The Low-Carb VHDL Tutorial

Table of Contents

List of Figures

List of Tables

List of Examples

1 Intent and Purpose

The purpose of this paper is to provide students with a tutorial to help develop the skills necessary to be able to use VHDL in the context of CPE 129/169, CPE 229/269, and CPE 329. Although there are many books regarding VHDL as well as many tutorials available on the internet, these sources are sometimes inadequate for several reasons. First, much of the information regarding VHDL is either needlessly confusing or poorly written. Secondly, the common approach is to introduce information and topics early on in the study which would be better presented later. This information has a tendency to be confusing and is easily forgotten if misunderstood or never applied. And lastly, the information presented is out of context with the approach and the learning objectives of CPE 129/169, CPE 229/269, and CPE 329.

The intent of this paper is to present topics in the context of the average CPE 129/169 student who has some knowledge of digital logic and has some skills with algorithmic programming languages such as Java or C. The information presented in this paper is focused on a base knowledge of the approach and function of VHDL. With a proper introduction to the basics of VHDL combined with a logical and intelligent introduction of VHDL concepts, students will be able to quickly and efficiently create useful VHDL code. In this way, students will be able to view VHDL as a valuable design, simulation, and test tool rather than another batch of throw-away technical knowledge encountered in some forgotten class or lab.

Lastly, VHDL is a powerful tool. The more you understand in the time you put into studying and working with VHDL, the more it will enhance your learning experience independently of your particular area of interest. The VHDL programming paradigm is also an interesting companion to algorithmic programming. It is well worth noting that VHDL and other hardware design languages that are used to create most of the digital integrated circuits found in the various electronic gizmos that currently overwhelm our modern lives. The concept of using software to design hardware that runs software will surely cause you endless hours of contemplation.

Disclaimer: This tutorial quickly brings you down the path to understanding VHDL and writing solid VHDL code. The ideas presented herein represent what generally the core ideas you'll need to get up and running with VHDL. This tutorial in no way presents a complete description of the VHDL language. In an effort to expedite the learning process, some of the fine details of VHDL have been omitted from this tutorial. Anyone who has the time and inclination should feel free to further explore the true depth of the VHDL language. There are many references and tutorials available on the web. If you find yourself becoming curious about what this tutorial is not telling you about VHDL, take a look at some of these references.

2 Introduction

VHDL has a rich and interesting history. But since knowing this history is probably not going to help you write better VHDL code, it will only be barely mentioned here. Consulting the proper text or search engine will yield this information for those who are interested. It is, however, worthy to state what the VHDL acronym stand for. Actually, the "V" in VHDL is short of yet another acronym: VHSIC or Very High-Speed Integrated Circuit. The HDL stands for Hardware Description Language. Obviously, the state of technical affairs these days has obviated the need for nested acronyms.

There are two primary purposes for hardware description languages such as VHDL. First, VHDL can be used to model digital circuits. Having a model of the circuit allows for simulation and testing of the design for proper operation. But maybe more importantly, the act of creating the model from VHDL code is a valuable and interesting learning experience in itself. Second, VHDL and other hardware description languages are used as one of the first steps in creating large digital integrated circuits. The VHDL code is used to magically[1] create digital circuits in a process known as *synthesis*.

These two purposes of VHDL match what we use it for in the CPE 129/169 and CPE 229/269. In the area of testing, there are other logic simulators are available to model the behavior of digital circuit designs. These packages are easy to use because they provide a graphical method to describe circuit designs. The tendency here is to prefer the graphical approach because it has such a nice learning curve. But, as you can easily imagine, your growing knowledge of digital concepts is accompanied by an ever-increasing complexity of digital circuits you'll be dealing with. The act of graphically connecting a bunch of lines on the computer screen quickly becomes tedious. The more intelligent approach is to be able to describe exactly how your digital circuit works (modeling it) without having to worry about the details of connecting massive quantities of signal lines. Having a working knowledge of VHDL will provide you with the tools to model digital circuits in an intelligent manner.

And finally, you'll be able to use your VHDL code to create actual functioning circuits. This allows you to implement relatively complex circuits in a relatively short period of time. The design methodology you'll be using allows you to dedicate more time to designing your circuits and less time "constructing" them. The days of placing, wiring, and troubleshooting multiple integrated circuits on a proto-board are gone. The emphasis is now placed on writing and maintaining a quality description of the circuit. The main requirement now is to learn the description language and the various design tools involved in the process.

[1] It's not really magic. There is actually a well-defined science behind it.

3 VHDL Invariants

There are several qualities of VHDL that you should know before moving forward. Although it's rarely a good idea to people to memorize anything, you should memorize the basic ideas presented in this section. Making these ideas second nature should help eliminate some of the drudgery involved in learning a new programming language while laying the foundation for creating visually pleasing VHDL source code.

3.1 Case Sensitivity

VHDL is not case sensitive. This means that the two statements in Figure 1 have the exact same meaning (don't worry about what the statement actually means though):

```
Dout <= A and B;
```

```
doUt <= a AND b;
```

Figure 1: An example of VHDL case insensitivity.

3.2 White Space

VHDL is not sensitive to white space (spaces and tabs) in the source document. The two statements in Figure 2 have the exact same meaning.

```
nQ   <=   In_a or In_b;
```

```
nQ <= in_a    OR         in_b;
```

Figure 2: An example showing VHDL's indifference to white space.

3.3 Comments

Appropriate use of comments increases both the readability and the understandability of any VHDL code. The general rule is to comment any line or section of code that may not be clear to some other reader of your code. The only inappropriate use of a comment is to state something that is patently obvious. Comments in VHDL begin with "`--`" (two consecutive dashes). The VHDL compiler ignores anything after the two dashes up to the end of the line in which the dashes appear. Figure 3 shows two types of commenting styles. Unfortunately, there are no block-style comments (comments that span multiple lines but don't require comment marks on every line) in available in VHDL

```
-- This next section of code is used to blah-blah
-- blah-blah blah-blah. This type of comment is the best
-- fake for block-style commenting.

PS <= NS_reg;      -- Assign next_state value to present_state
```

Figure 3: Two typical uses of comments.

3.4 Parenthesis

VHDL is relatively lax on its requirement for using parenthesis. Like other computer languages, there are precedence rules associated with the various operators in the VHDL language. Though it is possible to learn all these rules and write clever VHDL source code that will ensure the readers of your code will be scratching their heads, a better idea is to practice liberal use of parenthesis to ensure the human reader of your source code understands the purpose the code. Once again, the two statements appearing in Figure 4 have the same meaning. Note that extra white space has been added in addition to the parenthesis to make the statement on the right clear.

```
if x = '0' and y = '0' or z = '1' then
    blah;
    blah;
    blah;
end if;

if  ( ((x = '0') and (y = '0')) or (z = '1') )  then
    blah;
    blah;
    blah;
end if;
```

Figure 4: An example of parenthesis use that produces clarity.

3.5 VHDL Statements

Similar to other algorithmic programming languages, every VHDL statement is terminated with a semicolon. This fact helps when attempting to remove compile errors from VHDL code since they are easily forgotten during coding. The main challenge them becomes to know what constitutes a VHDL statement in order to know when to include semicolons. The VHDL compiler is not as forgiving as other languages when superfluous semicolons are place in the source code.

3.6 `if`, `case`, and `loop` Statements

As you soon will find out, the VHDL language contains **`if`**, **`case`**, and **`loop`** statements. A common source of frustration that occurs when learning VHDL is the classic dumb mistakes involving these statements. Always remember the rules stated below when writing or debugging your VHDL code and you'll save yourself a lot of time. Make a note of this section as one you may want to reread once you've had a formal introduction to these particular statements.

- Every **`if`** statement has a corresponding **`then`** component
- Each **`if`** statement is terminated with an "**`end if`**"
- If you need to use an "`else if`" construct, the VHDL version is "**`elsif`**"
- Each **case** statement is terminated with an "**`end case`**"
- Each **`loop`** statement has a corresponding "**`end loop`**" statement

3.7 Identifiers

An *identifier* refers to the name given to discern various items in VHDL. Examples of identifiers in higher level languages include variable names and function names. Examples of identifiers in VHDL include variable names, signal names, and port names (all of which will be discussed soon). Listed below are the hard and soft rules, i.e., you must follow them or you should follow them, regarding VHDL identifiers. Remember, intelligent choices for identifiers make your VHDL code more readable, understandable, and more impressive to coworkers, superiors, family, and friends. People should quietly mumble to themselves "this is impressive looking code… it must be good". A few examples of both good and bad choices for identifier names appear in Table 1.

- Identifiers should be self-commenting. In other words, the text you apply to identifiers should provide information as to the use and purpose of the item the identifier represents.
- Identifiers can be as long as you want (contain many characters). Shorter names make for more readable code, but longer names present more information. It's up to the designer to choose a reasonable identifier length.
- Identifiers can only contain some combination of letters (A-Z and a-z), digits (0-9), and the underscore character ('_').
- Identifiers must start with an alphabetic character.
- Identifiers must not end with an underscore and must never have two consecutive underscores.

Valid Identifiers		**Invalid Identifiers**	
`data_bus_val`	descriptive name	`3Bus_val`	begins with numeric character
`WE`	classic "write enable" acronym	`DDD`	not self-commenting
`div_flag`	a real winner	`mid_$num`	contains illegal character
`port_A`	provides some info	`last__value`	contains consecutive underscores
`in_bus`	input bus (a good guess)	`start_val_`	ends with underscore
`clk`	classic system clock name	`in`	uses VHDL reserved word
		`@#$%%$`	total garbage
		`this_sucks`	possibly true but try to avoid
		`Big_vAlUe`	valid way too ugly
		`pa`	possibly lacks meaning
		`sim-val`	illegal character (dash)

Table 1: Desirable and undesirable identifiers.

3.8 Reserved Words

There is a list of words that have been assigned special meaning by the VHDL language. These special words, usually referred to as reserved words, can therefore not be used for purposes of identifiers by person writing the VHDL code. A partial list of reserved words that you may be

more inclined to use appears Table 2. A complete list of reserved words appears in Appendix A. Notably missing from Table 2 are standard operator names such as AND, OR, XOR, etc.

access	**exit**	**mod**	**return**	**while**
after	**file**	**new**	**signal**	**with**
alias	**for**	**next**	**shared**	
all	**function**	**null**	**then**	
attribute	**generic**	**of**	**to**	
block	**group**	**on**	**type**	
body	**in**	**open**	**until**	
buffer	**is**	**out**	**use**	
bus	**label**	**range**	**variable**	
constant	**loop**	**rem**	**wait**	

Table 2: A short list of VHDL reserved words.

3.9 VHDL Coding Style

Coding style refers to the appearance of the VHDL source code. Obviously, the freedom provided by case insensitivity and indifference to white space creates a virtual coding anarchy. The emphasis in coding style is therefore placed on readability. Unfortunately, the level of readability of any document is subjective. Writing VHDL code is similar to writing code in other computer languages such as C and Java in that you have the ability to make the document more readable without changing the function of the document. This is primarily done through the use of indenting certain portions of the program and commenting where necessary.

Instead of stating a bunch of rules for you to follow as to how your document should look, you should instead strive to simply make your source code *readable*. Listed below are a few thoughts on the notion of a readable document.

- Chances are that if your VHDL source code is readable to you, it will be readable to others who may need to peruse your document. These other people may include someone who is helping you get the code working properly, someone who is assigning a grade to your document, or someone who signs your paycheck at the end of the day. These are the people you want to please.

- Your VHDL source code should be modeled after some other VHDL document that you find organized and readable. Any code you look at that is written down somewhere is most likely written by someone with more VHDL experience than a beginner such as yourself. Emulate the good parts of their style.

- Adopting a good coding style helps you write code without mistakes. As with other compilers you have experience with, you'll find that the VHDL compiler does a great job at knowing a document has error but a marginal job at telling you where or what the error is. Using a consistent coding style enables you to find errors both before compilation and after the compiler has found an error.

4 Basic VHDL Design Units

The “black box” approach to any type of design implies a hierarchical approach where varying amounts of detail are available at each of the different levels of the hierarchy. In the black box approach, units of action which share a similar purpose are grouped together and abstracted to a higher level. Once this is done, the module is referred to by its inherently more simple black box representation rather than thinking about the details of the circuitry that actually performs that functionality. This approach has two main advantages. First, it simplifies the design from a systems standpoint. Examining a circuit diagram containing appropriately named black boxes is much more understandable than staring at a circuit containing a countless number of logic gates. Second, the black box approach allows for the reuse of previously written and working code.

Not surprisingly, VHDL descriptions of circuits are based upon the black box approach. The two main parts of any hierarchical design are the black box and the stuff that goes in the black box. In VHDL, the black box is referred to as the *entity* and the stuff that goes inside is referred to as the *architecture*. For this reason, the VHDL entity and architecture are closely related. As you probably can imagine, creating the entity is relatively simple while a good portion of the VHDL learning curve is spent describing the architecture. Our approach here is to present an introduction to writing VHDL code by describing the *entity* and the moving onto the detail of writing the *architecture*. Familiarity with the *entity* will hopefully aid in your learning of the techniques to describing the architecture.

4.1 The Entity

The *entity* is VHDL’s version of the black box. The VHDL entity construct provides a method to abstract the functionality of a circuit description to a higher level. It provides a simple “wrapper” for the lower-level circuitry. This wrapper effectively describes how the black box interfaces with the outside world. Since VHDL is describing a digital circuit, the entity simply lists the various input and outputs to the underlying circuitry. In VHDL terms, the black box is described by an *entity declaration.* The syntax[2] of the entity declaration is shown in Figure 5.

```
entity entity_name is
   [port_clause]
end entity_name;
```

Figure 5: Generic form of an entity declaration.

The *entity_name* provides a method to reference the entity. The port_clause specifies the actual interface of the entity. The syntax of the port_clause is shown in Figure 6.

[2] The bold font is used to describe VHDL keywords while italics are used to show names that are supplied by the writer of the VHDL code.

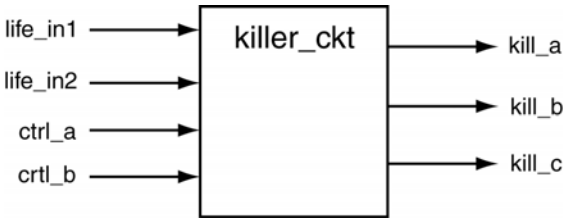
```
port (
   port_name : mode  data_type;
   port_name : mode  data_type;
   port_name : mode  data_type
      );
```

Figure 6: Syntax of the port_clause.

A "port" is essentially a signal that interfaces with the outside word. This signal can be either an input to the underlying circuit or an output from the underlying circuit to the outside world. The port_clause is nothing more than a list of the signals from the underlying circuit that are available to the outside world. The *port_name* is an indentifier used to differentiate the various signals. The *mode* specifies the direction of the signal relative to the entity where signals can either enter (inputs) or exit (outputs) the box. These input and output signals are associated with the keywords **in** and **out**, respectively. The *data_type* refers to the type of data that the port will handle. There are many data types available in VHDL though we'll deal primarily with the ***std_logic*** type. Information regarding the various VHDL data types will be discussed later.

Figure 7 shows an example of a black box and it VHDL code used to describe it. Listed below are a few things to note about the code in Figure 7. Most of things to note regard the readability and understandability of the VHDL code. The bolding of the VHDL keywords is done to remind you what the keywords are and have no function in actual VHDL code.

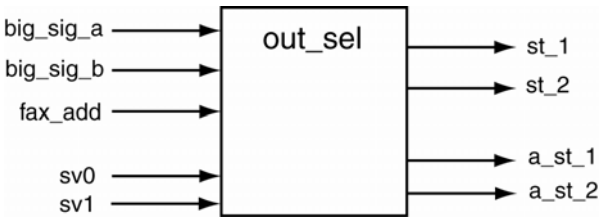
- Each port name is unique and has an associated mode and type.
- The VHDL compiler allows several port names to be included on a single line. The port names are delineated by commas.
- The inputs and outputs are listed consecutively in the entity.
- There is a feeble attempt to line up the colons in each *port_clause* line which makes the code more readable. Remember that white space is ignored by the compiler.
- A comment was included which almost says intelligent things.




```
------------------------------------------------------------------
-- Here's my interface description of the killer circuit
-- It does a lot of killer things.
------------------------------------------------------------------
entity killer_ckt is
   port ( life_in1 : in std_logic;
          life_in2 : in std_logic;
            crtl_a, ctrl_b : in std_logic;
            kill_a : out std_logic;
            kill_b : out std_logic;
            kill_c : out std_logic);
end killer_ckt;
```

Figure 7: Example black box and associated VHDL entity declaration.

Figure 8 provides another example of a black box diagram and its associated entity declaration. All of the ideas noted in Figure 7 are equally applicable in Figure 8.




```
------------------------------------------------------------
-- out_sel is used to select one inputs based on the
-- conditions of sv0 and sv1 blah blah blah
------------------------------------------------------------
entity out_sel is
   port (big_sig_a, big_sig_b : in  std_logic;
                      sv0, sv1 : in  std_logic;
                      fax_add  : in  std_logic;
                  st_1, st_2 : out std_logic;
              a_st_1, a_st_2 : out std_logic);
end out_sel;
```

Figure 8: An example of an input and output diagram of a circuit and its associated VHDL entity.

Hopefully, you're not finding these entity specifications too challenging. In fact, they're so straight forward, we'll throw in one last twist before we leave the realm of VHDL entities. Most the more meaningful circuits that you'll be designing, analyzing, and testing using VHDL have many similar and closely related inputs and outputs. These are commonly referred to as *bus* signals in computer lingo. Bus lines are made of more than one signal that differ in name by only a numeric character. In other words, each separate signal in the bus name contains the bus name plus a number to separate it from other signals in the bus. Individual bus signals are referred to as *elements* of the bus. As you would imagine, busses are used often in digital circuits.

Busses are easily described in the VHDL entity. All that is needed is new data type and a special notation to indicate when a signal is a bus or not. A few examples are shown in Figure 9. In these examples note that the mode remains the same but the type has changed. The ***std_logic*** data type now includes the word *vector* to indicate each signal name contains more than one signal. There are ways to reference individual members of each bus but we'll get to those details later.

