the clock enable is enabled only one-quarter of clock cycles, the clock enable should be named to represent that -- ce4. Signals that use this enable might be named <signal_name>_ce4. This will greatly aid you in your ability to specify multi-cycle constraints.

Following some basic rules on the use of signals and variables can greatly reduce common coding problems.

Signals

The rules for using signals are not complex. The most common problem is that signals can be various data types. The problem in VHDL is "*casting*" from one data type to another. Unfortunately, no single function can automatically cast one signal type to another. Therefore, the use of a standard set of casting functions is important to maintain consistency between designers.

Casting

Rules for Casting

- Use std_logic_arith, std_logic_unsigned/std_logic_signed packages.

This provides the essential conversion functions:
- conv_integer(<signal_name>): converts std_logic_vector, unsigned, and signed data types into an integer data type.
- conv_unsigned(<signal_name>, <size>): converts a std_logic_vector, integer, unsigned (change size), or signed data types into an unsigned data type.
- conv_signed(<signal_name>, <size>): converts a std_logic_vector, integer, signed (change size), or unsigned data types into a signed data type.
- conv_std_logic_vector(<signal_name>, <size>): converts an integer, signed, or unsigned data type into a std_logic_vector data type.
- ext(<signal_name>, <size>): zero extends a std_logic_vector to size <size>.
- sxt(<signal_name>, <size>): sign extends a std_logic_vector to size <size>.

All conversion functions can take for the <signal_name> data-type a std_logic_vector, unsigned, signed, std_ulogic_vector, or integer. <size> is specified as an integer value.

Inverted Signals

To reduce complication and to make the code easier to debug and test, it is generally recommended that you use active-high signals in hardware description languages. Generally, active-low signals make the code more complicated than necessary. If active-low signals are required at the boundaries of an FPGA, invert incoming signals at the FPGA top structural level. Also, for outbound signals, invert them at the FPGA top structural level. Consider this a rule of thumb.

However, for FPGAs in general and Xilinx FPGAs specifically, inverters are *free* throughout the device. There are inverters in the IOB, and a LUT can draw in an inverter as part of its functionality, without a loss in performance.

Often, ASIC designers will use active-low signals in their code to use less power in the part. The synthesis tool will map the logic based on a vendor's libraries. Therefore, the synthesis tool will infer active-low signals internally when it sees fit. For that matter, writing code that uses active-low signals does not necessarily infer active-low signals in the ASIC. Again, the synthesis tool makes these decisions based on the vendor's libraries. Let the synthesis tool do its job!

Rule of thumb

- Use only active-high signals in HDL. One exception is a signal with a dual purpose, such as a read or a write signal. In this case, a naming convention should be used to reduce complication – rw_l is an easily recognizable signal name that clearly defines that signal's role.
- Where active-low signals are required, use of a _l as a suffix generally makes the intent clear. E.g., <signal_name>_l. Use of _n is generally confusing.

Rule for Signals

- There are a few basic rules to follow when you use signals. Remember that ports are just signals with special rules that apply to them.

Entity Port Rules within the Bound Architecture:

- You can read from inputs, but you cannot assign to inputs.
- You can assign to outputs, but you cannot read from outputs.
  - See section 1, Signal Naming Conventions , rule number four, for help in skirting this limitation.
- You can both assign to and read from inouts.

Internal Signal Rules

- Never assign to a signal in more than one process, with the exception of a three-state signal.
- For a combinatorial process (no registers inferred), never assign to a signal and read from the same signal in the same process. This will eliminate infinite loops when performing behavioral simulation.
  - This is not true for a "clocked" process; i.e., a process that is used to register signals. A clocked process would only need to have an asynchronous set or reset signal and a clock in its sensitivity list. Therefore, this process would not execute again until there was a change on one of those signals.
- In a *clocked process*, never assign to a signal outside of the control of the *if rising_edge(clk)* statement (or reset statement if an asynchronous reset exists). This is a common coding mistake. In synthesis, it will infer a combinatorial signal. In a behavioral simulation, it will have the behavior of a signal clocked on the falling edge.

Filling out a Process Sensitivity List

- Within a combinatorial process, all signals that are read (which can change) must be in the sensitivity list.
  - This will insure the correct behavioral simulation. This includes any signals that are compared in if-then-else statements and case statements. It also includes any signal on the right-hand side of an assignment operator. Remember that this is only for signals that can change. A constant cannot change; thus, it does not need to be in the sensitivity list.
- Within a clocked process, only an asynchronous set or reset and the clock should be in the sensitivity list.
  - If others are added, the functionality of a behavioral simulation will still be correct. However, the simulation will be slower because that process will need to be evaluated or simulated whenever a signal in its sensitivity list changes.

Rules for Variables and Variable Use

Variables are commonly not understood and are therefore not used. Variables are also commonly used and not understood. Variables can be very powerful when used correctly. This warrants an explanation of how to properly use variables.

Variables are used to carry combinatorial signals within a process. Variables are updated differently than signals in simulation and synthesis.

In simulation, variables are updated immediately, as soon as an assignment is made. This differs from signals. Signals are not updated until all processes that are scheduled to run in the current delta cycle have executed (generally referred to as suspending). Thus, a variable can be used to carry a combinatorial signal within both a clocked process and a combinatorial process. This is how synthesis tools treat variables – as intended combinatorial signals.

Figure 13-10 shows how to use a variable correctly. In this case, the variable *correct_v* maintains its combinatorial intent of a simple two-input and-gate that drives an input to an or-gate for both the *a* and *b* registers.
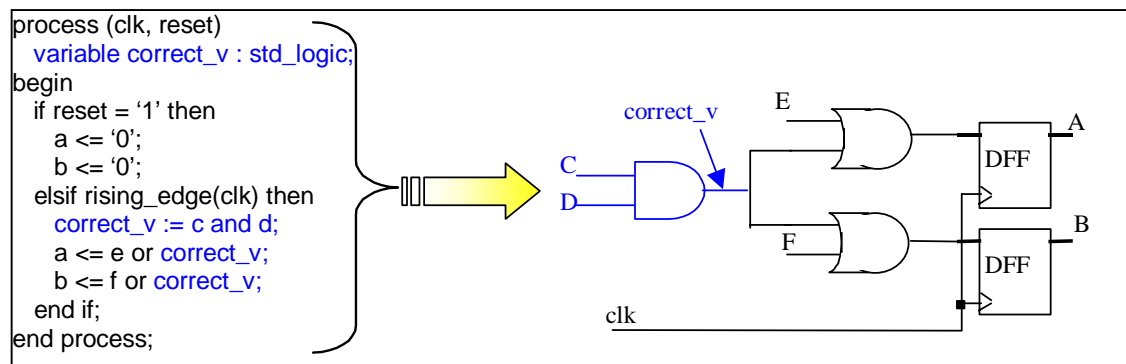


```
process (clk, reset)
  variable correct_v : std_logic;
begin
  if reset = '1' then
    a <= '0';
    b <= '0';
  elsif rising_edge(clk) then
    correct_v := c and d;
    a <= e or correct_v;
    b <= f or correct_v;
  end if;
end process;
```

