Coding Tips and Techniques for Synthesizeable, Reusable VHDL

Subbu Meiyappan
Ken Jaramillo
Peter Chambers

VLSI Technology, Inc.

© VLSI Technology, Inc, 1998-1999

Table Of Contents

Table Of Contents....2
Introduction....3
The Design Reuse Challenge....4
Hardware Description Languages....5
Features of VHDL which Promote Reusability....6
Reuse Tip #1: Generics....7
    The Effect of Generics on Synthesis....9
    Generics: Summary....10
Reuse Tip #2: Package of Constants....11
    Deferred Constants....14
    The Effect of a Package of Constants on Synthesis....15
    Package Of Constants: Summary....15
Reuse Tip #3: Generate Statements....16
    Generate Statements : Summary....18
Reuse Tip #4: Tying off Ports....19
Reuse Tip #5: Unconstrained Arrays....21
Reuse Tip #6: Configuration Specifications....25
Reuse Tip #7: Use of a Preprocessor for Enhancing VHDL Features....27
Reuse Tip #8: VHDL Attributes that aid Reuse and are Synthesizable....28
Reuse Tip #9: Use of Block Statements for Design Reuse....28
Reuse Tip #10: Leaving Unused Ports 'open'....29
References....29

Introduction

Design Reuse is the process of migrating or leveraging tested, high quality blocks of logic from one ASIC design to another. With the tremendous advances in semiconductor technology, it is becoming increasingly difficult to bridge the gap between what technology offers and what productivity allows. An internal source illustrates that by the year 2001, VLSI Technology will be able to produce 64 million transistors on a chip using 0.18 micron technology, which is approximately 16 million gates. Designing full-custom ASICs to occupy as much silicon area as possible is increasingly challenging. To achieve the highest level of silicon efficiency, designing semi-custom ASICs with highly reusable design entities has become the order of the day. Design reuse, the use of pre-designed and pre-verified design blocks, is the most promising opportunity to bridge the gap between available gate-count and designer productivity.

The design reuse movement is gaining momentum throughout the industry. One example of this is shown by groups such as Virtual Socket Interface Alliance (VSIA) who are working towards setting standards for tool interfaces and design practices for effective design reuse.

The Design Reuse Challenge

Design for reuse poses new and innovative challenges to a designer. Before being reusable, a design must be usable, which necessitates the use of good design practices. In order for a design to be reusable, the design must be:

- designed with the mindset of solving a general problem
- well coded, commented, and documented
- verified to a high level of confidence
- technology independent
- synthesis tool independent
- simulator independent
- application independent

Owing to the mounting pressures in schedules, some or all of these guidelines are usually bypassed rendering a design virtually non-reusable. But, if these guidelines are adhered to, they speed the design, verification, and debug processes of a project by reducing iterations throughout the coding and verification loop. Thus, the efficient use and reuse of designs plays a vital role in the creation of large ASICs with aggressive time-to-market schedules.

Hardware Description Languages

The days of schematic entry for huge designs are long gone and the design community now understands the need for Hardware Description Languages like VHDL and Verilog. Although HDLs have been used for some time, most designs today do not use the built in “design-reuse” features of the languages themselves. In other words, the purpose of the HDLs has not been thoroughly understood, and their features have been misused or underused. On an average, only 20% of the designs in the industry are considered reusable to any extent. With an increasing need for design reuse, the emphasis on coding techniques for design reuse is on the rise. VLSI fully understands the need for design reuse and has a plethora of reusable IP in the form of Functional System Blocks (FSB™) delivered through HDLi™, a design delivery mechanism proprietary to VLSI. This paper attempts to focus on developing reusable designs using the native language features of VHDL. Because we are attempting to reuse designs for silicon efficiency, this paper concentrates on design reuse techniques pertaining to synthesizeable VHDL.

While reading this paper the reader is assumed to have a basic understanding of digital design, VHDL, and synthesis. Further, unless otherwise explicitly stated, all of the examples in this paper conform to the VHDL1076-1987 standard published by the IEEE.

Features of VHDL which Promote Reusability

VHDL has been in use in the design community for over a decade. One of the primary intents of developing designs in VHDL was reusability, although this has not been the employed effectively until recently. VHDL is a feature-rich language that can be exploited well for reuse techniques, as is discussed in the following sections. The features of VHDL that facilitate reuse are:

- Generics
- Packages of constants
- Generate statements
- Unconstrained array aggregates
- VHDL attributes
- Block statements for inline design partitioning
- Looping Constructs
- Record data types for signal aliasing
- Configuration specifications
- Tying ports off to known constants
- Leaving unused output ports open and unconnected
- The “others” clause, and array aggregates

Reuse Tip #1: Generics

Generics have been used for writing parameterized models since the early days of VHDL. Generics can be used to write models of varying structure and behavior[2]. Here are some examples:

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


entity up_counter_en is
   generic (
            BIT_WIDTH   : INTEGER := 2;     -- Structure
            COUNT_ENABLE: INTEGER := 1;  -- Structure
            DOWN_COUNT  : INTEGER := 0; -- Behavior
            OutDelay    : TIME    := 3 ns   -- Behavior, Non-
                                            -- synthesizable
        );
   port (
         clk     : in  std_logic;
         reset_n : in  std_logic;
         en      : in  std_logic;
         count   : out std_logic_vector(BIT_WIDTH-1 downto 0)
         );
end up_counter_en;

architecture behav of up_counter_en is

signal count_s : std_logic_vector(BIT_WIDTH-1 downto 0);

begin
   process (clk, reset_n)
   begin
      if(reset_n = '0') then
         count_s <= (others => '0');
      elsif (clk'event and clk = '1') then
         if (COUNT_ENABLE = 1) then  -- removed if generic is false
            if (en = '1') then
               if (DOWN_COUNT = 0) then
                  count_s <= count_s + 1;
               else
                  count_s <= count_s - 1;
               end if;
            end if;
         else
            if (DOWN_COUNT = 0) then
               count_s <= count_s + 1;
            else
               count_s <= count_s - 1;
            end if;
         end if;
      end if;
   end process;

   count <= count_s after OutDelay;

end behav;
```

The example shown above illustrates the use of generics for modifying structure and behavior using the language features for simulation and synthesis. Selective features can be enabled and disabled by turning generics on and off. For example, if the `COUNT_ENABLE` generic is set to FALSE in line 4, then none of the logic described in lines 23-27 is ever elaborated or synthesized, but the parent design still has the ability to have a count enable. Using different values for `OutDelay` and `DOWN_COUNT` changes the behavior (although the `OutDelay` is ignored during synthesis) and changing `BIT_WIDTH` and/or `COUNT_ENABLE` modifies the structure of the design. Creating designs with generics enables the reuse of the design in various circumstances where different structure or behavior is needed. For example, a design may require two counters: one that counts up to 1024 and the other that counts up to 8. Designing two separate counters, one that is 10-bits wide and the other 3-bits wide, respectively has the following drawbacks:

- Unnecessary investment in time taken to design the counters
- Waste of resources
- Requires more verification time and more synthesis time

On the other hand, if a counter were designed with reuse in mind using the generic approach, a great deal of effort in terms of design, synthesis, and verification would have been averted. Although, the use of a counter as an example may not substantiate the use of generics, it is used as an illustrative example[1]. From the author's design experiences, the use of generics for parameterizing the structure and behavior is essential for design reuse applications. The following examples illustrate the instantiation of the counter shown above in an application that requires a 10-bit up counter and a 3-bit down counter

```
TenBit: counter
   generic map (
                 BIT_WIDTH    => 10,
                 COUNT_ENABLE => true,
                 DOWN_COUNT   => false
                )
   port map (
             clk     => my_clk,
             reset_n => my_reset_n,
             en      => TenBit_en,
             count   => TenBit_cnt
            );

ThreeBit: counter
   generic map (
                 3,
                 false,
                 true,
                 4 ns
                )

port map (
             my_clk,
             my_reset_n,
```

[1] Throughout this paper, the reader may find simplistic examples for various methods of parameterization. These examples should serve as a guideline for reusability and extending such examples for larger applications is left to the reuse-creativity of the reader.

```
                gnd,
                ThreeBit_cnt
               );
```

The example above illustrates the following points:

- Lines 1-12 instantiates the counter to be used as a 10-bit up counter with the count-enable logic turned on.
- The `TenBit` counter instantiation uses named association for its generics and ports.
- The generics whose values are not mapped in the instantiation assume default values.
- Lines 14-26 instantiates the same counter to be used as a 3-bit down counter with count-enable logic turned off.
- The `ThreeBit` counter instantiation uses positional association for its generics and ports. In general however, the authors strongly discourage the used of positional association, because changing a parameter or a port in the reusable design requires the modification in all of the instances.
- The use of generics can be of great help in terms of resources and time when multiple instances of the same design is required.

The Effect of Generics on Synthesis

The use of generics to parameterize designs not only helps create reusable design blocks, but also helps in optimizing unnecessary logic or modifying useful logic when it comes to synthesis. Some synthesis tools help create macros and templates when designs are parameterizable through generics, that can be used as a library element in subsequent designs for simulations and/or synthesis. Parameterizing bus/register widths through generics is a trivial case of the use of generics and is not dealt with in this article. The use of generics for optimizing logic and its effect on synthesis is discussed here. Consider the example of the counter shown above with the following values for the generics:

```
BIT_WIDTH    => 8
COUNT_ENABLE => true
DOWN_COUNT   => true
OutDelay     => 3 ns
```

When this design is elaborated during synthesis, the generic for `OutDelay` is ignored because the synthesis tools cannot handle time-delay elements in creating logic. The synthesis tool creates an 8-bit down-counter with the count_enable logic, as shown below.

VLSI Technology

Consider another case of the same counter with the following generics:

```
BIT_WIDTH    => 8
COUNT_ENABLE => false
DOWN_COUNT   => false
```

This creates an 8 bit up-counter without the `count_enable` logic. In cases where the gate count is an important factor, unwanted logic can be very efficiently optimized using this methodology. Thus the structure (changing the `BIT_WIDTH`) and/or the behavior (up-counter or down-counter, `count_enable` disabled or enabled) can be modified during design, synthesis, and simulation using this elegant approach to parameterization.

Generics: Summary

As we've seen from the sections above, generics can be used to add reusability to a design in a very readable manner. They are excellent for specifying widths of counters, buses, shift registers, etc. But they can be used for much more. As in the example, generics can be used to turn various features on and off. This allows one to use only the features which apply to the current project. Some more examples of the types of features that generics can be used to specify are as follows:

- FIFO depths
- Choosing between various bus interfaces (PCI, Arm System Bus, etc.)
- Choosing a specific architecture of a particular design (up/down counter, counter with enable, flip flop based register vs. latched based register, whether a register is read, write, or both, fixed arbitration scheme or round robin arbitration scheme for a bus arbiter, ripple carry adder vs. carry look ahead adder, etc.)
- Address of a register
- Power Up Value for a register
- Which bits in a register are supported and which are reserved
- Clock divide ratio for a clock divider circuit

This is just a small list of the types of things you might make generic in your design. But you can see that by making the design somewhat generic, it becomes easier for others to reuse.

One drawback to the approach of using generics can be seen when the generics are used in a hierarchy. In order for the lowest-level blocks to receive the generics, they must pass down through the hierarchy. This may involve passing generics through the blocks that do not use the value of the generics. Another drawback to using generics is that as the list of generics grows it becomes more cumbersome to carry them around at each point in the hierarchy. And still another drawback to using generics is that current synthesis tools tend to have limited support generics (for example, requiring all generics to be integer type). An efficient way to avoid these problems is the use of a package of constants.

Reuse Tip #2: Package of Constants

A VHDL package is a simple way of grouping a collection of related declarations that serve a common purpose. The package could be made visible to the appropriate blocks in the design by the use of library statements. This approach has added benefits:

- If a new parameter is to be added or changed, there is only one central package file that will be modified.
- Some synthesis tools do not allow the use of Boolean, string, or array types for generics. In such cases, a constant in a package could be used. Synthesis tools tend to allow most constant types.
- Packages can use TYPE statements for enumerated data types
- When translating from VHDL to Verilog, constants map easily as parameters.
- A package used for design can also be used for simulation for “design-aware” test-benches.

As an example for the use of a package of constants, we will change the previous counter example to use such a package. For illustrative purposes, we will assume that the package resides in a VHDL library, referred to as pkgs.

```
-- parameters package par_pkg.vhd

package par_pkg is
  constant BIT_WIDTH    : integer := 10;
  subtype CNT_WIDTH is integer range BIT_WIDTH-1 downto 0;
  constant COUNT_ENABLE: BOOLEAN := true;  -- Structure
  constant DOWN_COUNT  : BOOLEAN := false; -- Behavior
  constant OutDelay    : TIME    := 3 ns   -- Behavior
end par_pkg;

 -- reusable counter design counter.vhd
library pkgs;
use pkgs.par_pkg.all;

entity counter is
port (
  clk     : in  std_logic
  reset_n : in  std_logic;
  en      : in  std_logic;
  count   : out std_logic_vector(CNT_WIDTH)
)
end counter;

