The Shock and Awe

VHDL

Tutorial

©Copyright: 2007 by Bryan Mealy (01-01-2007)

Table of Contents

TABLE OF CONTENTS 2

LIST OF FIGURES 5

LIST OF TABLES 7

LIST OF EXAMPLES 8

1. INTENT AND PURPOSE 9

2. INTRODUCTION .... - 10 -

2.1 THE GOLDEN RULES OF VHDL .... - 11 -

3. VHDL INVARIANTS .... - 12 -

3.1 CASE SENSITIVITY .... - 12 -
3.2 WHITE SPACE .... - 12 -
3.3 COMMENTS .... - 12 -
3.4 PARENTHESIS .... - 13 -
3.5 VHDL STATEMENTS .... - 13 -
3.6 IF, CASE, AND LOOP STATEMENTS .... - 13 -
3.7 IDENTIFIERS .... - 14 -
3.8 RESERVED WORDS .... - 14 -
3.9 VHDL CODING STYLE .... - 15 -

4. BASIC VHDL DESIGN UNITS 16

4.1 THE ENTITY 16
4.2 THE ARCHITECTURE 19
4.3 IMPORTANT POINTS 20
4.4 EXERCISES: BASIC VHDL DESIGN UNITS 21

5. THE VHDL PROGRAMMING PARADIGM 23

5.1 CONCURRENT STATEMENTS 23
5.2 THE SIGNAL ASSIGNMENT OPERATOR “<=” 25
5.3 CONCURRENT SIGNAL ASSIGNMENT STATEMENTS 26
5.4 CONDITIONAL SIGNAL ASSIGNMENT 29
5.5 SELECTED SIGNAL ASSIGNMENT 32

5.6 THE PROCESS STATEMENT 36
5.7 IMPORTANT POINTS 36
5.8 EXERCISES: CONCURRENT, CONDITIONAL, AND SELECTIVE SIGNAL ASSIGNMENT 37

6. STANDARD MODELS IN VHDL ARCHITECTURES 41

6.1 VHDL DATAFLOW STYLE ARCHITECTURE 41
6.2 VHDL BEHAVIOR STYLE ARCHITECTURE 41
6.3 THE PROCESS STATEMENT 42
6.4 SEQUENTIAL STATEMENTS 43
6.4.1 SIGNAL ASSIGNMENT STATEMENTS 44
6.4.2 IF STATEMENTS 44
6.4.3 CASE STATEMENTS 47
6.5 CAVEATS REGARDING SEQUENTIAL STATEMENTS 50
6.6 IMPORTANT POINTS 51
6.7 EXERCISES: INTRODUCTION TO BEHAVIORAL MODELING 52

7. VHDL OPERATORS 54

7.1 LOGICAL OPERATORS 54
7.2 RELATIONAL OPERATORS 54
7.3 SHIFT OPERATORS 55
7.4 ALL THE OTHER OPERATORS 55
7.5 THE CONCATENATION OPERATOR 56
7.6 THE MODULUS AND REMAINDER OPERATORS 56

8. REVIEW (OF ALMOST EVERYTHING UP TO NOW) 58

9. USING VHDL FOR SEQUENTIAL CIRCUITS 60

9.1 SIMPLE STORAGE ELEMENTS USING VHDL 60
9.2 INDUCING MEMORY: DATAFLOW VS. BEHAVIOR MODELING 64
9.3 IMPORTANT POINTS 65
9.4 EXERCISES: BASIC MEMORY ELEMENTS 66

10. FINITE STATE MACHINE DESIGN USING VHDL 68

10.1 VHDL BEHAVIORAL REPRESENTATION OF FSMS 69
10.2 ONE-HOT ENCODING FOR FSMS 77
10.3 IMPORTANT POINTS 80
10.4 EXERCISES: BEHAVIORAL MODELING OF FINITE STATE MACHINES 82

11. STRUCTURAL MODELING USING VHDL 88

11.1 VHDL AND PROGRAMMING LANGUAGES: EXPLOITING THE SIMILARITIES 88
11.2 IMPORTANT POINTS 94

11.3 EXERCISES: STRUCTURAL MODELING 95

12. REGISTERS AND REGISTER TRANSFER LEVEL 97

12.1 IMPORTANT POINTS 102
12.2 EXERCISES: REGISTER TRANSFER LEVEL CIRCUITS 103

13. DATA OBJECTS 105

13.1 TYPES OF DATA OBJECTS 105
13.2 DATA OBJECT DECLARATIONS 105
13.3 VARIABLES AND THE ASSIGNMENT OPERATOR ":=" 106
13.4 SIGNALS VS. VARIABLES 106
13.5 DATA TYPES 107
13.6 COMMONLY USED TYPES 107
13.7 INTEGER TYPES 107
13.8 THE STD_LOGIC TYPE 108

14. LOOPING CONSTRUCTS 111

14.1 FOR AND WHILE LOOPS 111
14.2 FOR LOOPS 111
14.3 WHILE LOOPS 113
14.4 LOOP CONTROL: NEXT AND EXIT STATEMENTS 113
14.5 THE NEXT STATEMENT 113
14.6 THE EXIT STATEMENT 114

15. STANDARD DIGITAL CIRCUITS IN VHDL ..... - 115 -

15.1 RET D FLIP-FLOP (BEHAVIORAL MODEL) ..... - 115 -
15.2 FET D FLIP-FLOP WITH ACTIVE-LOW ASYNCHRONOUS PRESET (BEHAVIORAL MODEL) ..... - 115 -
15.3 8-BIT REGISTER WITH LOAD ENABLE (BEHAVIORAL MODEL) ..... - 116 -
15.4 SYNCHRONOUS UP/DOWN COUNTER (BEHAVIORAL MODEL) ..... - 116 -
15.5 SHIFT REGISTER WITH SYNCHRONOUS PARALLEL LOAD (BEHAVIORAL MODEL)- 117 -
15.6 8-BIT COMPARATOR (BEHAVIORAL MODEL) ..... - 117 -
15.7 BCD TO 7-SEGMENT DECODER (DATAFLOW MODEL) ..... - 118 -
15.8 4:1 MULTIPLEXER (BEHAVIORAL MODEL) ..... - 118 -
15.9 4:1 MULTIPLEXER (DATAFLOW MODEL) ..... - 119 -
15.10 3:8 DECODER ..... - 119 -

A *APPENDIX*: VHDL RESERVED WORDS 120

List of Figures

Figure 1: An example of VHDL case insensitivity. - 12 -
Figure 2: An example showing VHDL's indifference to white space. - 12 -
Figure 3: Two typical uses of comments. - 12 -
Figure 4: An example of parenthesis use that produces clarity and happiness. - 13 -
Figure 5: Generic form of an entity declaration. 16
Figure 6: Syntax of the port_clause. 16
Figure 7: Example black box and associated VHDL entity declaration. 17
Figure 8: An example of an input and output diagram of a circuit and its associated VHDL entity. 18
Figure 9: A few examples of bundles signals of varying content. 18
Figure 10: A black box example containing busses and its associated entity declaration. 19
Figure 11: Some common circuit that is well known to "execute" parallel operations. 24
Figure 12: VHDL code that describes the circuit of Figure 11. 24
Figure 13: Three equivalent sets of statements describing the circuit shown in Figure 11. 24
Figure 14: Algorithm code similar to the code in Figure 12. 25
Figure 15: Syntax for the concurrent signal assignment statement. 26
Figure 16: Solution to *EXAMPLE 1* 26
Figure 17: Solution to *EXAMPLE 2*. 28
Figure 18: Alternative but functionally equivalent architecture for *EXAMPLE 2*. 28
Figure 19: The syntax for the conditional signal assignment statement. 29
Figure 20: Solution to *EXAMPLE 3*. 30
Figure 21: Solution for *EXAMPLE 4*: A 4:1 MUX using a conditional signal assignment statement. 31
Figure 22: An alternative solution to EXAMPLE 4 accessing individual signals within a bundle. 32
Figure 23: Syntax for the selected signal assignment statement. 32
Figure 24: Solution to EXAMPLE 5. 33
Figure 25: Solution to *EXAMPLE 6*. 33
Figure 26: Solution to *EXAMPLE 7*. 34
Figure 27: The no-nonsense solution to standard function implementation problems. 36
Figure 28: Syntax for the process statement. 42
Figure 29: Entity declaration for circuit performing XOR function. 43
Figure 30: Dataflow and behavioral descriptions of *my_xor_fun* entity. 43
Figure 31: Syntax for the *if* statement. 44
Figure 32: Solution to *EXAMPLE 9*. 45
Figure 33: An alternate solution for *EXAMPLE 9* . 45
Figure 34: Solution to *EXAMPLE 10*. 46
Figure 35: Solution to *EXAMPLE 11*. 47
Figure 36: Syntax for the *case* statement. 48
Figure 37: Expanding the equations for *EXAMPLE 12*. 48
Figure 38: Solution to *EXAMPLE 12*. 49
Figure 39: An alternate solution for *EXAMPLE 12* . 49
Figure 40: Solution to *EXAMPLE 13*. 50
Figure 41: Examples of the concatenation operator. 56
Figure 42: Solution to *EXAMPLE 14*. 61
Figure 43: Solution to *EXAMPLE 15*. 62
Figure 44: Solution to *EXAMPLE 16*. 63
Figure 45: Solution to *EXAMPLE 19*. 64
Figure 46: Block diagram for a Moore-type FSM. 68
Figure 47: Model for VHDL implementations of FSMs. 69
Figure 48: Black box diagram for the FSM of *EXAMPLE 20*. 70
Figure 49: Solution of *EXAMPLE 20*. 70
Figure 50: Black box diagram of *EXAMPLE 20* including the state variable as an output. 72
Figure 51: Solution for *EXAMPLE 20* including state variable as an output. 72
Figure 52: Black box diagram for the FSM of *EXAMPLE 21*. 73
Figure 53: Solution for *EXAMPLE 21*. 74

Figure 54: Black Box diagram for the FSM of *EXAMPLE 22* 75
Figure 55: Solution for *EXAMPLE 22*........ 76
Figure 56: Modifications to convert entity of *EXAMPLE 22* to one-hot encoding........ 78
Figure 57: Modifications to convert state variables to use one-hot encoding........ 79
Figure 58: Modifications to convert state output of *EXAMPLE 22* to pseudo one-hot encoding. 79
Figure 59: The final solution to *EXAMPLE 24*. 80
Figure 60: Discrete gate implementation of a 3-bit comparator. 89
Figure 61: Entity and Architecture definitions for discrete gates. 90
Figure 62: Entity declaration for 3-bit comparator. 91
Figure 63: VHDL code for the top of the design hierarchy for the 3-bit comparator. 92
Figure 64: Alternative architecture for *EXAMPLE 25* using implied mapping. 93
Figure 65: An example of the same signal name crossing hierarchical boundaries. 93
Figure 66: The final solution to *EXAMPLE 28*. 98
Figure 67: The final solution of example *EXAMPLE 33*. 99
Figure 68: The final solution to *EXAMPLE 38* using a structural modeling approach. 101
Figure 69: Examples of integer declarations. 108
Figure 70: Examples of integer type declarations. 108
Figure 71: Examples of illegal assignment statements. 108
Figure 72: Declaration of the std_ulogic enumerated type. 109
Figure 73: Solution for *EXAMPLE 39*........ 110
Figure 74: The basic forms of the `for` and `while` loops. 111
Figure 75: Two equivalent for loops that (a) specify a range, (b) use a previously specifed range........ 112
Figure 76: `for` loops using the downto approach. 112
Figure 77: Two examples of while loops calculating a Fibonacci sum. 113
Figure 78: Examples of the two forms of next statements........ 113
Figure 79: Example of the two forms of `exit` statements. 114
Figure 80: VHDL code for D flip-flop. .... - 115 -
Figure 81: VHDL code for 8-bit register. .... - 116 -
Figure 82: VHDL code for Up/Down counter. .... - 116 -
Figure 83: VHDL code for shift register. .... - 117 -
Figure 84: VHDL code for Comparator. .... - 117 -
Figure 85: VHDL code for BCD to 7-Segment Decoder. .... - 118 -
Figure 86: VHDL code for 4:1 Multiplexor. .... - 118 -
Figure 87: VHDL code for 4:1 Multiplexor. .... - 119 -
Figure 88: VHDL code for 3:8 Decoder. .... - 119 -

List of Tables

Table 1: Desirable and undesirable identifiers. .... - 14 -
Table 2: A short list of VHDL reserved words........ - 15 -
Table 3: VHDL operators. 54
Table 4: Relational operators with brief explanations. 55
Table 5: Shift operators with examples. 55
Table 6: All the other operators not listed so far. 56
Table 7: Definitions of `rem` and `mod` operators. 57
Table 8: Examples of `rem` and `mod` operators. 57
Table 9: Part of the classic VHDL cheat sheet. 59
Table 10: Similarities between modules in "C" and VHDL. 89
Table 11: A comparision of entity and component declarations. 91
Table 12: Data object declaration forms. 105
Table 13: Example declarations for signal, variable, and constant data objects. 106
Table 14: Data types used in this tutorial. 107
Table 15: A complete list of VHDL reserved words. 120

List of Examples

1. Intent and Purpose

The purpose of this tutorial is to provide students with a guide to help develop the skills necessary to be able to use VHDL in the context of introductory and intermediate level digital design courses. These skills will allow students to not only navigate early courses, but also give them the skills and confidence to continue on with VHDL-based digital design and the development of skills required to solve more advanced digital design problems.