Figure 13-10 Correct Use of Variables

In Figure 13-11, you read from the variable *incorrect_v* before you assign to it. Thus, *incorrect_v* uses its previous value, therefore inferring a register. Had this been a combinatorial process, a latch would have been inferred.
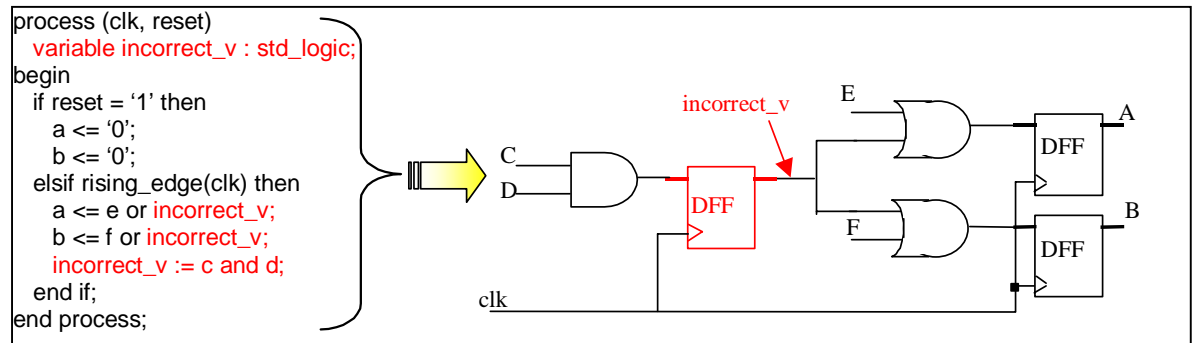

Figure 13-11 Incorrect Use of Variables

Rule for Variables

- Always make an assignment to a variable before it is read. Otherwise, variables will infer either latches (in combinatorial processes) or registers (in clocked processes) to maintain their previous value. The primary intent of a variable is for a combinatorial signal.

Packages are useful for creating modular and reusable code. There should be one or more packages used by a design team. These packages should include commonly used functions, procedures, types, subtypes, aliases, and constants. All team designers should familiarize themselves with the contents of these packages. If each designer were to create his or her own functions, procedures, types, subtypes, aliases, and constants, it could result in code that is difficult for other team members to use and read. Thus, when your team uses packages, it results in code that is more modular and more readable.

Package use can generally be broken down into the three types:

- The global package. This package is used on a company-wide basis, on each design. This package should include functions and procedures, such as reduction functions, for instance functions, and procedures that -- *and, or, and xor* (etc.) -- reduce individual buses. It should also include commonly used types and subtypes. This package should be created in a group setting by VHDL *experts* (or the most experienced in VHDL) who decide the best elements to have present in the package. This package should be used extensively and should have periodic reviews to determine what should be added to or taken away from the package. Because most divisions within a company work on the same type of projects, primarily, this package should contain the most widely and extensively used material that is common to all design teams.
- The project package. This package is used and created for a specific design project. The functions, procedures, types, subtypes, constants, and aliases are all specifically defined and created for the design at hand.
- The designer's packages. These packages are specific to a designer. Packages of this type should not be used extensively. If there is a need for something to be extensively used within the designer's package, it should be moved into the project package and possibly even the global package. Code readability and modularity is limited by the use of designer packages, as the type of function calls and types, etc. will not be readily understandable to all other designers in the group.

Package Contents

Constants

Used correctly, constants can ease the coding of complex and modular designs. Constants can be used in a variety of ways. They can be used to create ROMs, for modular coding, and to define what or how something should be used. For example, constants can be used in conjunction with generate statements to specify which portion of code to use (synthesize). Consider, for example, one portion of code written for an ASIC implementation and another portion written for a Xilinx implementation. The ASIC implementation should use gates to implement a multiplexer, while the Xilinx version should use three-state buffers to implement a multiplexer. Because some synthesis tools do not currently support configuration statements, a generate statement is the best solution.

Figure 13-12 shows an example of how constants can be used to define the logic created. Although this is a simple example, it illustrates the possibilities. By one change to the constant *ASIC*, an entirely different set of circuitry is synthesized throughout the design.

```
--within a package
constant asic : boolean := True;

. . .
-- within an architecture
generate_asic :
if asic = true then
mux_proc : process (addr, sel, data)

. . .
generate_fpga :
if asic = false then
tri_state_proc : process (addr, sel, data)

. . .
```

Figure 13-12 A Constant Guiding the Generation of Logic

Constants can aid modular coding. For example, you could define a constant that specifies the width of the address bus. One change to that constant in the package would make a modular change to everything in the design. See Figure 13-13. Using constants to define address and data-bus widths may be better than using generics. Generics are passed from the top-down, eliminating the possibility of synthesizing bottom-up. A bottom-up synthesis is generally preferable for decreased synthesis run-times because only the modules that change need to be resynthesized.

```
--within the package pack_ase_fpga
constant addrw : integer := 18;

use work.pack_ase_fpga.all;

entity profound is
port ( addr : in std_logic_vector (addrw-1 downto 0);

. . .
```

Figure 13-13 Address Width defined by a Constant

Rules for Defining Constants within a Package

• Define constants within a package when it can be used to improve the modularity of the code by guiding generate statements.
• Define constants in a package to define sizes and widths of buses. Constants used in this manner are generally more powerful than using generics because it allows the design to be synthesized in *any* manner, whereas generics allow only top-down synthesis.

Functions and Procedures

By definition, functions and procedures add modularity and reuse to code. Extensive use of functions and procedures from the global and project packages is encouraged. Rather than extensively using functions and procedures from a designer's package, the designer is encouraged to add the functions and procedures at a local level (within an architecture), to maintain readability for other designers and future reuse.

When defining functions and procedures, it is beneficial to use unsized vectors to pass signals. Using unsized vectors allows a modular use of the subprogram. In addition to using unsized vectors, use signal – range attributes to define the logic.

In the *function* example shown below in Figure 13-14, the input, named *vec,* is defined as a *std_logic_vector.* By not defining a sized vector, the actual size of the signal that is passed in will determine the implementation. The range attribute *'range* specifies the size of the intended logic. This function is modular; that is, it is not limited to being used for one specific vector size. A vector of any size can be passed into this function and correctly infer any amount of logic.

```
function parity (vec : input std_logic_vector) return std_logic is
   variable temp_parity : std_logic := '0';
begin
   for i in vec'range loop
      temp_parity := temp_parity xor vec(i);
   end loop;
   return temp_parity;
end function;
```

Figure 13-14 Modular Function Use

Rules for Functions and Procedures

- Extensive use of functions and procedures is encouraged. Predominately, the functions and procedures should be defined within either the global or the project packages.
- Create modular functions and procedures by not specifying the width of inputs and outputs. Then use range attributes to extract the needed information about the size of an object.