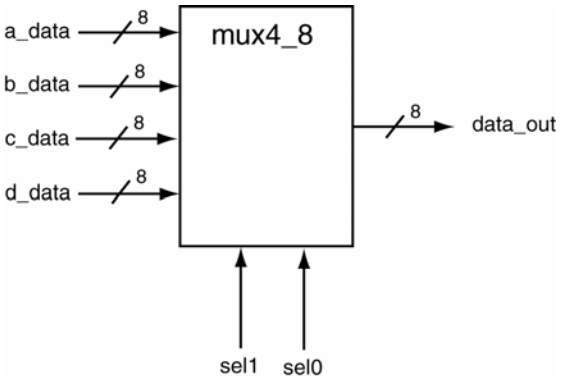
```
magic_in_bus     : in std_logic_vector(0 to 3);
big_magic_in_bus : in std_logic_vector(7 downto 0);
tragic_in_bus    : in std_logic_vector(16 downto 1);
data_bus_in_32   : in std_logic_vector(0 to 31);
mux_out_bus_16   : out std_logic_vector(0 to 15);
addr_out_bus_16  : out std_logic_vector(15 downto 0);
```

Figure 9: A few examples of bus signals of varying content.

As you can see by examining Figure 9, there are two possible methods to describe the signals in the bus. These two methods are shown in the argument lists that follow the data type declaration. The signals in the bus can be listed in one of two orders which is specified by the *to* and *downto*

keyword. Producing VHDL code with greater clarity should decide which of these orientations to use. Be sure to remember the orientation of signals when you are using them to describe your circuit.

A more appropriate introduction to busses would be to see how this notation is used to describe an actual black box. Figure 10 shows a black box followed by its entity declaration. Note that the black box uses a slash/number notation to indicate that the signal is a bus. The slash across the signal line indicates the signal is a bus and the associated number specifies the number of signals on the bus. Worthy of mention regarding the black box of Figure 10 is that the input lines *sel1* and *sel0* could have been made into a bus containing two signals.




```
-------------------------------------------------------------------------
-- Unlike the other examples, this is actually an interface
-- for a MUX that selects one of four bus line for the
-- output.
-------------------------------------------------------------------------
entity mux4_8 is
   port (  a_data : in  std_logic_vector(0 to 7);
           b_data : in  std_logic_vector(0 to 7);
           c_data : in  std_logic_vector(0 to 7);
           d_data : in  std_logic_vector(0 to 7);
        sel1,sel0 : in  std_logic;
   a_st_1, a_st_2 : out std_logic_vector(0 to 7));
end mux4_8;
```

Figure 10: A black box example containing busses and its associated entity declaration.

4.2 The Architecture

The VHDL entity declaration describes the interface or the external representation of the circuit. The *architecture* is describes what the circuit actually does. In other words, the VHDL architecture describes the internal implementation of the associated entity. As you can probably imagine, describing the external interface to a circuit is generally much easier than describing how the circuit is intended to operate. This statement becomes even more important as the circuits you're describing become more complex.

The most challenging part about learning VHDL is learning the myriad of ways to describe a circuit. The bulk of this tutorial is centered about the different methods available to describe circuits so not too much more will be mentioned about VHDL architectures at this point. Examples of VHDL entity-architecture pairs are presented throughout the remainder of this tutorial and none will be presented here. A few general statements regarding VHDL architectures are presented below.

- There can be any number of architectures describing a single entity. As you'll eventually discover, the VHDL coding style used in the architectures have a significant effect on the circuit is synthesized (how the circuit will be implemented in an actual device). This allows the VHDL code the flexibility of designing with specific characteristics such as particular physical size or operational speed.

- There are several common models that architectures can use to describe circuits. These are *dataflow*, *behavioral*, and *structural* models as well as hybrid versions of these models. These models are described in later sections of this tutorial.

5 The VHDL Programming Paradigm

The last section introduced the idea of the basic design units of VHDL: the entity and the architecture. Most all the time was spent describing the entity simply because there is so much less involved when compared to the architecture. Remember, the entity declaration is used to describe the interface of a circuit to the outside world. The architecture is used to describe how the circuit is intended to function.

Before we get into the details of architecture specification, we must step back and remember what it is we're trying to do with VHDL. We are, for one reason or another, describing a digital circuit. Realizing this is massively important. The tendency for students with computer programming backgrounds is to view VHDL as just another programming language they need to learn to pass another class. Although many students have used this approach to pass the basic digital classes, this is a bad approach. When viewed correctly, VHDL represents a completely different approach to programming. This problem most likely arises because VHDL has many similarities to other programming languages. The main similarity is that they both use a syntactical and rule-based language to describe something abstract. But, the difference is that they are describing two different things. Realizing this fact will help you to truly understand the VHDL programming paradigm and language, to churn out more meaningful VHDL code, and illuminate a nice contrast between a language that describes hardware and the language used to execute software on that hardware.

5.1 Concurrent Statements

The heart of most programming languages are the statements that form a majority of the associated source code. These statements represent finite quantities of "actions" to be taken. A statement in an algorithmic programming language such as C or Java represents an action to be taken by the processor. Once the processor finishes one action, it moves onto the next action specified somewhere in the associated source code. This makes sense and is comfortable to us as humans because just like the processor, we generally are only capable of doing one thing at a time. This description lays the foundation for an algorithm in that the processor does great job at following a set of rules which are essentially the direction provided by the source code. When the rules are meaningful, the processor can do amazing things.

VHDL programming is different. Whereas we view a processor to step one-by-one through a set of statements, VHDL has the ability to "execute" a virtually unlimited number of statements at the same time. The key to remember here is that we are designing hardware. Parallelism, or things happening *concurrently*, in the context of hardware is a straight-forward concept. You're most likely already both familiar and comfortable with this concept.

Figure 11 shows a simple example of a circuit that operates in parallel. As you know, the gates are generally stupid in the output of the gates are a function of the gate inputs. Anytime the input changes, there is a possibility that the output will change. This is true of all the gates in Figure 11 or in any digital circuit in general. The key here is that the changes in the input to these gates can happen in at the same time.

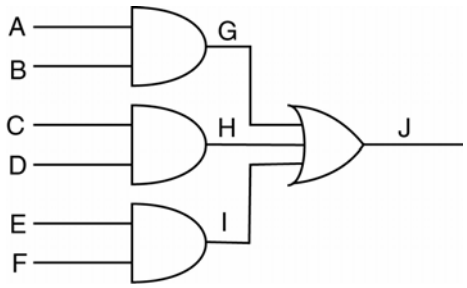


Figure 11: Some common circuit that is well known to "execute" parallel operations.

So here's the trick. Since most of us are human, we're only capable of reading one line of text at a time and in a sequential manner. This same limitation follows us around when we try to write some text, not to mention entering some text into a computer. So how then are we going to describe some "thing" that is inherently parallel? We don't have this problem when discussing something inherently sequential such as standard algorithmic programming. When writing code using an algorithmic programming language, there is generally only one processing element to do all the work. The processing element generally does only one thing at a time which fits nicely with our basic limitation as humans.

The VHDL programming paradigm built around the concept of expression parallelism and concurrency with textual descriptions of circuits. The heart of VHDL programming is the *concurrent statement*. These are statements that look a lot like the statements in algorithmic languages but they are significantly different because they express concurrency of execution. As a brief introduction to this concept, the code that implements the circuit shown in Figure 11 is listed for your convenience.

Figure 12 lists the code that implements the circuit shown in Figure 11. This code shows four *concurrent signal assignment* statements. The "<=" construct is referred to as a *signal assignment operator*. The reality is that we can't write these four statements at the same time but we can interpret these statements as actions that are happening at the same time or *concurrently*. Once again, the concept of concurrency is a key concept in VHDL. Keep this in mind that anytime you are dealing with VHDL code. If you feel that algorithmic style of thought creeping into your soul, try to snap out of it quickly. Concurrent signal assignment will be discussed more completely in the next section.

```
G <= A AND B;
H <= C AND D;
I <= E AND F;
J <= G OR H OR I;
```

Figure 12: VHDL code that describes the circuit of Figure 11.

Figure 13 shows some "C" code that looks similar to the code listed in Figure 12. In this case, the logic functions were replaced with addition operators and the signal assignment operators were replaced by the assignment operator. The statements in this piece of code are executed sequentially as opposed to concurrently as is the case for the VHDL code of Figure 12. Once again, although the two snippets of code look somewhat similar, they have completely different meanings.

```
G =  A + B;
H =  C + D;
I =  E + F;
J =  G + H + I;
```

Figure 13: Algorithm code similar to the code in Figure 12.

5.2 The Signal Assignment Operator "<="

Algorithmic programming languages always have some type of *assignment* operator. In "C", this is the well-known "=" sign. In these languages, the assignment operator signifies a transfer of data from the right side of the operator to the left side. VHDL uses two consecutive characters to represent the assignment operator: "<=". This combination was chosen because it is different from the assignment operators in most other common algorithmic programming languages. The operator is officially known as a *signal assignment* operator to highlight its true purpose. The signal assignment operator specifies a relationship between signals. In other words, the signal on the left side of the signal assignment operator is dependent upon the signals on the right side of the operator.

With these new insights into VHDL, you should be able to understand the code of Figure 12 and its relationship to Figure 11. The statement "`G <= A AND B;`" indicates that the value of the signal named "G" represents an ANDing of the signals A and B. The similar statement in written in an algorithmic programming language, "`G = A + B;`" indicates that the value represented by variable A is added to the value represented by variable B and the result is then represented by variable G. The distinction between these two types of statements should be becoming clearer.

There are four types of concurrent statements that are examined in this tutorial. We've already briefly discussed the concurrent signal assignment statement and we'll soon examine it further and put it in context of an actual circuit. The three other types of concurrent statements that of immediate interest to us are *process statements, conditional signal assignments,* and *selected signal assignments.*

In essence, the four types of statements represent tools which you can use to implement digital circuits. You will soon be discovering the versatility of these statements. Unfortunately, this versatility effectively adds a fair amount of steepness to the learning curve. As you know from your experience in other programming languages, there are always multiple ways to do the same things. Stated differently, several seemingly different pieces of code can actually produce the same result. The same is true for VHDL code. Keep this in mind when you look at any of the examples given in this tutorial. Any VHDL code used to solve a problem is more than likely one of many possible solutions to that problem.

5.3 Concurrent Signal Assignment Statements

The general form of a concurrent signal assignment statement is shown in Figure 14. In this case, *target* is a signal that receives the values of the *expression*. An expression is defined by either a constant, a signal, or an set of operators that operate on other signals and evaluate to some value. Examples of expressions used in VHDL code are shown in the examples that follow.

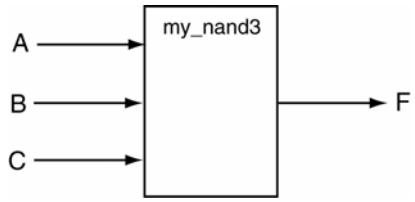
```
target <= expression;
```

Figure 14: Syntax for the concurrent signal assignment statement.

EXAMPLE 1

Write the VHDL code to implement a three input NAND gate. The three input signal names are A, B, and C while the output signal name is F.

Solution: It's good practice to always draw a diagram of the thing you're designing. Furthermore, though we could draw a diagram showing the familiar symbol for the NAND gate, we'll choose to keep the diagram general and take the black box approach instead. Remember, the black box is a nice aid when it comes to writing the entity declaration.




```
-- header files and library stuff
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity my_nand3 is
    port ( A,B,C : in std_logic;
                F : out std_logic);
end my_nand3;

architecture ex_nand3 of my_nand3 is
begin
    F <=  NOT (A AND B AND C);
--
--  An alternative approach for this statement:
--  F <= A NAND B NAND C;
--
end ex_nand3;
```

Figure 15: Solution to *EXAMPLE 1*

This example contains a few new ideas that are worth further mention.

- There are header files and library files that must be included in your VHDL code in order for your code to correctly compile. These few lines of code are listed at the top of the code in Figure 15. The listed lines are more than is needed for this example but they will be required in later examples.

- This example highlights the use of several logic operators. The logic operators available in VHDL are **AND**, **OR**, **NAND**, **NOR**, **XOR**, and **XNOR**. The **NOT** operator is technically speaking not a logic but is available also.

EXAMPLE 1 demonstrates the use of the concurrent signal assignment (CSA) statement in a working VHDL program. But since there is only one CSA statement, the concept of concurrency is not readily apparent. The idea behind this statement is that the output is changed anytime one of the input signals changes. In other words, the output F is reevaluated anytime a signal on the input expression changes. The idea of concurrency is more clearly demonstrated in the *EXAMPLE 2*.

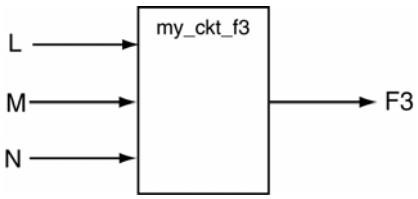
EXAMPLE 2

Write VHDL code to implement the function expressed in the following truth table.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: The first step in the process is to reduce the given function. While this is not mandatory it may help shorten your time spent entering VHDL code. We hope the VHDL compiler would somewhere along the way automatically reduce such functions but that is probably too much to hope for. The reduced equation is given below. The black box diagram and associated VHDL code is shown in Figure 16.

$$F3 = \overline{L}\,\overline{M}N + LM$$




```
--------------------------------------------------------------------
-- header files and library stuff: It needs to
-- be here but it has been deleted to save space
--------------------------------------------------------------------
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
              F3 : out std_logic);
end my_ckt_f3;

architecture f3_1 of my_ckt_f3 is
   signal A1, A2 : std_logic; -- intermediate signals
begin
   A1 <= ((NOT L) AND (NOT M) AND N);
   A2 <= L AND M;
   F3 <= A1 OR A2;
end f3_1;
```

Figure 16: Solution to *EXAMPLE 2*.

The previous example contains a few new ideas and concepts. Note that the library files and header information no longer appears in the example. This will be true with the remainder of the examples in this tutorial. This information is redundant and somewhat system specific. More importantly, this example demonstrates the use of signal statements. These statements are used to provide what some people prefer to call intermediate results. This approach is analogous to declaring extra variables in an algorithmic programming language to be used for specifically for storing intermediate results. Note that the signal statements are similar to the port clause statement appearing in the entity declaration except the mode specification is missing. This approach is not mandatory by any means in that this code can be written without using the signal statements as shown in architecture declaration of Figure 17.

```
architecture f3_2 of my_ckt_f3 is
begin
   F3 <= ((NOT L) AND (NOT M) AND N) OR (L AND M);
end f3_2;
```

Figure 17: Alternative but functionally equivalent architecture for *EXAMPLE 2*.

Despite the fact that the architectures of f3_1 and f3_2 of Figure 16 and Figure 17 appear different, they are functionally equivalent. This is because all the statements are concurrent signal assignment statements. Despite the fact that the f3_1 architecture contains three CSAs, they are functionally equivalent to the CSA in f3_2 because each of the three statements is effectively executed *concurrently*.

EXAMPLE 2 demonstrates that you can easily convert a function listed in truth-table format to VHDL code. The conversion of the simplified function to CSAs was somewhat straight-forward. The ease at which these functions can be implemented into VHDL code was almost trivial. But

then again, the function in *EXAMPLE 2* was not overly complicated. The point is that CSAs are useful statements. But as functions become more complicated (more inputs and outputs), an equation entry approach such as this becomes tedious. Luckily, there are a few other types of concurrent constructs that mitigate the tedium.

5.4 Conditional Signal Assignment

Concurrent signal assignment statements associated one target with one expression. The term *conditional signal assignment* is used to describe statements that have only one target but can have more than one associated expression. Each of the expressions is associated with a certain condition. The individual conditions are evaluated sequentially until the first condition evaluates to TRUE. In this case, the associated expression is evaluated and assigned to the target. Only one assignment is used per assignment statement.

The syntax of the conditional signal assignment is shown below. The *target* in this case is the name of a signal. The *condition* is based upon the state of some other signals in the given circuit. Note that there is only one signal assignment operator associated with the conditional signal assignment statement.

```
target <= expression when condition else
          expression when condition else
          expression;
```

Figure 18: Snytax for the conditional signal assignment statement.

The conditional signal assignment statement is probably easiest to understand in the context of a circuit. For our first example, let's simply redo the *EXAMPLE 2* using conditional signal assignment instead of concurrent signal assignment.

EXAMPLE 3

Write VHDL code to implement the function expressed in the following truth table. Use only conditional signal assignment statements in your VHDL code.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: The entity declaration does not change from *EXAMPLE 2* so the solution only needs a new architecture description. The solution is listed in Figure 19.

```
architecture f3_3 of my_ckt_f3 is
begin
   F3 <= '1' when (L = '0' AND M = '0' AND N = '1')  else
         '1' when (L = '1' AND M = '1') else
         '0';
end f3_3;
```

Figure 19: Solution to *EXAMPLE 3.*

There are a couple of interesting points to note about this solution.

- It's not much of an improvement over the VHDL code written using concurrent signal assignment. In fact, it looks a bit less efficient in terms of the amount of stuff in the code.

- If you look carefully at this code and notice that there is in fact one target and a bunch of expressions and conditions. The last expression is the catch all condition in that if none of the above conditions evaluate as TRUE, the last expression is assigned to the target.

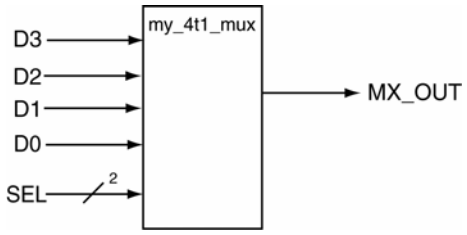
- We've used a *relational operator*. There are actually six different relational operators available in VHDL. The only two we will use in this paper are the "=" and "/=" relational operators which are the "is equal to" and "is not equal to" operators, respectively. More on operators is mentioned later.

There are more intelligent uses of the conditional signal assignment statement. One of the classic one is for the implementation of a MUX. The next example is one of the classics.

EXAMPLE 4

Write the VHDL code that implements a 4:1 MUX using a single conditional signal assignment statement. The inputs to the MUX are data inputs D3, D2, D1, D0 and a two-input control bus SEL. The single output is MX_OUT.