Although there are many online books and tutorials available dealing with VHDL, these sources are often troublesome for several reasons. First, much of the information regarding VHDL is either needlessly confusing or poorly written. Material with these characteristics seems to be written from the standpoint of someone who is painfully intelligent or has forgotten that their audience may be seeing the material for the first time. Secondly, the common approach for most VHDL manuals is to introduce too much extraneous information and topics too early. Most of this material would best be presented later in the presentation. Material presented in this manner has a tendency to be confusing and is easily forgotten if misunderstood or never applied. The approach taken by this manual is to provide students with only what they need to know to quickly get them up and running in VHDL. As with all learning, once you obtained and applied some useful information, it is much easier to build on what you know as opposed to continually adding information that is not directly applicable to the subjects at hand.

The intent of this tutorial is to present topics in the context of the average 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 basic VHDL concepts, students should 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 similar 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 is controlled by 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 online VHDL references and tutorials. 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 yields 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. The "HDL" acronym is vastly important. First, VHDL is a true computer language with the accompanying set of syntax and usage rules. But different from higher-level computer languages, VHDL is used to primarily to "describe hardware".

The tendency for most people familiar with a higher-level computer language such as C or Java is to view VHDL as just another computer language. This is not altogether a bad approach in that such a view facilitates a quick understanding of the language syntax and structure. The common mistake made by this approach is to attempt to program in VHDL as you would program a higher-level computer language. Higher-level computer languages are sequential in nature; VHDL is not. VHDL is used to describe hardware. Another way to look at this is that higher-level computer languages are used to describe algorithms (sequential execution) and VHDL is used to describe hardware (parallel execution). This inherent difference necessarily should encourage you to rethink how you write your VHDL code. Attempts to write VHDL code with a higher-level language style generally results in VHDL code that no one understands. Moreover, the tools used to synthesize the circuits described by this type of code have a tendency to generate circuits that generally don't work correctly and have bugs that are nearly impossible to trace. And if the circuit does actually work, it will most likely be inefficient due to the fact that the resulting hardware was needlessly large and overly complex. This problem is compounded as the size and complexity of your circuits becomes greater.

There are two primary purposes for hardware description languages such as VHDL. First, VHDL can be used to *model digital circuits and systems*. Although the word *model* is one of those overly used words in engineering, in this context is simply refers to a description of something that presents a certain level of detail. The nice thing about VHDL is that the level of detail is unambiguous due to the rich syntax rules associated with VHDL. In other words, VHDL provides everything that is necessary in order to describe any digital circuit. Likewise, a digital circuit/system is any circuit that processes or stores digital information. Secondly, having some type of circuit model allows for the subsequent simulation and/or testing of the circuit. The VHDL model can also be translated into a form that can be used to generate actual working circuits. The VHDL model is magically[1] interpreted by software tools in such a way as to create actual digital circuits in a process known as *synthesis*.

These two main purposes of VHDL match what VHDL is used for in introductory and intermediate digital design courses. Although 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 model circuits. The tendency here is to prefer the graphical approach because it has such a comfortable 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 to digital circuit design is to start with a system that is to be able to describe exactly how your digital circuit works (in other word, 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.

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

Modeling digital circuits with VHDL represents a modern digital design paradigm. In other words, the emphasis in modern digital design to develop skills that are immediately applicable and can be easily built upon. Modern digital design is more about appropriately[2] modeling digital circuits and maintaining a quality description of the circuit. All that is left now is learning the description language and the associated digital design paradigm.

2.1 The Golden Rules of VHDL

Before you start, here are a couple of items that you should never forget when working with VHDL.

VHDL is a hardware design language. Although most people have probably had previously exposure to some type of higher-level computer language, these skills are only indirectly applicable to VHDL. When you are working with VHDL, you are not "programming", you are "designing hardware". Your VHDL code should reflect this fact. If your VHDL code appears too similarly to code of a higher-level computer language, it's probably bad VHDL code. This is vitally important.

Have a general concept of what your hardware should look like. Although VHDL is vastly powerful, if you don't understand the basic digital constructs, you'll likely not be able to generate efficient digital circuits. Digital design is similar to higher-level language programming in that even the most complicated programming at any level can be decomposed in to some simple programming constructs[3]. There is a strong analogy to digital design in that even the most complicated digital circuits can be described in terms of basic digital constructs[4]. In other words, if you are not able to roughly envision the digital circuit you're trying to model in terms of basic digital circuits, you'll probably misuse VHDL, thus angering the VHDL gods. VHDL is cool, but it is not as magic as it initially appears to be.

[2] The word appropriately here means that you stay within the designated boundaries of the modeling system. In this case, you understand the VHDL design paradigm and you use it to your advantage that than fighting it.
[3] It's the structured programming thing all over again…
[4] Basic digital constructs include MUXes, decoders, basic logic, etc.

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 shown in Figure 1 have the exact same meaning (don’t worry about what the statement actually means though). Keep in mind that Figure 1 shows an example of VHDL case sensitivity and not good VHDL coding practices.

`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. Once again, Figure 2 is not an example of good VHDL coding style. Note that Figure 2 once again advertised that VHDL is not case sensitive.

`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

Comments in VHDL begin with “`--`“ (two consecutive dashes). The VHDL synthesizer ignores anything after the two dashes and 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.

Appropriate use of comments increases both the readability and the understandability of VHDL code. The general rule is to comment any line or section of code that may not be clear to a reader of your code besides yourself. The only inappropriate use of a comment is to state something that is patently obvious. It’s hard to image code that has too few comments: don’t be shy; use lots of comments. Research has shown that using lots of appropriate comments is actually a sign of high intelligence.

```
-- 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_reg <= 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 a few 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 lower statement more clear. Don't worry about the syntax presented in Figure 4.

```
if x = '0' and y = '0' or z = '1' then
    blah;  -- some useful statement
    blah;  -- some useful statement
end if;

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

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

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 often omitted during initial coding. The main challenge them becomes to know what constitutes a VHDL statement in order to know when to include semicolons. The VHDL synthesizer 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)
		`DDE_SUX`	no comment…

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 as 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, indifference to white space, and lax rules on parenthesis creates a virtual coding anarchy. The emphasis in coding style is therefore placed on readability. Unfortunately, the level of readability of any document, particularly coding text, 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, using self-commenting identifiers, and provided proper comments when and 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 code, or someone who signs your paycheck at the end of the day. These are the people you want to please. These people are most likely massively busy and more than willing to make a subjective glance at your code. Nice looking code will slant such subjectivity in your favor.

- If in doubt, 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 while on the path to creating an even more readable 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.

- A properly formatted document explicitly presents information about your design that would not otherwise be readily apparent. This is particularly true with using proper indentation.

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 (which of course can be other black boxes). 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 properly describing the architecture. Our approach here is to present an introduction to writing VHDL code by describing the *entity* and the moving onto the details 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 of the underlying circuitry. In VHDL terms, the black box is described by an *entity declaration.* The syntax[5] of the entity declaration is shown in Figure 5. 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.

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

Figure 5: Generic form of an entity declaration.

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

Figure 6: Syntax of the port_clause.

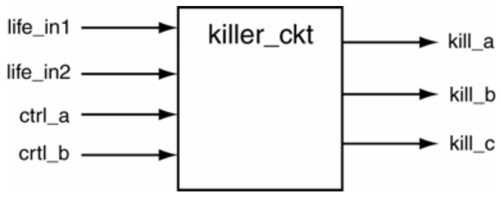
A “port” is essentially a conduit that interfaces a signal inside the box with a signal on the outside world. This signal can be either an input to the underlying circuit from the outside world or an output from the underlying circuit to the outside world. The *port_clause* is nothing more than a list of the signals from the

[5] 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. The concept of boldness is for readability only; your VHDL synthesizer won’t have a use for it.

underlying circuit that are available to the outside world which is why it is often referred to as an interface specification. 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[6]. 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 but 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 the 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 data type. This is a requirement.
- The VHDL compiler allows several port names to be included on a single line. The port names are delineated by commas. This is not a requirement; you should therefore strive for readability.
- The port names are somewhat lined up in a feeble attempt to increase readability. This again is not a requirement but you should always be striving for readability. Remember that white space is ignored by the compiler.
- A comment was included which simulates the telling of almost intelligent things.
- A black box diagram of the circuit is also provided. Once again, drawing some type of diagram helps with any VHDL code that you may be writing. Remember… don't be a wuss; draw a picture.

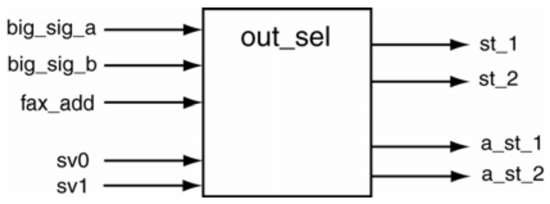


```
----------------------------------------------------------------
-- Here's my interface description of the killer circuit
-- It does a lot of killer things.
----------------------------------------------------------------
entity killer_ckt is
   port (     life_in1 : in std_logic;
              life_in2 : 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.

[6] There are actually other mode specifiers but we'll not discuss them until a later time.



```
--------------------------------------------------------------
-- 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. Unfortunately, the word *bus* in digital lingo also refers to established data transfer protocols. To disambiguate the word bus, we'll be using the word *bundle* to refer to a set of similar signals and *bus* to refer to a protocol.

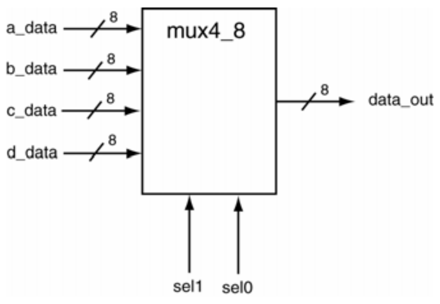
Busses are easily described in the VHDL entity. All that is needed is new data type and a special notation to indicate when a signal is a bundle or not. A few examples are shown in Figure 9. In these examples note that the mode remains the same but the type has changed. The ***std_logic*** data type 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 bundles signals of varying content.

As you can see by examining Figure 9, there are two possible methods to describe the signals in the bundle. 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 not to forget the orientation of signals when you are using this notation in your VHDL model.

A more appropriate introduction to bundles 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 bundle. The slash across the signal line indicates the signal is a bundle and the associated number specifies the number of signals in the bundle. 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;
          data_out: 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 need to be mentioned about VHDL architectures at this point. Examples of VHDL entity-architecture pairs are presented throughout the remainder of this tutorial. 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.

4.3 Important Points

- The entity declaration describes the inputs and outputs to a circuit. This set of signals is often referred to as the interface to the circuit since these signals are what the circuitry external to the entity uses to interact with the given circuit.

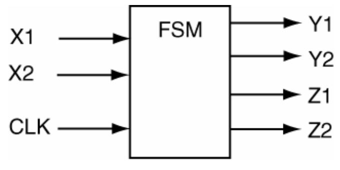
- Signals described in the entity declaration include a mode specifier and a type. The mode specifier can be either an *in* or an *out* while the type is either a *std_logic* or *std_logic_vector*.

- The word *bundle* is preferred over the word *bus* when dealing with multiple signals that share a similar purpose. The word *bus* has other connotations that are not consistent with the bundle definition.

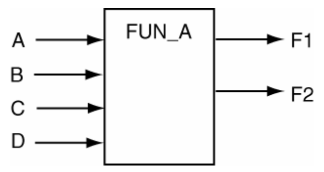
- Multiple signals that share a similar purpose should be declared as a bundle using a *std_logic_vector* type. Bundled signals such as these are always easier to work with in VHDL as compared to scalar types such as *std_logic*.