Types, Subtypes, and Aliases

Types and subtypes are encouraged for readability. Types defined at the global and project level are generally required, and they help to create reusable code.

Aliases can be used to clarify the intent, or meaning, of a signal. In most cases, the intent of a signal can be clearly identified by its name. Thus, aliases should not be used extensively. While aliases can help to clarify the purpose of a signal, they also add redirection, which may reduce the readability of the code.

Although aliases are not used in conjunction only with types and subtypes, it is useful for examples to be included here. In Figure 13-15 there are two types defined: a record and an array. For this example, aliases can be used to clarify the use of the signal *rx_packet.data* (*rx_data*) and the intent of the signal *data_addr(0)* (*data_when_addr0)*. In this example, the alias *data_when_addr0* is used in place of *data_array(0),* this provides more meaning to the "slice" of data than data_array(0) provides. Whenever the alias *data_when_addr0* is seen in the code, the intent is obvious. The use of the alias *rx_data* simply provides a shortened version of the signal *rx_packet.data* while its use and intent are maintained.

```
package alias_use is
type opcode is record
        parity : std_logic;
        address: std_logic_vector(7 downto 0);
        data : std_logic_vector(7 downto 0);
        stop_bits : std_logic_vector(2 downto 0);
end record;
type data_array_type is array (0 to 3) of std_logic_vector (31 downto 0);
end package;

architecture rtl of alias_use is
   signal addr : std_logic_vector (11 downto 0);
   signal data_array : data_array_type;
   alias data_when_addr0 : std_logic_vector(31 downto 0) is data_array(0);
signal rx_packet : opcode;
alias rx_parity is rx_packet.parity;
alias rx_addr is rx_packet.address;
alias rx_data is rx_packet.data;
alias rx_stop is rx_packet.stop_bits;

begin
   data_when_addr0 <= data when addr = x"000" else (others => '0');
   rx_data <= data_when_addr0;
. . .
```

Figure 13-15 Use of Types and Aliases

Rules for Types, Subtypes, and Alias use

- Types and subtypes are encouraged on a global or project basis to facilitate reusable code.
- Alias use is encouraged when it clearly promotes readability without adding complex redirection.

Technology-Specific Code (Xilinx) Section 4

It is desirable to maintain portable, reusable code. However, this is not always possible. There are cases for each technology vendor where instantiation of blocks is required. Furthermore, writing what is intended to be generic code will not always provide the best solution for a specific technology. The tradeoffs between instantiation versus technology-specific code are discussed below.

Instantiation

Although instantiation of Xilinx primitives is largely unneeded and unwanted, there are some specific cases where it must be done -- and other occasions when it should be done. While some of the components that need to be instantiated for a Xilinx implementation vary, those covered here are specific for Synplify, Synopsys, Exemplar, and XST. This section will describe situations where deviation from reusable code is required.

Required Instantiation

Specific top-level (FPGA) components require instantiation, including the boundary scan component, digital delay-locked loop components (DLL) or digital clock manager (DCM), startup block, and I/O pullups and pulldowns.

Inputs and outputs, other than LVTTL, can be specified in the synthesis tool. However, it is more advantageous to specify the I/O threshold level in the Xilinx Constraints Editor. This will write a constraint into the Xilinx UCF (User Constraint File), which is fed into the Xilinx implementation tools.

To instantiate Xilinx primitives, you will need to have a correct component declaration. This information can be inferred directly from the *Xilinx Libraries Guide*, found in the online documentation.

Rules for Required Instantiations for Xilinx

- Boundary Scan (BSCAN)
- Digital Clock Manager (DCM) or Delay-Locked Loop (DLL). Instantiating the DCM/DLL provides access to other elements of the DCM, as well as elimination of clock distribution delay. This includes phase shifting, 50-50 duty-cycle correction, multiplication of the clock, and division of the clock.
- IBUFG and BUFG. IBUFG is a dedicated clock buffer that drives the input of the DCM/DLL. BUFG is an internal global clock buffer that drives the internal FPGA clock and provides the feedback clock to the DCM/DLL.
- DDR registers. DDR registers are dedicated Double-Data Rate (DDR) I/O registers located in the input or output block of the FPGA.
- Startup. The startup block provides access to a Global Set or Reset line (GSR) and a Global Three-State line (GTS). The startup block is not inferred because routing a global set or reset line on the dedicated GSR resources is slower than using the abundant general routing resources.
- I/O pullups and pulldowns (pullup, pulldown).

Simulation of Instantiated Xilinx Primitives

Correct behavioral simulation will require certain simulation files. These can be found in the Xilinx directory structure: $Xilinx/vhdl/src/unisims. Note that unisims are similar to simprims, except that: unisims *do not* have component timing information enabled. Whereas, simprims have the timing information enabled but require an SDF file (from Xilinx place and route) to supply the timing information (post place and route timing simulation).

Within the unisim directory, several VHDL files need to be compiled to a **unisim** library. They can then be accessed by specifying the library unisim and using the use statement. For example:
- library unisim;
- use unisim.vcomponents.all;

The VHDL files must be compiled in a specific order because there are dependencies between the files. The compilation order is:
1. unisim_VCOMP.vhd
2. unisim_VPKG.vhd
3. unisim_VITAL.vhd

For post-place-and-route timing simulation, the simprim files need to be compiled into a **simprim** library. The VHDL files for simprims are in: $Xilinx/vhdl/src/simprims. The correct package compilation order is:
1. simprim_Vcomponents.vhd
2. simprim_Vpackage.vhd
3. simprim_VITAL.vhd

Simulation files rules

- Unisims are used for behavioral and post-synthesis simulation.
- Simprims are used for post place-and-route timing simulation.

Non-Generic Xilinx-Specific Code

This section is used to describe situations where Xilinx-specific coding may be required to get a better implementation than can be inferred from either generic code or ASIC-specific coding.

Three-State Multiplexers

Generic coding of multiplexers is likely to result in an and-or gate implementation. However, for Xilinx parts, gate implementation of multiplexers is generally not advantageous. Xilinx parts have a very fast implementation for multiplexers of 64:1 or less. For multiplexers greater than 64:1, the tradeoffs need to be considered. Multiplexers implemented with internal three-state buffers have a near consistent implementation speed for any size multiplexer.