Solution: For this example we need to start from scratch. This of course included the now famous black box diagram and the associated entity statement. The VHDL portion of the solution is shown in Figure 20.




```
------------------------------------------------------------------
-- entity and architecture of 4:1 Multiplexor implemented using
-- conditional signal assignment.
------------------------------------------------------------------
entity my_4t1_mux is
    port ( D3,D2,D1,D0 : in std_logic;
                  SEL : in std_logic_vector(1 downto 0);
               MX_OUT : out std_logic);
end my_4t1_mux;

architecture mux4t1 of my_4t1_mux is
begin
   MX_OUT <= D3 when (SEL = "11") else
             D2 when (SEL = "10") else
             D1 when (SEL = "01") else
             D0 when (SEL = "00") else
             '0';
end mux4t1;
```

Figure 20: Solution for *EXAMPLE 4*: A 4:1 MUX using a conditional signal assignment statement.

There are a couple of things to note in the solution provided in Figure 20.

- The solution looks somewhat efficient compared to the amount of logic that would have been required if CSA statements were used. The VHDL code appears nice and is pleasing to the eyes which are qualities required for readability.

- The "=" relational operator is used in conjunction with a bus signal. In this case, the values on the bus SEL bus lines are accessed using double quotes around the specified values.

- For the sake of completeness, we've included every possible condition for the SEL signal plus a catch-all **else** statement. We could have changed the line containing '0' to D0 and removed the line associated with the SEL condition of "00". This would be functionally equivalent to the solution shown in but not nearly as impressive looking.

Remember, a conditional signal assignment is a type of concurrent statement. In this case, the statement is executed any time a change occurs on the conditional signals. This is similar to the concurrent signal assignment statement where the statement is executed any time there is a change in any of the signals listed on the right side of the signal assignment operator.

Though it's still early in the VHDL learning game, you've been exposed to a lot of concepts and syntax. The conditional signal assignment is maybe a bit less intuitive than the concurrent signal assignment. There is an alternative way to think of it though. If you think about it, the conditional signal assignment statement is similar in function to **if-else** constructs from programming

languages. We'll touch more upon the relationship once we start talking about sequential statements.

5.5 Selected Signal Assignment

Selective signal assignment statements are the third form of concurrent statements that we'll examine. These statements can have only one target signal and only one expression determines what the choices are based upon. The syntax for the selected signal assignment statement is shown below.

```
with choose_expression select
   target <= {expression when choices, }
             expression when choices;
```

Figure 21: Syntax for the selected signal assignment statement.

EXAMPLE 5

Write VHDL code to implement the function expressed in the following truth table. Use only selected signal assignment statements in your VHDL code.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: This is yet another version of the my_ckt_f3 example originally appearing in *EXAMPLE 2*. The black box diagram and the entity declaration for this example are shown in Figure 16 . The solution is shown in Figure 22.

```
architecture f3_4 of my_ckt_f3 is
begin
   with ( (L = '0' AND M = '0' and N = '1') or
          (L = '1' AND M = '1') ) select

     F3 <= '1' when '1',
           '0' when '0',
           '0' when others;
end f3_4;
```

Figure 22: Solution to *EXAMPLE 5*.

EXAMPLE 6

Write the VHDL code that implements a 4:1 MUX using a single selected signal assignment statement. The inputs to the MUX are data inputs D3, D2, D1, D0 and a two-input control bus SEL. The single output is MX_OUT.

Solution: This is a repeat of *EXAMPLE 4* except a selected signal assignment operator is used instead of a conditional signal assignment operator. The entity declaration does not change but is listed again in Figure 23. The black box diagram for this example is the same as shown in Figure 20 and is not repeated here.

```
-------------------------------------------------------------------
-- entity and architecture of 4:1 Multiplexor using selective
-- signal assignment.
-------------------------------------------------------------------
entity my_4t1_mux is
    port ( D3,D2,D1,D0 : in std_logic;
                  SEL : in std_logic_vector(1 downto 0);
               MX_OUT : out std_logic);
end my_4t1_mux;

architecture mux4t1_2 of my_4t1_mux is
begin
   with SEL select
   MX_OUT <= D3  when "11",
             D2  when "10",
             D1  when "01",
             D0  when "00",
             '0' when others;
end mux4t1_2;
```

Figure 23: Solution to *EXAMPLE 6*.

Once again, there are a few things of interest in the solution of *EXAMPLE 6* which are listed below.

- The VHDL code has several similarities to the solution of *EXAMPLE 4*. The general appearance is the same. Both solutions are also much more efficient than the solution would have been if only CSAs were used.

- A different catch-all expression is in the selected signal assignment statement. As opposed to the final else in the conditional assignment statement, the selected signal assignment statement uses the **when others** VHDL keyword. In the case of *EXAMPLE 4*, the output is assigned the constant value of '0' when the other listed conditions of the *chooser_expression* are not met.

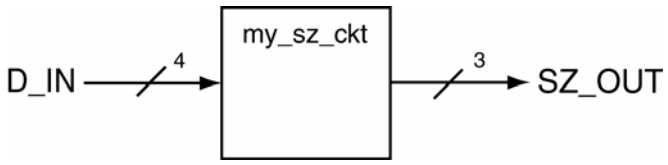
- The circuit used in this example was a 4:1 MUX. In this case, each of the conditions of the *chooser_expression* is accounted for in the body of the selected signal assignment statement. This is not a requirement however. The only requirement here is that the line containing the **when others** keywords appears in the final line of the statement.

EXAMPLE 7

Write the VHDL code that implements the following circuit. The circuit contains an input bus containing four signals and an output bus containing three signals. The input bus, D_IN, represents a 4-bit binary number. The output bus, SZ_OUT, is used to indicate the magnitude of the 4-bit binary input number. The relationship between the input and output is shown in the table below. Use a selected signal assignment statement in the solution.

input range of D_IN	output value for SZ_OUT
0000 → 0011	100
0100 → 1001	010
1010 → 1111	001
unknown condition	000

Solution: The solution to *EXAMPLE 7* is shown in Figure 24.




```
------------------------------------------------------------------
-- A decoder-type circuit for example 7 using selective signal
-- assignment
------------------------------------------------------------------
entity my_ckt is
    port ( D_IN   : in std_logic_vector(3 downto 0);
           SX_OUT : out std_logic_vector(2 downto 0));
end my_ckt;

architecture spec_dec of my_ckt is
begin
   with D_IN select
   SX_OUT <= "100"  when "0000" | "0001" | "0010" | "0011",
             "010"  when "0100" | "0101" | "0110" | "0111"
                               | "1000" | "1001",
             "001"  when "1010" | "1011" | "1100" | "1101"
                               | "1110" | "1111",
             "000"  when others;
end spec_dec;
```

Figure 24: Solution to *EXAMPLE 7*.

The only comment for the solution of *EXAMPLE 7* is that the vertical bar is used as a selection character in the *choices* section of the selected signal assignment statement. This increases the readability of the code as it does with the similar constructs in algorithmic programming languages

Once again, the selected signal assignment statement is one form of a concurrent statement. This is verified by the fact that there is only one signal assignment statement in the body of the selected signal assignment statement. The selected signal assignment statement is evaluated each time there is a change in the *chooser_expression*

The final comment regarding selected signal assignment is similar to the final comment regarding selected signal assignment. You should recognize the general form of the selected signal assignment statement to be similar to *switch* statements in algorithmic programming languages such as “C” and Java. Once again, this relationship is mentioned in much more depth once we are ready to talk about sequential statements.

5.6 The Process Statement

The process statement is the final concurrent statement we’ll take a look at. Before we do that, however, we need to take a few steps back and explore a few other VHDL principles and definitions that we’ve been excluding up to now. Remember, there are a thousand ways to learn things. This is especially true when learning programming languages where there are usually many different and varied solutions to the same problem. This difference is highlighted by the many different and varied approaches that appear VHDL books and tutorials that exist. So… now is not the time to learn about the process statement. We’ll do that right after we pick up a few more standard VHDL concepts.

6 Standard Architectures in VHDL

As you may remember, the VHDL architecture describes the function of some VHDL entity declaration. The architecture is comprised of two parts: the declaration section followed by a collection of concurrent statements. We've studied three types of concurrent statements thus far: concurrent signal assignment, conditional signal assignment, and selected signal assignment. Concurrent statements pass information to other concurrent statements though signals. We were about to described another concurrent statement, the *process* statement, before we got side-tracked here in this section.

There are three different approaches to writing VHDL architectures. These approaches are known as *dataflow style*, *structural style*, and *behavioral style* architectures. The standard approach to learning VHDL is to introduce each of these architectural styles individually and design a few circuits using that style. Although this approach is good from the standpoint of keeping things simple while immersed in the learning process, it's also somewhat misleading because more complicated VHDL circuits generally use a mix of these three styles. Keep this fact in mind in the following discussion of these styles. We will, however, put most of our focus on dataflow and behavioral architectures. Structural modeling is essentially a method to combine a set of VHDL models. For this reason, it is less of a design method and more of an approach for interfacing previously design modules.

The reason we choose to slip the discussion of the different architectures at this point is that you already have some familiarity with one of the styles. Up to this point, all of our circuits have used dataflow style of architectures. We're now at the point of talking about behavioral style of architectures which is primarily centered around the another concurrent statement known as a *process* statement. If it seems confusing, some of the confusion should go away once we start dealing with actual circuits and real VHDL code.

6.1 VHDL Dataflow Style Architecture

A *dataflow* style architecture specifies a circuit as a concurrent representation of the flow of data through the circuit. In the dataflow approach, circuits are described by showing the input and output relationships between the various built-in components of the VHDL language. The built-in components of VHDL include operators such as AND, OR, XOR, etc. The three forms of concurrent statements we've talked about up until now (concurrent signal assignment, conditional signal assignment, and selective signal assignment) are all statements that are found in dataflow style architectures. If you were to re-examine some of the examples we've done so far, you can in fact see how the data flows through the circuit. These signal assignment statements effectively describe how the data flows from the signals on the right side of the assignment operator (<=) to the signal on the left side of the operator.

The dataflow style of architecture has its strong points and weak points. It is good that you can see the flow of data in the circuit by examining the VHDL code. The dataflow models also allow you to make an intelligent guess as to how the actual logic will appear should you decide to synthesize the circuit. In other words, the dataflow style has a heavy influence on the synthesized hardware. The dataflow style works fine for small and primitive circuits. But as circuits become more complicated, it is usually advantageous to switch to behavioral style models.

6.2 VHDL Behavior Style Architecture

In comparison to the dataflow style architecture, the behavioral style architecture provides no details as to how the design is implemented in actual hardware. VHDL code written in a behavioral style does not necessarily reflect how the circuit is implemented when it is synthesized. Instead, the behavioral style models how the circuit outputs will react to (or behave) the circuit inputs. It is the VHDL synthesizer tool that decides the actual circuit description. In one sense, behavioral style modeling is the ultimate "black box" approach to designing circuits.

The heart of the behavioral style architecture is the *process* statement. This is fourth type of concurrent statement that we'll work with. As you will see, the process statement is significantly different from the other three concurrent statements in several ways. The major difference lies in the process statement's approach to concurrency, which is the major sticking point in learning to deal with this new concurrent statement.

6.3 The Process Statement

To understand the process statement, we'll first examine the similarities between it and the concurrent signal assignment statement. Once you grasp these similarities, we'll start discussing the differences between the statements.

The syntax for the process statement is shown in Figure 25. The main thing to notice in this figure is the body of the process statement comprises of *sequential statements*. The main difference between concurrent signal assignment statements and process statements lies with these sequential statements. But once again, let's stick to the similarities before we dive into the differences.

```
label: process(sensitivity_list)
begin
   {sequential_statements}
end process label;
```

Figure 25: Syntax for the process statement.

Figure 26 shows an entity declaration for a XOR function. Figure 27 shows both a dataflow and behavioral style of architecture for the entity of Figure 26. The main difference between the two architecture descriptions is the presence of the *process* statement in the listed code.

Recall that the concurrent signal assignment statement in the dataflow description operates as follows. Since it is a concurrent statement, anytime there is a change in the signals listed on the right side of the signal assignment operator causes the signal on the left side of the operator to be re-evaluated. For the behavioral architecture description, anytime there is a change in signals in process *sensitivity list*, all of the sequential statements in the process are evaluated. Evaluation of the process statement is controlled by the signals that are placed in the process sensitivity list. The concurrent signal assignment statement in the dataflow architecture is evaluated anytime there is a change in a signal on the right side of the signal assignment operator. The approaches are effectively the same except the syntax is significantly different.

So here's where it gets strange. Even though both of the architectures listed in Figure 27 have the exact same signal assignment statement (`F <= A XOR B;`), execution of the statement in the

behavioral style architecture is controlled by what signals appear in the process sensitivity list. The statement appearing in the dataflow model is evaluated anytime there is a change in signal A or signal B. This is a functional difference rather than a cosmetic difference.

```
entity my_xor_fun is
    port ( A,B : in std_logic;
               F : out std_logic);
end my_xor_fun;
```

Figure 26: Entity declaration for circuit performing XOR function.

```
architecture my_xor_dataflow of my_xor_fun is
begin
   F <= A XOR B;
end my_xor_dataflow;


architecture my_xor_behavioral of my_xor_fun is
begin
xor_proc: process(A,B)
   begin
      F <= A XOR B;
   end process xor_proc;

end my_xor_behavioral;
```

Figure 27: Dataflow and behavioral descriptions of *my_xor_fun* entity.

The other main difference between dataflow and behavioral architectures is that the body of the process statement contains only *sequential statements*. Our next order of business is to explore a few types of sequential statements.

6.4 Sequential Statements

The term “sequential statement” is derived from the fact that the statements within the body of a process are executed sequentially. Execution of the sequential statements (the statements appearing in the process body) is started when a change in the signal contained in the process sensitivity list occurs. Execution of statements within the process body continues until the end of the process body is reached.

There are three types of sequential statements that we’ll be discussing. We’ll not say too much about the first type though because we’ve already been dealing with it. The other two types of statements are the *if statement* and the *case statement*. The nice part about both of these statements is that you’ve worked with them before in algorithmic programming languages. The structure and function of the VHDL *if* and *case* statements is exactly the same. Keep this in mind when you read the descriptions that follow.

6.4.1 Signal Assignment Statements

The sequential style of a signal assignment statement is syntactically equivalent to the concurrent signal assignment statement. Another way to look at it is that if a signal assignment statement appears inside of a process than it is a sequential statement. Otherwise, it is a concurrent signal assignment statement. To drive the point home, the signal assignment statement in the dataflow style architecture of Figure 27 is a concurrent signal assignment statement while the same statement in the behavioral style architecture is a sequential signal assignment statement. The functional differences were already covered in the discussion regarding process statements.

6.4.2 IF Statements

The *if* statement is used to create a branch in the execution flow of the sequential statements. Depending on the conditions listed in the body of the *if* statement, either the instructions associated with one or none of the branches is executed then the *if* statement is processed. The general form of the *if* statement is shown in Figure 28.

```
if (condition) then
   { sequence of statements }
elsif (condition) then
   { sequence of statements }
else
   { sequence of statements }
end if;
```

Figure 28: Syntax for the *if* statement.

The concept of the *if* statement should be familiar to you in two regards. First, its form and function are similar to the *if*-genre of statements found in most algorithmic programming languages. The syntax, however, is a bit different. Secondly, the VHDL *if* statement is the sequential equivalent to the VHDL conditional signal assignment statement. These two statement essentially do the same thing but the *if* statement is a sequential statement found in a *process* body while the conditional signal assignment statement is one form of concurrent signal assignment.

Yet again, there are a couple of interesting things to note about the listed syntax for the *if* statement.

- The parenthesis placed around the *condition* expressions are optional. They should be included in most cases to increase the readability of the VHDL source code.

- Each *if* type statement contains an associated *then* keyword. The final *else* clause has no *then* keyword associated with it.

- As written in Figure 28, the *else* clause is a catch-all statement. If none of the previous conditions evaluate as true, then the sequence of statements associated with the final *else* clause is executed. The way the *if* statement is shown in Figure 28 guarantees that at least one of the listed sequence of statement will be executed.

- The final *else* clause is optional. Not including the final *else* clause presents the possibility that none of the sequence of statements associated with the conditions will be evaluated.

EXAMPLE 8

Write some VHDL code that implements the following function using an *if* statement:

$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC$$

Solution: Although it is not directly stated in the problem description, the VHDL code for this solution utilizes a behavioral architecture. This is because the problem states that an *if* statement should be used. The VHDL code for the solution is shown in Figure 29. We've opted to leave out the black box diagram in this case since the problem is relatively simple and thus does not demonstrate the power of behavioral modeling.

```
entity my_ex_7 is
    port ( A,B,C : in std_logic;
           F_OUT : out std_logic);
end my_ex_7;


architecture dumb_example of my_ex_7 is
begin
proc1: process(A,B,C)
   begin
      if (A = '1' and B = '0' and C = '0') then
         F_OUT <= '1';
      elsif (B = '1' and C = '1') then
         F_OUT <= '1';
      else
         F_OUT <= '0';
      end if;
   end process proc1;
end dumb_example;
```

Figure 29: Solution to *EXAMPLE 8*.

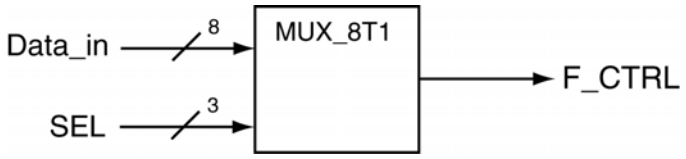
This is probably not the best was to implement a function but it does show an *if* statement in action. Just to drive the point further into the ground, an alternate architecture for the solution of *EXAMPLE 8* is shown in Figure 30. A more intelligent use of the *if* statement is demonstrated in *EXAMPLE 9*.

```
architecture bad_example of my_ex_7 is
begin
proc1: process(A,B,C)
   begin
      if (A = '0' and B = '0' and C = '0') or
         (B = '1' and C = '1') then
         F_OUT <= '1';
      else
         F_OUT <= '0';
      end if;
   end process proc1;
end bad_example;
```

Figure 30: An alternate solution for *EXAMPLE 8* .

EXAMPLE 9

Write some VHDL code that implements the 8:1 MUX shown in below. Use an *if* statement in your implementation.