4.4 Exercises: Basic VHDL Design Units

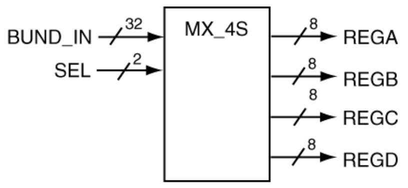
1. What is referred to by the word *bundle*?
2. Why is the word bundle more appropriate to use than the word *bus*?
3. What is a common method of representing bundles in black box diagrams?
4. Why is it considered a good approach to always draw a black box diagram when using VHDL to model digital circuits?
5. Write VHDL entity declarations that describe the following black box diagrams:



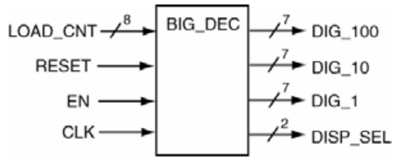
(a)



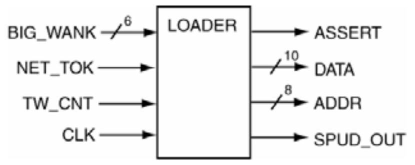
(b)



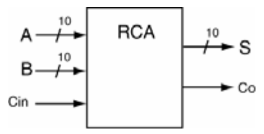
(c)



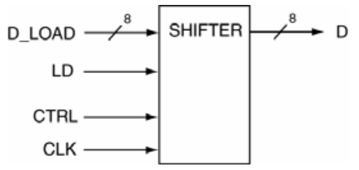
(d)



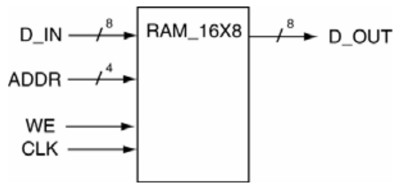
(e)



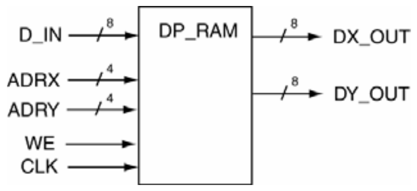
(f)



(g)



(h)



(i)

6. Provide black box diagrams that are defined by the following VHDL entity declarations:

```
entity ckt_a is
   port ( in_a : in std_logic;
          in_b : in std_logic;
          in_c : in std_logic
         out_f : out std_logic);
end ckt_a;
```

(a)

```
entity ckt_b is
   port (     LDA,LDB : in std_logic;
              ENA,ENB : in std_logic;
          CTRLA,CTRLB : in std_logic;
            OUTA,OUTB : out std_logic);
end ckt_b;
```

(b)

```
entity ckt_c is
   port (   bun_a,bun_b_bun_c : in std_logic_vector(7 downto 0);
                 ld_a,ldb,ldc : in std_logic;
          reg_a, reg_b, reg_c : out std_logic_vector(7 downto 0);
end ckt_c;
```

(c)

```
entity ckt_d is
   port ( big_bunny : in std_logic_vector(31 downto 0);
                 mx : in std_logic_vector(1 downto 0);
           byte_out : out std_logic_vector(7 downto 0));
end ckt_d;
```

(d)

```
entity ckt_e is
   port ( RAM_CS,RAM_WE,RAM_OE : in std_logic;
              SEL_OP1, SEL_OP2 : in std_logic_vector(3 downto 0);
                   RAM_DATA_IN : in std_logic_vector(7 downto 0);
                   RAM_ADDR_IN : in std_locic_vector(9 downto 0);
                  RAM_DATA_OUT : out std_logic_vector(7 downto 0));
end ckt_e;
```

(e)

```
entity ckt_f is
   port ( rss_bytes, rss_sux, rss_dogface : in std_logic;
              worthless, way_bad, go_away : in std_logic_vector(23 downto 0);
                 big_joke, insecure, lazy : out std_logic_vector(32 downto 0);
                                      SMD : out std_logic_vector(7 downto 0));
end ckt_f;
```

(f)

7. The following two entity declarations contain two of the most common syntax errors made in VHDL. What are they?

```
entity ckt_a1 is
   port ( J,K : in std_logic;
          CLK : in std_logic
            Q : out std_logic;)
end ckt_a;
```

(a)

```
entity ckt_d is
   port ( mr_fluffy : in std_logic_vector(15 downto 0);
           mux_ctrl : in std_logic_vector(3 downto 0);
           byte_out : out std_logic_vector(3 downto 0);
end ckt_d;
```

(b)

5. The VHDL Programming Paradigm

The last section introduced the idea of the basic design units of VHDL: the entity and the architecture. Most of 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 is 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 and once we finish that one thing, we move onto the next thing. 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 significantly different. Whereas a processor steps one-by-one through a set of statements, VHDL has the ability to "execute" a virtually unlimited number of statements at the same time (in other words, in parallel). Once again, the key to remember here is that we are designing hardware. Parallelism, or things happening *concurrently*, in the context of hardware is a much more straight-forward concept in hardware land than it is in the world of software. If you've had any introduction to basic digital hardware, you're most likely already both familiar and comfortable with the concept of concurrency.

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 that any gate input changes, there is a possibility that the gate 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. Once changes to the gate inputs occur, the inputs are re-evaluated and the gate outputs may change accordingly. Although the circuit in Figure 11 only shows a few gates, this idea of concurrent operation of all the elements in the circuit is the same in all digital circuits no matter how large or complex they are.

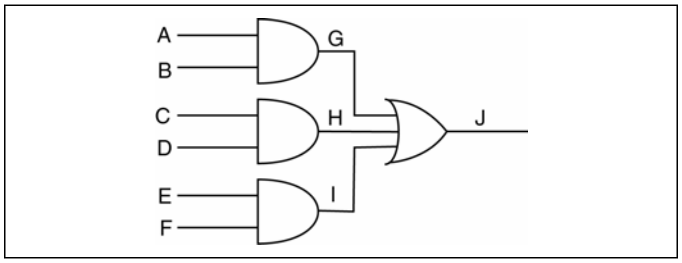


Figure 11: Some common circuit that is well known to "execute" parallel operations.

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 use text to describe some circuit that is inherently parallel? We didn'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 and in a sequential manner 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 the VHDL statements by definition express concurrency of execution.

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 better stated, actions that occur *concurrently*. Once again, the concept of concurrency is a key concept in VHDL. Keep this notion in mind 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 is discussed more completely in the next section.

As a consequence of the concurrent nature of VHDL statements, the three chunks of code appearing in Figure 13 are equivalent to the code shown in Figure 12. Once again, since the statements are interpreted as occurring concurrently, the order that these statements appear in your VHDL source code make no difference. Generally speaking, it would be a better idea to describe the circuit as shown in Figure 12 since it reflects somewhat of a natural organization of statements.

```
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.

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

Figure 13: Three equivalent sets of statements describing the circuit shown in Figure 11.

Figure 14 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. In other words, the statements shown in Figure 14 are designed to be executed by some type of processing element. This processing element executes one statement and then moves onto the next statement. Once again, although the two snippets of code appear somewhat similar, they have completely different meanings. Keep in mind that if you were to rearrange the statements shown in Figure 14, they would have a completely different meaning than the similar VHDL code due to the fact that the statements in VHDL are interpreted as occurring concurrently.

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

Figure 14: 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 are 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'll 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: several considerably different pieces of VHDL code an actually generate the exact same hardware. 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. Some of the VHDL models presented in this tutorial are presented to show that something "can" be done a certain way, but that does not necessarily mean they "should" be presented in that way.

5.3 Concurrent Signal Assignment Statements

The general form of a concurrent signal assignment statement is shown in Figure 15. In this case, *target* is a signal that receives the values of the *expression*. An expression is defined by a constant, a signal, or a 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.

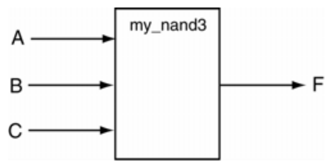
```
target <= expression;
```

Figure 15: 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 circuit 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 exa_nand3 of my_nand3 is
begin
    F <=  NOT (A AND B AND C);
end exa_nand3;

architecture exb_nand3 of my_nand3 is
begin
  F <= A NAND B NAND C;
end exb_nand3;
```

Figure 16: 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 16. The listed lines are more than is needed for this example but they will be required in later examples. These lines will not be included in any of the following 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. Moreover, these logic operators are considered *binary* operators in that they operate on the two values appearing on the left and right side of the operator. The **NOT** operator is a *unary* operator in that it only operates on the value appearing to the right of the operator.

- Two architectures have been provided in this solution; they are both associated with the same entity. This is fair common practice in complex circuits but is not overly useful in most VHDL design.

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 any concurrent statement in VHDL 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. This is a key concept in truly understanding the VHDL (so you may want to read that sentence a few more times). The idea of concurrency is more clearly demonstrated in the *EXAMPLE 2*.

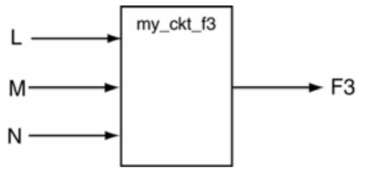
EXAMPLE 2

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, especially if you can't think of an easier way to represent the function. We hope the VHDL compiler would somewhere along the way automatically reduce such functions but we'll deal with concept later. The reduced equation is given below. The black box diagram and associated VHDL code is shown in Figure 17.

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



```
----------------------------------------------------------------------
-- 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_2 of my_ckt_f3 is
begin
   F3 <= ((NOT L) AND (NOT M) AND N) OR (L AND M);
end f3_2;
```

Figure 17: 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. More importantly, this example demonstrates the use of signal statements.

An alternative solution to *EXAMPLE 2* is provided in Figure 18. Although this is only the second VHDL example you've seen in this tutorial, it represents a massively important concept in VHDL. The solution shown in Figure 18 uses some special statements in order to implement the circuit. These special statements are used to provide what is often referred to as *intermediate results*. This approach is analogous to declaring extra variables in an algorithmic programming language to be used for specifically for storing intermediate results. The need for intermediate results in VHDL is provided by the declaration of extra signal values which are often referred to *intermediate signals*. Note in Figure 18 that the declaration of the intermediate signals is similar to the port declarations appearing in the entity declaration except the mode specification is missing. The intermediate signals must be declared within the body of the architecture because they have no linkage to the outside world and thus do not appear in the entity declaration. In other words, the outside world does not need to know about these signals so they only need appear in the architecture. Note that the intermediate signals are declared in the architecture body but appear before the begin statement.

```
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 18: Alternative but functionally equivalent architecture for *EXAMPLE 2*.

Despite the fact that the architectures of f3_1 and f3_2 of Figure 17 and Figure 18 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*.

Although the approach of using intermediate signal is not mandatory for this example, their use brings up some good points. First, the use of intermediate signals is the norm for most VHDL models. The use of intermediate signals was optional in this example due to the fact that the example was modeling a relatively simple circuit. As circuits become more complex, there are many occasions where intermediate signals must be used. Secondly, using intermediate signals is somewhat of a tool that you'll often need to use in your VHDL models. The thought here is that you're trying to describe a digital circuit using a textual description language: you'll often need to use intermediate signals in order to accomplish your goal of modeling the circuit. The important idea here is that the use of intermediate signals allows you to more easily model digital circuit but does not make the generated hardware more complicated. The tendency in using VHDL is to think that since there is more text written on your page, that the circuit you're describing and/or the resulting hardware is larger or more complex. This is simply not true. The main theme of VHDL is that you should use the VHDL tools at your disposal in order to model your circuits in the simplest possible way. Simple circuits have a higher probability of being understood and actually working. But most importantly, a simple VHDL model is not related to the length of the actual VHDL code.

And lastly, *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, particularly an equation that has been reduced by you the digital circuit designer, 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 that can be assigned to the target. Each of the expressions is associated with a certain condition. The individual conditions are evaluated sequentially in the conditional signal assignment statement until the first condition evaluates as TRUE. In this case, the associated expression is evaluated and assigned to the target. Only one assignment is applied per assignment statement.

The syntax of the conditional signal assignment is shown in Figure 19. 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 19: The syntax 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 to *EXAMPLE 3* is listed in Figure 20.

```
-- the entity is the same as before
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 20: 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 associated expressions are the single digits surrounded by single quotes; the associated conditions follow the **when** keyword. In other words, there is only one signal assignment operator used for each conditional signal assignment statement.

- The last expression in the signal assignment statement is the catch all condition. If none of the conditions listed above the final expression evaluate as TRUE, the last expression is assigned to the target.

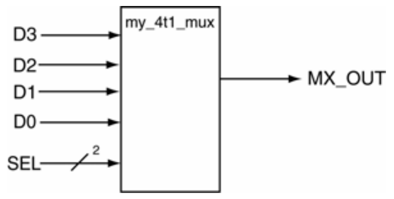
- The solution uses *relational operators*. There are actually six different relational operators available in VHDL. Two of the more common relational operators are the “=” and “/=” relational operators which are the “is equal to” and “is not equal to” operators, respectively. More on operators is provided later.

There are more intelligent uses of the conditional signal assignment statement. One of the classic ones is for the implementation of a multiplexor (MUX). The next example is a typical conditional signal assignment implementation of a MUX.

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 includes the now famous black box diagram and the associated entity statement. The VHDL portion of the solution is shown in Figure 21.



```
-----------------------------------------------------------------
-- 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 21: 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 21.

- 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 bundle SEL lines are accessed using double quotes around the specified values. In other word, signal quotes are used to describe values of single signals while double quotes are used to describe values associated with multiple signals, or bundles.

- 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. Generally speaking, you should clearly provide all the options in the code and not rely on a catch all statement for intended signal assignment.

Remember, a conditional signal assignment is a type of concurrent statement. In this case, the conditional signal assignment statement is executed any time a change occurs on the conditional signals (the signals listed in the expressions on the right side of the signal assignment operator). 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 in algorithmic programming languages. We'll touch more upon the relationship once we start talking about sequential statements.

The concept of working with bundles is massively important in VHDL. Generally speaking, if you can use a bundle as opposed to individual signals, you should. *EXAMPLE 4* referenced bundled signals in the condition portion of the conditional signal assignment statement. Often times is required that individual signals within a bundle be accessed. When this is the case, there is special syntax used to access individual signals from the bundle. Be sure to note that the code shown in Figure 22 is equivalent to but probably not as clear as the code shown in Figure 21. Be sure to note the similarities and differences.