Three-state multiplexers are implemented by assigning a value of "Z" to a signal. Synthesis further requires concurrent assignment statements. An example is shown in Figure 13-16. For this example, there is a default assignment made to the signal *data_tri* to 'Z'. The case statement infers the required multiplexing, and the concurrent assignment statements to the signal *data* infer internal three-state buffers. With those concurrent assignment statements, synthesis can only resolve the signal values is by using three-states. Without the concurrent assignment statements, synthesis would implement this in gates, despite the default assignment to "Z."

```
process (r1w0, addr_integer, data_regs1)
begin  -- process
   for i in 0 to 3 loop  -- three-state the signal
         data_tri(i) <= (others => 'Z');
   end loop;  -- i
   if r1w0 = '1' then
         case addr_integer is
            when 0 to 3 =>
                  data_tri(0) <= data_regs1(0);
            when 4 to 7 =>
                  data_tri(1) <= data_regs1(1);
            when 8 to 11 =>
                  data_tri(2) <= data_regs1(2);
            when 12 to 15 =>
                  data_tri(3) <= data_regs1(3);
         end case;
   end if;
end process;
-- concurrent assignments to data
data <= data_tri(0);
data <= data_tri(1);
data <= data_tri(2);
data <= data_tri(3);
```

Figure 13-16 Three-state Implementation of 4:1 Multiplexer

Rules for Synthesis Three-State Implementation

- Use a default assignment of "Z" to the three-state signal.
- Make concurrent assignments to the actual three-stated signal.

Memory

While memory can be inferred for Xilinx, it most likely cannot be inferred for the ASIC by using the same code. It is very likely that two separate implementations will be required. This section will describe the methodology used to infer Xilinx-specific memory resources. It is generally advantageous to instantiate the use of memory resources to make it easier to change for other technology implementations. While it is not always required, Xilinx's CORE Generator™ system program can generate RAM for instantiation. The CORE Generator™ system created memory must be used for dual-ported block RAMs, but it can also be used for creating other types of memory resources. The CORE Generator™ system does provide simulation files, but it is seen as a black box in synthesis; therefore, it will not provide timing information through that block.

RAM and ROM

The Xilinx LUT-RAM is implemented in the look-up tables (LUTs). Each slice has 32-bits of memory. A slice can have three basic single-port memory configurations: 16x1(2), 16x2, or 32x1. The Xilinx slices and CLBs can be cascaded for larger configurations.

LUT-RAM memory is characterized by synchronous write and asynchronous read operation. It also is not able to be reset; however, it can be loaded with initial values through a Xilinx user constraint file (UCF). Inference of Xilinx LUT-RAM resources is based on the same behavior described in the code shown in Figure 13-17. Dual-port LUT-RAM can also be inferred by adding a second read address. Dual-port RAM has similar functionality with a synchronous write port and two asynchronous read ports.

```
type ram_array is array (0 to 15) of std_logic_vector (5 downto 0);
signal ram_data : ram_array;

...
begin
process(clk)  --synchronous write
begin
if clk'event and clk = '1' then
   if we = '1' then
      ram_data(conv_integer(addr_sp)) <= data_to_ram;
   end if;
end if;
end process;


-------------------------------------------------------
-- for single port, use the same address as
-- is used for the write
-------------------------------------------------------
-- asynchronous read – dual port
ram_data_dp <=ram_data(conv_integer(addr_dp));
```

Figure 13-17 Xilinx LUT-RAM Inference

ROM inference is driven by constants. Example code for inferring LUT-ROM is shown in Figure 13-18.

```
type rom is arrary (0 to15) of std_logic_vector (3 downto 0);

-- 16x4 ROM in Xilinx LUT's
constant rom_data : rom := (x"F", x"A", x"7", x"0", x"1", x"5",
x"C", x"D", x"9", x"4", x"8", x"2", x"6", x"3", x"B", x"E");

...
begin
-- ROM read
data_from_rom <= rom_data(conv_integer(addr));

...
```

Figure 13-18 LUT-ROM Inference

Single-port block RAM inference is driven by a registered read address and a synchronous write. The example shown Figure 13-19 has this characterization. In the past, block RAM has been easily inferred, simply by having the registered address and synchronous write. Synthesis tools can only infer simple block RAMs. For example, you cannot infer a dual-port RAM with a configurable aspect ratio for the data ports. For these reasons, most dual-port block RAMs should be block-RAM primitive instantiations or created with the CORE Generator™ system.

```
type ram_array is array (0 to 127) of std_logic_vector (7 downto 0);
signal ram_data : ram_array;

...
begin
process(clk)  --synchronous write
begin
if clk'event and clk = '1' then
    addr_q0 <= addr;  -- registered address/pipelined address
    if we = '1' then
        ram_data(conv_integer(addr)) <= data_to_ram;
    end if;
end process;

data_from_ram <= ram_data(conv_integer(addr_q0));

...
```

Figure 13-19 Virtex Block RAM Inference

Rules for Memory Inference

- For single- or dual-port RAM implemented in LUTs, describe the behavior of a synchronous write and an asynchronous read operation.
- For ROM inference in LUTs, create an array of constants.
- Single-port block RAM is inferred by having a synchronous write and a registered read address (as shown in the example above, Figure 13-19).
    - For other configurations of the Xilinx block RAM, use the CORE Generator™ system.

CORE Generator System

The CORE Generator™ system may be used for creating many different types of ready-made functions. One limiting factor of the CORE Generator™ system is that synthesis tools cannot extract any timing information; it is seen as a black box.

The CORE Generator™ system provides three files for a module:
- Implementation file, <module_name>.ngc.
- Instantiation template, <module_name>.vho
- Simulation wrapper, <module_name>.vhd

For behavioral and post-synthesis simulation, the simulation wrapper file will have to be used. To simulate a CORE Generator™ module, the necessary simulation packages must be compiled. More information on using this flow and generating the necessary files can be found in the CORE Generator tool under Help →Online Documentation.

The CORE Generator™ system provides simulation models in the $Xilinx/vhdl/src/XilinxCoreLib directory. There is a strict order of analysis that must be followed, which can be found in the analyze_order file located in the specified directory. In addition, Xilinx provides a Perl script for a

fast and easy analysis of different simulators. To compile the XilinxCoreLib models with
ModelSim or VSS, use the following syntax at a command prompt:

➢ xilinxperl.exe $Xilinx/vhdl/bin/nt/compile_mti_vhdl.pl coregen
➢ xilinxperl.exe $Xilinx/vhdl/bin/nt/compile_vss_vhdl.pl coregen

Comparators

Compare logic is frequently implemented poorly in FPGAs. Compare logic is inferred by the use
of <, <=, >, and >= VHDL operators. For a Xilinx implementation, this logic is best implemented
when described with and-or implementations. When possible, look for patterns in the data or
address signals that can be used to implement a comparison with gates, rather than compare logic.
If a critical path includes comparison logic, an implementation that would use and-or logic should
be considered.

Rule for Comparator Implementation

- If a critical path has comparator logic in it, then try to implement the comparison by using
  and-or gates.