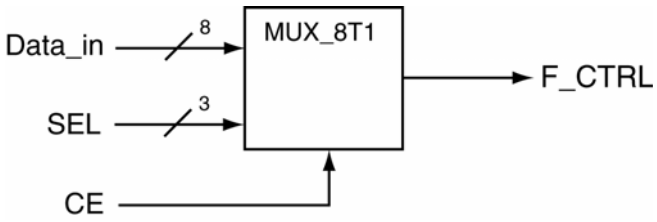
Solution: The solution to *EXAMPLE 9* is shown in Figure 31.

```
entity mux_8t1 is
    port ( Data_in : in std_logic_vector (7 downto 0);
               SEL : in std_logic_vector (2 downto 0);
            F_CTRL : out std_logic);
end mux_8t1;

architecture my_8t1_mux of mux_8t1 is
begin
my_proc: process (Data_in,SEL)
   begin
      if (SEL = "111")    then F_CTRL <= Data_in(7);
      elsif (SEL = "110") then F_CTRL <= Data_in(6);
      elsif (SEL = "101") then F_CTRL <= Data_in(5);
      elsif (SEL = "100") then F_CTRL <= Data_in(4);
      elsif (SEL = "011") then F_CTRL <= Data_in(3);
      elsif (SEL = "010") then F_CTRL <= Data_in(2);
      elsif (SEL = "001") then F_CTRL <= Data_in(1);
      elsif (SEL = "000") then F_CTRL <= Data_in(0);
      else  F_CTRL <= '0';
      end if;
   end process my_proc;
end my_8t1_mux;
```

Figure 31: Solution to *EXAMPLE 9*.

EXAMPLE 10

Write some VHDL code that implements the 8:1 MUX shown in below. Use as many *if* statements as you deem necessary to implement your design. In the black box diagram shown below, the CE input is a chip enable. When CE = '1', the output acts like the MUX of *EXAMPLE 9*. When CE is '0', the output of the MUX is '0'.



Solution: The solution to *EXAMPLE 10* is similar to the solution of *EXAMPLE 9*. Note in this solution that the *if* statements can be nested to attain various effects. The solution to *EXAMPLE 10* is shown in Figure 32.

```
entity mux_8t1_ce is
    port ( Data_in : in std_logic_vector (7 downto 0);
               SEL : in std_logic_vector (2 downto 0);
                CE : in std_logic;
           F_CTRL : out std_logic);
end mux_8t1_ce;


architecture my_8t1_mux of mux_8t1_ce is
begin
my_proc: process (Data_in,SEL,CE)
   begin
      if (CE = '0') then
          F_CTRL <= '0';
      else
         if (SEL = "111")    then F_CTRL <= Data_in(7);
         elsif (SEL = "110") then F_CTRL <= Data_in(6);
         elsif (SEL = "101") then F_CTRL <= Data_in(5);
         elsif (SEL = "100") then F_CTRL <= Data_in(4);
         elsif (SEL = "011") then F_CTRL <= Data_in(3);
         elsif (SEL = "010") then F_CTRL <= Data_in(2);
         elsif (SEL = "001") then F_CTRL <= Data_in(1);
         elsif (SEL = "000") then F_CTRL <= Data_in(0);
         else  F_CTRL <= '0';
         end if;
      end if;
   end process my_proc;
end my_8t1_mux;
```

Figure 32: Solution to *EXAMPLE 10*.

6.4.3 Case Statements

The *case* statement is somewhat similar to the *if* statement in that a sequence of statements are executed if an associated expression evaluates as true. The *case* statement differs from the *if* statement in that the resulting choice is made on depending upon the value of the single control expression. Only one of the set of sequential statements are executed for each execution of the *case* statement and is dependent upon the first *when* branch to evaluate as true. The syntax for the *case* statement is shown in Figure 33.

```
case (expression) is
   when choices =>
       {sequential statements}
   when choices =>
       {sequential statements}
   when others =>
       {sequential statements}
end case;
```

Figure 33: Syntax for the *case* statement.

Once again, the concept of the *case* statement should be familiar to you in several regards. First, its can somewhat be considered a different and more compact form of the *if* statement. It is not as functional, however, for the reason described above. Secondly, the *case* statement is similar in both form and function to case or switch statements in other algorithmic programming languages. And finally, the VHDL *case* statement is the sequential equivalent to the VHDL selected signal assignment statement. These two statements essentially have the same capabilities but the *case* statement is a sequential statement found in a *process* body while the selected signal assignment statement is one form of concurrent signal assignment. The "`when others`" line is not required.

EXAMPLE 11

Write some VHDL code that implements the following function using an *case* statement:

$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC$$

Solution: This falls into the category of not being the best way to implement a circuit using VHDL. It does, however, illustrate another useful feature in the VHDL. The first part of this solution requires that we list the function as a sum of minterm. This is down by multiplying the non-minterm product term given in the example by 1. In this case, 1 is equivalent to $(A + \overline{A})$. The operation is shown below.

$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC$$
$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC(A + \overline{A})$$
$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + ABC + \overline{A}BC)$$

The other part of the solution is shown in Figure 34. An interesting feature in this solution was the grouping of the three input signals which allowed for the use of a *case* statement in the solution. This approach required the declaration of an intermediate signal which was appropriately labeled "ABC". Once again, this was probably not the most efficient method to implement a function but it does highlight the need to be resourceful and creative when describing the behavior of digital circuits.

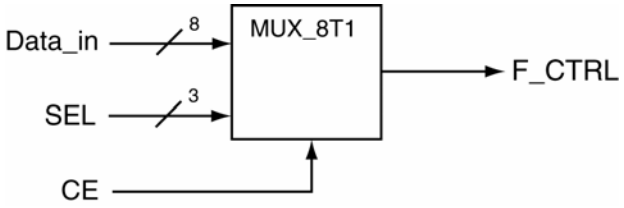
```
entity my_example is
    port ( A,B,C : in std_logic;
           F_OUT : out std_logic);
end my_example;


architecture my_soln_exam of my_example is
   signal ABC: std_logic_vector(2 downto 0);
begin
   ABC <= (A,B,C); -- group signals for case statement
my_proc: process (ABC)
   begin
      case (ABC) is
          when "100" =>  F_OUT <= '1';
          when "011" =>  F_OUT <= '1';
          when "111" =>  F_OUT <= '1';
          when others => F_OUT <= '0';
      end case;
   end process my_proc;
end my_soln_exam;
```

Figure 34: Solution to *EXAMPLE 11*.

One of the main items that should be emphasized in any VHDL program is the readability. In the next problem, we redo *EXAMPLE 10* but use a *case* statement instead of *if* statements.

EXAMPLE 12

Write some VHDL code that implements the 8:1 MUX shown in below. Use a case statement in your design. In the black box diagram shown below, the CE input is a chip enable. When CE = '1', the output acts like the MUX of *EXAMPLE 9*. When CE is '0', the output of the MUX is '0'.



Solution: This solution to *EXAMPLE 12* is shown in Figure 35. The entity declaration is repeated below for your convenience. This solution places the *case* statement in the body of an *if* construct. In case you've not noticed it yet, the number of possible solutions to a given problem increase as the problems become more complex.

```
entity mux_8t1_ce is
    port ( Data_in : in std_logic_vector (7 downto 0);
                SEL : in std_logic_vector (2 downto 0);
                 CE : in std_logic;
            F_CTRL : out std_logic);
end mux_8t1_ce;


architecture my_case_ex of mux_8t1_ce is
begin
   my_proc: process (SEL,Data_in,CE)
   begin
      if (CE = '1') then
         case (SEL) is
            when "000" =>  F_CTRL <= Data_in(0);
            when "001" =>  F_CTRL <= Data_in(1);
            when "010" =>  F_CTRL <= Data_in(2);
            when "011" =>  F_CTRL <= Data_in(3);
            when "100" =>  F_CTRL <= Data_in(4);
            when "101" =>  F_CTRL <= Data_in(5);
            when "110" =>  F_CTRL <= Data_in(6);
            when "111" =>  F_CTRL <= Data_in(7);
            when others => F_CTRL <= '0';
         end case;
      else
          F_OUT <= '0';
      end if;
   end process my_proc;
end my_case_ex;
```

Figure 35: Solution to *EXAMPLE 12*.

7 VHDL Operators

This tutorial has only implicitly mentioned the available VHDL operators. This section presents a complete list of operators as well as a few examples of their use. A complete list of operators is shown in Table 3. This is followed by brief descriptions of some of the less obvious operators.

Operators in VHDL are grouped into seven different types: logical, relational, shift, addition, unary, multiplying, and "others". The ordering of this operator list is *somewhat* important because it presents the operators in the order of precedence. The word "somewhat" is italicized because your VHDL code should never rely on operator precedence to describe circuit behavior. Reliance on obscure precedence rules tends to make the VHDL code cryptic and hard to understand. A liberal use of parenthesis is a better approach to VHDL coding.

The first column of Table 3 lists the operators in precedence order with the logical operators having the highest precedence. Although there is a precedence order in the types of operators, there is not precedence order within each type of operator. In other words, the operators appearing in the rows are presented in no particular order. This means that the operators are applied to the given operands in the order they appear in the associated VHDL code.

Operator Type							
logical	`and`	`or`	`nand`	`nor`	`xor`	`xnor`	`not`
relational	`=`	`/=`	`<`	`<=`	`>`	`>=`	
shift	`sll`	`srl`	`sla`	`sra`	`rol`	`ror`	
addition	`+`	`-`					
unary	`+`	`-`					
multiplying	`*`	`/`	`mod`	`rem`			
other	`**`	`abs`	`&`				

Table 3: VHDL operators.

7.1 Logical Operators

The logical operators are generally self-explanatory in nature. They have also been used throughout this tutorial. The only thing worthy to note here is that the not operator has been included in the group of logical operators despite the fact that it is not technically a logic operation.

7.2 Relational Operators

The logical operators are also generally self-explanatory in nature. Many of them have been used in this tutorial. A complete list of relational operators is provided in Table 4.

Operator	Name	Explanation
=	equivalence;	is some value equivalent to some other value?
/=	not-equivalence;	is some value not equivalent to some other value?
<	less than;	is some value less than some other value?
<=	less than or equal;	is some value less than or equal to some other value?
>	greater than;	is some value greater than some other value?
>=	greater than or equal;	is some value greater than or equal to some other value?

Table 4: Relational operators with brief explanations.

7.3 Shift Operators

There are three types of shift operators: simple shift, arithmetic shift, and rotates. Although these operators basically shift bits either left-to-right or right-to-left, there are a few basic differences which are listed below. The shift operators are listed in Table 5.

- Both the simple and arithmetic shifts *stuff* zeros into one end of the operand that is affected by the shift operation. In other words, zeros are fed into one end of the operand while bits are essentially lost from the other end. The difference between simple and arithmetic shifts is that in arithmetic shift, the sign-bit is never changed. For arithmetic shift lefts, zeros are stuffed in the right end of the operand. For arithmetic shift rights, the sign-bit (the left-most bit) is propagated right (the value of the left-most bit is fed into the left end of the operand).

- Rotate operators grab a bit from one end of the word and stuff it into the other end. This operation is done independent of the value of the individual bits in the operand.

Operator		Name	Example	Result
simple	ssl	shift left	`result <= "110111" ssl 2`	`"011100"`
	ssr	shift right	`result <= "110111" ssr 3`	`"000110"`
arithmetic	sla	shift left arithmetic	`result <= "110011" sla 2`	`"101100"`
	sra	shift right arithmetic	`result <= "110011" sra 3`	`"100010"`
rotate	rol	rotate left	`result <= "101000" rol 2`	`"100010"`
	ror	rotate right	`result <= "101001" ror 2`	`"011010"`

Table 5: Shift operators with examples.

7.4 All the Rest of the Operators

The other groups of operators are generally used with numeric types. Since this tutorial does not present numerical operations in detail, the operators are briefly listed below. Special attention is given to the **`mod`**, **`rem`**, and "**`&`**" operators. These operators are also limited to operating on specific types which are also not listed here.

Operator		Name	Comment
addition	+	addition	
	-	subtraction	
unary	+	identity	
	-	negation	
multiplying	*	multiplication	
	/	division	often limited to powers of two
	mod	modulus	see note below
	rem	remainder	see note below
other	**	exponentiation	often limited to powers of two
	abs	absolute value	
	&	concatenation	see note below

Table 6: All the other operators not listed so far.

7.4.1 The Concatenation Operator

The concatenation operator, "&", is often a useful operator when dealing with digital circuits. There are many times when you'll find a need to tack together two separate values. Some examples of the concatenation operators are presented in Figure 36.

```
signal A_val, B_val : std_logic_vector(3 downto 0);
signal C_val : std_logic_vector(6 downto 0);
signal D_val : std_logic_vector(8 downto 0);

C_val <= A_val & "00";
C_val <= "11" & B_val;
C_val <= '1' & A_val & '0';
D_val <= "0001" & A_val;
D_val <= A_val & B_val;
```

Figure 36: Examples of the concatenation operator.

7.4.2 The Modulus and Remainder Operators

There is often confusion about the differences between the remainder and modulus operators, **rem** and **mod**, and the difference in their operation on negative and positive numbers. The definitions that VHDL uses for these operators are shown in Table 7 while a few examples of these operators are provided in Table 8. A general rule followed by many programmers is to avoid using the mod operator when dealing with negative numbers. As you can see from the examples below, the answers are sometime counter-intuitive.

Operator	Name	Satisfies this Conditions	Comment
rem	remainder	1. the sign of (X **rem** Y) has the same sign as X 2. abs (X **rem** Y) < abs (B) 3. X = (X / Y) * Y + (X **rem** Y)	abs = absolute value
mod	modulus	1. the sign of (X **mod** Y) is the same sign as the sign of 2. abs(X **mod** Y) < abs (Y) 3. X = Y * N + (X **mod** Y)	N is an integer value

Table 7: Definitions of `rem` and `mod` operators.

rem	mod
8 **rem** 5 = 3	8 **mod** 5 = 3
-8 **rem** 5 = -3	-8 **mod** 5 = 2
8 **rem** -5 = 3	8 **mod** -5 = -2
-8 **rem** -5 = -3	-8 **mod** -5 = -3

Table 8: Examples of `rem` and `mod` operators.

8 Review

VHDL is a programming language used to design, test, and implement digital circuits. The basic design units in VHDL are the *entity* and the *architecture* which exemplify the general hierarchical approach of VHDL. The entity represents the black box diagram of the circuit or the interface of the circuit to the outside world while the architecture encompasses all the under details of how the circuit behaves.

The VHDL architectures are comprised of statements that described the behavior of the circuit. Because this is a hardware description language, the statements in VHDL are primarily considered to execute concurrently. The idea of concurrency is one of the main themes of VHDL as one would expect since a digital circuit can be model as a set of logic gates that operate with concurrently.

The main concurrent statement types in VHDL are the *concurrent signal assignment* statement, the *conditional signal assignment* statement, the *selected signal assignment* statement, and the *process* statement. The process statement is a concurrent statement which is comprised of exclusively *sequential* statements. The main types of sequential statements are the *signal assignment* statement, the *if* statement, and the *case* statement.
The *if* statement is a sequential version of *conditional signal assignment* statement while the *case* statement is a sequential version of the *selected signal assignment* statement. The syntax of these statements and examples are given in the following table.

Concurrent Statements		Sequential Statements
Concurrent Signal Assignment	⇔	Signal Assignment
target <= *expression*;		*target* <= *expression*;
A <= B **AND** C; DAT <= (D **AND** E) **OR** (F **AND** G);		A <= B **AND** C; DAT <= (D **AND** E) OR (F **AND** G);
Conditional Signal Assignment	⇔	*if* statements

target <= *expressn* **when** *condition* **else**
expressn **when** *condition* **else**
expressn;
		if (*condition*) **then**
		{ sequence of statements }
		elsif (*condition*) **then**
		{ sequence of statements }
		else --(the else is optional)
		{ sequence of statements }
		end if;

F3 <= '1' **when** (L='0' **AND** M='0') **else** '1' **when** (L='1' **AND** M='1') **else** '0';		**if** (SEL = "111") **then** F_CTRL <= D(7); **elsif** (SEL = "110") **then** F_CTRL <= D(6); **elsif** (SEL = "101") **then** F_CTRL <= D(1); **elsif** (SEL = "000") **then** F_CTRL <= D(0); **else** F_CTRL <= '0'; **end if;**
Selective Signal Assignment	⇔	*case* statements
with *chooser_expression* **select** target <= *expression* **when** *choices*, *expression* **when** *choices;*		**case** (*expression*) **is** **when** *choices* => {sequential statements} **when** *choices* => {sequential statements} **when others** => -- (optional) {sequential statements} **end case**;
with SEL **select** MX_OUT <= D3 **when** "11", D2 **when** "10", D1 **when** "01", D0 **when** "00", '0' **when others**;		**case** (ABC) **is** **when** "100" **=>** F_OUT **<=** '1'; **when** "011" **=>** F_OUT **<=** '1'; **when** "111" **=>** F_OUT **<=** '1'; **when others =>** F_OUT **<=** '0'; **end case;**
Process		
label: **process**(*sensitivity_list*) **begin** {*sequential_statements*} **end process** *label*;		

proc1: **process**(A,B,C)
begin
if (A = '1' **and** B = '0') **then**
F_OUT <= '1';
elsif (B = '1' **and** C = '1') **then**
F_OUT <= '1';
else
F_OUT <= '0';
end if;
end process proc1;		

9 Using VHDL for Sequential Circuits

All the circuits we've examined up until now have been combinatorial logic circuits. In other words, none of the circuits we've examined so far were able to store information. This section shows the some of the various methods used to describe sequential circuits. We limit our discussion to VHDL behavioral models for several different flavors of D flip-flops. It is possible and in some cases desirable to use dataflow models to describe flip-flops but it is much easier to use behavior models.

The few approaches to designing flip-flops shown in the next section cover just about all the possible functionality you could imagine adding to a D flip-flop. Once you understand these basics, you'll be on your way to understanding how to use VHDL to design finite state machines.

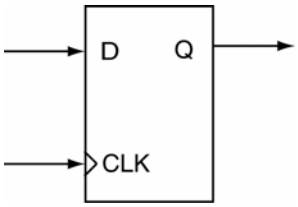
9.1 Simple Storage Elements Using VHDL

The general approach to learning about storage elements is to study the properties of the basic cross-coupled cell. From there, some type of enable logic is added to create a basic latch. The concept of a clocking signal is added and the device becomes a flip-flop. And finally, some type of pulse narrowing circuitry is added and we arrive at the edge-triggered flop-flop.