```
-----------------------------------------------------------------
-- entity and architecture of 4:1 Multiplexor implemented using
-- conditional signal assignment. The conditions access the
-- individual signals of the SEL bundle in this model.
-----------------------------------------------------------------
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(1) = '1' and SEL(0) = '1') else
             D2 when (SEL(1) = '1' and SEL(0) = '0') else
             D1 when (SEL(1) = '0' and SEL(0) = '1') else
             D0 when (SEL(1) = '0' and SEL(1) = '0') else
             '0';
end mux4t1;
```

Figure 22: An alternative solution to EXAMPLE 4 accessing individual signals within a bundle.

5.5 Selected Signal Assignment

Selective signal assignment statements are the third form of concurrent statements that we'll examine. As with conditional signal assignment statements, selective signal assignment statements only have one assignment operator. Selective signal assignment statements differ from conditional assignment statements in that assignments are based upon the evaluation of one expression. The syntax for the selected signal assignment statement is shown in Figure 23.

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

Figure 23: 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 17 . The solution is shown in Figure 24.

```
-- yet another solution to the F3 example
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 24: Solution to EXAMPLE 5.

One thing to notice about the solution listed in Figure 24 is the use of the “when others” clause as the final entry in the selective signal assignment statement. In actuality, the middle clause (“`‘0’ when ‘0’`”) could be removed from the solution without changing the meaning of the statement. In general, it is considered good VHDL programming practice to include all the *expected* cases in the selective signal assignment statement followed by the “when others” clause. It is generally bad programming (and sometime impossible) to include all the *possible* cases in the selective signal assignment statement.

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 25. The black box diagram for this example is the same as shown in Figure 21 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 25: 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 5*. The general appearance is the same. Both solutions are also much more pleasing to the eye than if the MUX had been modeled using only concurrent signal assignment statements.

- A **when others** clause is used again. In the case of *EXAMPLE 6*, the output is assigned the constant value of '0' when the other listed conditions of the *chooser_expression* are not met.

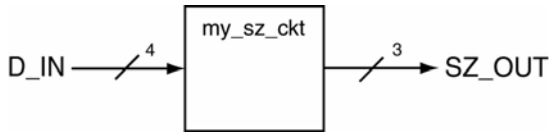
- The circuit used in this example was a 4:1 MUX. In this case, each of the conditions of the *chooser_expression* is accounted for in the body of the selected signal assignment statement. This is not a requirement however. The only requirement here is that the line containing the **when others** keywords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

EXAMPLE 7

Write the VHDL code that implements the following circuit. The circuit contains an input bundle containing four signals and an output bundle containing three signals. The input bundle, **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: This is an example of a generic decoder-type circuit. The solution to *EXAMPLE 7* is shown in Figure 26.



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

architecture spec_dec of my_sz_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 26: 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* listed in the first line of the selective signal assignment statement.

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.

EXAMPLE 8

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: This is the same old example problem. The problem with the previously solutions to this example are that they required the user to somehow reduce the function before it was implemented. In this modern day of circuit digital circuit design, you score the most points when you allow the VHDL synthesizer to do the work for you. The solution to this example hopefully absolves you from ever again using a Karnaugh map, or God forbid, Boolean algebra, to reduce a function.

The first thing to note about the problem is that it is presented in truth table form. In order to make the solution more straight-forward, you should first convert the problem to a more compact representation. In other words, this problem cries out for compact minterm form. This equivalent representation is shown below:

$$\boxed{F3(L,M,N) = \sum(1,6,7)}$$

The final solution to the previous example is shown in Figure 27. Listed below are a couple of new thoughts regarding the provided solution.

- The VHDL model indicates no attempt to reduce the code. Any reduction of the code is thus a responsibility of the VHDL synthesizer.
- The entity declaration describes the inputs as single signals. This is not the best approach to the coding style used in this problem. The solution is to create a bundle out of the individual input signals and use that bundle in the selective signal assignment statement. This indicates another common situation when an intermediate signal must be used. The intermediate bundle signal is created by the signal declaration. Assignment of the **L**, **M**, and **N** signals to the bundle is accomplished using the concatenation operator: “&”.

- Only the TRUE cases (the cases where the function outputs are ‘1’) are listed in this solution. The when others clause is used to handle the all of the other cases. This approach saves listing all of the cases where the function output is ‘0’.

```
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
              F3 : out std_logic);
end my_ckt_f3;

architecture f3_8 of my_ckt_f3 is
   signal t_sig : std_logic_vector(2 downto 0); -- declaring the bundle
begin
   t_sig <= (L & M & N); -- assigning the bundle using concatenation operator

   with (t_sig) select
   F3 <= ‘1’ when “001” | “110” | “111”,
         ‘0’ when others;
end f3_8;
```

Figure 27: The no-nonsense solution to standard function implementation problems.

5.6 The Process Statement

The process statement is the final concurrent statement we’ll 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.

5.7 Important Points

- The entity/architecture pair form the interface and functional description, respectively, of how a digital circuit should operate.

- The main design consideration in VHDL modeling supports the fact that digital circuits operate in parallel. In other words, the various design units in a digital design process and store information independently of each other. This is the major difference between VHDL and higher-level computer programming languages.

- The major signal assignment types in VHDL are concurrent signal assignment, conditional signal assignment, selective signal assignment, and process statements. Each concurrent statement is interpreted as acting in parallel (concurrent) to other concurrent statements.

- Signals that are declared as OUTs in the entity declaration cannot appear on the right side of a signal assignment operator. The characteristic is prevented from being a problem by the declaration and use of intermediate signals. Intermediate signals are similar to signals declared in entities except that they contain no mode specifier. Intermediate signals are declare in the architecture body before the begin statement.

- There are generally multiple approaches it modeling any given digital circuit. In other words, various types of concurrent statements can be used to describe the same circuit. The designer should strive for clarity in digital modeling and allow the VHDL synthesizer to sort out the details.

5.8 Exercises: Concurrent, Conditional, and Selective Signal Assignment

1. For the following function descriptions, write VHDL models that implement these functions using concurrent signal assignment.

(a) $F(A,B,C) = \sum(2,3,5)$

(b) $F(A,B,C) = m_2 + m_5$

(c) $F(A,B,C,D) = \sum(4,3,15)$

(d) $F(W,X,Y,Z) = \sum(12,14,15)$

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

(f) $\overline{F}(A,B,C) = \sum(5,7)$

(g) $\overline{F}(W,X,Y) = \sum(2,3,4,7)$

(h) $\overline{F}(R,S,T) = \sum(0,1,4,7)$

(i) $F(A,B,C) = \prod(1,2)$

(j) $F(A,B,C) = M_1 \cdot M_6 \cdot M_7$

(k) $F(L,M,N) = \prod(1,6)$

(l) $F(W,X,Y,Z) = \prod(1,4,5,8,13)$

(m) $F(A,B,C,D) = (\overline{A} + \overline{B}) \cdot (B + C + \overline{D}) \cdot (\overline{A} + D)$

(n) $\overline{F}(A,B,C) = \prod(0,2,7)$

(o) $\overline{F}(A,B,C) = \prod(0,1,5)$

(p) $\overline{F}(W,X,Y,Z) = \prod(5,7)$

2. For the following function descriptions, write VHDL models that implement these functions using both conditional and selective signal assignment.

(a) $F(A,B,C) = \sum(2,3,5)$

(b) $F(A,B,C) = m_2 + m_5$

(c) $F(A,B,C,D) = \sum(4,3,15)$

(d) $F(W,X,Y,Z) = \sum(12,14,15)$

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

(f) $F(A,B,C) = \prod(1,2)$

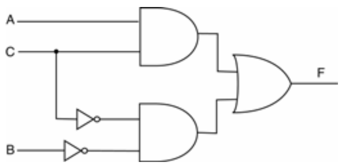
(g) $F(A,B,C) = M_1 \cdot M_6 \cdot M_7$

(h) $F(L,M,N) = \prod(1,6)$

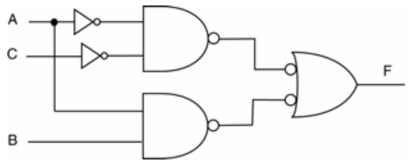
(i) $F(W,X,Y,Z) = \prod(1,4,5,8,13)$

(j) $F(A,B,C,D) = (\overline{A} + \overline{B}) \cdot (B + C + \overline{D}) \cdot (\overline{A} + D)$

3. Implement the following functions using concurrent signal assignment.

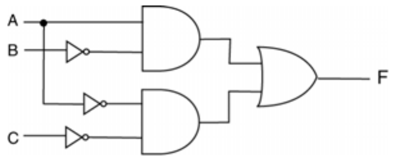


(a)

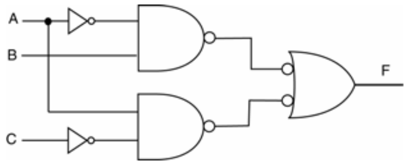


(b)

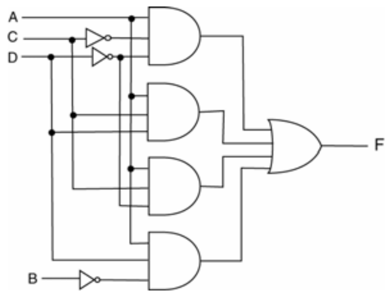
4. Implement the following functions using concurrent, conditional, or selective signal assignment.



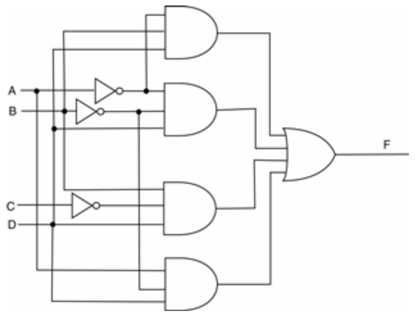
(a)



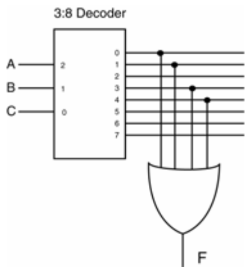
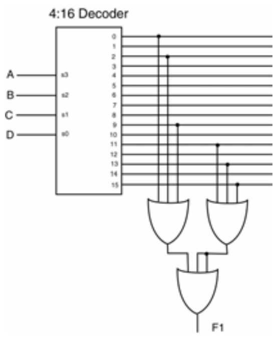
(b)



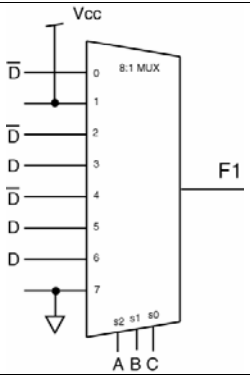
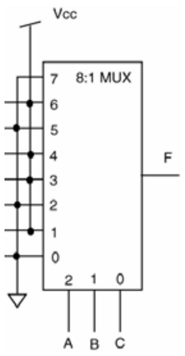
(c)



(d)

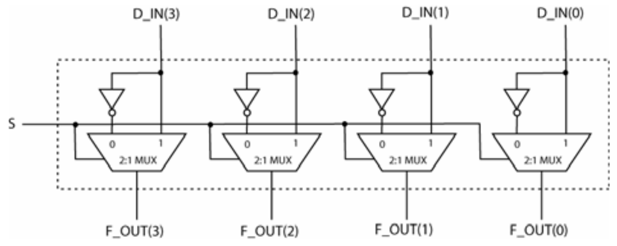
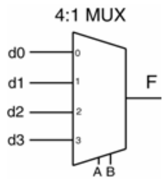
	

(e) (f)

(g) (h)

5. Model the following circuits using concurrent, conditional, or selective signal assignment.

(a) (b)

6. Provide a VHDL model of an 8-input AND gate using concurrent, conditional, and selective signal assignment.

7. Provide a VHDL model of an 8-input OR gate using concurrent, conditional, and selective signal assignment.

8. Provide a VHDL model of an 8:1 MUX using both conditional and selective signal assignment.

9. Provide a VHDL model of a 3:8 decoder using both conditional and selective signal assignment; consider the decoder's outputs to be active high.

10. Provide a VHDL model of a 3:8 decoder using both conditional and selective signal assignment; consider the decoder’s outputs to be active low.

6. Standard Models in VHDL Architectures

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 describe 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 mixture 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 an existing set of VHDL models. In other words, structural modeling supports the interconnection of black boxes but does not have the ability to describe the logic functions used to model the circuit operation. For this reason, it is less of a design method and more of an approach for interfacing previously designed 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 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. In other words, if you exclusively use concurrent, conditional, and selective signal assignment statement in your VHDL models, you have used a dataflow model. If you were to re-examine some of the examples we've done so far, you can in fact sort of see how the data flows through the circuit. To put this in another way, if you have a working knowledge of digital logic, it's fairly straight-forward to imagine the underlying circuitry in terms of standard logic gates. 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. Dataflow modeling works fine for small and relatively simple circuits. But as circuits become more complicated, it is often 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. Whereas in dataflow modeling you somewhat needed to have a feel for the underlying logic in the circuit, behavioral models provide you with various tools to describe how the circuit will behave and leaves the implementation details up to the synthesis tool. In other words, dataflow modeling describes how the circuit should look in terms of gates whereas behavioral modeling describes how the circuit should act. For these reasons, behavioral modeling is considered higher-up on the circuit abstraction level as compared to dataflow models. It is the VHDL synthesizer tool that decides the actual circuit implementation. 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 28. 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. The *label* listed in Figure 28 is optional but should always be included to promote the self-commenting of your VHDL code.