architecture behav of counter is
signal count_s : std_logic_vector(CNT_WIDTH); -- Same as count
begin
process (clk, reset_n)
begin
  if(reset_n = '0') then
    count_s <= (others => '0');
  elsif (clk'event and clk = '1') then
    if (COUNT_ENABLE) then  -- removed if generic is false
      if (en = '1') then
        if (not DOWN_COUNT) then
          count_s <= count_s + 1;
        else
          count_s <= count_s - 1;
        end if;
      end if;
    else
      if (not DOWN_COUNT) then
        count_s <= count_s + 1;
      else
        count_s <= count_s - 1;
      end if;
    end if;
  end if;
end process;
  count <= count_s after OutDelay;
end behav;
```

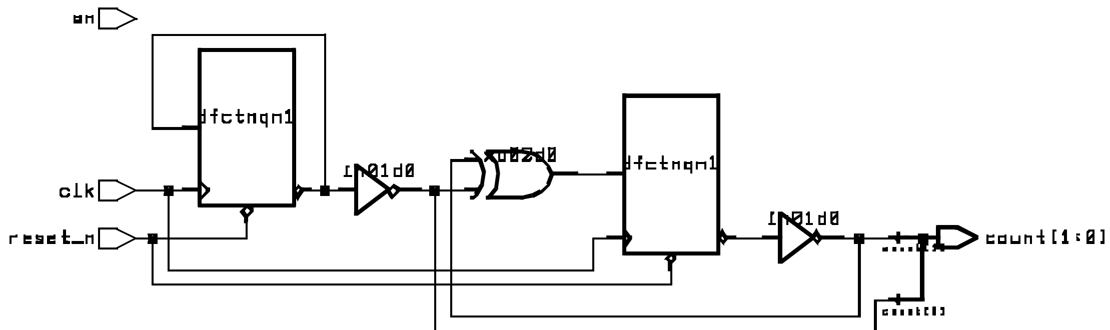
As shown in the above example, use of a package of constants is very similar to the use of generics for parameterization. In addition, it has the following advantages:

- The parameters in the package can be referenced by any design entity that needs the information without any overhead.
- In order to change the structure of the design, all that is necessary is to change the value in the package and the change is reflected in all the units the parameter is referenced.

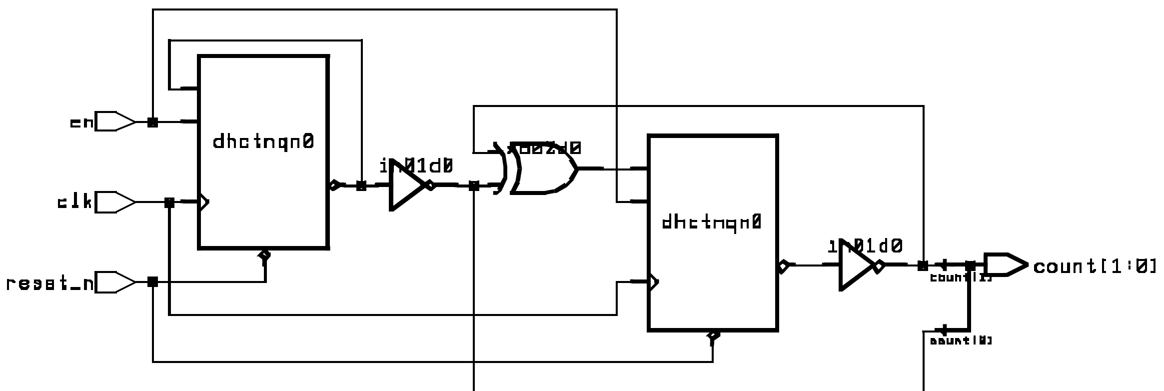
VLSI Technology

- A package of constants can use subtypes and enumerated data types to reference the parameters for reusability and readability.
- A central package can serve as a package of parameters to parameterize the entire design.
- It is relatively simple to use arrays and other composite data types for parameterization inside a package.
- A package can be worked on separately as a design unit that can be worked on independently and reused in different parts of a model.
- Some non-synthesizeable constructs in generic statements are usually synthesizeable when used in a package.
- The package may also contain other constants and information that may or may not be used for parameterization, yet may be used in the design. The package serves as a common placeholder for this type of shared information.
- A package of parameters provides better code structure, provides efficient organization, and is self-documenting.

Examples of Synthesized Counters with Different Parameters

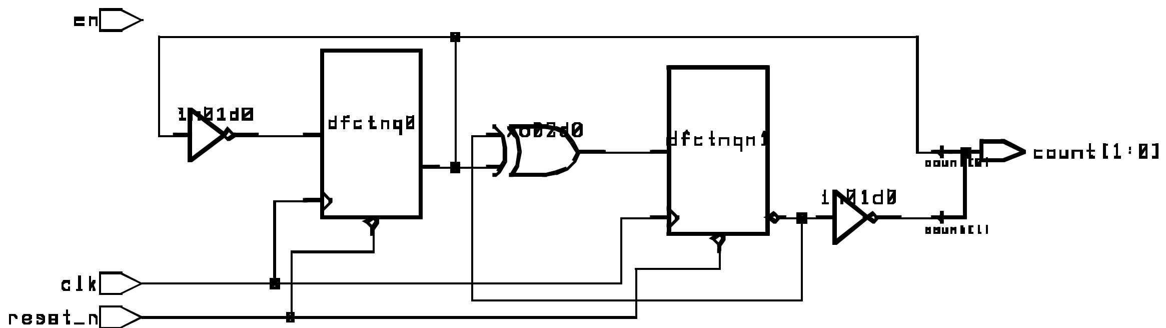


Example 1: Enable Logic is Disabled and the Counter Mode is Set to Count Up



Example 2: An Up Counter with Count Enable

VLSI Technology



Example 3: A Down Counter with No Enable

Deferred Constants

A *deferred* constant is one in which the constants are declared in a package, but are not initialized in the package. Instead, the deferred constants are initialized in the design that uses the constants. In other words, the binding of the constants to a value is “deferred”. Such deferred constants have to be bound before they are referenced.

```
-- Deferred Constants Parameters Package par_def_pkg.vhd

package par_def_pkg is
  constant BIT_WIDTH    : integer := 10;  -- Cannot be deferred
  subtype  CNT_WIDTH is integer range BIT_WIDTH-1 downto 0;
  constant COUNT_ENABLE: BOOLEAN;         -- Deferred
  constant DOWN_COUNT   : BOOLEAN;         -- Deferred
  constant OutDelay     : TIME     := 3 ns -- Behavior
end par_pkg;

