"Design reuse" is the process of migrating high-quality intellectual property (IP) from one ASIC design to another. With the tremendous advances in semiconductor technology, it is increasingly difficult to bridge the productivity gap between what technology offers and what silicon productivity allows. Designing full-custom ASICs to occupy as much silicon area as possible is becoming increasingly challenging. To achieve the highest level of silicon efficiency, designing semicustom ASICs with highly reusable design entities has become today's challenge.

The use of predesigned and preverified design blocks to achieve a high level of design reuse is the most promising technique for bridging the gap between available gate count and designer productivity. Designing a complex chip requires an HDL-based design. Effective HDL generation with design reuse in mind will help you create IP cores that are usable in multiple chip designs.

The design-reuse challenge

Designing for reuse poses new and innovative challenges to a designer. Before being reusable, a design must be usable, which means design using good design practices. A reusable design must be designed with the mindset of solving a general problem; well-coded, commented, and documented; verified to a high level of confidence; and independent of technology, design tools, and applications.

Because of mounting time-to-market pressures, designers often bypass some or all of these guidelines, rendering a design virtually nonreusable. However, following these guidelines speeds designing, verifying, and debugging a project by reducing iterations throughout the coding and verification loops. The efficient use and reuse of designs play a vital role in the creation of large ASICs with aggressive design schedules.

Although chip designers have used HDLs for some time, most designs today do not use the built-in "design-reuse" features of the languages. In other words, designers do not thoroughly understand the purpose of the HDLs and misuse or underuse their features. On an average, only 20% of the designs in the industry are reusable. With an increasing need for design reuse, the emphasis on coding techniques for design reuse is on the rise. This article covers developing reusable designs using the native-language features of VHDL and design-reuse techniques pertaining to synthesizable VHDL. Unless stated otherwise, the VHDL discussed complies with the VHDL-87 standard.

VHDL features promoting reusability

Chip designers have used VHDL for more than a decade. One of the primary intents of developing
designs in VHDL is reusability, although, designers, until recently, have not effectively employed this

You can exploit the feature-rich VHDL for reuse techniques. VHDL features include

generics, packages of constants, generate statements, unconstrained arrays, VHDL attributes, block

statements for inline-design partitioning, record data types for data bundling, configuration

specifications, the ability to tie ports off to known constants, the ability to leave unused output ports
open and unconnected, array aggregates, functions, and procedures.

**Tip 1: Generics**

You use generics to write parameterized models of varying structure and behavior (Reference 1). Listing 1 provides a simple example of a synchronous counter with modifiable structure and behavior. You accomplish this modification through the use of VHDL generics. This example illustrates the use of generics for modifying structure and behavior using the language’s features for simulation and synthesis. You can enable and disable selective features by turning generics on and off. For example, if you set the COUNT_ENABLE generic to FALSE in line 8, then none of the logic
described in lines 32 to 38 is elaborated or synthesized, but the parent design can still have a count enable. Using different values for OutDelay and DOWN_COUNT changes the structure of the design. Creating designs with generics enables design reuse in various circumstances where you need different structure or behavior. For example, a design may require two counters: one that counts to 1024 and another that counts to eight. Designing two separate counters—one that is 10 bits wide and one that is 3 bits wide—has drawbacks of unnecessary investment in design, verification, and synthesis time.

If you use the generic approach to design a counter with reuse in mind, you save a great deal of
design, synthesis, and verification time. The use of generics for parameterizing structure and

behavior is essential for design reuse-applications. The following examples illustrate the

instantiation of the counter in Listing 1 in an application that requires a 10-bit up-counter and a 3-bit
down-counter

The example in Listing 2 illustrates the following points:

- Lines 3 to 14 instantiate the counter 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.
- Unmapped generic values in the instantiation assume default values.
- Lines 18 to 30 instantiate the same counter 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, it is not advisable to use positional association because changing a parameter or port in the reusable design requires the same modification in all instances of that design.
- The use of generics can help greatly in resources and time when you need multiple instances of the same design.

The use of generics to parameterize designs helps not only to create reusable design blocks, but also to remove unnecessary logic or to modify useful logic during synthesis. Some synthesis tools help create macros and templates when you parameterize designs through generics. You can use the feature thus created as a library element in subsequent designs for simulations or synthesis. Parameterizing bus and register widths through generics is a simple example of the use of generics.

Consider the example of the counter in Listing 1 with the following generic values:
When synthesis elaborates this design, the synthesis tool ignores the generic for OutDelay because the tool cannot handle time-delay elements in mapping logic. The synthesis tool creates a 2-bit down-counter with the count_enable logic, as the following examples illustrate.