This study of VHDL descriptions of storage elements starts at the edge-triggered D flip-flop. The VHDL examples presented are the basic edge-triggered D flip-flop with an assortment of added functionality.

EXAMPLE 13

Write the VHDL code that describes a D flip-flop shown below. Use a behavioral model in your description.



Solution: The solution to *EXAMPLE 13* is shown in Figure 37. Listed below are a few interesting things to note about the solution.

- The given architecture body describes the *my_d_ff* version of the *d_ff_x* entity.
- Because example requested the use of a behavioral model, the architecture body is comprised primarily of a *process* statement. The statements within the *process* are executed sequentially. The *process* is executed each time a change is detected in any of the signals in the *process*'s *sensitivity list.* In this case, the statements within the *process* are executed each time there is a change in logic level of the *D* or *CLK* signals.

- The *rising_edge()* construct is used in the *if* statement to indicate that changes in the circuit output only on the rising edge of the *CLK* input. The *rising_edge()* construct is actually an example of a VHDL function which has been defined in one of the included libraries. The way the VHDL code is written makes the circuit synchronous since changes in the circuit's output are synchronized with the rising edge of the clock signal. In this case, the action is a transfer of the logic level on the *D* input to the *Q* output.

- The process is given a label: *dff*. This is not required by the VHDL language but the addition of process labels promotes a self-commenting nature of the code and increases its readability and understandability.

```
------------------------------------------------------------
-- Description of a simple D Flip-Flop
------------------------------------------------------------
entity d_ff_x is
    port ( D, CLK : in std_logic;
                 Q : out std_logic);
end d_ff_x;

architecture my_d_ff of d_ff_x is
begin
dff:  process (D, CLK)
      begin
         if (rising_edge(CLK)) then
            Q <= D;
         end if;
      end process dff;
end my_d_ff;
```

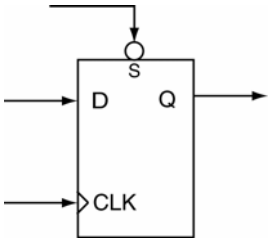
Figure 37: Solution to *EXAMPLE 13*.

The D flip-flop is best known and loved for its ability to store (save, remember) a single bit. The way that the VHDL code listed in Figure 37 is able to store a bit is not obvious, however. The bit-storage capability in the VHDL is *implied* by the both the VHDL code and the way the VHDL code is interpreted. The implied storage comes about as a result of not providing a condition that indicates what should happen if the listed **`if`** condition is not met. In other words, if the **`if`** condition is not met, the device does not change the current value of Q and therefore must "remember" that current value. The remembering of the current value, or state, constitutes the famous bit storage quality of a flip-flop.

The explanation in the previous paragraph is vitally important to anyone who is required to generate VHDL descriptions of circuits. Even if you'll only be using VHDL to design combinatorial circuits, you will most likely be faced with understanding these concepts. One of the classic warnings generated by the VHDL synthesizer is notification that your VHDL code has generated a "latch". Despite the fact that this is "only a warning", if you did not intend to generate a latch, you should strive modify your VHDL code in such as way as to remove this warning. Assuming you did not intend to generate a latch, the cause of your problem is that you've not explicitly provided an output state for all the possible input conditions. Because of this, your circuit will need to remember the previous output state so that it can provide an output in the case where you've not explicitly listed the current input condition.

EXAMPLE 14

Write the VHDL code that describes a D flip-flop shown below. Use a behavioral model in your description. Consider the *S* input to be an active-low, synchronous input that sets the D flip-flop outputs when asserted.



Solution: The solution to *EXAMPLE 14* is shown in Figure 38. There are a few things of interest regarding this solution.

- The *S* input to the flip-flop is made synchronous by only allowing it to affect the operation of the flip-flop on the rising edge of the system clock.

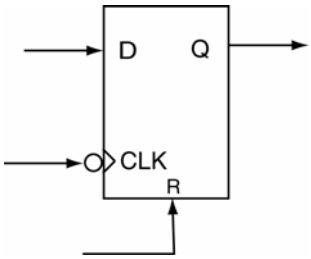
- On the rising edge of the clock, the *S* input takes precedence over the *D* input because the state of the *S* input is checked prior to examining the state of the *D* input. In an `if-else` statement, once one condition evaluates are true, none of the other conditions are checked. In other words, the *D* input is transferred to the output only the rising edge of the clock and only if the *S* input is not asserted.

```
-----------------------------------------------------------
-- Description of RET D Flip-flop with active-low
-- synchronous set input.
-----------------------------------------------------------
entity d_ff_ns is
    port (   D,S :  in  std_logic;
             CLK :  in  std_logic;
               Q :  out std_logic);
end d_ff_ns;

architecture my_d_ff_ns of d_ff_ns is
begin
dff:  process (D,S,CLK)
      begin
         if (rising_edge(CLK)) then
            if (S = '0') then
               Q <= '1';
            else
               Q <= D;
            end if;
         end if;
      end process dff;
end my_d_ff_ns;
```

Figure 38: Solution to *EXAMPLE 14*.

EXAMPLE 15

Write the VHDL code that describes a D flip-flop shown below. Use a behavioral model in your description. Consider the R input to be an active-high, asynchronous input that resets the D flip-flop outputs when asserted.



Solution: The solution to *EXAMPLE 15* is shown in Figure 39. You can probably glean the most information about asynchronous and synchronous inputs by comparing the solutions to *EXAMPLE 14* and *EXAMPLE 15*. A couple of interesting points are listed below.

- The reset input is independent of the clock and takes priority over the clock. This prioritizing is done by making the reset condition the first condition in the `if` statement. Evaluation of the other conditions continues if the *R* input does not evaluate to a 1.

- The *falling_edge()* function is used to make the D flip-flop falling-edge-triggered. Once again, this function is defined in one of the included libraries.

```
------------------------------------------------------------
-- Description of FET D Flip-flop with active-high
-- asynchronous reset input.
------------------------------------------------------------
entity d_ff_r is
    port (   D,R :  in  std_logic;
             CLK :  in  std_logic;
               Q :  out std_logic);
end d_ff_r;


architecture my_d_ff_r of d_ff_r is
begin
dff:  process (D,R,CLK)
      begin
         if (R = '1') then
            Q <= '0';
         elsif (falling_edge(CLK)) then
            Q <= D;
         end if;
      end process dff;
end my_d_ff_r;
```

Figure 39: Solution to *EXAMPLE 15*.

The solutions of *EXAMPLE 14* and *EXAMPLE 15* represent what can be considered the standard VHDL approaches to handling synchronous and asynchronous inputs, respectively. The general forms of these solutions are actually considered templates for synchronous and asynchronous inputs by several VHDL references. As you will see later, these solutions form the foundation to finite state machine design using VHDL.

10 Finite State Machine Design Using VHDL

Finite state machines (FSMs) are generally used as controllers in digital designs. At this point in your digital design career, you've probably designed quite a few state machines on paper, but there was no real point for the design. You're now to the point where your designs still don't have much point but you'll be able to implement and test them using actual hardware if you so choose. The first step in this process is to learn how to model FSMs using VHDL.

As you'll see in the next section, simple FSM designs are just a step beyond the sequential circuits described in the sequential circuits section. The techniques you learn in this section will allow you to quickly and easily design relatively complex FSMs which can be used as controllers in digital circuits.

A block diagram for a standard Moore-type FSM is shown in Figure 40. This diagram looks fairly typical but some different names are used for the some of the blocks in the design. The *next state decoder* is a block of combinatorial logic that uses the current external inputs and the current state of the FSM to decide upon the next state of the FSM. In other words, the inputs to this block are decoded and to produce an output that represents the next state of the FSM. The next state becomes the present state of the FSM when the clock input to the *state registers* block becomes active. The state registers block is storage elements that store the present state of the machine. The inputs to the *output decoder* represent are used to generate the desired external outputs. The inputs are decoder via combinatorial logic to produce the external outputs. Because the external outputs are only dependent upon the current state of the machine, this FSM is classified as a Moore FSM.

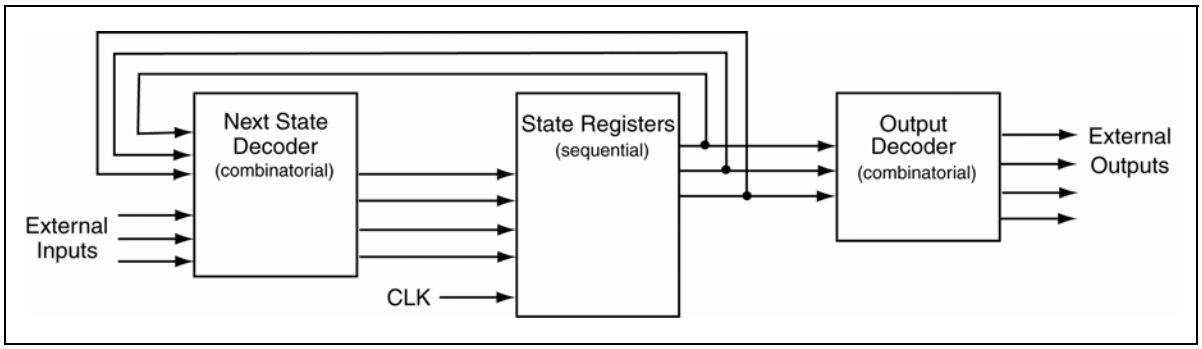


Figure 40: Block diagram for a Moore-type FSM.

The FSM model shown in Figure 40 is probably the model of a Moore-type FSM that you are used to thinking about. This is most likely because as a learning exercise you were required to generate the combinatorial logic required to implement the next state decoder and the output decoder. But we want to think about FSMs in the context of VHDL. The true power of VHDL starts to emerge in its dealings with FSMs. As you'll see, the versatility of VHDL behavioral model removes the need for large paper designs of endless K-maps and endless combinatorial logic.

There are several different approaches used to model FSMs using VHDL. The many different possible approaches are a result of the general versatility of VHDL as a programming language. What we'll describe in this section is probably the clearest approach for FSM implementation. A block diagram of the approach we'll use in the implementation of FSMs is shown in Figure 41.

Although it does not look that much clearer, you'll soon be finding the FSM model shown in Figure 41 to be a straight-forward method to implement FSMs. The approach we will use divides the FSM into two VHDL processes. One process, the *Synchronous Process*, handles all the matters regarding clocking and other controls associated with the storage element. The other process, the *Combinatorial Process*, handles all the matters associated with the Next State Decoder and the Output Decoder of Figure 40. Note that these two blocks in Figure 40 are both comprised of combinatorial logic.

There is some new lingo used in the description of signals used in Figure 41. The inputs labeled *Parallel Inputs* are used to signify inputs that act in parallel to each of the storage elements. These inputs would include enables, presets, clears, etc. The inputs labeled *State Transition Inputs* include external inputs that control state transitions. These inputs also include external inputs used to decode Mealy-type external outputs. All of the other inputs and outputs listed in Figure 41 should seem familiar and are therefore self-explanatory.

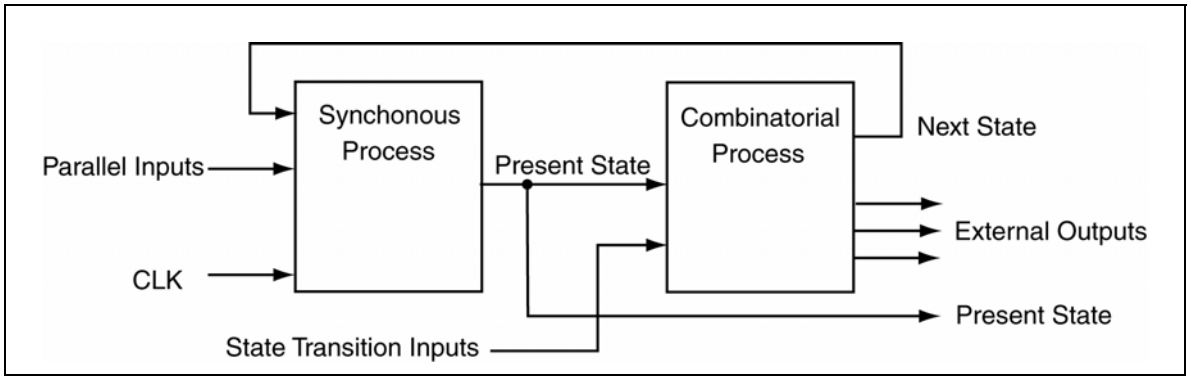
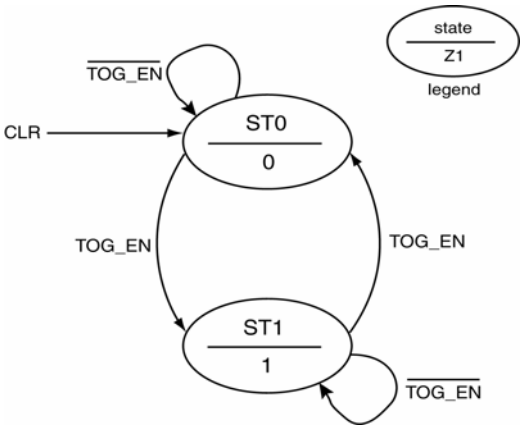


Figure 41: Model for VHDL implementations of FSMs.

One final comment before we begin… Although there are many different methods that can be used to described FSMs using VHDL, two of the more common approaches are the are the *dependent* and *independent PS/NS* styles. Only cover the dependent style in this tutorial because it is clearer than the independent PS/NS style. The model shown in Figure 41 is actually a model of the dependent PS/NS style of FSMs. One you understand the dependent PS/NS style of VHDL FSM modeling, understanding of the independent PS/NS style is painless. More information on the independent PS/NS coding style is found in the class text.

EXAMPLE 16

Write the VHDL code that implements the FSM shown below. Use a dependent PS/NS coding style in your implementation.



Solution: This problem represents a basic FSM implementation. It is somewhat instructive in show the black box diagram which aids in the writing of the entity description. The black box diagram is shown in Figure 42 and the solution is shown in Figure 43.

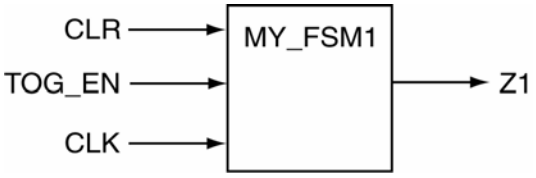


Figure 42: Black box diagram for the FSM of *EXAMPLE 16*.

```
entity my_fsm1 is
    port (   TOG_EN : in  std_logic;
            CLK,CLR : in  std_logic;
                 Z1 : out std_logic);
end my_fsm1;

architecture fsm1 of my_fsm1 is
   type state_type is (ST0,ST1);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,CLR)
   begin
     if (CLR = '1') then PS <= ST0;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,TOG_EN)
   begin
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '0';  -- Moore output
            if (TOG_EN = '1') then NS <= ST1;
            else  NS <= ST0;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (TOG_EN = '1') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when others => -- the catch-all condition
            Z1 <= '0';  -- arbitrary; it should never
            NS <= ST0;  --  make it to these two statement
      end case;
   end process comb_proc;
end fsm1;
```

Figure 43: Solution of *EXAMPLE 16*.

And of course, this solution has many things worth noting in it. The more interesting things are listed below.

- We've declared a special VHDL *type* to represent the states in this FSM. This is an example of how VHDL handles enumeration types. There is an internal numerical representation for the listed state types but we only deal with the more expressive textual equivalent.

- The synchronous process is equal in form and function to the simple D flip-flops we examined in the section on Sequential Circuits. The only difference is we've substituted PS and NS for D and Q, respectively.

- Even though this is about the simplest FSM you could hope for, the code looks somewhat complicated. But if you examine it closely, you can see that everything is nicely compartmentalized in the solution. There are two processes. The synchronous process

handles the asynchronous reset and the assignment of a new state upon the arrival of the system clock. The combinatorial process handles the outputs not handled in the synchronous process, the outputs, and the generation of the next state of the FSM.

- Because the two processes operate concurrently, they can be considered as working in a lock-step manner. Changes to the NS signal that are generated in the combinatorial process forces an evaluation of the synchronous process. When the changes are actually instituted in the synchronous process on the next clock edge, the changes in the PS signal causes the combinatorial process to be evaluated.

- The case statement in the combinatorial process provides a *when* clause for each state of the FSM. This is the standard approach for the dependent PS/NS coding style. A *when others* clause has also been provided. The signal assignments that are part this catch-all clause are arbitrary since the code should never actually make it there. This statement is provided for a sense of completeness.

- The Moore output is a function of only the present state. This is expressed by the fact that the assignment of the Z1 output is unconditionally evaluated in each *when* clause of the case statement in the combinatorial process.

There is one final thing to note about *EXAMPLE 16*. In an effort to keep the example simple, we disregarded the digital values of the state variables. This is indicated in the black box diagram shown in Figure 42 by the fact that the only output of the FSM is signal Z1. This is reasonable in that it could be considered that only one output was required in order to control some other device or circuit. The state variable is represented internally and its precise representation is not important since the state variables are not provided as outputs.

In FSMs such as counters and as a means to test any FSM, the state variables are also outputs of the FSM. To show this situation, we'll provide a solution to *EXAMPLE 16* with the state variables as outputs. The black box diagram of this solution is shown in Figure 44 and the alternate solution is shown in Figure 45.

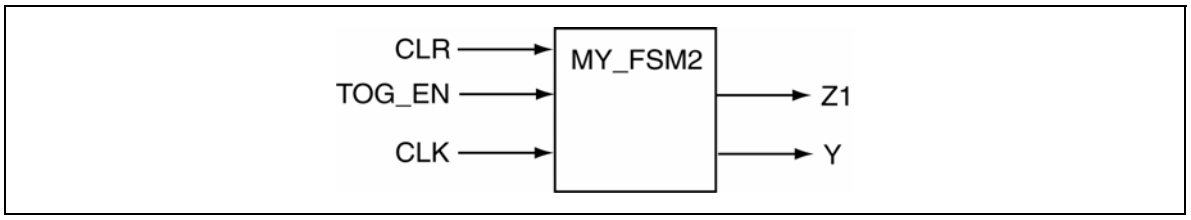


Figure 44: Black box diagram of *EXAMPLE 16* including the state variable as an output.