-- Reusable Counter Design counter.vhd
library pkgs;
use pkgs.par_pkg.all;

entity counter is
port (
  clk      : in  std_logic
  reset_n  : in  std_logic;
  en       : in  std_logic;
  count    : out std_logic_vector(CNT_WIDTH)
)
end counter;

architecture behav of counter is

constant COUNT_ENABLE: BOOLEAN := false; -- Deferred binding
constant DOWN_COUNT   : BOOLEAN := false; -- Deferred binding

signal count_s : std_logic_vector(CNT_WIDTH); -- Same as count
begin
…
…
…
end behav;
```


VLSI Technology

This way, any change to the package will not require recompilation or resynthesis of the design counter.

The Effect of a Package of Constants on Synthesis

The package of constants has the same effect as the use of generics in modifying structure and/or behavior during synthesis. The package of constants has an added advantage that composite data types can be used very effectively for the purposes of readability, and yet preserving the synthesizability. And it is a little easier to synthesize a design which uses a package of constants then one which uses generics. By easier we don't mean it's easier for the synthesis tool, but rather that it is generally easier for an engineer to learn how to get the synthesis tool to use the package of constants then to use a design which uses generics. But this last fact shouldn't be considered as a major defect in the use of generics. It should also be borne in mind that some synthesis tools take longer run times when composite data types are used.

Package Of Constants: Summary

We've seen that a package of constants can be used much the same way as generics. Packages of constants tend to be easier to use than generics if there are many parameters involved. And packages tend to be a better supported by the synthesis tools than generics.

However, the use of a package of constants has the following drawbacks:

- Multiple instances of the design with different parameters cannot be used in a single design unit.
- A change in a package that uses non-deferred constants will cause the designs referring the package to be recompiled and/or resynthesized even if a design is not affected by a parameter.
- Needs a separate file/library to be maintained.

The use of a package of constants should be compared against the use of generics for parameterization after considering the intended scope of the application. As a general practice, it is recommended to use a package of constants for designs that have a lot of parameters and will not be instantiated multiple times within a large design. For example, a memory controller design (that translates host/CPU cycles into memory cycles) is very unlikely to be instantiated multiple times in a design. Such designs should use a package of constants. Designs such as bus interfaces, counters, adders, LFSRs etc, should use generics for parameterization.

Reuse Tip #3: Generate Statements

Many digital systems can be implemented as regular iterative compositions of subsystems. Memories are a good example, being composed of a rectangular array of storage cells. Indeed, designers prefer to find such implementations, as they make it easier to produce compact, proven, area-efficient layout, thus reducing cost. If a design can be expressed as a repetition of some subsystem, we should be able to describe the subsystem once then describe how it is to be repeatedly instantiated, rather than describe each instantiation individually [2].

Generate Statements can be used very effectively to produce iterative structures of a design. Generate statements are concurrent VHDL constructs that may contain further concurrent statements that are to be replicated. Generate statements when used in conjunction with the generics or constants can be used to generate repetitive structures in an efficient way. Consider an example where a 32-bit on-chip data-bus has to be driven off-chip using 8 output enables through an output PAD (IO).

```
entity chip is
   port (
         data : inout std_logic_vector (31 downto 0);
          .
          .
          .
        );
architecture pads of chip is

   signal dataout : std_logic_vector (data'range); -- output data
   signal datain  : std_logic_vector (data'range); -- input data
   signal dataoe  : std_logic_vector (data'length/4-1 downto 0);

   Component pad is
      Port (
            Pad_di  : out   std_logic; -- this is the input data
                                       -- from the pad
            Pad_do  : in    std_logic; -- this will be driven off-
                                       -- chip
            Pad_oe  : in    std_logic; -- output enable
            Pad     : inout std_logic  -- pad itself
           );
   end pad;

begin
   data_pad : for i in data'range generate
      u_datapad : pad
         port map (
                   pad_di => dataint(i),
                   pad_do => dataout(i),
                   pad_oe => dataoe(i/4),
                   pad    => data(I)
                  );
   end generate;
```


VLSI Technology

This code will instantiate 32 pad cells for the data bus. Note the use of the '*range* and '*length* attributes. These also promote reuse in that they use the previously defined bus widths for the data bus. Also note the use of the "`i/4`" in the assignment of the output enable signals to the pad cell. The synthesis tool should be intelligent enough to truncate the division to an integer value to give to proper assignment of `dataoe(3)` to `data(31:24)`, `dataoe(2)` to `data(23:16)`, and so on.

The next example illustrates the use of generate statements with iterative structures of concurrent statements to create a register from a flip-flop.

```
Reg:
For I in 0 to REG_WIDTH generate
   BitFlop: process (reset_n, clk)
   Begin
      If (reset_n = '0') then
         Q(I) <= '0';
      Elsif (clk'event and clk = '1') then
         If (write_en = '1') then
            Q(I) <= D(I);
         End if;
      End if;
   End process;
End generate;
```

Generate statements can also be used to conditionally create, modify or remove structures. This involves code level optimization, where unwanted structures are removed during elaboration time. With the use of generics and/or package of constants this technique can be very useful in creating a reusable design. With the use of conditional generate statements, logic that implements a certain feature-set can be enabled or disabled, instead of hand-removing the code or optimizing via synthesis. As an example of conditional code inclusion/exclusion, an output can be synchronous to the clock or combinatorial as set by the constant :

```
CONSTANT SYNC_OUTPUTS : BOOLEAN : TRUE;
```

Then a synchronous or a combinatorial output can be generated as shown below:

```
sync:
   if (SYNC_OUTPUTS) generate
      process(clk)
         if (clk'event and clk = '1') then
            if(rd = '1') then
               q <= d;
            end if;
         end if;
      end process;
   end generate;

comb:
if (not SYNC_OUTPUTS) generate
   process(rd)
      if (rd = '1') then
         q <= d;
      end if;
   end process;
```