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

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

This code creates an 8-bit up-counter without the count-enable logic. If gate count is an important parameter, you can efficiently optimize unused logic using this method. You can modify the structure (changing the BIT_WIDTH) or behavior (up-or down-counter, count_enable disabled or enabled) during design, synthesis, and simulation using this elegant approach to parameterization.

Generics are excellent for specifying widths of counters, buses, shift registers, and other designs, but as Listing 2 shows, you can also use generics to turn various features on and off. This technique lets you use only the features that apply to your current project. You can use generics to specify such features as FIFO depths; bus interface, such as PCI or ARM System Bus; architecture, such as up/down counter, flip-flop-based register versus latched-based register, and ripple-carry adder versus carry-look-ahead adder; register address; power-on-reset value for a register; supported and reserved bits in a register; clock-divide ratio for a clock-divider circuit; and number of buffers in a clock tree.

If you make the design somewhat generic, others can more easily reuse it. One drawback of the generic approach occurs when you use generics in a hierarchy. To apply the generics to the lowest level of the hierarchy, the generics must pass down through the hierarchy. This passing down may involve generics having to go through blocks that do not use the value of the generics. Another drawback of using generics is that, as the list of generics grows, it becomes more cumbersome to carry them around at each point in the hierarchy. A third drawback is that some synthesis tools have limited support for generics. For example, a synthesis tool may require all generics to be of type integer. An efficient way to avoid these problems is to use a package of constants.

**Tip 2: Constants**

A VHDL package is a simple way of grouping a collection of related declarations that serve a common purpose. You can make the package visible to the appropriate design blocks by using library statements. Using library statements means that adding or changing a parameter requires you to modify only one package file. Also, some synthesis tools do not allow the use of Boolean, string, enumerated, or array types for generics. In such cases, using library statements allows you to use a constant in a package. Most synthesis tools allow most data types, and using library statements lets packages use TYPE statements for enumerated data types. A package of constants also lets you use the same package for design and simulation in "design-aware" testbenches.

As an example of the use of a package of constants, consider changing the counter in Listing 2 to use such a package. Also, assume that the package resides in the "pkgs" VHDL library (Listing 3). This counter example shows that using a package of constants is similar to using generics for parameterization. In addition, using a package of constants allows any design entity to reference the parameters in the package without any overhead. Also, to change the structure of the design, you
have to change only the parameter value in the package, and you can see the change in all the units referencing the parameter. A package of constants can also use subtypes and enumerated data types to reference the parameters for reusability and readability, and a central package can serve as a package of parameters to parameterize an entire design. Further, using a package makes it relatively simple to use arrays and other composite data types for parameterization.

You can work on a package separately as a design unit, create the package independently of the design, and reuse the package in different parts of a model. Some nonsynthesizeable constructs in generic definitions, such as enumerated data types, become synthesizeable when you use them in a package. The package may contain other constants and information that you may use for parameterization, yet the design may still use this information. The package serves as a common place-holder for this type of shared information. Furthermore, a package of parameters provides better code structure, provides efficient organization, and is self-documenting.

Figure 1 shows the parameterized counter for different values of generics and constants. The counter was synthesized using Synopsys' (www.synopsys.com) Design Compiler with a 0.2-µm standard-cell library with the BIT_WIDTH parameter set to 2 in all synthesis tests.

In the counter of Figure 1, COUNT_ENABLE is false (unconnected en enable signal), BIT_WIDTH is 2, and DOWN_COUNT is false (a conventional up-counter). In the counter of Figure 2, an up-counter with count enable, COUNT_ENABLE is true (connected en enable signal), BIT_WIDTH is 2, and DOWN_COUNT is false. Also in the counter of Figure 3, a down-counter with no enable, COUNT_ENABLE is false (unconnected en enable signal), BIT_WIDTH is 2, and DOWN_COUNT is true. These three examples show how you can modify counter structure and behavior by using different values of generics and constants while eliminating unnecessary gates.

A deferred constant is one in which you declare but do not initialize the constants in a package. Instead, you initialize the deferred constants in the design that uses the constants. In other words, you "defer" the binding of the constants. You must bound deferred constants before you reference them so that any change to the package does not require design-counter recompilation or resynthesis (Listing 4).

Using a package of constants has the same effect as using generics modify structure or behavior during synthesis. The package of constants also allows you to effectively use composite data types for readability and still preserve design synthesizability. Furthermore, it is easier to synthesize a design that uses a package of constants than one that uses generics. In other words, it is easier for an engineer to learn how to get the synthesis tool to use the package of constants than to use a design that uses generics. Some synthesis tools have longer runtimes for designs with composite data types.