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

Figure 28: Syntax for the process statement.

Figure 29 shows an entity declaration for a XOR function. Figure 30 shows both a dataflow and behavioral style of architecture for the entity of Figure 29. 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 any of the signals listed on the right side of the signal assignment operator, the signal on the left side of the operator is 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. These two 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 30 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 29: 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 30: 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 initiated when a change in the signal contained in the process sensitivity list occurs. Generally speaking, execution of statements within the process body continues until the end of the process body is reached. The strangeness evokes a philosophical dilemma: The process statement is a concurrent statement yet it is comprised of sequential statements. This is actually a tough concept to grasp. After years of contemplation I’m only starting to grasp the reality of this strange contradiction. The key to understanding sequential evaluation of statements occurring in a concurrent statement remains hidden in the interpretation of VHDL code by the synthesizer. And since the ins and outs of this interpretation are not always readily apparent, some implementation details must be taken for granted until the time comes when you really need to fully understand the process. The solution to not fully comprehending this distinction is somewhat overcome by keeping your process statements as simple as possible. The tendency is to use the process statement as a repository for a bunch of loosely related sequential statements. Although syntactically correct, the code is not understandable in the context of digital circuit generation. You should strive to keep your process statements simple. Divide up your intended functionality into several process statements that communicate with each other rather than one giant, complicated, bizarre process statement. Remember, process statements are concurrent statement: they all can be executed concurrently. Try to take advantage of this feature in order to simplify your circuit descriptions.

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’s analog in the dataflow models. 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 strikingly similar. 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 30 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 31.

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

Figure 31: 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 31, 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 31 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 *if* statement will be evaluated. This has deep ramifications that we'll discuss later.

EXAMPLE 9

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 32. 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 32: Solution to *EXAMPLE 9*.

This is probably not the best was to implement a logic 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 9* is shown in Figure 33. A more intelligent use of the *if* statement is demonstrated in *EXAMPLE 10*.

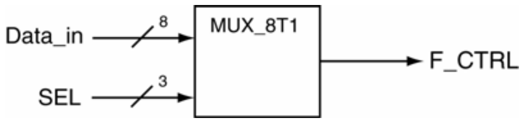
```
architecture bad_example of my_ex_7 is
begin
   proc1: process(A,B,C)
   begin
      if (A = ‘0’ and B = ‘0’ and C = ‘0’) or (B = ‘1’ and C = ‘1’) then
         F_OUT <= ‘1’;
      else
         F_OUT <= ‘0’;
      end if;
   end process proc1;
end bad_example;
```

Figure 33: An alternate solution for *EXAMPLE 9* .

One final comment on process statements now that we've discussed a few pieces of code. Process statements can be preceded with an optional label. A label should always be included with process statements as a form of self-commenting. This of course means that the label should be meaningful in terms of describing the purpose of the process statement. Providing good label names is somewhat of an art form but keep in mind that it's easier to provide a meaningful name to a process that is not trying to do too much.

EXAMPLE 10

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 10* is shown in Figure 34.

```
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_mux: 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_mux;
end my_8t1_mux;
```

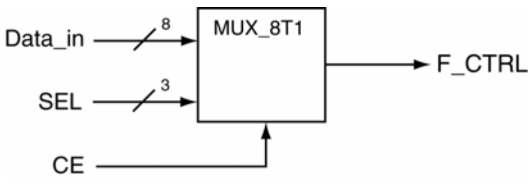
Figure 34: Solution to *EXAMPLE 10*.

The solution to *EXAMPLE 10* shown in Figure 34 uses some new syntax. The entity uses bundle signal but the associated architecture needs to access individual elements of these bundles. The solution is to use the bus index operator to access internal signals of the bus. This is comprised of a (number representing an index) placed in parenthesis. Bus index operators are used extensively in VHDL and were previously mentioned in this tutorial. The solution to *EXAMPLE 10* shows a more typical use of the operator than was previously mentioned.

One other thing to notice about the solution in *EXAMPLE 10* is that every possible combination of the select variable is accounted for in the code. It would be possible to remove the final elsif statement in the code shown in Figure 34 and place the associated signal assignment in the else clause. But this is not considered good VHDL practice and should be avoided at all costs. The justification for this is that it will modify the readability of the code but not alter the hardware generated by the code.

EXAMPLE 11

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 10*. When CE is ‘0’, the output of the MUX is ‘0’.



Solution: The solution to *EXAMPLE 11* is strangely similar to the solution of *EXAMPLE 10*. Note in this solution that the *if* statements can be nested to attain various effects. The solution to *EXAMPLE 11* is provided in Figure 35.

```
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_mux: 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_mux;
end my_8t1_mux;
```

Figure 35: Solution to *EXAMPLE 11*.

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 sole dependent upon the first *when* branch to evaluate as true. The syntax for the *case* statement is shown in Figure 36.

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

Figure 36: 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 but should be used for general good VHDL programming.

EXAMPLE 12

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

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

Solution: This solution 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 $(\mathrm{A} + \overline{\mathrm{A}})$. This factoring operation is shown in Figure 37.

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

Figure 37: Expanding the equations for *EXAMPLE 12*.

The other part of the solution is shown in Figure 38. 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 38: Solution to *EXAMPLE 12*.

Another similar approach to *EXAMPLE 12* is to use the “don’t care” feature built into VHDL. This allows the logic function to be implemented without having to massage the inputs. As with everything, if you have to modify the problem before you arrive at the solution, you stand a finite chance of creating an error that would not have otherwise been created had you taken a more clever approach. A different architecture for the *EXAMPLE 12* solution is shown in Figure 39. One possible drawback of using a *don’t care*s in your VHDL code is that some synthesizers and some simulators do not handle them well. I personally avoid them at all costs and seek a more “definitive” method of modeling the circuit’s I’m dealing with.

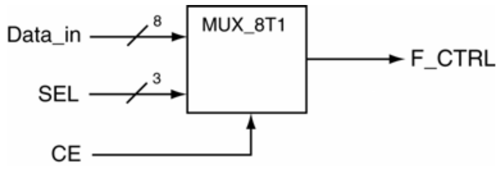
```
-- a solution that uses a don’t care
architecture my_soln_exam2 of my_example is
   signal ABC: std_logic_vector(2 downto 0);
begin
   ABC <= A & B & C; -- group signals for case statement

   my_proc: process (ABC)
   begin
      case (ABC) is
          when “100” =>  F_OUT <= ‘1’;
          when “-11” =>  F_OUT <= ‘1’;
          when others => F_OUT <= ‘0’;
      end case;
   end process my_proc;
end my_soln_exam2;
```

Figure 39: An alternate solution for *EXAMPLE 12* .

One of the main items that should be emphasized in any VHDL program is the readability. In the next problem, we’ll redo *EXAMPLE 11* but use a *case* statement instead of *if* statements.

EXAMPLE 13

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 10*. When CE is ‘0’, the output of the MUX is ‘0’.



Solution: This solution to *EXAMPLE 13* is shown in Figure 40. 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 increases as the circuits you're implementing 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_mux: 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_mux;
end my_case_ex;
```

Figure 40: Solution to *EXAMPLE 13*.

One massively important item in the solution to *EXAMPLE 13* is the fact that a *case* statement was embedded into an *if* statement. The technical term for this style of coding is as you would guess: *nesting*. Nesting sequential statements is typical in behavioral models and is used often. This is actually one of the features that make behavioral modeling so much more powerful than dataflow modeling. The reality is that conditional and selective signal assignment statements can not be nested. Bummer!

6.5 Caveats Regarding Sequential Statements

As you begin to work with sequential statements, you tend to start getting the feeling that you're doing algorithmic programming using a higher-level language. This is due to the fact that sequential statements have a similar look and feel to some of the programming constructs in higher-level languages. The bad part of this tendency is when and if your VHDL coding approach becomes similar to that of the higher-level language. Since I've seen this happen over and over with students who are learning VHDL, I feel it is appropriate to remind you once again that ***VHDL is not programming***: it's hardware design. You are, generally speaking, not implementing algorithms in VHDL, you are ***describing hardware***. It's a totally different paradigm. I've toiled over too many pieces of VHDL code that attempt to use a single process statement in order to implement a relatively complex circuit. Although the code appears like it should work in terms of the provided statements, this is an illusion based on the fact that your mind is interpreting the statements in terms of a higher-level language. The reality is that VHDL is somewhat mysterious in that you’re trusting that the VHDL synthesizer to magically know what you’re trying to describe. If you don’t understand the ins and outs of VHDL at a low level, you’re circuit is not going to synthesize properly. Personally, I feel I understand simple VHDL behavioral models. But once the models become complex, my understanding quickly fades.

The solution to this dilemma is really simple: keep your VHDL models simple, particularly your process statements. The best approach is to keep your process statements centered about a single function and have lots of process statements that communicate with each other. The bad approach is to have one massive process statement that does everything for you. The magic of VHDL is that if you provide simple code to the synthesizer, it’s more than likely going to provide you with a circuit that works and an implementation that is simple and eloquent. If you provide with synthesizer with complicated VHDL code, the circuit, may work and the circuit may be efficient in both time and space considerations, but probably not. As opposed to higher-level languages where small amounts of code often translate directly to code of relatively high efficiency, efficiency in VHDL code is obtained by compact and simple partitioning of the VHDL code based on the underlying hardware constructs. In other words, simple VHDL models are better but the simplicity is generally obtained by proper partitioning and description of the model. So try to fight off the urge to impress your friends with the world’s shortest VHDL model; your hardware friends will know better.

6.6 Important Points

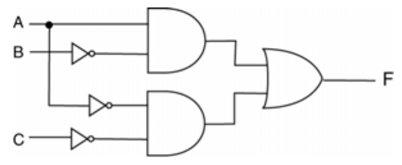
- The three main flavors of VHDL modeling styles include dataflow, behavioral, and structural models.
- VHDL behavioral models by definition use process statements.
- VHDL dataflow models by definition use concurrent signal assignment, conditional signal assignment, and/or selective signal assignment.
- The process statement is a concurrent statement. Statements appearing within the process statement are sequential statements.
- The ***if*** statement has a direct analogy to the conditional signal assignment statement used in dataflow modeling.
- The ***case*** statement has a direct analogy to the selective signal assignment statement used in dataflow modeling.
- Both the ***case*** statement and the ***if*** statement can be nested. Concurrent, conditional, and selective signal assignment statements can not be nested.

6.7 Exercises: Introduction to Behavioral Modeling

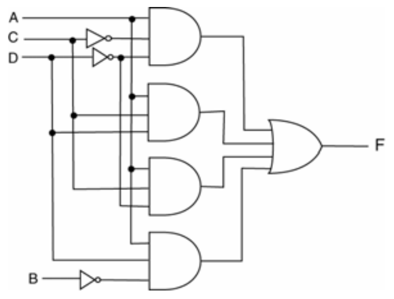
1. For the following function, write VHDL behavioral models that implement these functions using both a case statements and if statements (two separate models for each function).

   (a) $F(A,B,C) = \sum(0,3,6)$
   (b) $F(A,B,C) = m_1 + m_6$
   (c) $F(A,B,C,D) = \overline{A}C + \overline{B}C + BC$
   (d) $F(W,X,Y) = \prod(1,2,6,7)$
   (e) $F(A,B,C) = M_2 \cdot M_5$
   (f) $F(A,B,C,D) = (\overline{A} + \overline{B}) \cdot (B + C + \overline{D}) \cdot (\overline{A} + D) \cdot (C + D)$

2. For the following function, write VHDL behavioral models that implement these functions using both a case statements and if statements (two separate models for each function).

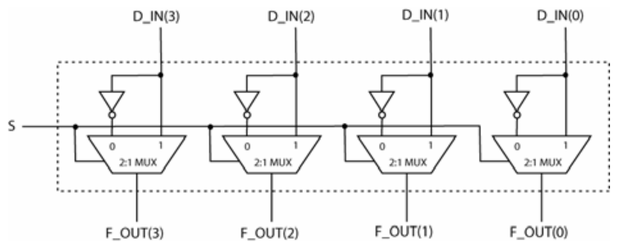


(a)

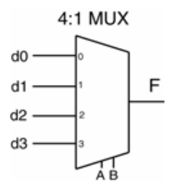


(b)

3. Model the following circuits using concurrent, conditional, or selective signal assignment.



(a)



(b)

4. Provide a VHDL model of an 8-input AND gate using a process statement.

5. Provide a VHDL model of an 8-input OR gate using a process statement.

6. Provide a VHDL model of an 8:1 MUX using a process statement. Include a models that use if statements and case statements (two separate models).

7. Provide a VHDL model of a 3:8 decoder using a process statement. Include a models that use if statements and case statements (two separate models). Consider the outputs to be active low.

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. Although you may not have an immediate need to use some of these operators, you should be aware that they exist. And although there are some special things you should know about some of these operators, not too much information is presented in this section.

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**			
others	******	**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 relational 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 other 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.

7.5 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. The concatenation operator has been seen in some previous example solutions. Some more examples of the concatenation operators are presented in Figure 41.