Xilinx Clock Enables

Clock enables are easily inferred, either explicitly or implicitly. Clock enables are very useful for
maintaining a synchronous design. They are highly preferable over the unwanted *gated clock*.
However, not all technologies support clock enables directly. For those architectures that do not
support clock enables as a direct input to the register, it will be implemented via a feedback path.
This type of implementation is not a highly regarded implementation style. Not only does it add a
feedback path to the register, it also uses more logic because FPGA architecture requires two extra
inputs into the LUT driving the register.

The Xilinx architecture supports clock enables as a direct input to a register. This is highly
advantageous for a Xilinx implementation. However, the designer must be certain that the logic
required to create the clock enable does not infer large amounts of logic, making it a critical path.

In the example shown below (Figure 13-20), there is an explicit inference of a clock enable and an implicit inference of clock enables. In the first section, a clock enable is via explicitly testing for a terminal count. In the second section of code, the clock enables are implied for the signals *cs* and *state*. The clock enable for *cs* is inferred by not making an assignment to *cs* in the state *init*. The clock enable for the signal *state* is inferred by not defining all possible branches for the if-then-else statement, highlighted in red. When the if-then-else condition is false, *state* must hold its current value. Clock enables are inferred for these conditions when they are in a *clocked process*. For a *combinatorial process*, it would infer latches.

```
process (clk)  -- Explicit inference of a clock enable
  begin  -- process
    if rising_edge(clk) then
       if tc  = '1' then
           cnt <= cnt + '1';
       end if;
    end if;
  end process;

process (clk, reset)  -- Implicit inference of a clock enable
  begin  -- process
   if reset = '1' then
      state <= (others => '0');
      cs <= "00";
   elsif rising_edge(clk) then
      case (state) is
          when init => -- inference of a clock enable for signal cs
              state <= load;
          when fetch =>
              if (a = '1' and b = '1') then  -- inference of a clock enable for signal state
                 state <= init;
              end if;
              cs <= "11";
          when others => null;
      end case;
   end if;
  end process;
```

Figure 13-20 Clock Enable Inference

Rules for Clock Enable Inference

- Clock enables can only be inferred in a *clocked process*.
- Clock enables can be inferred explicitly by testing an enable signal. If the enable is true, the signal is updated. If enable is false, that signal will hold its current value.
- Clock enables can be implicitly inferred two ways:
  - Not assigning to a signal in every branch of an if-then-else statement or case statement. Remember that latches will be inferred for this condition in a *combinatorial process* (see section 5, Inadvertent latch Inference).
  - Not defining all possible states or branches of an if-then-else or case statement.

Pipelining with SRL

In Xilinx FPGAs, there is an abundance of registers; there are two registers per slice. This is sufficient for most registered signals. However, there are times when multiple pipeline delays are required at the end of a path. When this is true, it is best to use the Xilinx SRL (Shift Register LUT). The SRL uses the LUT as a shiftable RAM to create the effect of a shift register. In Figure 13-21an example of how to infer the SRL is shown. This will infer a shift register with 16 shifts (width = 4). Although this will infer registers for an ASIC, it will infer the SRL when you are targeting a Xilinx part. The behavior that is required to infer the SRL is highlighted in blue. The size could be made parameterizable by using constants to define the signal widths (section 3, Constants). It could also be made into a procedure with parameterized widths and sizes.

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

entity srltest is
  port(clk, en : in  std_logic ;
       din : in std_logic_vector(3 downto 0);
       dout : out std_logic_vector(3 downto 0)) ;
end srltest ;

architecture rtl of srltest is
   type srl_16x4_array is array (15 downto 0) of std_logic_vector(3 downto 0);
   signal sreg  : srl_16x4_array ;
begin
 dout  <= sreg(15) ;  -- read from constant location
 srl_proc : process (clk, en)
 begin
    if rising_edge(clk) then
         if (en = '1') then
             sreg <= sreg(14 downto 0) & din ; -- shift the data
                                 -- Current Value sreg (15:1)  sreg(0)
                                 -- Next Value     sreg  (14:0)   din
         end if;
    end if;
```

Figure 13-21 Inference of Xilinx Shift Register LUT (SRL)

Rules for SRL Inference

- No reset functionality may be used directly to the registers.
    - *If a reset is required, the reset data must be supplied to the SRL until the pipeline is filled with reset data.*
- You may read from a constant location or from a dynamic address location. In Xilinx Virtex™-II parts, you may read from two different locations: a fixed location and a dynamically addressable location.

Technology-Specific Logic Generation – Generate Statements

This section has outlined ways that Xilinx-specific coding will differ from other solutions. Because many styles may exist for a similar block of code (for example a multiplexer), to get the optimal implementation, use VHDL generate statements. This is the best solution for a couple of reasons. Although configuration statements are commonly used to guide the synthesis of multiple implementation styles, some synthesis tools currently do not fully support them. Also, with generate statements, a change to a single constant will change the type of logic generated (ASIC or FPGA).

An example of using generate statements was covered in section 3, in the Figure 13-12.

Coding for Synthesis   Section 5

The main synthesis issues involve coding for minimum logic level implementation (i.e., coding for speed, max frequency); inadvertent logic inference; and fast, reliable, and reusable code.

Synchronous Design

The number one reason that a design does not work in a Xilinx FPGA is that the design uses asynchronous techniques.   To clarify, the primary concern is asynchronous techniques used to insert delays to align data, not crossing clock domains. Crossing clock domains is often unavoidable, and there are good techniques for accomplishing that task via FIFOs. There are no good techniques to implement an asynchronous design. First, and most important, the actual delay can vary based on the junction temperature. Second, for timing simulations, Xilinx provides only maximum delays. If a design works based on the maximum delays, this does not mean that it will work with actual delays. Third, Xilinx will stamp surplus –6 (faster) parts with a –5 or –4 (slower speed) speed-grade. However, if the design is done synchronously there will be no adverse effects.

Clocking

In a synchronous design, only one clock and one edge of the clock should be used. There are exceptions to this rule. For example, by utilizing the 50/50 duty-cycle correction of the DCM/DLL, in a Xilinx FPGA you may safely use both edges of the clock because the duty-cycle will not drift.

Do not generate internal clocks. Primarily, do not generate gated clocks because these clocks will glitch, propagating erroneous data. The other primary problems with internally generated clocks are clock-skew related problems. Internal clocks that are not placed on a global clock buffer will incur clock skew, making it unreliable. Replace these internally generated clocks with either a clock enable signal or generate divided, multiplied, phase shifted, etc. clocks with a clock generated via the DCM/DLL.

Rules for Clock Signals

- Use one clock signal and one edge.
- Do not generate internal clock signals because of glitching and clock-skew related problems.

Local Synchronous Sets and Resets

Local *synchronous* sets and resets eliminate the glitching associated with local *asynchronous* sets and resets. An example of such a problem is associated with the use of a binary counter that does not use the maximal binary count. For example, a four-bit binary counter has 16 possible binary counts. However, if the design calls only for 14 counts, the counter needs to be reset before it has reached its limit. An example of using local asynchronous resets is highlighted in red in Figure 13-22. A well-behaved circuit is highlighted in blue, in the Figure 13-23. For the binary counter that is using a local asynchronous reset, there will be glitching associated with the binary transitions, which will cause the local asynchronous reset to be generated. When this happens, the circuit will propagate erroneous data.