You can use a package of constants in much the same way that you use generics. Packages of constants are easier to use than are generics if a lot of parameters are involved. Packages also typically have better support of synthesis tools than do generics. However, using a package of constants means that you cannot use multiple instances of a design with different parameters in a single design unit. Instead, you need a unique entity and a unique package for each recurring design unit. Also, a change in a package that uses nondeferred constants causes recompilation or resynthesis of the designs referring the package even if a parameter does not affect the design. Also, a package of constants requires you to maintain a separate file or library.

Compare using a package of constants with using generics for parameterization after considering the intended scope of an application. As a general practice, use a package of constants for designs that have many parameters and are not instantiated multiple times within a large design. For
example, a memory-controller design that translates host/CPU cycles into memory cycles is unlikely to be instantiated multiple times in a design. Such designs should use a package of constants. You should use generics for designs such as bus interfaces, counters, adders, and linear-feedback shift registers.

**Tip 3: Generate statements**

You can implement many digital systems, such as memories, as regular iterative compositions of subsystems. For example, memories comprise rectangular arrays of storage cells. Designers prefer such implementations, because they make it easier to produce compact, proven, area-efficient layouts, thus reducing cost. If you can express a design as a repetition of some subsystem, you should be able to describe the subsystem once and then describe how it is to be repeatedly instantiated, rather than describe each instantiation individually (Reference 2).

You can use generate statements to effectively produce iterative structures of a design. Generate statements are concurrent VHDL constructs that may contain further concurrent statements for replication. When you use generate Statements in conjunction with generics or constants, they can efficiently generate repetitive structures. Consider a situation in which you need to drive a 32-bit off-chip data bus from on-chip using eight output enables through an output pad (Listing 5). This example instantiates 32 pad cells for the data bus. Note the use of the "range" and "length" attributes. These attributes also promote reuse in that they use the previously defined bus widths for the data bus. Also note the use of "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.

Listing 6 illustrates the use of generate statements with iterative structures of concurrent statements to create a register from a flip-flop. You can also use generate statements to conditionally create, modify, or remove structures. This technique involves code-level optimization, which removes unwanted structures during elaboration time. With the use of generics or packages of constants, this technique can be useful in creating a reusable design.

Using conditional generate statements, you can enable or disable logic that implements certain features instead of manually removing the code or optimizing via synthesis. As an example of conditional code inclusion and exclusion, you can synchronize an output to the clock or combinatorially set it with the constant CONSTANT SYNC_OUTPUTS : BOOLEAN : TRUE; This technique lets you generate a synchronous or a combinatorial output (Listing 7).

The generate statement is a powerful tool to control the inclusion or exclusion of logic. It is useful for designs that repeatedly use blocks of logic, such as flip-flops, in an iterative structure. These blocks form registers, pad cells, and many other structures. Many designers use generate statements to instantiate cells, as the pads example illustrates, but you can also use generate statements to conditionally create, modify, or remove sections of VHDL code. Generate statements are powerful tools promoting design reuse. A few more examples that show the application of generate statements are choosing implementation of a latch-based or flip-flop-based register; including fixed, round-robin, or another arbitration scheme in a bus-arbiter design; and including only those bits of an interrupt controller that you know you are going to use. Consider the case in which registered interrupts are entering the interrupt controller. If these inputs go through a substantial amount of combinatorial logic before being routed to other registers, then the use of generate statements to include only the necessary flip-flops will help a synthesis tool to significantly reduce the gate count. Be aware that some synthesis tools cannot optimize across flip-flops. In these cases, even if we know that an input, such as an unused interrupt, is always tied high, the synthesis tool can't use this information to reduce the gate count of the synthesized design.
Tip 4: Ports

In many instances, you can selectively disable logic by tying off certain ports to default values. When synthesized with a top-down approach, the synthesis tool uses "optimization by constant propagation"—optimizing that path and taking into consideration that tied-off value. You can later remove the tied-off ports from the entity. Consider a three-AND-gate design (Figure 4a). If you tie one of the inputs to a zero (Figure 4b), then the resulting logic eliminates all the AND gates and the output, F, is always at logic 0.

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

The same situation is true for port outputs. By leaving unused port outputs open (zo = > open), you can eliminate the logic that creates these outputs when you adopt a top-down synthesis approach.

Tip 5: Unconstrained arrays

Using unconstrained arrays is a helpful method of reusing designs for variable-width implementations. You should be careful when using attributes such as "range" and "length" in the design to avoid runtime and elaboration-time errors. Unconstrained arrays are particularly suitable for address, data, and register widths. You can use these arrays for formal parameters in functions and procedures as well as for entity ports.