```
end generate;
```

Generate Statements : Summary

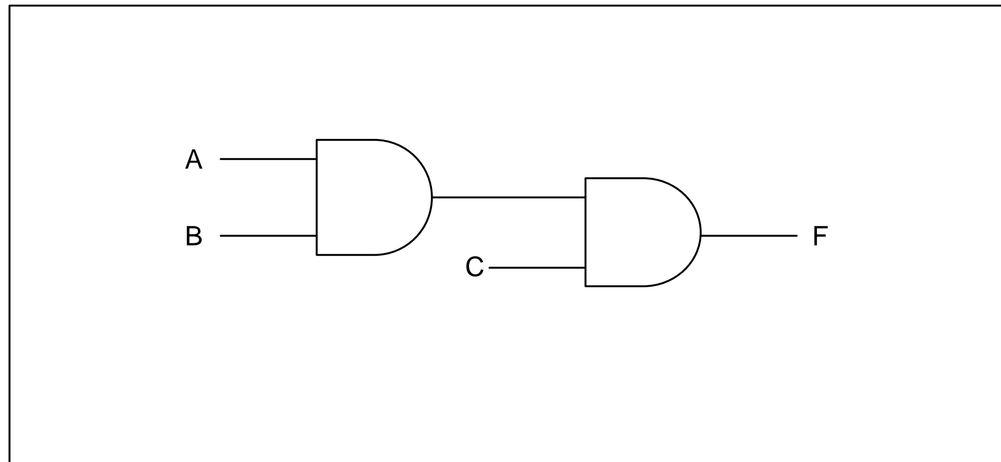
The generate statement is a powerful tool to control the inclusion or exclusion of logic. It is very useful for designs which have blocks of logic which are used repeatedly in an iterative structure such as flip flops which together form registers, pad cells, and many other structures. Many designers use generate statements to instantiate cells like the pads example. But don't forget that it can be used to conditionally create, modify, or remove sections of VHDL code as well. Generate statements are powerful tools in promoting design reuse.

Here are a few more examples of the uses of generate statements:

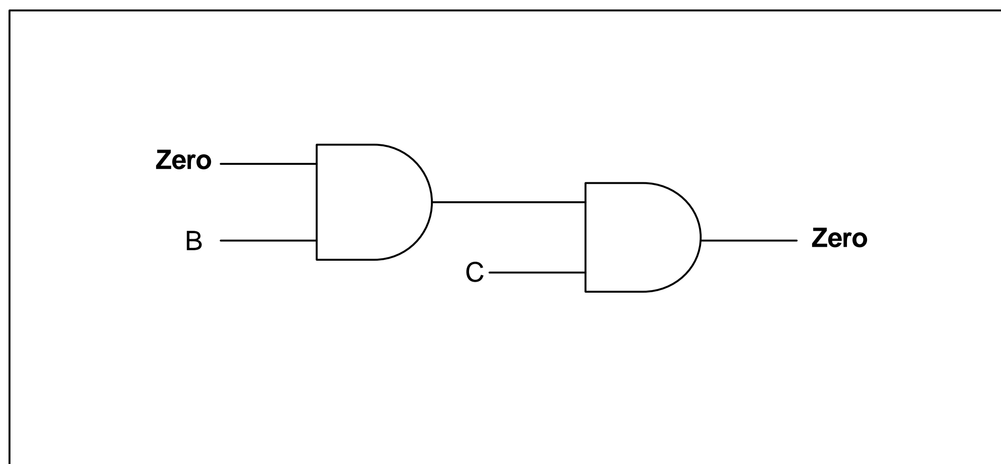
- Choosing implementation of a register (latch or flip-flop based)
- Including one or more arbitration schemes in a bus arbiter design (fixed arbitration, round robin, etc…)
- Only including bits of an interrupt controller that we know we're going to use. This may sound trivial, but consider the case where the interrupt bits coming into the interrupt controller are registered. If these inputs then go through a substantial amount of combinatorial logic before being routed to some other registers, then using generate statements to only include the necessary flip flops will help the synthesis tools reduce the gate count significantly. Synthesis tools generally cannot optimize across flip flops. So even if we know that an input is always tied high (an unused interrupt) the synthesis tool can't use this information to reduce the gate count of the synthesized design. Depending on the design, this technique can result in significant gate counts. However, this technique won't be as readable as it would be by including all register bits. So use it only if it results in a significant gate count.

Reuse Tip #4: Tying off Ports

In many instances logic can be selectively disabled by tying off certain ports to default values. When synthesized with a top-down approach, the synthesis tool will optimize that path, taking that tied-off value into consideration. The tied-off ports can later be removed from the entity. For example, in a design which has a series of AND gates, as shown below:



If one of the inputs is tied to a ‘0’ as shown below then the resulting logic will eliminate all the three AND gates and the output (zo) will be at logic ‘0’ always.



This is also referred to as constant propagation.