```
-- Asynchronous local reset and internally generated clock
process (clk, reset, cnt_reset)
begin  -- process
-- global and local async. reset
    if (reset = '1' or cnt_reset = '1') then
            tc <= '0';
            cnt <= "0000";
    elsif rising_edge(clk) then
            if cnt = "1110" then
                cnt_reset <= '1';
                tc <= '1';
            else
                cnt <= cnt + 1;
                tc <= '0';
                cnt_reset <= '0';
            end if;
    end if;
end process;
-- internally generated clock - tc
process (tc, reset)
begin  -- process
    if reset = '1' then
            data_en <= (others => '0');
    elsif rising_edge(tc) then
            data_en <= data;
    end if;
```

Figure 13-22 Local Asynchronous Reset and TC & Well-Behaved Synchronous Reset & CE

```vhdl
-- Synchronous Local reset and clock enable use
process (clk, reset)
   variable tc : std_logic := '0';
begin  -- process
   if reset = '1' then  -- global asynchronous reset
         cnt <= "0000";
         data_en <= (others => '0');
   elsif rising_edge(clk) then
     if cnt = "1110" then
           cnt <= "0000";   -- local synchronous reset
           data_en <= data;  -- terminal count clock enable
     else
           cnt <= cnt + '1';
           tc := '0';
     end if;
   end if;
end process;
```

Figure 13-23 Local Asynchronous Reset and TC & Well-Behaved Synchronous Reset & CE

Rule for Local Set or Reset Signals

- A local reset or set signal should use a synchronous implementation.

Pipelining

Pipelining is the act of inserting registers into one path to align that data with the data in another path, such that both paths have an equal amount of latency. Pipelining may also decrease the amount of combinatorial delay between registers, thus increasing the maximum clock frequency. Pipelines are often inserted at the end of a path by using a shift register implementation. Shift registers in Xilinx's Virtex™ parts are best implemented in the LUT as an SRL, as described in section 4. Signal naming for pipelined signals is covered in section 1.

Registering Leaf-Level Outputs and Top-Level Inputs

A very robust technique, used in synchronous design, is registering outputs of leaf-levels (sub-blocks). This has several advantages:

- No optimization is needed across hierarchical boundaries.
- Enables the ability to preserve the hierarchy.
- Bottom-up compilation.
- Recompile only those levels that have changed.
- Enables hierarchical floorplanning.
- Increases the capability of a guided implementation.
- Forces the designer to keep like-logic together.

Similarly, registering the top-level inputs decreases the input to clock (ti2c) delays; therefore, it increases the chip-to-chip frequency.

Rules for the Hierarchical Registering of Signals

- Register outputs of leaf-level blocks.
- Register the inputs to the chip's top-level.

Clock Enables

The use of clock enables increases the routability of a Xilinx implementation and maintains synchronous design. The use of clock enables is the correct alternative to gated clocks.

Clock enables increase the routability of the design because the registers with clock enables will run at a reduced clock frequency. If the clock enable is one-half the clock rate, the clock enabled datapaths are placed-and-routed once the full clock frequency paths have been placed-and-routed. The clock enable should have a timing constraint placed on it so that the Xilinx implementation tools will recognize the difference between the normal clock frequency and the clock-enabled frequency. This will place a lower priority on routing the clock-enabled paths.

Gated clocks will introduce glitching in a design, causing incorrect data to be propagated in the data stream. Therefore, gated clocks should be avoided.

Using signals generated by sequential logic as clocks is a common error. For example, you use a counter to count through a specific number of clock cycles, producing a registered terminal count. The terminal count is then used as a clock to register data. This internal clock is routed on the general interconnect. The skew on internally generated clocks can be so detrimental that it causes errors. This may also cause race conditions if the data is resynchronized with the system clock. This error is illustrated in Figure 13-. The text highlighted in red is the implementation using the terminal count as a clock.

Instead, generate the terminal count one count previous, and use the terminal count as a clock enable for the data register. The text highlighted in blue is the well-behaved implementation using the terminal count as a clock enable. An explanation of the reset signals is covered in the next section - 5.

It may be useful to generate clock enables by using a state machine. The state machine can be encoded at run time by the synthesis tool. Thus a one-hot, gray, or Johnson encoding style could be used. It is also possible to produce precisely placed clock enables by using a linear feedback shift register (LFSR), also known as a pseudo-random bitstream generator (PRBS generator). Xilinx provides application notes on the use of LFSRs.

Clock enables for Xilinx implementations are further discussed in section 4.

Rules for Clock Enable

- Use clock enables in place of gated clocks.
- Use clock enables in place of internally generated clocks.

Finite State Machines

Coding for Finite State Machines (FSM) involves includes analyzing several tradeoffs.

Encoding Style