VHDL allows the use of unconstrained-array types that let you indicate the type of index values without specifying the index bounds. Unconstrained arrays are useful for making designs that you can reuse in different applications just by modifying their bit widths. The previous counter example uses unconstrained arrays for the count output (Listing 8). This technique lets you 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. You cannot synthesize this design by itself, and you have to instantiate it in a top-level entity to bind the array values to a finite range (Listing 9). You must synthesize the code in Listing 9 in a top-down manner so that you can synthesize the counter along with the rest of the design.

Another use of unconstrained arrays occurs in functions and procedures. You should write functions and procedures that you design for synthesis as generically as possible, independently of bit widths. Consider an example of a binary-code-to-gray-code converter. To create a gray code from a binary code, use the algorithm in Figure 5a. Figure 5b is an example of how to convert binary 100 to its gray-code equivalent of 110. Table 1 shows the gray codes for the 3-bit binary values that the algorithm of Figure 5a creates. You hard-code and optimize this algorithm for a 3-bit case. When the design has to accommodate more counts, the function has to change, requiring you to revalidate all the logic. Writing a generic function that is independent of the bit-vector lengths makes efficient reuse possible. Listing 10 is a bit-width-independent implementation for the binary-code-to-gray-code converter. As another example, consider the functions and procedures in the IEEE std_logic libraries. Most of these functions and procedures are implemented using unconstrained arrays to support efficient reuse.

Tip 6: VHDL attributes

A few attributes of composite types are useful in creating reusable designs. The attributes "left," "right," "range," "length," "low," and "high" are synthesizable and make the code independent of data type. Refer to the examples using unconstrained arrays (Listing 8 and Listing 9), where the function Gray2bin and the entity counter use the "range" attribute to promote reusability.
Tip 7: Configuration specs

You use configuration specifications to bind component instances to design entities. You can also use these configurations to pass parameters such as generics at the top-most level in a testbench, to select an architecture for an entity, or to override port mappings in an instantiation. Some synthesis tools do not support configuration specifications.

Consider the previous counter example that illustrates the use of generics for parameterization. Listing 11 illustrates the same counter with another architecture that buffers the counter outputs with a generate statement. The counter is now instantiated in a top-level design using two instances of the counter (Listing 12). A configuration specification configures the counter in the entity top, as shown in Listing 13. Configuration specifications let you configure various levels of the design’s hierarchy.

Tip 8: Block statements

Block statements are VHDL constructs that allow inline design partitioning. For example, if you partition a design such that the datapath exists in a separate VHDL entity, then you can partition the architecture for that entity using block statements. Block statements are a method of grouping related logic. Block statements also provide the ability to declare signals within the blocks and, if you remove the block, unnecessary signals do not remain unconnected in the code. You can combine a generate statement with the block statement to selectively include or exclude blocks.

Tip 9: Unused ports

In a hierarchical design, if you do not use certain ports in an entity, then the usual practice is to connect them to a dummy signal. From a top-down synthesis approach, this scenario makes the synthesizer assume that you’ve connected the signal to a net. You can avoid this problem by leaving the port unconnected or by specifying with the VHDL keyword "open."

Tip 10: Preprocessors

In many situations, designers cannot accomplish what they want using the available features. In some cases, it is desirable to see only the code that is relevant to the design. In such cases, you can use a preprocessor to add, eliminate, or modify code for a specific application, through the use of preprocessor directives.

Author info

Subbu Meiyappan is a senior design engineer at VLSI Technology. He has worked for the company for nearly three years, designing, developing, synthesizing, simulating, and validating high-performance IP blocks for PCI, ARM-ASB-based devices, and high-performance ASICs. He has a BE from Annamalai University (Annamalai Nagar, India) and an MS from Tennessee Technological University (Cookeville, TN). His interests include computer architecture, design automation, volleyball, and travel.

Ken Jaramillo is a staff engineer at VLSI Technology (www.vlsi.com). In his three years with the company, he has worked on high-speed networking designs, such as fiber-distributed data interfaces, Firewire, and high-speed satellite modems. He has a BSEE from the University of Missouri (Kansas City, MO) and a BScCoE from the University of Missouri (Columbia, MO). His hobbies include basketball, rock climbing, and travel.

Peter Chambers is an engineering fellow at VLSI Technology, where he has worked for six years
developing many PCI-based designs, ASICs, chip sets, and reusable IP cores. He has a BS from the University of Exeter (UK) and an MS from Arizona State University (Tempe, AZ). He is a member of both IEE and IEEE.

**REFERENCE**