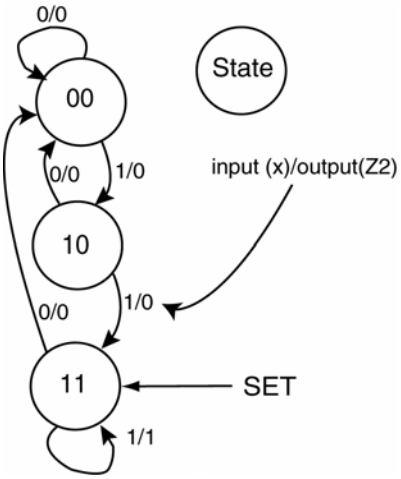
```
entity my_fsm2 is
    port (   TOG_EN : in  std_logic;
           CLK, CLR : in  std_logic;
                  Y : out std_logic;
                 Z1 : out std_logic);
end my_fsm2;


architecture fsm2 of my_fsm2 is
   type state_type is (ST0,ST1);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,CLR)
   begin
     if (CLR = '1') then PS <= ST0;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,TOG_EN)
   begin
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '0';  -- Moore output
            if (TOG_EN = '1') then NS <= ST1;
            else  NS <= ST0;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (TOG_EN = '1') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when others => -- the catch-all condition
            Z1 <= '0';  -- arbitrary; it should never
            NS <= ST0;  --  make it to these two statement
      end case;
   end process comb_proc;

   with PS select
      Y <= '0' when ST0,
           '1' when ST1,
           '0' when others;

end fsm2;
```

Figure 45: Solution for *EXAMPLE 16* including state variable as an output.

Note that the VHDL code in shown in Figure 45 differs in only two areas from the code shown in Figure 43. The first area is the modification of the entity declaration to account for the state variable output Y. The second area is the inclusion of the selective signal assignment statement which assigns a value of state variable output Y based on the condition of the state variable. The selective signal assignment statement is evaluated each time a change in signal PS is detected. Remember, there are three concurrent statements in the VHDL code shown in Figure 43: two process statements and a selective signal assignment statement. We'll consider the state variables as outputs in the FSM examples that follow.

EXAMPLE 17

Write the VHDL code that implements the FSM shown below. Use a dependent PS/NS coding style in your implementation. Consider the state variables as output.



Solution: The state diagram shown in the problem description indicates that this is a three-state FSM with one Mealy-type output. Since there are three states, the solution requires at least two state variables to handle the three states. The black box diagram of the solution is shown in Figure 46. Note that the two state variables are handled as a bus signal.

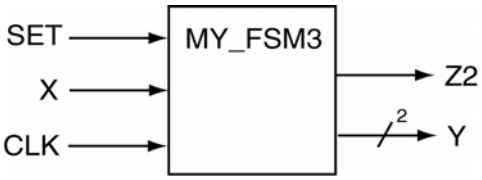


Figure 46: Black box diagram for the FSM of *EXAMPLE 17*.

```
entity my_fsm3 is
    port (      X : in  std_logic;
              CLK : in  std_logic;
              SET : in  std_logic;
                Y : out std_logic_vector(1 downto 0);
               Z2 : out std_logic);
end my_fsm3;


architecture fsm3 of my_fsm3 is
   type state_type is (ST0,ST1,ST2);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,SET)
   begin
     if (SET = '1') then PS <= ST2;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,X)
   begin
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z2 <= '0';  -- Mealy output always 0
            if (X = '0') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z2 <= '0';  -- Mealy output always 0
            if (X = '0') then NS <= ST0;
            else  NS <= ST2;
            end if;
         when ST2 =>    -- items regarding state ST2
            -- Mealy output handled in the if statement
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST2;  Z2 <= '1';
            end if;
         when others => -- the catch all condition
             Z2 <= '1'; NS < ST0;
      end case;
   end process comb_proc;

   with PS select
      Y <= "00" when ST0,
           "10" when ST1,
           "11" when ST2,
           "00" when others;
end fsm3;
```

Figure 47: Solution for *EXAMPLE 17*.

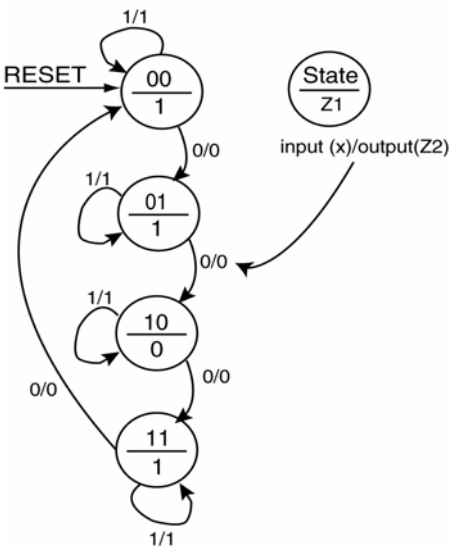
As usual, there are a couple of fun things to note about the solution for *EXAMPLE 17*. Most importantly, you should note the similarities between this solution and the previous solution.

- The FSM has one Mealy-type output. The solution essentially treats this output as a Moore-type output in the first two *when* clauses of the *case* statement. In the final *when* clause, the Z2 output appears in both sections of the *if* statement. It's the fact that the Z2 output is different in the context of the ST2 state that makes it a Mealy-type output.

- Two state variables were required since the state diagram contained more than two states. The solution opted to make these outputs busses which had the effect of slightly changing the form of the selected signal assignment statement appearing at the end of the architecture description.

EXAMPLE 18

Write the VHDL code that implements the FSM shown below. Use a dependent PS/NS coding style in your implementation. Consider the state variables as output.



Solution: The state diagram indicates that the solution will contain four states, one input, and two outputs. This is a hybrid FSM in that the *if* contains both a Mealy and Moore-type output. The black box diagram for the solution is shown in Figure 48 and the actual solution is shown in Figure 49.

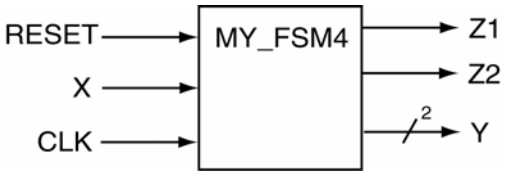


Figure 48: Black Box diagram for the FSM of *EXAMPLE 18* .

```
entity my_fsm4 is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                 Z1,Z2 : out std_logic);
end my_fsm4;

architecture fsm4 of my_fsm4 is
   type state_type is (ST0,ST1,ST2,ST3);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '1') then PS <= ST0;
     elsif (rising_edge(CLK)) then PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,X)
   begin
      -- Z1: the Moore output; Z2: the Mealy output
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST2; Z2 <= '0';
            else  NS <= ST1; Z2 <= '1';
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST2; Z2 <= '0';
            else  NS <= ST1; Z2 <= '1';
            end if;
         when ST2 =>    -- items regarding state ST2
            Z1 <= '0';  -- Moore output
            if (X = '0') then NS <= ST3; Z2 <= '0';
            else  NS <= ST2; Z2 <= '1';
            end if;
         when ST3 =>    -- items regarding state ST3
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST3;  Z2 <= '1';
            end if;
         when others => -- the catch all condition
            Z1 <= '1';  Z2 <= '0';  NS <= ST0;
      end case;
   end process comb_proc;

   with PS select
      Y <= "00" when ST0,
           "01" when ST1,
           "10" when ST2,
           "11" when ST3,
           "00" when others;
end fsm4;
```

Figure 49: Solution for *EXAMPLE 18*.

10.1 One-Hot Encoding for FSMs

The approach taken in the previous FSM examples was to use *full encoding* for the sequential portion of the state machines. The full encoding approach minimizes the number of flip-flops used to store the state variables. The closed form equation describing the number of flip-flops required for a given FSM is shown in Equation 1. The bracket-like symbols used in Equation 1 indicates a *ceiling* function[3].

$$\#(\text{flip-flops}) = \lceil \log_2(\#\text{states}) \rceil$$

Equation 1: Relation between the number of states and number of flip-flops for full encoding.

For one-hot encoded FSMs, only one flip-flop is asserted at any given time. This requires that each distinct state be represented by one flip-flop. In this encoding scheme, the number of flip-flops required to implement a FSM is equal to the number of states in the FSM. The closed form of this relationship is shown in Equation 2.

$$\#(\text{flip-flops}) = \#(\text{states})$$

Equation 2: Relation between the number of states and number of flip-flops for one-hot encoding.

The question naturally arises as to how VHDL can be used to implement one-hot encoded FSMs. A few comments are in order here. Probably the most straight-forward approach involves the tools you are working with rather then the VHDL code itself. In all likelihood, there is a setting in the development software you are using that allows you to select the method that is used to represent the state variables. You can quickly generate a one-hot encoded FSM by selecting the proper option embedded in your development software. This approach requires no modifications to the VHDL FSMs that have been described thus far. The downside of this approach is that you're denied the learning experience associated with implementing the VHDL code that explicitly induces one-hot encoding in your FSM. Since we're concerned with learning VHDL, we need to look at the process of explicitly encoding one-hot FSMs.

The tendency in moving toward one-hot encoded FSMs is to generate an output that mimics the values associated with the one-hot encoded state variables. While this approach appears to the outside world that the design has been one-hot encoded, it actually does not alter the default state variable encoding approach used by the development software. This pseudo one-hot encoded approach is sometimes adequate so we'll list it here. As you'll see, the required modifications to the standard FSM are minimal. The approach we've described thus far can be quickly converted from full encoding to one-hot encoding. The modular approach we used to implement FSMs is what expedites the conversion process. These changes are limited to how the outputs are assigned

[3] The ceiling function $y = \lceil x \rceil$ assigns y to the smallest integer that is greater or equal to x.

to the state variables. Modifications to the full encoded approach as thus limited to the entity declaration and the assignment of the output variables.

Figure 50 shows the modifications to the entity declaration required to convert the full encoding used in *EXAMPLE 18* to a pseudo one-hot encoding. Figure 51 shows how the number of declared state variables is modified to represent the FSM using one-hot encoding. Figure 52 shows the required modifications to the state variable output assignment in order to complete the full encoding to one-hot encoding conversion of *EXAMPLE 18*. Note in Figure 52 that default case is assigned a valid one-hot state instead of the customary *all zero* state. As you can see by examining these figures, only the external outputs have been modified.

```
-- full encoded approach
entity my_fsm4 is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                 Z1,Z2 : out std_logic);
end my_fsm4;
```

```
-- one-hot encoding approach
entity my_fsm4 is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(3 downto 0);
                 Z1,Z2 : out std_logic);
end my_fsm4;
```

Figure 50: Modifications *required* to convert entity of *EXAMPLE 18* to pseudo one-hot encoding.

```
   type state_type is (ST0,ST1,ST2,ST3);
   signal PS,NS : state_type;
```

Figure 51: Modifications *required* to convert state variables to pseudo one-hot encoding.

```
-- full encoded approach
with PS select
   Y <= "00" when ST0,
        "01" when ST1,
        "10" when ST2,
        "11" when ST3,
        "00" when others;
end fsm4;
```

```
-- pseudo one-hot encoded approach
with PS select
   Y <= "1000" when ST0,
        "0100" when ST1,
        "0010" when ST2,
        "0001" when ST3,
        "1000" when others;
end fsm4;
```

Figure 52: Modifications to convert state output of *EXAMPLE 18* to pseudo one-hot encoding.

To convert you FSM to true one-hot encoding that is independent of the development software, you must make a few additional modifications to your design. Two of the modifications are identical to the VHDL code shown in Figure 50 and Figure 52 and are repeated in Figure 53 and Figure 55. The only differences are found in Figure 52. What this snippet of code does is force the declared types to have certain characteristics by use of the VHDL attribute modifier. Forcing the state variables to be truly encoded using one-hot encoding requires two steps as shown in Figure 54. These two lines of code essentially force the VHDL synthesizer to represent each state

of the FSM with its own storage element. In other words, each state is represented by the "string" modifier as listed. This forces four bits per state to be remembered by the FSM implementation which essentially requires four flip-flops.

```
-- full encoded approach
entity my_fsm4 is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                 Z1,Z2 : out std_logic);
end my_fsm4;
```

```
-- one-hot encoded approach
entity my_fsm4 is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(3 downto 0);
                 Z1,Z2 : out std_logic);
end my_fsm4;
```

Figure 53: Modifications *required* to convert entity of *EXAMPLE 18* to true one-hot encoding.

```
  type state_type is (ST0,ST1,ST2,ST3);
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of state_type: type is "1000 0100 0010 0001";
  signal PS,NS : state_type;
```

Figure 54: Modifications *required* to convert state variables to true on-hot encoding.

```
-- full encoded approach
with PS select
   Y <= "00" when ST0,
        "01" when ST1,
        "10" when ST2,
        "11" when ST3,
        "00" when others;
end fsm4;
```

```
-- one-hot encoded approach
with PS select
   Y <= "1000" when ST0,
        "0100" when ST1,
        "0010" when ST2,
        "0001" when ST3,
        "1000" when others;
end fsm4;
```

Figure 55: Modifications to convert state variable outputs of *EXAMPLE 18* to true one-hot encoding.

11 Structural Modeling Using VHDL

As was mentioned earlier, there are generally three approaches to writing VHDL code: dataflow modeling, behavioral modeling, and structural modeling. This document has opted to only deal with dataflow and behavioral models up to this point. This section presents a basic introduction to structural modeling.

As digital designs become more complex, it becomes less likely that any one design can be designated as any one of the three types of VHDL models. We've already seen this property in our dealings with FSMs where we mixed process statements (behavioral modeling) with selective signal assignment statements (dataflow modeling). The result was a hybrid VHDL model. By its very nature, structural modeling is a likewise hybrid VHDL model. Most complex designs could be considered structural models, i.e., if they are implemented using sound coding procedures.

The design of complex digital circuits using VHDL should closely resemble the structure of complex computer programs. Many of the techniques and practices used to construct large and well structured computer programs written in higher-level languages should also be applied when using VHDL to describe digital designs. This common structure we are referring to is the ever so popular *modular* approach to coding. The term structural modeling is the terminology that VHDL uses for the modular design. The VHDL modular design approach directly supports hierarchical design which is essential when attempting to understand complex digital designs.

The benefits of modular design to VHDL are similar to the benefits that modular design or object oriented design provides for higher-level computer languages. Modular designs promote understandability by packing low-level functionality into modules. These modules can be easily reused in other designs thus saving the designer time by removing the need to reinvent and retest the wheel. The hierarchical approach extends beyond code written on the file level. VHDL modules can be placed in appropriately named files and libraries in the same way as higher-level languages.

And finally, after all the commentary regarding complex designs, we present a few simple examples. Though the structural approach is most appropriately used in complex digital designs, the examples presented are rather simplistic in nature. These examples show the essential details of VHDL structural modeling. It is up to the designer to conjure up digital designs where a structural modeling approach would be more appropriate.

11.1 VHDL and Computer Programming Languages: Exploiting the Similarities

The main tool for modularity in higher-level languages such as C is the *function*. In other less useful computer languages, similar modularity is accomplished through the use of the methods, procedures, and subroutines. The approach used in C is to 1) name the function interface you plan on writing (the function declaration), 2) code what the function will do (the function body), 3) let the program know it exists and is available to be called (the proto-type), and 4) call the function from the main portion of the code. The approach used in VHDL is similar: 1) name the module you plan to describe (the entity), 2) describe what the module will do (the architecture), 3) let the program know the module exists and can be used (component declaration), and 4) use the module in your code (component instantiation, or mapping). The similarities between these two approaches are listed in Table 9.

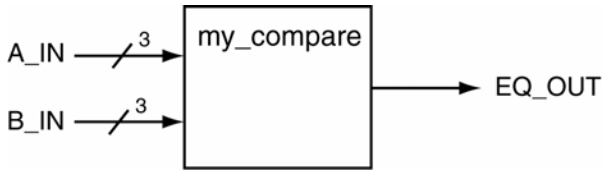
"C" programming language	VHDL
Describe function interface	the entity
Describe what the function does (coding)	the architecture
Provide a function prototype to main program	component declaration
Call the function from main program	component instantiation (mapping)

Table 9: Similarities between modules in "C" and VHDL.

It's best to use these principles in an example. Our approach is to describe a template-type approach to VHDL structural design using a simple and well-known combinational circuit.

EXAMPLE 19

Design a 3-bit comparator using a VHDL structural modeling. The interface to this circuit is described in the diagram below.



Solution: A comparator is one of the classic combinatorial circuits that every digital design student must derive at some point in their careers. The solution presented here implements the discrete gate version of the circuit which is shown in Figure 56. Once again, the solution presented here is primarily an example of a VHDL structural model and does not represent the most efficient method to represent a comparator using VHDL.

The approach of this solution is to model each of the discrete gates as individual "systems". They are actually simple gates but the interfacing requirements of the VHDL structural approach are the same regardless of whether the circuit elements are simple gates or complex digital subsystems.

The circuit shown in Figure 56 contains some extra information that relates to the VHDL structural implementation. First, the dashed line represents the boundary of the VHDL entity i.e., signals that cross this boundary must appear in the entity declaration for this implementation. Second, each of the internal signals (signals that do not cross the dashed entity boundary) have been given names. This is a requirement for VHDL structural implementations as these signals must be assigned to the various sub-modules on the interior of the design (somewhere in the architecture).

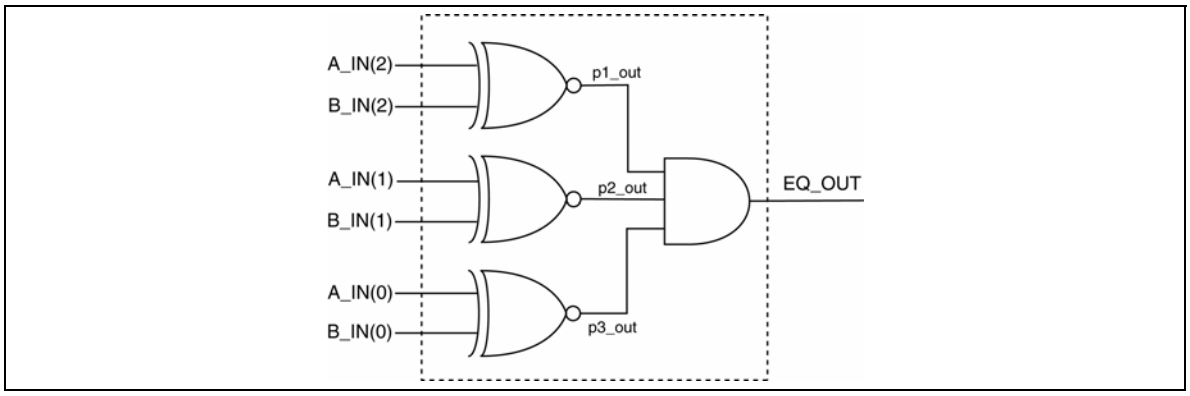


Figure 56: Discrete gate implementation of a 3-bit comparator.