Enumerated types in VHDL allow the FSM to be encoded by the synthesis tool. However, the encoding style used will not be clearly defined in the code but rather in the synthesis tool. Therefore, good documentation should be provided -- stating specifically which encoding style was used. By default, most synthesis tools will use binary encoding for state machines with less than five states: one-hot for 5 to 24 states and gray for more than 24 states (or similar). Otherwise, synthesis will use one-hot encoding. One-hot encoding is the suggested implementation for Xilinx FPGAs because Xilinx FPGAs have abundant registers. Other encoding styles may also be used -- specifically gray encoding. For a gray-encoding style, only one-bit transitions on any given state transition (in most cases); therefore, less registers are used than for a one-hot implementation, and glitching is minimized. The tradeoffs for these encoding styles can easily be analyzed by changing a synthesis FSM attribute and running it through synthesis to get an estimate of the timing. The timing shown in synthesis will most likely not match the actual implemented timing; however, the timing shown between the different encoding styles will be relative, therefore providing the designer a good estimate of which encoding style to use.

Another possibility is to specifically encode the state machine. This is easily done via the use of constants. The code will clearly document the encoding style used. In general, one-hot is the suggested method of encoding for FPGAs -- specifically for Xilinx. A one-hot encoding style uses more registers, but the decoding for each state (and the outputs) is minimized, increasing performance. Other possibilities include gray, Johnson (ring-counter), user-encoded, and binary. Again, the tradeoffs can easily be analyzed by changing the encoding style and synthesizing the code.

Regardless of the encoding style used, the designer should analyze illegal states. Specifically, are all the possible states used? Often, state machines do not use all the possible states. Therefore, the designer should consider what occurs when an illegal state is encountered. Certainly, a one-hot implementation does not cover all possible states. For a one-hot implementation, many illegal states exist. Thus, if the synthesis tool must decode these states, it may become much slower.   The code can also specifically report what will happen when an illegal state is encountered by using a "when others" VHDL case statement. Under the "when others" statement, the state and all outputs should be assigned to a specific value. Generally, the best solution is to return to the reset state. The designer could also choose to ignore illegal states by encoding "don't care" values ('X') and allow the synthesis tool to optimize the logic for illegal states. This will result in a fast state machine, but illegal states will not be covered.

Rules for Encoding FSMs

- For enumerated-types, encode the state machine with synthesis-specific attributes. Decide if the logic should check for illegal states.
- For user-encoded state machines, the designer should analyze whether the logic should check for illegal states or not, and the designer should accordingly write the "when others" statement. If the designer is concerned with illegal states, the state machine should revert to the reset state. If the designer is not concerned with illegal states, the outputs and state should be assigned "X" in the "when others" statement.
- Xilinx suggests using one-hot encoding for most state machines. If the state machine is large, the designer should consider using a gray or Johnson encoding style and accordingly analyze the tradeoffs.

FSM VHDL Processes

Most synthesis tools suggest coding state machines with three process statements: one for the next state decoding, one for the output decoding, and one for registering of outputs and state bits. This is not as concise as using one process statement to implement the entire state machine; however, it

---

allows the synthesis tools the ability to better optimize the logic for both the outputs and the next-state decoding. Another style is to use two processes to implement the state machine: one for next state *and* output decoding and the other process for registering of outputs and state bits.

The decision to use one, two, or three process statements is entirely left up to the discretion of the designer. Moore state machines (output is dependent only on the current state) generally have limited decoding for the outputs, and the state machine can, therefore, be safely coded with either one or two process statements. Mealy state machine (outputs depend on the inputs and the current state) output decoding is generally more complex, and, therefore, the designer should use three processes. Mealy state machines are also the preferred style for FSMs because it is advantageous to register the outputs of a sub-block (as described above in section 5). Mealy state machines will have the least amount of latency with registered outputs. Mealy state machines can be used with a look-ahead scheme. Based on the current state and the inputs, the outputs can be decoded for the next state. For simple state machines where the output is not dependent on the inputs, a Moore implementation is equivalent to a look-ahead scheme. That is, the outputs can be decoded for the next state and appropriately registered to reflect the next state (rather than reflecting the current state). The purpose of this scheme is to introduce the least amount of latency when registering the outputs.

Rules for FSM Style

- Generally, use three process statements for a state machine: one process for next-state decoding, one for output decoding, and one for the registering of state bits and outputs.