```
a1: and3
   port map (
             a  => a,
             b  => '0', -- Valid in VHDL 93, Tied off to vss <=
                        -- '0' in 87
             c  => c,
             zo => zo
            );
```

The same is true for port outputs. By leaving unused port outputs open (`zo => open`) we can optimize away the logic which creates them.

Reuse Tip #5: Unconstrained Arrays

Use of unconstrained arrays is a great way to reuse designs for variable-width implementations. Care should be taken while using attributes like ‘range, ‘length etc. inside the design to avoid run-time and elaboration-time errors. Unconstrained arrays are particularly suitable for address, data, and register widths. They may also be used for formal parameters in functions and procedures.

VHDL allows the use of unconstrained array types that let us indicate the type of index values without specifying the bounds. Unconstrained arrays come in handy to make reusable designs that may be reused in various applications, by just modifying their bit-widths. As an example, our counter example can be designed using unconstrained arrays for the count output as shown below:

```
entity counter is
port (
  clk     : in  std_logic
  reset_n : in  std_logic;
  count   : out std_logic_vector     -- Unconstrained array
)
end counter;

architecture behav of counter is
signal count_s : std_logic_vector(count’range) -- Same as count
begin
process (clk, reset_n)
begin
  if(reset_n = ‘0’) then
    count_s <= (others => ‘0’);
  elsif (clk’event and clk = ‘1’) then
    count_s <= count_s + 1;
  end if;
end process;
count <= count_s;
end behav;
```

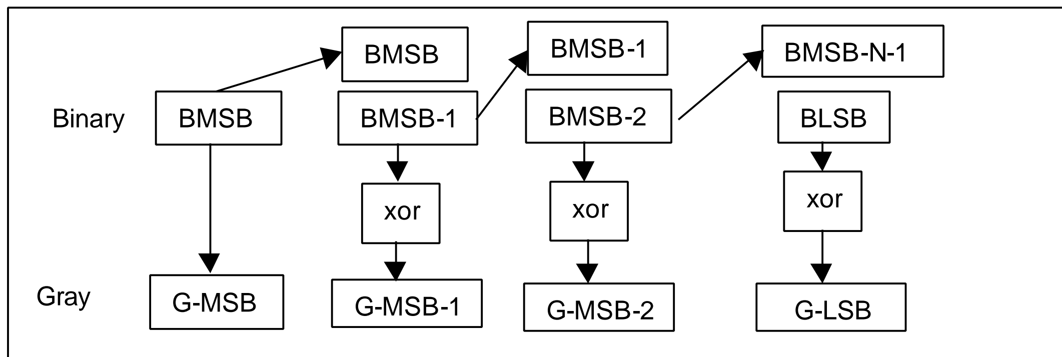
This lets us connect the counter entity to array signals of any size or with any range of index values. Note the use of the VHDL attribute ‘range to create a signal of the same width and range specification as the port count itself. Note that this design is not synthesizable by itself and has to be instantiated in a top level entity to bind the array values to a finite range as in the following example:

VLSI Technology

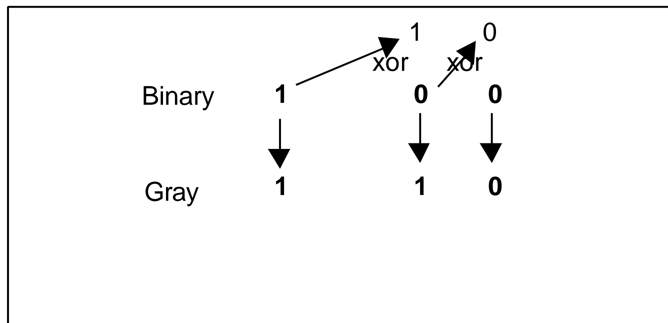
```
architecture inst of top is
begin
signal my_count : std_logic_vector(7 downto 0);
signal clk, reset_n : std_logic;

cnt: counter
port map
  ( clk => clk,
    reset_n => reset_n,
    count => my_count
  );
..
..
..
end inst;
```

Another use of unconstrained arrays is in functions and procedures. Functions and procedures that are intended to be synthesizable should be written as general as possible, independent of bit widths.

Consider an example of a binary code to gray code converter. In order to create a gray code from a given binary code we use the algorithm shown in the figure below:



Here is an example where we convert binary 100 to its gray code equivalent of 110:



The following table shows the gray codes for three-bit binary values:

Binary	Gray
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

If this algorithm were hard coded and optimized for a 3-bit case, then when the design has to accommodate more counts, this function has to change, and hence the entire logic has to be revalidated. By writing a generic function independent of the bit-vector lengths, efficient reuse is possible. A bit-width independent implementation for the binary code to gray code converter is shown below. If one takes a look at some of the functions and procedures in the IEEE `std_logic` libraries, most of the functions and procedures are implemented using unconstrained arrays to support efficient reuse.

VLSI Technology