```
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(7 downto 0);

C_val <= A_val & “00”;
C_val <= “11” & B_val;
C_val <= ‘1’ & A_val & ‘0’;
D_val <= “0001” & C_val(3 downto 0);
D_val <= A_val & B_val;
```

Figure 41: Examples of the concatenation operator.

7.6 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 (X) 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 (of Almost Everything Up to Now)

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 other details of how the circuit behaves.

The VHDL architectures are comprised of statements that describe the behavior of the digital 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.

Coding styles in VHDL fall under the category of *dataflow*, *behavioral*, and *structural* models. Exclusive use of process statements indicates a behavioral model. The use of concurrent, conditional, and selective signal assignment indicate the use of a dataflow model. VHDL code describing more complex digital circuits will generally contain both features of all of these types of modeling.

The information provided in Table 9 is what I refer to as my VHDL cheat sheet. Since I make no effort whatsoever to memorize VHDL syntax, I keep this sheet next to me as I perform my VHDL modeling. Developing a true understanding of VHDL is what’s going to make you into a good hardware designer. The ability to memorize VHDL syntax proves nothing; so why bother memorizing stuff?

Concurrent Statements		**Sequential Statements**
Concurrent Signal Assignment **(dataflow model)**	⇔	**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 **(dataflow model)**	⇔	***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 **(dataflow model)**	⇔	***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 **(behavioral model)**		

opt_label: **process**(*sensitivity_list*)
begin
{*sequential_statements*}
end process *opt_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;		

Table 9: Part of the classic VHDL cheat sheet.

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 storage elements in VHDL, 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 (FSMs)*. This tutorial will examine FSMs in a later section.

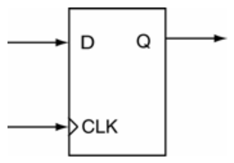
9.1 Simple Storage Elements Using VHDL

The general approach to learning about storage elements in digital design is to study the properties of a basic cross-coupled cell. The cross coupled cell forms what is referred to as a *latch*. The concept of a clocking signal is added and the device in order to enhance its controllability. And finally, some type of pulse narrowing circuitry is added to the clocking signal and we arrive at the flop-flop. The flip-flop is nothing more than an edge-sensitive bit-storage device.

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 14

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



Solution: The solution to *EXAMPLE 14* is shown in Figure 42. 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.

```
-------------------------------------------------------------
-- Model 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 42: Solution to *EXAMPLE 14*.

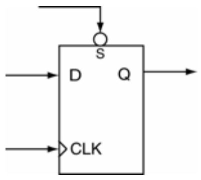
The D flip-flop is best known and loved for its ability to store (save, remember) a single bit. The way that the VHDL code listed in Figure 42 is able to store a bit is not obvious, however. The bit-storage capability in the VHDL is *implied* by the both the VHDL code and the way the VHDL code is interpreted. 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. If you have not specified what the outputs should be for every possible set of input conditions, the option taken by VHDL is not to change the current output. By definition, if the inputs change to an unspecified state, the outputs remain unchanged. In this case, the outputs associated with the previous set of inputs can be thought of as being remembered. It is this mechanism, as strange and interesting as it is, that is used to induce memory in VHDL.

In terms of the D flip-flop shown in *EXAMPLE 14*, the only time the output is specified is for that delta time associated with the rising edge of the clock. The typical method used to provide a catch-all condition in case the **`if`** condition is not met is with an **`else`** clause. Generally speaking, a quick way to tell if you’ve induced a memory element is to look for the presence of an **`else`** clauses associated with the **`if`** statement.

The previous two paragraphs are vastly important to understanding VHDL; the concept of inducing memory in VHDL is massively important to digital circuit design. By definition, the design of sequential circuits is dependent on this concept. This somewhat cryptic method used by VHDL to induce memory elements is a byproduct of behavioral modeling based solely on the interpretation of the VHDL source code. 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 15

Write the VHDL code that describes a D flip-flop shown on the right. 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 15* is shown in Figure 43. There are a few things of interest regarding this solution.

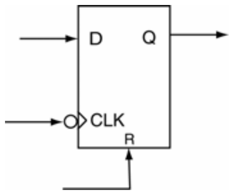
- The *S* input to the flip-flop is made synchronous by only allowing it to affect the operation of the flip-flop on the rising edge of the system clock.
- On the rising edge of the clock, the *S* input takes precedence over the *D* input because the state of the *S* input is checked prior to examining the state of the *D* input. In an **`if-else`** statement, once one condition evaluates as 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.

```
-----------------------------------------------------------------
-- RET D Flip-flop model 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 43: Solution to *EXAMPLE 15*.

EXAMPLE 16

Write the VHDL code that describes a D flip-flop shown on the right. 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 16* is shown in Figure 44. You can probably glean the most information about asynchronous and synchronous inputs by comparing the solutions to *EXAMPLE 15* and *EXAMPLE 16*. 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.

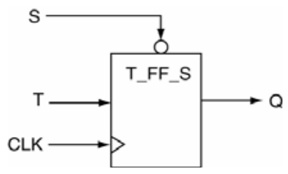
```
---------------------------------------------------------------------
-- FET D Flip-flop model with active-high asynchronous reset input.
---------------------------------------------------------------------
entity d_ff_r is
    port (   D,R :  in  std_logic;
             CLK :  in  std_logic;
               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 44: Solution to *EXAMPLE 16*.

The solutions of *EXAMPLE 15* and *EXAMPLE 16* 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.

EXAMPLE 17

Write the VHDL code that describes a T flip-flop shown on the right. Use a behavioral model in your description. Consider the S input to be an active-low, asynchronous input that sets the T flip-flop outputs when asserted.



Solution: The solution to *EXAMPLE 18* is shown in Figure 45. This example has some massively important techniques associated with it that are well worth mentioning below.

- A unique quality of the D flip-flop is demonstrated in this implementation of a T flip-flop. The output of a D flip-flop is only dependent upon the D input and is not a function of the present output of the flip-flop. The output of a T flip-flop is dependent upon both the T input and the current output of the flip-flop. This adds a certain amount of extra complexity to the T flip-flop model as compared to the D flip-flop as is shown in Figure 45. The T flip-flop model in Figure 45 uses a temporary signal in order to use the current state of the flip-flop as in input. In other words, since Q appears as a port to the entity it must be assigned a mode specifier, and in this case, it has been assigned a mode specifier of “out”. Signals that are declared as outputs can therefore not appear on the right side of a signal assignment operator. The standard approach to bypassing this apparent limitation in VHDL is to use intermediate signals which, as opposed to port signals, do not have mode specifications and can thus be used as either inputs or outputs (can appear on both sides of the signal assignment operator) in the body of the architecture. The approach is to manipulate the intermediate signal in the body of the architecture but to also use a concurrent signal assignment statement to assign the intermediate signal to the appropriate output. Note that in the key statement in the solution shown in Figure 45 that the intermediate signal appears on both sides of the signal assignment operator. This is a widely used approach in VHDL; please take time to understand and absorb it. And lastly on this note, there are other mode specifications that would allow you a different approach to bypassing this problem (namely, the use of the “buffer” mode specification), but you should never use these in VHDL. The approach presented here is considered a good use of VHDL.

- This code uses the characteristics equation of a T flip-flop in its implementation. We technically used a characteristic equation when we implemented the D flip-flop but since the characteristic equation of a D flip-flop is relatively trivial ($Q^+ = D$), you may not have been aware of it.

- Where there are certain advantages to using T flip-flops in come conditions, D flip-flops are generally the storage element of choice using VHDL. If you don’t have a specific reason for using some type of flip-flop other than a D flip-flop, you probably shouldn’t.

```
-----------------------------------------------------------------
-- RET T Flip-flop model with active-low asynchronous set input.
-----------------------------------------------------------------
entity t_ff_s is
    port ( T,S,CLK :  in  std_logic;
                 Q :  out std_logic);
end t_ff_s;

architecture my_t_ff_s of t_ff_s is
   signal t_tmp : std_logic; -- intermediate signal declaration
begin
   tff:  process (T,S,CLK)
   begin
      if (S = ‘0’) then
         Q <= ‘1’;
      elsif (rising_edge(CLK)) then
         t_tmp <= T XOR t_tmp; -- temp output assignment
      end if;
   end process tff;

   Q <= t_tmp; -- final output assignment

end my_t_ff_s;
```

Figure 45: Solution to *EXAMPLE 19*.

9.2 Inducing Memory: Dataflow vs. Behavior Modeling

A major portion of digital design deals with sequential circuits. Generally speaking, most sequential circuit design is synchronized to a clock edge. In other words, output changes in sequential circuits generally only occur on an active clock edge. The introduction to memory elements in VHDL presented in this section may lead the reader to think that memory in VHDL is only associated with behavioral modeling, but this is not the case. The same concept of inducing memory holds for dataflow modeling as well: not explicitly specifying an output for every possible input condition generates memory. And on this note, checking for unintended memory element generation is one of the duties of the digital designer. As you would imagine, memory elements add an element of needless complexity to the synthesized circuit.

One common approach to learning the syntax and mechanics of new computer languages is to implement the same task in as many different ways as possible. This approach utilizes the flexibility of the language and is arguably a valid approach to learning a new language. This is also the case in VHDL. But, probably more so in VHDL than other languages, there are specific ways of doing things and these things should always be done in these specific ways. Although it would be possible to generate flip-flops using dataflow models, most knowledgeable people examining your VHDL code would not initially be clear as to what exactly you're doing. As far as generating synchronous memory elements go, the methods outlined in this section are simply the optimal method of choice. This is one area not be clever with.

9.3 Important Points

- Storage elements in VHDL are induced by not specifying output conditions for every possible input condition.

- Unintended generation of storage elements is generally listed by the synthesizer as "latch generation". Once again, latches are generated when there is an existing input condition to a circuit that does not have a corresponding output specification.

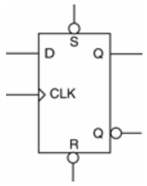
- Memory elements can be induced by both dataflow and behavioral models.

- If a signal declared in the entity declaration has a mode specifier of out, that signal cannot appear on the right side of a signal assignment operator. This limitation is bypassed by using intermediate signals for any functional assignments and later assigning the intermediate signal to the output signal using a concurrent signal assignment statement.

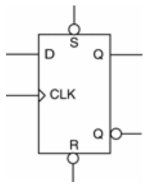
- The mode specification of *buffer* should never be used in VHDL unless you're an old fart who does not know any better are not interested in learning the best approach to using VHDL.

9.4 Exercises: Basic Memory Elements

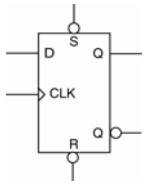
1. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



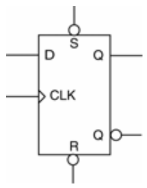
2. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume the S input takes precedence over the R input in the case where both are asserted simultaneously.



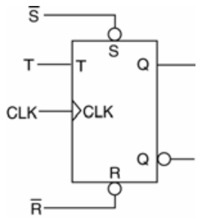
3. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are synchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



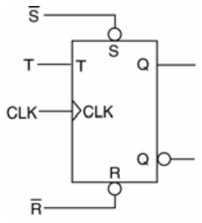
4. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. If both the S and R inputs are asserted simultaneously, the output of the flip-flop will toggle.



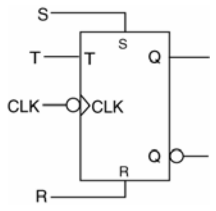
5. Provide a VHDL behavioral model of the T flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously. Implement this flip-flop first using an equation description of the outputs and then using a behavioral description of the outputs.



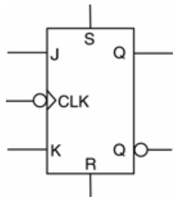
6. Provide a VHDL behavioral model of the T flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



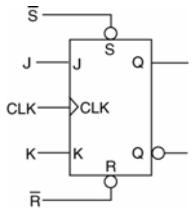
7. Provide a VHDL behavioral model of the T flip-flop shown at the right. The S and R inputs are an active high asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



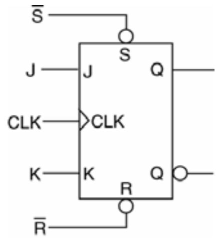
8. Provide a VHDL behavioral model of the JK flip-flop shown on the right. The S and R inputs are an asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously. Implement this flip-flop first using an equation description of the outputs and then using a behavioral description of the outputs.