The first part of the solution is to provide entity and architecture implementations for the individual gates shown in Figure 56. We need to provided as least one definition of an XNOR gate and a 3-input AND gates. We only need to provide one definition of the XNOR gate despite the fact that three are shown in the diagram. The modular VHDL approach allows us to reuse circuit definitions and we take advantage of this feature. These definitions are shown in Figure 57.

```
-------------------------------------------------------------
-- Descriptions of XNOR function
-------------------------------------------------------------
entity big_xnor is
    Port ( A,B : in std_logic;
           F : out std_logic);
end big_xnor;

architecture ckt1 of big_xnor is
begin
      F <= not ( (A and (not B)) or ( (not A) and B) );
end ckt1;

-------------------------------------------------------------
-- Description of 3-input AND function
-------------------------------------------------------------
entity big_and3 is
    Port ( A,B,C : in std_logic;
              F : out std_logic);
end big_and3;

architecture ckt1 of big_and3 is
begin
      F <= ( A and B and C );
end ckt1;
```

Figure 57: Entity and Architecture definitions for discrete gates.

The implementations shown in Figure 57 present no new VHDL details. The new information is contained in how the circuit elements listed in Figure 57 are used as components in a larger circuit. The procedures for implementing a structural VHDL design can be summarized in the following steps. These steps assume that the entity declarations for the interior modules already exit.

1. Generate the entity declaration
2. Declare design units used in design
3. Declare required internal signals
4. Instantiate and Map design units

Step One: The first step in a structural implementation is identical to the standard approach we've used for the implementing other VHDL circuits: the entity. The entity declaration is derived directly from dashed box in Figure 56 and is shown in Figure 58.

```
-----------------------------------------------------
-- Interface description of 3-bit comparator
-----------------------------------------------------
entity my_compare is
    Port ( A_IN : in std_logic_vector(2 downto 0);
           B_IN : in std_logic_vector(2 downto 0);
           EQ_OUT : out std_logic);
end my_compare;
```

Figure 58: Entity declaration for 3-bit comparator.

Step Two: The next step is to declare the design units that are used in the circuit. In VHDL lingo, declaration refers to the act of making a particular design unit available to be used in a particular design. Note that the act of declaring a design unit, by definition, transforms your circuit into a hierarchical design. The declaration of a design unit makes the unit available to be placed into the design hierarchy. For our design, we need to declare two separate design units: the XOR gate and a 3-input AND gate.

There are two factors involved in declaring a design unit: 1) how to do it, and, 2) where to place it. A component declaration can be viewed as a modification of the associated entity declaration. The difference is that the word *entity* is replaced with the word *component* and the word *component* must also follow the word *end* to terminate the instantiation. The best way to do this is by cutting, pasting, and modifying the original entity declaration. The resulting component declaration is placed in the architecture declaration after the **`architecture`** line and before the **`begin`** line. The two component declarations and their associated entity declarations are shown in Table 10. Figure 59 shows the component declarations as they appear in working VHDL code.

`entity big_xnor is`
`  Port ( A,B : in std_logic;`
`  F : out std_logic);`
`end big_xnor;`
	`component big_xnor`
	`  Port ( A,B : in std_logic;`
	`  F : out std_logic);`
	`end component;`

`entity big_and3 is`
`  Port ( A,B,C : in std_logic;`
`  F : out std_logic);`
`end big_and3;`
	`component big_and3`
	`  Port ( A,B,C : in std_logic;`
	`  F : out std_logic);`
	`end component;`

Table 10: A comparision of entity and component declarations.

Step Three: The next step is to declare internal signals used by your design. The required internal signals for this design are the signals that are not intersected by the dashed line shown in Figure 56. These three signals are similar to local variables used in higher-level programming languages in that they must be declared before they can be used in the design. These signals effectively

provide an interface between the various design units that are instantiated in the final design. For this design, three signals are required and used as the outputs of the XOR gates and inputs to the AND gate. Internal signal declarations such as these appear with the component declarations in the architecture declaration after the **`architecture`** line and before the **`begin`** line. Note that the declaration of intermediate signals is similar to the signal declaration contain in the entity body. The only difference is that the intermediate signal declaration does not contain the mode specifier. The signal declarations are included as part of the final solution shown in Figure 59.

Step Four: The final step is to create *instances* of the required modules and map these *instances* of the various components in the architecture body. Technically speaking, as the word "instance" implies, the appearance of instances of design units is the main part of the *instantiation* process. In some texts, the process of instantiation includes what we've referred to as component declaration but we've opted not to do this here. The approach presented here is to have *declaration* refer to the component declarations before the **`begin`** line while *instantiation* refers to the creation of individual instances after the **`begin`** line. The mapping process is therefore included in our definition of instantiation.

The process of mapping provides the interface requirements for the individual components in the design. This mapping step associates external connections from each of the components to signals in the next step upwards in the design hierarchy. Each of the signals associated with individual components "maps" to either an internal or external signal in the higher-level design. Each of the individual mappings includes a unique name for the particular instance as well as the name of the original entity. The actual mapping information follows the VHDL key words of: **`port map`**. All of this information appears in the final solution shown in Figure 59.

The process of mapping provides the interface requirements for the individual components in the design. This mapping step associates external connections from each of the components to signals in the next step upwards in the design hierarchy. Each of the signals associated with individual components "maps" to either an internal or external signal in the higher-level design. Each of the individual mappings includes a unique name for the particular instance as well as the name of the original entity. The actual mapping information follows the VHDL key words of: **`port map`**. All of this information is appears in the final solution shown in Figure 59.

```
entity my_compare is
    Port ( A_IN : in std_logic_vector(2 downto 0);
           B_IN : in std_logic_vector(2 downto 0);
           EQ_OUT : out std_logic);
end my_compare;

architecture ckt1 of my_compare is

   -- XNOR gate
   component big_xnor is
      Port ( A,B : in std_logic;
                F : out std_logic);
   end component;

   -- 3-input AND gate
   component big_and3 is
      Port ( A,B,C : in std_logic;
                   F : out std_logic);
   end component;

   -- intermediate signal declaration
   signal p1_out,p2_out,p3_out : std_logic;

begin
   x1: big_xnor port map (A => A_IN(2),
                          B => B_IN(2),
                          F => p1_out);

   x2: big_xnor port map (A => A_IN(1),
                          B => B_IN(1),
                          F => p2_out);

   x3: big_xnor port map (A => A_IN(0),
                          B => B_IN(0),
                          F => p3_out);

   a1: big_and3 port map (A => p1_out,
                          B => p2_out,
                          C => p3_out,
                          F => EQ_OUT);
end ckt1;
```

Figure 59: VHDL code for the top of the design hierarchy for the 3-bit comparator.

It is worthy to note that the solution shown in Figure 59 is not the only approach to use for the mapping process. The approach shown in Figure 59 uses what is referred to a *direct mapping* of components. In this manner, each of the signals in the interface of the design units are listed and are directly associated with the signals they connect to in the higher-level design by use of the "**=>**" operator. This approach has several potential advantages: it is explicit, complete, orderly, and allows the signals to be listed in any order. The other approach to mapping is to use *implied* mapping. In this approach, connections between external signals from the design units are associated with signals in the higher-level design by order of their appearance in the mapping statement. This differs from direct mapping because only signals from the higher-level design appear in the mapping statement instead. The association between signals in the design units and the higher-level design are implied by the ordering of the signal as they appear in the component or entity declaration. This approach uses less space in the source code but requires that signals be placed in the proper order. An alternate but equivalent architecture for the previous example using implied mapping is shown in Figure 60.

```
architecture ckt2 of my_compare is

   component big_xnor is
      Port ( A,B : in std_logic;
               F : out std_logic);
   end component;

   component big_and3 is
      Port ( A,B,C : in std_logic;
                 F : out std_logic);
   end component;

   signal p1_out,p2_out,p3_out : std_logic;

begin
   x1: big_xnor port map (A_IN(2),B_IN(2),p1_out);
   x2: big_xnor port map (A_IN(1),B_IN(1),p2_out);
   x3: big_xnor port map (A_IN(0),B_IN(0),p3_out);
   a1: big_and3 port map (p1_out,p2_out,p3_out,EQ_OUT);
end ckt2;
```

Figure 60: Alternative architecture for *EXAMPLE 19* using implied mapping.

Due to the fact that this design was relatively simple, it was able to bypass one of the interesting issues that arises when using structural modeling. Often when dealing with structural designs, different levels of the design will contain the same signal name. The question arises as to whether the synthesizer is able to differentiate between the signal names across the hierarchy. VHDL synthesizers, like compilers for higher-level languages, are able to handle such instances. Signals with the same names are mapped according to the mapping presented in the component instantiation statement. Probably the most common occurrence of this is with clock signals. In this case, a component instantiation such as the one shown in Figure 61 is both valid and commonly seen in designs containing a system clock. Name collision does not occur because the signal name on the left side of the "=>" operator is understood to be internal to the component while the signal on the right side is understood to reside in the next level up in the hierarchy.

```
   x5: some_component port map (CLK => CLK,
                                CS => CS);
```

Figure 61: An example of the same signal name crossing hierarchical boundaries.

12 Data Objects

This tutorial has been specifically written to minimize the introduction of the theory behind VHDL in order to leverage the digital knowledge you probably already have. Many of the concepts presented thus far have been implicitly presented in the context of example problems. In this way, you've probably been able to generate quality VHDL code but were constrained to using the VHDL style presented in the examples. In this section, we'll present some of the underlying details and theories that surround VHDL as a backdoor approach to presenting tools that will allow you to use VHDL describe the behavior of more complex digital circuits.

A good place to start is with the definition of VHDL *objects*. An *object* is an item in VHDL that has both a name (associated identifier) and a specific type. There are four types of objects and many different data types in VHDL. Up to this point, we've only used signal data objects and `std_logic` data types. Two new data objects and several new data types are discussed in this section.

12.1 Types of Data Objects

There are four types of data objects in VHDL: signals, variables, constants, and files. One of the purposes of this section is to present some background information regarding variables which will be used later in this tutorial. The idea of constants will also be briefly mentioned since they are generally straight-forward to understand and use once the concepts of signals and variables are understood. File data objects are not discussed in this tutorial.

12.1.1 Data Object Declarations

The first thing to note about the data objects is the similarity in their declarations. The forms for the three data objects we'll be discussing are listed in Table 11. For each of these declarations, the bold-face font is used to indicate VHDL keywords. The form for the signal object should seem familiar since we've used it extensively up to this point. Note that each of the data objects can optionally be assigned initial values. As you know, signal declarations do not usually provided initial values as opposed to constants which generally do. Example declarations for these three flavors of data objects are provided in Table 12. These examples include several new data types which will be discussed in Section 12.2.

Data Object	Declaration Form
signal	`signal signal_name : signal_type := initial_value;`
variable	`variable variable_name : variable_type := initial_value;`
constant	`constant constant_name : constant_type := initial_value;`

Table 11: Data object declaration forms.

Data Object	Example Declarations

signal	`signal sig_var1 : std_logic := '0';`
	`signal tmp_bus : std_logic_vector(3 downto 0) := "0011";`
	`signal tmp_int : integer range -128 to 127 := 0;`
	`signal my_int : integer;`

variable	`variable my_var1, my_var2 : std_logic;`
	`variable index_a : integer range (0 to 255) := 0;`
	`variable index_b : integer := -34;`

constant	`constant sel_val : std_logic_vector(2 downto 0) := "001";` `constant max_cnt : integer := 12;`

Table 12: Example declarations for signal, variable, and constant data objects.

12.1.2 Variables and the Assignment Operator "**:=**"

Although variables are similar to signals, variables are not as functional for the several reasons mentioned in this section. Variables can only be declared and used inside of processes, functions, and procedures (this tutorial does not discuss functions and procedures). Implied in this statement is the sequential nature of variable assignment statements in that all statements appearing in the body of a process are sequential. One of the early mistakes made by VHDL programmers is attempting to use variables outside of processes.

The signal assignment operator, "**<=**", was used to transfer the value of one signal to another with dealing with signal data objects. When working with variables, the assignment operator "**:=**" is used to transfer the value of one variable data object to another. As you can see from Table 12, the assignment operator is overloaded which allows it to be used to assign initial values to the three listed forms of data objects.

12.1.3 Signals vs. Variables

The use of signals and variables can be somewhat confusing because these data objects can seem relatively similar. Generally speaking, a signal is can be thought of as representing a "wire" or some type of physical connection in a design. Signals thus represent a means to interface VHDL modules which includes connections to the outside word (I/O). In terms of circuit simulation,

signals can be *scheduled* to take on multiple values at specific times in the simulation. The specifics of simulating circuits using VHDL are not covered in this tutorial so the last statement may not carry much meaning to you. The important difference here is that events can be scheduled for signals while for variables, they cannot. The assignment of variables is considered to happen immediately and cannot have a list of scheduled events.

Variables cannot always be modeled as wires in a circuit. They also have no concept of memory since they cannot store events. With all this in mind, you may wonder the appropriate place to use variables. The answer is variables should only be used as iteration counters in loops or as temporary values when executing an algorithm that performs some type of calculation. It is possible to use variables outside of these areas, but it should be avoided.

12.2 Data Types

Not only does VHDL have many defined data types, VHDL also allows you to define your own types. This tutorial, however, only deals with a few of the most widely used types. In this section, the types that have already been discussed are listed and a few more popular and useful types are introduces.

12.2.1 Commonly Used Types

The types used thus far in this tutorial as well as two new types are listed in Table 13. The std_logic and std_logic_vector types have been used extensively in this tutorial. These types are more complex then has been previously stated and is discussed further in Section 12.2.3. The enumerated type was used during the discussion of Finite States Machines in Section 10. The integer type was cryptically mentioned in Section 12.1.1 but will be discussed further along with the Boolean type in this section.

Type	Example	Usage
std_logic	`signal my_sig : std_logic;`	all examples
std_logic_vector	`signal busA : std_logic_vector(3 downto 0);`	all examples
enumerated	`type state_type is (ST0,ST1,ST2,ST3);`	EXAMPLE 18
Boolean	`variable my_test : boolean := false;`	
integer	`signal iter_cnt : integer := 0;`	EXAMPLE 20

Table 13: Data types used in this tutorial.

12.2.2 Integer Types

The use of integer types aids in the design of algorithmic-type VHDL code. This type of coding allows VHDL to describe the behavior of complex digital circuits. As you progress in your digital studies, you'll soon find yourself in need of more complex descriptive VHDL tools; data types such as integers partially fills that desire. This section briefly looks at integer types as well as the definition of user specified integer types.

The range of the integer type is [-2,147,483,647 to 2,147,483,647]. These numbers should seem familiar since they represent the standard 32-bit range for a signed number: (-2^{31} to $+2^{31}$). Other types similar to integers included *natural* and *positive* types. These types are basically integers with shifted ranges. For example, the natural and positive types range from 0 and 1 to the full 32-bit range, respectively. Examples of integer declarations are shown in Figure 62.

```
signal my_int : integer range 0 to 255 := 0;
variable max_range : integer := 255;
constant start_addr : integer:= 512;
```

Figure 62: Examples of integer declarations.

Although it would be possible to use only basic integer declarations in your code, VHDL allows you to define you own integer types with their own personalized range constraints. These special types should be used where possible to make you code more readable. These type definitions use the **type**, **range**, and **to** (or **downto**) keywords in their definitions. Example of integer-type declarations are provided in Figure 63.

```
type scores is range 0 to 100;
type years is range -3000 to 3000;
type apples is range 0 to 15;
type oranges is range 0 to 15;
```

Figure 63: Examples of integer type declarations.

Although each of the types listed in Figure 63 are basically integers, they are still considered different types and cannot be assigned to each other. In addition to this, any worthy VHDL synthesizer will do range checks on your integer types. In the context of the definitions presented in Figure 63, each of the statements in Figure 64 is illegal.

```
signal score1 : scores := 100;
signal my_apple : apples := 0;
signal my_orange : oranges := 0;

my_apple <= my_orange;  -- different types
my_orange <= 24;        -- out of range
my_score <= 110;        -- out of range
```

Figure 64: Examples of illegal assignment statements.

12.2.3 The `std_logic` Type

One of the data types not used or listed in this tutorial is the *bit* type. This type can take on the values of '1' or '0' only. While this set of values for the *bit* types seems appropriate for designing digital circuits, it's actually somewhat limited. Due to its versatility and a more complete range of possible values, the *std_logic* type is most often preferred over *bit* types. The std_logic type is officially defined in the STANDARD *package* and provides a serves to provide a standard that can be used by all VHDL programmers.

The std_logic type is officially defined as an enumerated type. Two of the possible enumerations of course include '1' and '0'. The actual definition is shown in Figure 65. As you can see, this definition lists "std_ulogic" as opposed to the "std_logic" you're used to using. The std_logic type is a *resolved* version of the std_ulogic type. The exact meaning of resolution is beyond the scope of this tutorial and can be safely overlooked.

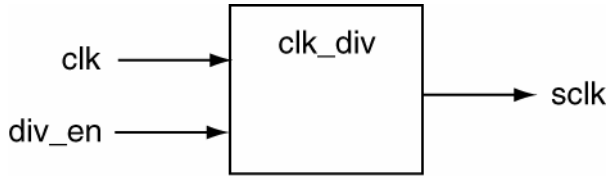
```
type std_ulogic is ( 'U', -- uninitialized
                     'X', -- forcing unknown
                     '0', -- forcing 0
                     '1', -- forcing 1
                     'Z', -- high impedance
                     'W', -- weak unknown
                     'L', -- weak 0
                     'H', -- weak 1
                     '-'  -- unspecified (don't care)
                   );
```

Figure 65: Declaration of the std_ulogic enumerated type.