```
function Bin2Gray( x : std_logic_vector ) return std_logic_vector
is
variable y : std_logic_vector( x'range );
begin
 for j in x'range loop
   if j = x'left then
     y(j) := x(j);
   else
     y(j) := x(j) xor x(j+1);
   end if;
 end loop;
return  y;
end;
```

Reuse Tip #6: Configuration Specifications

Configuration specifications are used to bind the component instances to design entities. Configurations can be used for a variety of purposes:

- For passing parameters as generics at the top-most level in a test bench;
- For choosing between architectures for a particular entity;
- For overriding port mappings in an instantiation.

Note that some synthesis tools do not support configuration specifications. The following example illustrates the use of a fairly elaborate configuration specification:

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

architecture buffered of counter is

signal count_s : std_logic_vector(BIT_WIDTH-1 downto 0);

component non_inv_buffer
  port (i : in std_logic;
        z : out std_logic);
end component;

begin
  process (clk, reset_n)
  begin
  if(reset_n = '0') then
    count_s <= (others => '0');
  elsif (clk'event and clk = '1') then
    if (COUNT_ENABLE = 1) then  -- removed if generic is false
      if (en = '1') then
        if (DOWN_COUNT = 0) then
          count_s <= count_s + 1;
        else
          count_s <= count_s - 1;
        end if;
      end if;
    else
      if (DOWN_COUNT = 0) then
        count_s <= count_s + 1;
      else
        count_s <= count_s - 1;
```


VLSI Technology

```
        end if;
      end if;
    end if;
  end process;

bufG : for  index in BIT_WIDTH-1 downto 0 generate
    nib_i : non_inv_buffer port map (
                                   i => count_s(index),
                                   z => count(index)
                                   );
end generate bufG;

end buffered;


library ieee;
use ieee.std_logic_1164.all;

entity top is
end top;
-- Some top level that uses different counters
architecture inst of top is

  constant BIT_WIDTH : integer := 8;
  signal clk : std_logic;
  signal gp_count, wd_count : std_logic_vector ( bit_width - 1 downto 0 );
  signal en : std_logic;
  signal reset_n : std_logic;

  component counter
    port (
      clk     : in  std_logic;
      reset_n : in  std_logic;
      en      : in  std_logic;
      count   : out std_logic_vector(BIT_WIDTH-1 downto 0)
    );
  end component;

begin

gpcntr: counter port map (
  clk => clk,
  reset_n => reset_n,
  en => en,
  count => gp_count
);

wdcntr: counter port map (
  clk => clk,
  reset_n => reset_n,
  en => en,
  count => wd_count
);
```

```
-- Other stuff
end inst;

library cntr_lib, vlsi_lib;

configuration cntr_cfg of top is
  for inst
    for gpcntr : counter use entity cntr_lib.counter(behav)
                    generic map (COUNT_ENABLE => 1,
                                 DOWN_COUNT   => 1);
    end for;
    for wdcntr   : counter use entity cntr_lib.counter(buffered)
                              generic map (COUNT_ENABLE => 1,
                                           DOWN_COUNT   => 0);
        for buffered
          for bufG
            for nib_i : non_inv_buffer use entity vlsi_lib.ni01d2(rtl);
            end for;
          end for;      -- End bufG
        end for;        -- End buffered
    end for;            -- End wdcntr

  end for;              -- End inst
end cntr_cfg;
```

Reuse Tip #7: Use of a Preprocessor for Enhancing VHDL Features

There are number of situations when the designer is not able to accomplish what they want using the available features. And in some cases, it is desirable to see only the code that is relevant to the design. In such cases, a preprocessor can be used to add, eliminate, or modify code for a specific application, through the use of preprocessor directives.

For example, VLSI developed a tool referred to as HDLi. It is used as a vehicle for reusable IP delivery. HDLi uses a subset of its own commands to modify VHDL/Verilog code for a specific application. The desired design features are specified through an interactive GUI. This solution has few limitations in terms of maintainability, testability and the numerous combinations of logic that can possibly be produced.

.

Reuse Tip #8: VHDL Attributes that aid Reuse and are Synthesizable

A few attributes of composite types are very useful in creating reusable designs. The attributes

- 'left
- 'right
- 'range
- 'length
- 'low
- 'high

are synthesizable and make the code independent of the data type. Refer to the example of the unconstrained arrays where the function Gray2bin and the entity counter use the **'range** attribute, thus promoting reusability.

Reuse Tip #9: Use of Block Statements for Design Reuse

Block statements are VHDL constructs that allow inline design partitioning. For example, if a design has been partitioned such that the data path exists in a separate VHDL entity, then the architecture for that entity, can in turn be partitioned using block statements. Block statements are a great way to group related logic together. Block statements also provide the facility to declare signals within the blocks and hence if that block is removed, unnecessary signals will not hang in the code. A generate statement can be combined with the block statement to selectively include/exclude blocks.

Reuse Tip #10: Leaving Unused Ports 'open'

In a hierarchical design, if certain ports are unused in a specific entity, then the usual practice is to connect them to a dummy signal. From a top-down synthesis approach, this makes the synthesizer assume that the signal is connected to a net. This can be avoided by leaving the port unconnected or specifying via the VHDL keyword **open.**

References

The following documents provide a variety of useful references for writing synthesizable VHDL:

1) Meiyappan, S, Chambers, P, "Design Reuse Using Scripting Methodologies," DesignCon 98, On-Chip System Design Conference, Santa Clara, CA, Jan 1998, PP 629-643

2) Ashenden, P.J, The Designer's Guide to VHDL, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

3) Smith, Douglas J., "HDL Chip Design," Doone Publications, 1996.

4) http://www.vlsi.com/products/design.shtml

VLSI Technology