9. Provide a VHDL behavioral model of the JK flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



10. Provide a VHDL behavioral model of the JK flip-flop shown on the right. The S and R inputs are active low synchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



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 section dealing with memory elements in VHDL. 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 46. 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 the Next State Decoder block are decoded and to produce an output that represents the next state of the FSM. The circuitry in Next State Decoder is generally the excitation equations for the storage elements (flip-flops) in the State Register block. 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* 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.

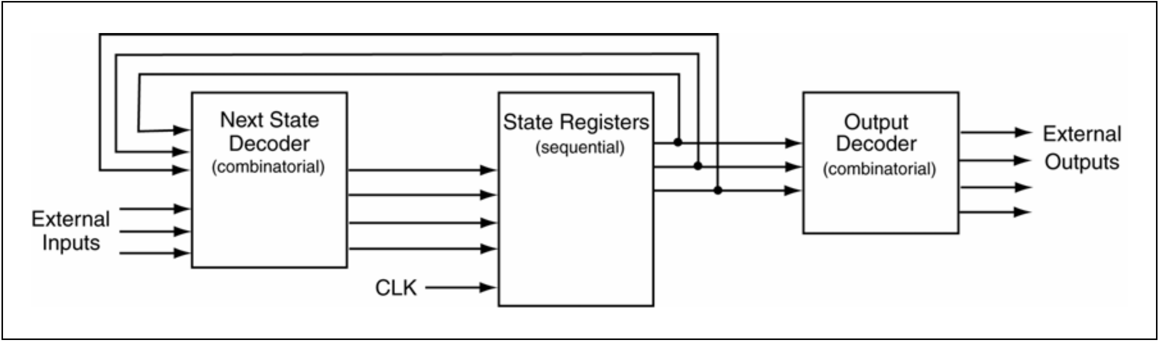


Figure 46: Block diagram for a Moore-type FSM.

The FSM model shown in Figure 46 is probably the model of a Moore-type FSM that you have previously been introduced to. 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 modeling 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 47.

Although it does not look that much clearer, you'll soon be finding the FSM model shown in Figure 47 to be a straight-forward method to implement FSMs. The approach we will use divides the FSM into two VHDL processes. One process, referred to as 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 46. Note that the two blocks in Figure 46 are both comprised of solely of combinatorial logic.

There is some new lingo used in the description of signals used in Figure 47; this description is outlined and described below:

- 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.

- The *Present State* signals are used by the Combinatorial Process box for both next state decoding and output decoding. The diagram of Figure 47 also shows that the Present State variables can also be provided as outputs to the FSM but is not required.

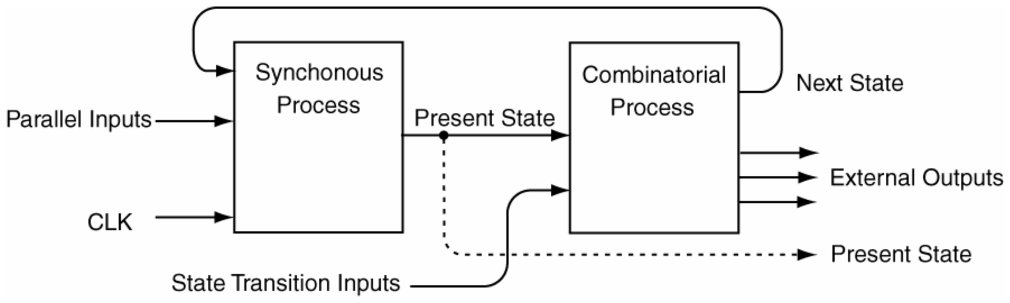


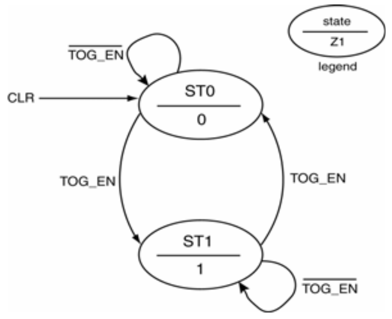
Figure 47: Model for VHDL implementations of FSMs.

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. I've opted to only cover the dependent style in this tutorial because it is clearer than the independent PS/NS style. The model shown in Figure 47 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 or any other style is relatively painless. More information on the other FSM coding styles is found in various VHDL texts or on the web.

10.1 VHDL Behavioral Representation of FSMs

EXAMPLE 20

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



Solution: This problem represents a basic FSM implementation. It is somewhat instructive to show the black box diagram which as an aid in the writing the entity description. Starting FSM problems with the drawing of a black box diagram is always a healthy approach particularly when dealing with FSMs. Often times with FSM problems, it sometimes becomes challenging to discern the FSM inputs from the outputs. Drawing a diagram partially alleviates this problem. The black box diagram is shown in Figure 48 and the solution to *EXAMPLE 20* is shown in Figure 49.

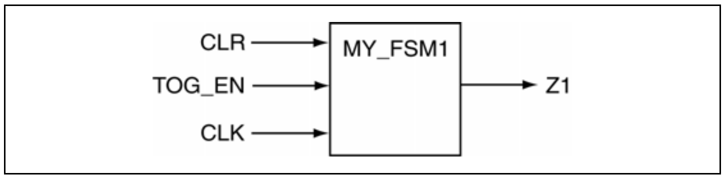


Figure 48: Black box diagram for the FSM of *EXAMPLE 20*.

```
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
     -- take care of the asynchronous input
     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
      Z1 <= ‘0’;        -- pre-assign output
      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 49: Solution of *EXAMPLE 20*.

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,* state_type, to represent the states in this FSM. This is an example of how enumeration types are used in VHDL. As with enumeration types in other higher-level computer languages, there are internal numerical representations for the listed state types but we only deal with the more expressive textual equivalent. In this case, the type we've created is called a state_type and we've declared two variables of this type: PS and NS. The key thing to note here is that a state_type is something we've created and is not a native VHDL type.

- 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. The key thing to note here is that the storage element is associated with the PS signal only. Note that PS is not specified for every possible combination of inputs.

- 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. And so on and so forth.

- 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 and represents good VHDL coding practice.

- 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. In other words, the Z1 variable is inside the *when* clause but outside of the *if* statement in the when clause. This is course because the Moore outputs are only a function of the states and not the external inputs. Note that it is the external input that controls the which state the FSM transitions to from any given state. You'll see later that Mealy outputs, due their general nature, are assigned inside the *if* statement.

- The Z1 output is pre-assigned as the first step in the combinatorial process. Pre-assigning it in this fashion prevents the unexpected latch generation for the Z1 signal. When dealing with FSMs, there is a natural tendency for the FSM designer to forget to specify an output for the Z1 variable in each of the states. Pre-assigning it prevents latches from being generated and can arguable clean up the source code. The pre-assignment make no difference to the VHDL code because if multiple assignments are made within the code, only the final assignment takes affect. In other words, only the final assignment is considered once the process terminates.

There is one final thing to note about *EXAMPLE 20*. 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 48 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 variable is not provided as an output.

In some FSM designs, the state variables are provided as outputs. To show this situation, we'll provide a solution to *EXAMPLE 20* with the state variables as outputs. The black box diagram of this solution is shown in Figure 50 and the alternate solution is shown in Figure 51.

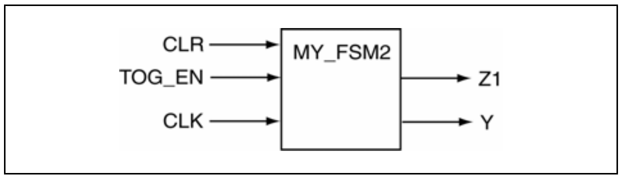


Figure 50: Black box diagram of *EXAMPLE 20* including the state variable as an output.

```
entity my_fsm2 is
    port (   TOG_EN : in  std_logic;
            CLK,CLR : in  std_logic;
               Y,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
      Z1 <= '0';

      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;

   -- assign values representing the state variables
   with PS select
      Y <= '0' when ST0,
           '1' when ST1,
           '0' when others;

end fsm2;
```

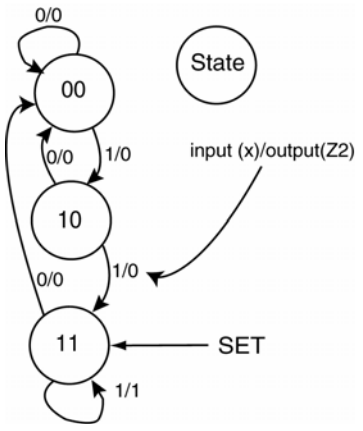
Figure 51: Solution for *EXAMPLE 20* including state variable as an output.

Note that the VHDL code in shown in Figure 51 differs in only two areas from the code shown in Figure 49. 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. Once again, since we have declared an enumeration type for the state variables, we have no way of knowing exactly how the synthesizer has opted to represent the state variable. The selective signal assignment statement in the code of Figure 51 only makes it appear like there is one state variable and the states are represented with a '1' and a '0'. In reality, there are methods we can use to control how the state variables are represented and we'll deal with those soon.

And lastly, there are three concurrent statements in the VHDL code shown in Figure 49: two process statements and a selective signal assignment statement.

EXAMPLE 21

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



Solution: The state diagram shown in the problem description indicates that this is a three-state FSM with one Mealy-type external output and one external input. 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 52. Note that the two state variables are handled as a bundled signal.

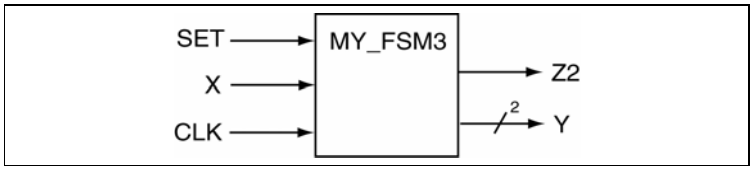


Figure 52: Black box diagram for the FSM of *EXAMPLE 21*.

```
entity my_fsm3 is
    port ( X,CLK,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
         Z2 <= '0';     -- pre-assign FSM outputs
         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;

   -- faking some state variable outputs
   with PS select
      Y <= "00" when ST0,
           "10" when ST1,
           "11" when ST2,
           "00" when others;
end fsm3;
```

Figure 53: Solution for *EXAMPLE 21*.

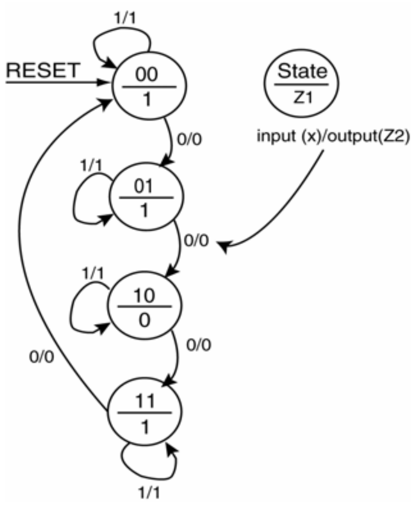
As usual, there are a couple of fun things to note about the solution for *EXAMPLE 21*. Most importantly, you should note the similarities between this solution and the previous solution.

- 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. The fact that the Z2 output is different in the context of state ST2 that makes it a Mealy-type output and therefore a Mealy-type FSM.

- When faking the state variable outputs (keeping in mind that the actual state variables are represented with enumeration types), two signals are required since the state diagram contains more than two states (and less than five states). The solution opted to represent these outputs as a bundle which has the effect of slightly changing the form of the selected signal assignment statement appearing at the end of the architecture description.

EXAMPLE 22

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



Solution: The state diagram indicates that the solution will contain four states, one external input, and two external outputs. This is a hybrid FSM in that the *if* contains both a Mealy and Moore-type output but in this case, the FSM would be considered a Mealy-type FSM. The black box diagram for the solution is shown in Figure 54 and the actual solution is shown in Figure 55.

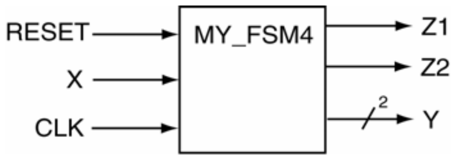


Figure 54: Black Box diagram for the FSM of *EXAMPLE 22* .

```
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
      Z1 <= ‘0’;   Z2 <= ‘0’; -- pre-assign the outputs
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= ‘1’;  -- Moore output
            if (X = ‘0’) then NS <= ST1; Z2 <= ‘0’;
            else  NS <= ST0; 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 55: Solution for *EXAMPLE 22*.

So if you’ve haven’t noticed by now, implementing FSMs using VHDL behavioral model is remarkable straight-forward. It’s actually a cookbook approach it’s so straight-forward. In reality, I rarely code a FSM from scratch; I usually opt to grab some previous FSM I’ve coded and start from there. Keep in mind that real engineering is rarely cookbook. For FSM problems, the engineering is in the testing and creation of the state diagram. So don’t get too comfortable with behavioral modeling of FSMs; the real fun is generating a FSM that solves a given problem.