- Use a Mealy look-ahead state machine with registered outputs whenever possible, or use a Moore state machine with next-state output decoding and registered outputs to incur the minimum amount of latency.

Logic Level Reduction

To minimize the number of cascaded logic levels, we need to follow a few simple rules of coding.

If-Then-Else and Case Statements

If-then-else and case statements can cause unwanted effects in a design. Specifically, nested If-then-else and case statements may cause extra levels of logic inference. This occurs because if-then-else statements generally infer priority-encoded logic. However, one level of an if-then-else will not necessarily create priority-encoded logic. For that matter, synthesis tools generally handle if-then-else or case statements very well and create parallel logic rather than priority encoded logic.

Often, a nested if statement can be combined in the original if statement and result in a reduced amount of inferred logic. A simple example is shown in Figure 13-24, which shows how priority encoded logic creates cascaded logic. Nested case statements can have the same effect, as can the combination of nested case and if-then-else statements.
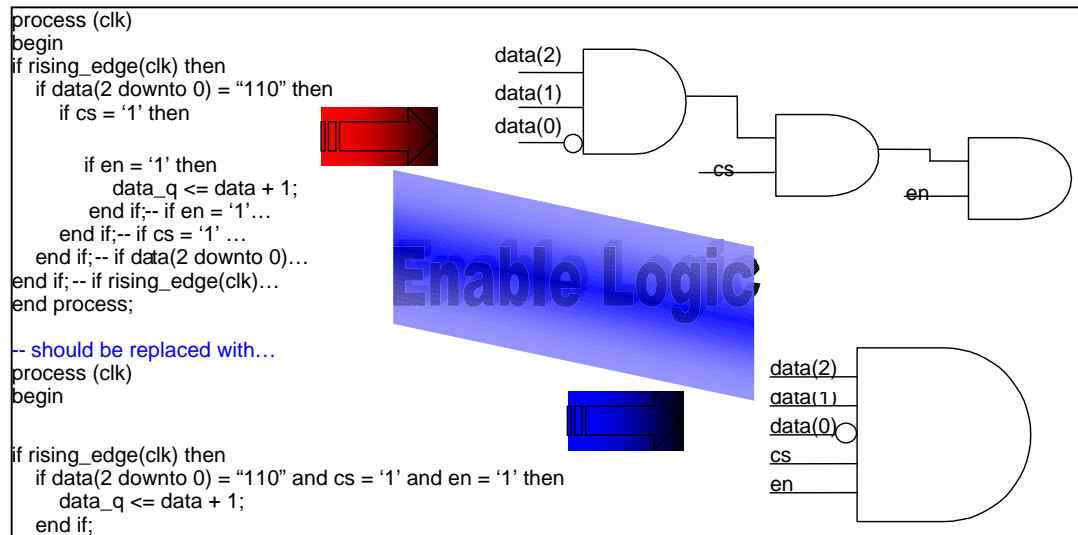

Figure 13-24 Priority Encoded Logic

Priority-encoded logic can be generated for other reasons. The use of overlapping conditions in if-then-else branches causes the generation of priority-encoded logic. This condition should be avoided. There are times that priority-encoded logic must be used and may be intended. If the selector expressions in the if-then-else statement branches are not related, then priority-encoded logic will be created. Although this may be the intent, its use should be cautioned.

Rules for If-Then-Else and Case Statements

- Limit the use of nested if-then-else and case statements.
- Avoid overlapping conditions in if-then-else statements – this condition will infer priority-encoded logic.
- Avoid using mutually exclusive branch expressions if possible. This condition will always infer priority-encoded logic.
    - *Instead, use mutually exclusive if-then statements for each expression (if possible).*

For Loops

Similar to the use of if-then-else and case statements, "for loops" can create priority-encoded logic. While for loops can be a very powerful tool for creating logic, the designer should evaluate their effects.

A simple example of the adverse effect of for loops is shown in Figure 13-25. Fortunately, this is a situation that most tools handle well, but in our goal of creating reusable (portable) code, this situation should be avoided.
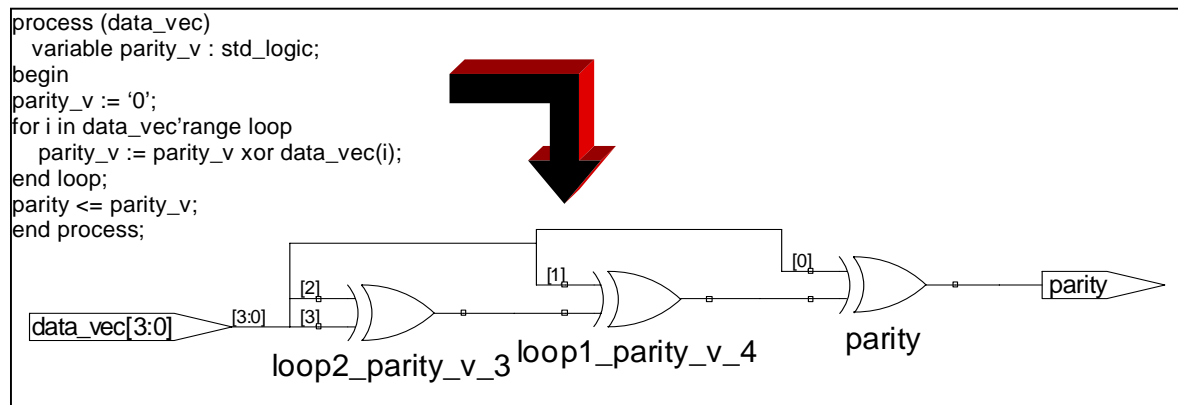
```
process (data_vec)
  variable parity_v : std_logic;
begin
parity_v := '0';
for i in data_vec'range loop
    parity_v := parity_v xor data_vec(i);
end loop;
parity <= parity_v;
end process;
```


Figure 13-25 For-Loop Cascaded Logic Implementation

Rule for For Loops

- Be cautious of using for loops for creating logic. Evaluate the logic created by the synthesis tool. There will likely be another way to write the code to implement the same functionality with the logic implemented in parallel.

Inadvertent Latch Inference

Inadvertent latch inference is a common problem that is easily avoided. Latches are inferred for two primary reasons: one, not covering all possible branches in if-then-else statements and two, not assigning to each signal in each branch. This is only a problem in a *combinatorial process*. For a *clocked process*, registers with clock enables are synthesized (as covered in the Xilinx-Specific Coding section, section 4).

A latch inference example for each of these cases is shown in Figure 13-26. For section one, each possible state was not covered. This is a very common mistake for one-hot encoded state machines. Section one is an example of Moore FSM output decoding. The latch inference would be eliminated by the use of a final "else'" statement with an assignment in that branch to the signal "cs."
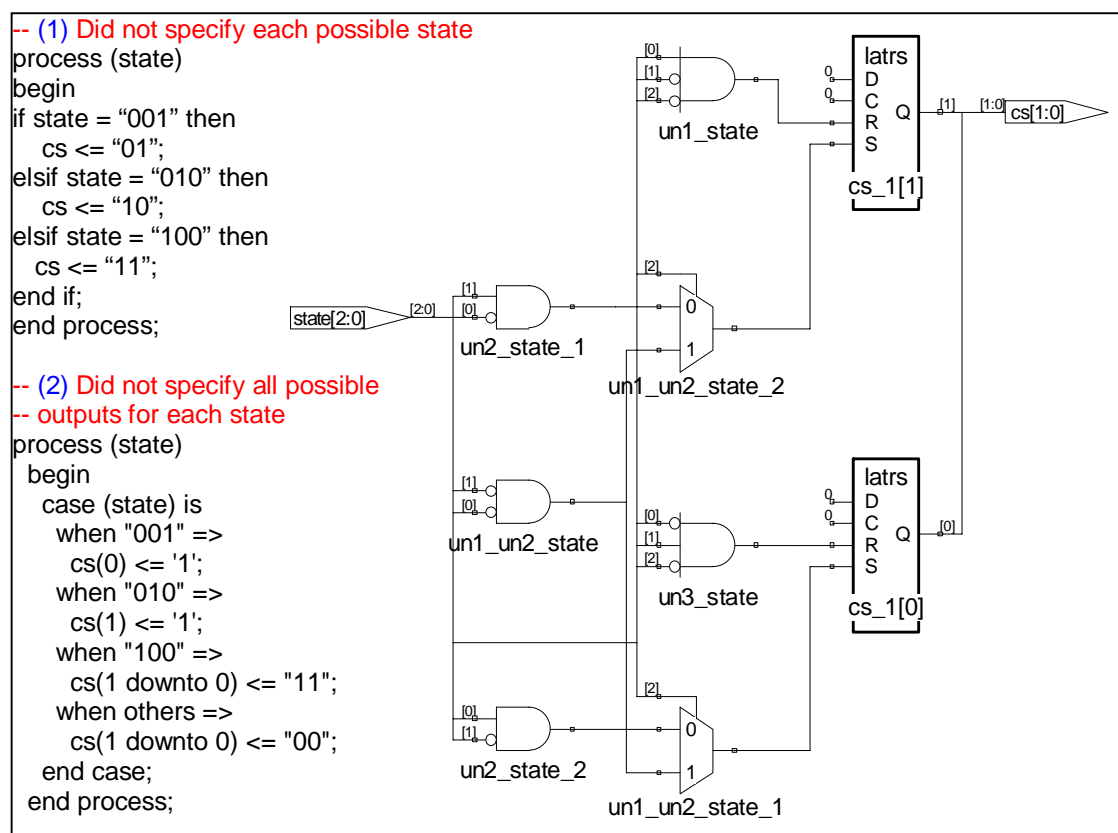

Figure 13-26 Latch Inference

For the second section of code, the latches are inferred because each signal is not assign in each state.

In Figure 13-27, the inference of latches is eliminated by covering all possible branches and assignging to each signal in each branch. The fixes are highlighted in blue. For the case implementation, the default assignment to cs before the case statement specifies a default assignment for each state. This way, each bit is changed depending on the state. This si equivalent to making a signal assignment in each branch. For the if-then-else statement, adding the else clause solves the problem.

```
-- (1) Fixed Case implementation        (2) Fixed if-then-else implementation
process (state)                          process (state)
 begin                                    begin
  cs <= "00";                               if state = "001" then
  case (state) is                             cs <= "01";
   when "001" =>                             elsif state = "010" then
    cs(0) <= '1';                          cs <= "10";
   when "010" =>                             elsif state = "100" then
    cs(1) <= '1';                          cs <= "11";
   when "100" =>                             else
    cs <= "11";                           cs <= "00";
   when others =>                             end if;
    cs <= "00";                          end process;
  end case;
 end process;
```

Figure 13-27 Elimination of Inadvertent Latch Inference

Rules for Avoidance of Latch Inference

- Cover all possible branches.
- Assign to all signals at all branches.