The std_ulogic type uses the VHDL *character* type in its definition. Although there are nine values in the definition shown in Figure 65, this tutorial only deals with '0', '1', 'Z', and '-'. The 'Z' if generally used when dealing with bus structures. This allows a signal or set of signals (a bus) to have the possibility of being driven by multiple sources without the need to generate resolution functions. When a signal is "driven" to its high impedance state, the signal is not driven from that source and is effectively removed from the circuit. And finally, since the characters used in the std_ulogic type are part of the definition, they must be used as listed. Use of lower-case letters will generate an error.

EXAMPLE 20

Design a clock divider circuit that reduces the frequency of the input signal by a factor of 64. The circuit has two inputs as shown in the diagram. The div_en input allows the clk signal to be divided when asserted and the sclk output will exhibit a frequency 1/64 that of the clk signal. When div_en is not asserted, the sclk output remains low. Frequency division resets when the div_en signal is reasserted.



Solution: As usual for more complex concepts and circuits, there are a seemingly infinite number of solutions. A solution that uses several of the concepts discussed in this section is presented in Figure 66. Some of the more important issues in this solution are listed below

- The type declaration for my_count appears in the architecture body before the begin statement.

- A constant is used for the max_count variable. This allows for quick adjustments in the clock frequency. In this example, this concept is somewhat trivial because the max_count variable is used only once.

- The variable is declared inside of the process (after the process **`begin`** line).

```
entity clk_div is
    Port ( clk : in std_logic;
           div_en : in std_logic;
           sclk : buffer std_logic);
end clk_div;


architecture my_clk_div of clk_div is
   type my_count is range 0 to 100;
   constant max_count : my_count := 63;
begin
   my_div: process (clk, div_en)

      variable div_count : my_count := 0;

   begin
      if (rising_edge(clk)) then    -- look for clock edge
         if (div_en = '1') then     -- divider enabled
            if (div_count = max_count) then
               sclk <= not sclk;    -- toggle output
               div_count := 0;      -- reset count
            else
               div_count := div_count + 1;
            end if;
         else                       -- divider disabled
            div_count := 0;         -- reset count
            sclk <= '0';            -- turn off output
         end if;
      end if;
   end process my_div;
end my_clk_div;
```

Figure 66: Solution for *EXAMPLE 20***.**

13 Looping Constructs

As the circuits you are required to design become more and more complex, you'll find yourself searching for more functionality and versatility in from the VHDL code. You'll probably find what you're looking for in various looping constructs which are yet another form of VHDL statements. This section provides descriptions of several types of looping constructs and some of details regarding their use.

There are two types of loops in VHDL: **`for`** loops and **`while`** loops. The names of these loops should seem familiar from your experience with higher-level computer programming languages. Generally speaking, you can leverage your previous experience with these loop types when describing the behavior of digital circuits. The comforting part is that since these two types of loops are both sequential statements (and thus can only appear in processes). You'll also be able to apply the algorithmic thinking and designing skills you developed coding higher-level computer languages to the circuits you'll be describing using VHDL. The syntax is slightly different but the basic structured programming concepts are the same.

13.1 `for` and `while` Loops

The purpose of a loop construct is to allow something to happen (lines of code to be processed) iteratively (over and over again). These two types of loops of course share this functionality. As you probably remember from higher-level language programming, the syntax of the language is such that you can use either type of loop in any given situation by clever modification of the code. The same is true in VHDL. But although you can be clever in the way you design your VHDL code, the best approach to make the code readable and understandable. Keeping this concept in mind underscores the basic functional difference between **`for`** and **`while`** loops. This basic difference can be best illuminated by examining the form of the loops which are provided in Figure 67.

```
label: for index in a_range loop
   sequential statements...
end loop label;
```

```
label: while (condition) loop
   sequential statements...
end loop label;
```

Figure 67: The basic forms of the `for` and `while` loops.

The major difference between these two loops lies in the number of iterations the loop will perform. This difference can be classified as under what conditions the circuit will terminate its iterations. If you know the number of iterations the loop requires, you should use a **`for`** loop. As you'll see in the examples that follow, the **`for`** loops allow you to explicitly state the number of iterations that a loop performs. The **`while`** loop should be used when you do not know the number of iterations a loop needs to perform. In this case, the loop stops iterating when the terms stated in the condition are met. Using these loops in this manner constitute good programming practices. The loop labels are listed in italics to indicate that they are optional. These labels should be always be used to clarify the associated VHDL code. Use of loop labels is an especially good idea when nested loops are used and when loop control statements are applied.

13.1.1 `for` Loops

The basic form of the **`for`** loop was shown in Figure 67. This loop uses some type of index value to iterate through a range of discrete values. There are two options that can be applied as to the range of discrete values: 1) the range can be specified in the **`for`** loop statement or 2) the loop can use a previously declared range.

`for cnt_val in 0 to 24 loop`
`  -- sequential_statements`
`end loop;`
	`type my_range is range 0 to 24;`
	
	`for cnt_val in my_range loop`
	`  -- sequential_statements`
	`end loop;`

`for cnt_val in 24 downto 0 loop`
`  -- sequential_statements`
`end loop;`
	`type my_range is range 24 downto 0;`
	
	`for cnt_val in my_range loop`
	`  -- sequential_statements`
	`end loop`

(a)	(b)

Figure 68: Two equivalent for loops that (a) specify a range, (b) use a previously specifed range.

The index variable used in the **`for`** loop contains some strange qualities which are listed below. Although your VHDL synthesizer should be able to flag these errors, you should still keep these in mind when you use a **`for`** loop and you'll save yourself a bunch of debugging time. Also note that the loop body has been indented to make the code more readable. Enhanced readability of the code is always good.

- The index variable does not need to be declared (it's done implicitly).
- Assignments cannot be made to the index variable. The index variable can, however, be used in calculations within the loop body.
- The index variable can only step through the loop in increments of one.
- The identifier used for the index variable can be the same as another variable or signal; no name collisions will occur. The index variable will effectively hide identifiers with the same name inside the body of the loop. Using the same identifier for two different values constitutes bad programming practice and should be avoided.
- The specified range for the index (when specified outside of the loop declaration) can be enumerated types.

And lastly, as shown in Figure 69, **`for`** loops can also apply the **`downto`** option. This option makes more sense when the range is specified in the **`for`** loop declaration.

```
for cnt_val in 24 downto 0 loop
   -- sequential_statements
end loop;
```

(a)

```
type my_range is range 24 downto 0;

for cnt_val in my_range loop
   -- sequential_statements
end loop
```

(b)

Figure 69: `for` loops using the downto approach.

13.1.2 `while` Loops

`while` loops are somewhat more simple than **`for`** loops due to the fact that they do not contain an index variable. The major difference between the **`for`** and **`while`** loops is that the **`for`** loop declaration contains a built-in loop termination criteria. The first thing you should remember about **`while`** loops is that the associated code should contain some way of exiting the loop. Examples of **`while`** loops are shown in Figure 70. Needless to say, the VHDL code appearing in Figure 70(b) should have used a **`for`** loop instead of a **`while`** loop because the number of iterations is known.

```
constant max_fib : integer := 2000;
variable fib_sum : integer := 1;
variable tmp_sum : integer := 0;

while (fib_sum < max_fib) loop
   fib_sum := fib_sum + tmp_sum;
   tmp_sum := fib_sum;
end loop;
```

(a)

```
constant max_num : integer := 10;
variable fib_sum : integer := 1;
variable tmp_sum : integer := 0;
variable int_cnt : integer := 0;

while (int_cnt < max_num) loop
   fib_sum := fib_sum + tmp_sum;
   tmp_sum := fib_sum;
   int_cnt := int_cnt + 1;
end loop;
```

(b)

Figure 70: Two examples of while loops calculating a Fibonacci sum.

13.2 Loop Control: `next` and `exit` Statements

Similar to higher-level computer languages, VHDL provides some extra loop control options. These options include the **`next`** statement and the **`exit`** statement. These statements are similar to their counterparts in the higher-level language in the control they can exert over loops. These two loop-control constructs are available for use in either **`for`** or **`while`** loops.

13.2.1 The `next` Statement

The **`next`** statement allows for the loop to bypass the remaining statements within the body of the loop and start immediately at the next iteration. In **`for`** loops, the index variable is

incremented automatically before the start of the upcoming iteration. In **`while`** loops, it is up to the programmer to ensure that the loop operates properly when the **`next`** statement is used. There are two forms of the **`next`** statement and both forms are shown in the examples of Figure 71. These are two examples that use the **`next`** statement and do not necessarily represent good programming practices or contain meaningful code.

```
variable my_sum : integer := 0;

for cnt_val in 0 to 50 loop
   if (my_sum = 20) then
      next;
   end if;
   my_sum := my_sum + 1;
end loop;
```

```
variable my_sum : integer := 0;

while (my_sum < 300) loop
   next when (my_sum = 20);
   my_sum := my_sum + 1;
end loop;
```

Figure 71: Examples of the two forms of next statements.

13.2.2 The `exit` Statement

The **`exit`** statement allows for the immediate termination of the loop and can be used in both **`for`** loops and **`while`** loops. Once the **`exit`** statement is encountered in the flow of VHDL code, control is returned to the statement following the **`end loop`** statement associated with the given loop. The **`exit`** statement works in nested loops as well. The two forms of the **`exit`** statement are similar to the two forms of the **`next`** statement. Examples of these forms are provided in Figure 72.

```
variable my_sum : integer := 0;

for cnt_val in 0 to 50 loop
   if (my_sum = 20) then
      exit;
   end if;
   my_sum := my_sum + 1;
end loop;
```

```
variable my_sum : integer := 0;

while (my_sum < 300) loop
   exit when (my_sum = 20);
   my_sum := my_sum + 1;
end loop;
```

Figure 72: Example of the two forms of `exit` statements.

14 Standard Digital Circuits in VHDL

As you know or as you'll be finding out soon, even the most complex digital circuit is comprised of a relatively small set of standard digital circuits plus some associated control signals. This list of standard digital circuits is a mixed bag of combinatorial sequential devices such as MUXes, decoders, counters, comparators, registers, etc. The art of digital design using VHDL is centered about the proper selection and interfacing of these devices. The actual creation and testing of these devices is de-emphasized.

The most efficient approach to utilizing standard digital circuits using VHDL is to use existing code for these devices and modify them according to the needs of your particular design. This approach allows you to utilize your current knowledge of VHDL to quickly an efficiently design complex digital circuits. The following figures list a set of standard digital devices and the VHDL code used to describe them. The following circuits represented in various sizes and widths. Note that the following circuit descriptions represent *possible* VHDL descriptions but are by no means the only descriptions. They do however provide starting points for you to modify them for your own design needs.

14.1 RET D Flip-flop

```
-----------------------------------------------------------------
--  D flip-flop: RET D flip-flop with single output
--
--  Required signals:
-------------------------------------------------
--  CLK,D: in STD_LOGIC;
--  Q: out STD_LOGIC;
-----------------------------------------------------------------
process (CLK,D)
begin
   if (rising_edge(CLK)) then
      Q <= D;
   end if;
end process;
```

Figure 73: VHDL code for D flip-flop.

14.2 8-Bit Register with Chip Select

```
-----------------------------------------------------------------
--  Register: 8-bit Register with chip select.
--
--  Required signals:
-------------------------------------------------
--  CLK,CS: in STD_LOGIC;
--  D_IN: in STD_LOGIC_VECTOR(7 downto 0);
--  D_OUT: out STD_LOGIC_VECTOR(7 downto 0);
-----------------------------------------------------------------
process (CLK,CS)
begin
   if (CS = '1' and rising_edge(CLK)) then  -- positive logic for CS
      D_OUT <= D_IN;
   end if;
end process;
```

Figure 74: VHDL code for 8-bit register.

14.3 Synchronous up/down counter (with other features)

```
-------------------------------------------------------------------
-- Counter: synchronous up/down counter with asynchronous
-- reset and synchronous parallel load.
--
--  Required signals:
-------------------------------------------------
--  CLK, RESET: in STD_LOGIC;
--  LOAD, UP: in STD_LOGIC;
--  DIN: in STD_LOGIC_VECTOR(7 downto 0);
--  COUNT: inout STD_LOGIC_VECTOR(7 downto 0);
-------------------------------------------------------------------
process (CLK, RESET)
begin
   if (RESET = '1') then    -- positive logic input
      COUNT <= "00000000";
   elsif (rising_edge(CLK)) then
      if (LOAD = '1') then    -- positive logic input
          COUNT <= DIN;
      else
         if (UP = '1') then
            COUNT <= COUNT + 1;
         else
            COUNT <= COUNT - 1;
         end if;
      end if;
   end if;
end process;
```

Figure 75: VHDL code for Up/Down counter.

14.4 Shift Register with Synchronous Parallel Load

```
-------------------------------------------------------------------
-- Shift Register: One direction shift register with synchronous
-- parallel load.
--
-- Required signals:
-------------------------------------------------
-- CLK, D_IN: in STD_LOGIC;
-- P_LOAD: in STD_LOGIC;
-- P_LOAD_DATA: in STD_LOGIC_VECTOR(7 downto 0);
-- D_OUT: out STD_LOGIC;
--
-- Required intermediate signals:
signal REG_TMP: STD_LOGIC_VECTOR(7 downto 0);
-------------------------------------------------------------------
process (CLK)
begin
   if (rising_edge(CLK)) then
      if (P_LOAD = '1') then
         REG_TMP <= P_LOAD_DATA;
      else
         REG_TMP <= REG_TMP(6 downto 0) & D_IN;
      end if;
   end if;
   D_OUT <= REG_TMP(3);
end process;
```

Figure 76: VHDL code for shift register.

14.5 8-Bit Comparator

```
-------------------------------------------------------------------
-- Comparator: Implemented as a sequential circuit.
--
-- Required signals:
-------------------------------------------------
-- CLK: in STD_LOGIC;
-- A_IN, B_IN : in STD_LOGIC_VECTOR(7 downto 0);
-- ALB, AGB, AEB : out STD_LOGIC
-------------------------------------------------------------------
process(CLK)
begin
   if (rising_edge(CLK)) then
      if ( A_IN < B_IN ) then ALB <= '1';
      else ALB <= '0';
      end if;

      if ( A_IN > B_IN ) then AGB <= '1';
      else AGB <= '0';
      end if;

      if ( A_IN = B_IN ) then AEB <= '1';
      else AEB <= '0';
      end if;
   end if;
end process;
```

Figure 77: VHDL code for Comparator.

14.6 BCD to 7-Segment Decoder

```
-------------------------------------------------------------------
-- BCD to 7-Segment Decoder: Implemented as combinatorial circuit.
-- Outputs are active low; Hex outputs are included. The SSEG format
-- is ABCDEFG (segA, segB etc.)
--
-- Required signals:
-------------------------------------------------
-- BCD_IN : in STD_LOGIC_VECTOR(3 downto 0);
-- SSEG : out STD_LOGIC_VECTOR(6 downto 0);
-------------------------------------------------------------------
with BCD_IN select
   SSEG <= "0000001" when "0000",   -- 0
           "1001111" when "0001",   -- 1
           "0010010" when "0010",   -- 2
           "0000110" when "0011",   -- 3
           "1001100" when "0100",   -- 4
           "0100100" when "0101",   -- 5
           "0100000" when "0110",   -- 6
           "0001111" when "0111",   -- 7
           "0000000" when "1000",   -- 8
           "0000100" when "1001",   -- 9
           "0001000" when "1010",   -- A
           "1100000" when "1011",   -- b
           "0110001" when "1100",   -- C
           "1000010" when "1101",   -- d
           "0110000" when "1110",   -- E
           "0111000" when "1111",   -- F
           "1111111" when others;   -- turn off all LEDs
```

Figure 78: VHDL code for BCD to 7-Segment Decoder.

14.7 4:1 Multiplexer

```
-----------------------------------------------------------------
-- A 4:1 multiplexer implemented as behavioral model using case
-- statement.
--
-- Required signals:
-------------------------------------------------
-- SEL: in STD_LOGIC_VECTOR(1 downto 0);
-- A, B, C, D:in STD_LOGIC;
-- MUX_OUT: out STD_LOGIC;
-------------------------------------------------------------------------
process (SEL, A, B, C, D)
begin
   case SEL is
      when "00" => MUX_OUT <= A;
      when "01" => MUX_OUT <= B;
      when "10" => MUX_OUT <= C;
      when "11" => MUX_OUT <= D;
      when others => NULL;
   end case;
end process;
```

Figure 79: VHDL code for 4:1 Multiplexor.

14.8 3:8 Decoder

```
-----------------------------------------------------------------
-- Decoder: 3:8 decoder with active high outputs implemented as
-- combinatorial circuit with selective signal assignment statement
--
-- Required signals:
-------------------------------------------------
-- D_IN: in STD_LOGIC_VECTOR(2 downto 0);
-- FOUT: out STD_LOGIC_VECTOR(7 downto 0);
-----------------------------------------------------------------
with D_IN select
   F_OUT <= “00000001” when “000”,
            “00000010” when “001”,
            “00000100” when “010”,
            “00001000” when “011”,
            “00010000” when “100”,
            “00100000” when “101”,
            “01000000” when “110”,
            “10000000” when “111”,
            “00000000” when others;
```

Figure 80: VHDL code for 3:8 Decoder.

A *Appendix*: VHDL Reserved Words

Table 14 provides a complete list of VHDL reserved words.

abs	**downto**	**library**	**postponed**	**srl**
access	**else**	**linkage**	**procedure**	**subtype**
after	**elsif**	**literal**	**process**	**then**
alias	**end**	**loop**	**pure**	**to**
all	**entity**	**map**	**range**	**transport**
and	**exit**	**mod**	**record**	**type**
architecture	**file**	**nand**	**register**	**unaffected**
array	**for**	**new**	**reject**	**units**
assert	**function**	**next**	**rem**	**until**
attribute	**generate**	**nor**	**report**	**use**
begin	**generic**	**not**	**return**	**variable**
block	**group**	**null**	**rol**	**wait**
body	**guarded**	**of**	**ror**	**when**
buffer	**if**	**on**	**select**	**while**
bus	**impure**	**open**	**severity**	**with**
case	**in**	**or**	**signal**	**xnor**
component	**inertial**	**others**	**shared**	**xor**
configuration	**inout**	**out**	**sla**	
constant	**is**	**package**	**sll**	
disconnect	**label**	**port**	**sra**	

Table 14: A complete list of VHDL reserved words.