10.2 One-Hot Encoding for FSMs

In reality, there are many different methods that can be used to encode the state variables[7]. If the exact form of the representation used is important to you, they you will need to take the necessary steps in order to control how the state variables are represented by the synthesizer. There are to approaches to control state variable representation. The first approach is to allow the synthesizing tool to handle the details. Since every FSM we've seen up to this point has used enumeration types to represent the state variables, the synthesizer could choose to actually represent them by an encoding scheme of its own choosing. The reality is that the tools generally have an option to select the desired encoding scheme. The downside of this approach is that you're denied the learning experience associated with implementing the VHDL code that explicitly induces your desired encoding scheme. After all, you may have some special encoding scheme you need to use but is not supported by the development tools. The second approach to encoding the state variables is to specify them directly in VHDL. The approach to specifying the state variables in the VHDL code is presented in this section.

The approach taken in the previous FSM examples was to pretend like we were using *full encoding* for the state variables of the FSM. The full encoding approach minimizes the number of storage elements (flip-flops) used to store the state variables. The closed form equation describing the number of flip-flops required for a given FSM as a function of the number of states is shown in Equation 1. The bracket-like symbols used in Equation 1 indicates a *ceiling* function[8]. Note the nice comfortable binary relationship expressed by this equation so much so that full encoding is often referred to as binary encoding.

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

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 one-hot encoding, the number of flip-flops required to implement a FSM is therefore equal to the number of states in the FSM. The closed form of this relationship is shown in Equation 2.

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

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. If you want total control of the process, you'll need to grab control away from the synthesizer. And since we're concerned with learning VHDL, we need to look at the process of explicitly encoding one-hot FSMs.

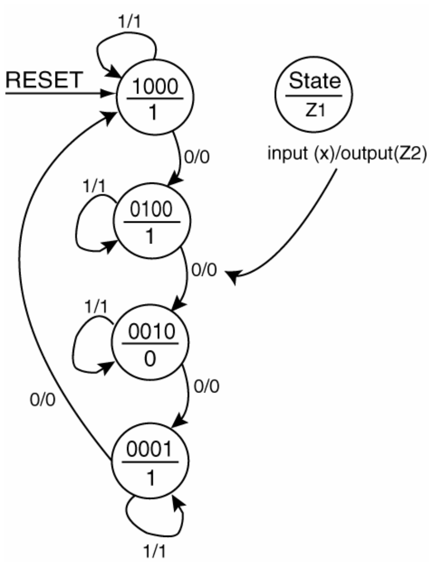
The modular approach we used to implement FSMs expedites the conversion process from using enumeration types to actually specifying how the state variables are represented. These changes required from our previous approach are limited to how the outputs are assigned to the state variables and how the state variables are forced to be represented by certain bit patterns. Modifications to the full encoded approach as thus limited to the entity declaration (you'll need more variables to represent the states), the declaration of the state variables (you'll need to explicitly declare the bit patterns associated with each state), and the assignment of the state variables to the outputs (in this case, we're actually not faking it like we were in other examples).

[7] In this case, encoding refers to the act of assigning a unique pattern of 1's and 0's to the each of the state in order to disambiguate them from other states.

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

EXAMPLE 23

Write the VHDL code that implements the FSM shown on the right. Use a dependent PS/NS coding style in your implementation. Consider the listed state variables as output. Use one-hot encoding for the state variables. This problem is *EXAMPLE 22* all over again but uses true one-hot encoding for the state variables.



Solution: The state diagram shows four states, one external input X, two external outputs Z1 and Z2 with the Z2 output being a Mealy output. This is a Mealy machine that indicates one-hot encoding should be used to encode the state variables. We'll approach this solution in pieces, bits and pieces.

Figure 56 shows the modifications to the entity declaration required to convert the full encoding used in *EXAMPLE 22* to a pseudo one-hot encoding. Figure 57 shows the required modifications to the state variable output assignment in order to move from enumeration types to a special form of assigned types. Forcing the state variables to be truly encoded using one-hot encoding requires these two extra lines of code as is shown in Figure 57. 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. Note in Figure 58 that default case is assigned a valid one-hot state instead of the customary all zero state. You should strongly consider comparing and contrasting these three figures. The total solution is shown in

```
-- 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 56: Modifications to convert entity of *EXAMPLE 22* to one-hot encoding.

```
-- the approach to for enumeration types
type state_type is (ST0,ST1,ST2,ST3);
signal PS,NS : state_type;
```

```
-- the approach used for explicitly specifying state bit patters
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 57: Modifications to convert state variables to use one-hot encoding.

```
-- fake 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 58: Modifications to convert state output of *EXAMPLE 22* to pseudo one-hot encoding.

```
entity my_fsm4_oh is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(3 downto 0);
                 Z1,Z2 : out std_logic);
end my_fsm4_oh;

architecture fsm4_oh of my_fsm4_oh is
   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;
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
      Z1 <= ‘0’;   Z2 <= ‘0’; -- pre-assign the outputs
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= ‘1’;  -- Moore output
            if (X = ‘0’) then NS <= ST1; Z2 <= ‘0’;
            else  NS <= ST0; 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;

   -- 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_oh;
```

Figure 59: The final solution to *EXAMPLE 24.*

10.3 Important Points

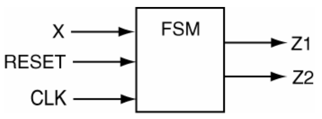
- Modeling FSMs from a state diagram is a straight-forward process using VHDL behavioral modeling. The process is so straight-forward that it is often considered cookie cutter. The real engineering involved in implementing FSM is in the generation of the state diagram that solved the problem at hand.

- Due to the general versatility of VHDL, there are many approaches that can be used to model FSMs using VHDL. The approach used in this tutorial details only one of those styles but is generally considered the most straight-forward of the styles.

- The actual encoding of the FSM's state variables when enumeration types are used is left up to the synthesis tool. If a preferred method of variable encoding is desired, using the attribute approach detail in this section is a simple but viable alternative.

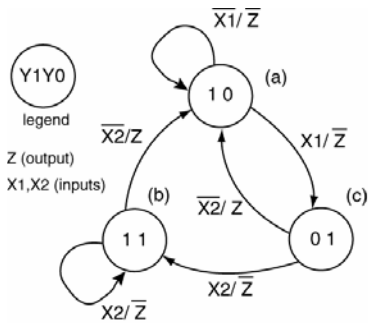
10.4 Exercises: Behavioral Modeling of Finite State Machines

1. Draw the state diagram associated with the following VHDL code. Be sure to provide a legend and completely label everything.

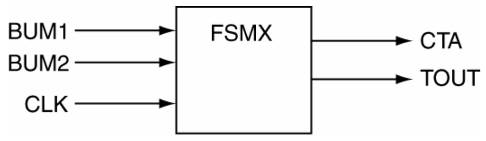


```
entity fsm is
   port (  X,CLK : in  std_logic;
           RESET : in  std_logic;
           Z1,Z2 : out std_logic;
end fsm;

architecture fsm of fsm is
   type state_type is (A,B,C);
   signal PS,NS : state_type;
begin

   sync_proc: process(CLK,NS,RESET)
   begin
      if (RESET = '0') then PS <= C;
      elsif (rising_edge(CLK)) then PS <= NS;
      end if;
   end process sync_proc;

   comb_proc: process(PS,X)
   begin
      case PS is
         Z1 <= '0';    Z2 <= '0';

         when A =>
            Z1 <= '0';
            if (X = '0') then NS <= A; Z2 <= '1';
            else  NS <= B; Z2 <= '0';
            end if;

         when B =>
            Z1 <= '1';
            if (X = '0') then NS <= A; Z2 <= '0';
            else  NS <= C; Z2 <= '1';
            end if;

         when C =>
            Z1 <= '1';
            if (X = '0') then NS <= B; Z2 <= '1';
            else  NS <= A; Z2 <= '0';
            end if;

         when others =>
            Z1 <= '1'; NS <= A; Z2 <= '0';

      end case;
   end process comb_proc;

end fsm;
```

2. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.

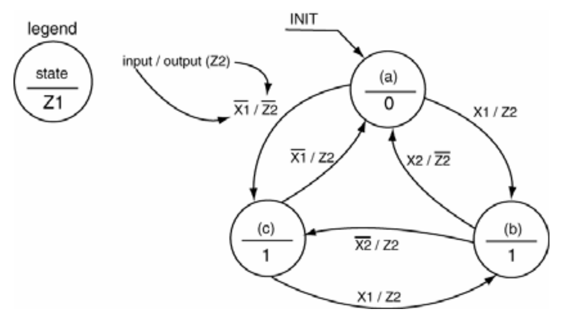


3. Draw the state diagram associated with the following VHDL code. Be sure to provide a legend and completely label everything.



```vhdl
entity fsmx is
    Port ( BUM1,BUM2 : in std_logic;
             CLK : in std_logic;
            TOUT,CTA : out std_logic);
end fsmx;


architecture my_fsmx of fsmx is
   type state_type is (S1,S2,S3);
    signal PS,NS : state_type;
begin
   sync_p: process (CLK,NS)
   begin
      if (rising_edge(CLK)) then
         PS <= NS;
      end if;
   end process sync_p;

   comb_p: process (CLK,BUM1,BUM2)
   begin
      case PS is

         when S1 =>
            CTA <= '0';
            if (BUM1 = '0')  then
               TOUT <= '0';
               NS <= S1;
            elsif (BUM1 = '1') then
               TOUT <= '1';
               NS <= S2;
            end if;

         when S2 =>
            CTA <= '0';
            TOUT <= '0';
            NS <= S3;

         when S3 =>
            CTA <= '1';
            TOUT <= '0';
            if (BUM2 = '1')  then
               NS <= S1;
            elsif (BUM2 = '0') then
               NS <= S2;
            end if;

         when others =>  CTA <= '0'; TOUT <= '0'; NS <= S1;
      end case;
   end process comb_p;

end my_fsmx;
```

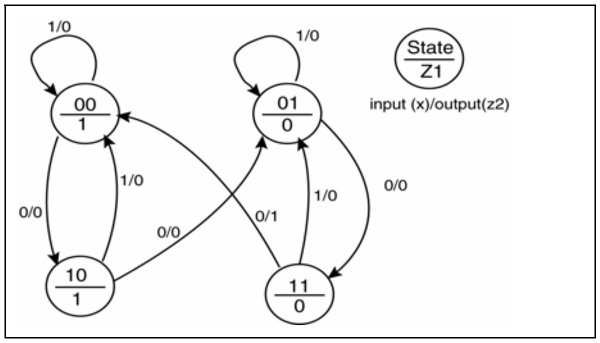
4. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right.



Draw the state diagram associated with the following VHDL code. Consider the outputs Y to be representative of the state variables. Be sure to provide a legend. Indicate the states with both the state variables and their symbolic equivalents.

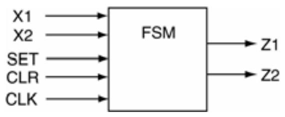
```
entity fsm is
port (     X,CLK : in  std_logic;
           RESET : in  std_logic;
           Z1,Z2 : out std_logic;
               Y : out std_logic_vector(2 downto 0));
end fsm;

architecture my_fsm of fsm is
   type state_type is (A,B,C);
   attribute ENUM_ENCODING: STRING;
   attribute ENUM_ENCODING of state_type: type is "001 010 100";
   signal PS,NS : state_type;
begin

sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '0') then PS <= C;
     elsif (rising_edge(CLK)) then PS <= NS;
     end if;
   end process sync_proc;

comb_proc: process(PS,X)
   begin
      case PS is
         when A =>
            Z1 <= '0';
            if (X = '0') then NS <= A; Z2 <= '1';
            else  NS <= B; Z2 <= '0';
            end if;
         when B =>
            Z1 <= '1';
            if (X = '0') then NS <= A; Z2 <= '0';
            else  NS <= C; Z2 <= '1';
            end if;
         when C =>
            Z1 <= '1';
            if (X = '0') then NS <= B; Z2 <= '1';
            else  NS <= A; Z2 <= '0';
            end if;
         when others =>
            Z1 <= '1'; NS <= A; Z2 <= '0';
      end case;
   end process comb_proc;

with PS select
   Y <= "001" when A,
        "010" when B,
        "100" when C,
        "001" when others;
end my_fsm;
```

5. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



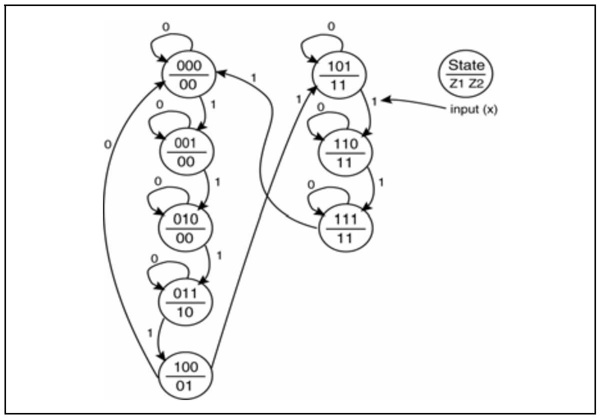
6. Draw the state diagram that corresponds to the following VHDL model and s*tate whether the FSM is a **Mealy** or **Moore** machine*. Be sure to label everything.

```
entity fsm is
    Port ( CLK,CLR,SET,X1,X2 : in std_logic;
                       Z1,Z2 : out std_logic);
end fsm;
```

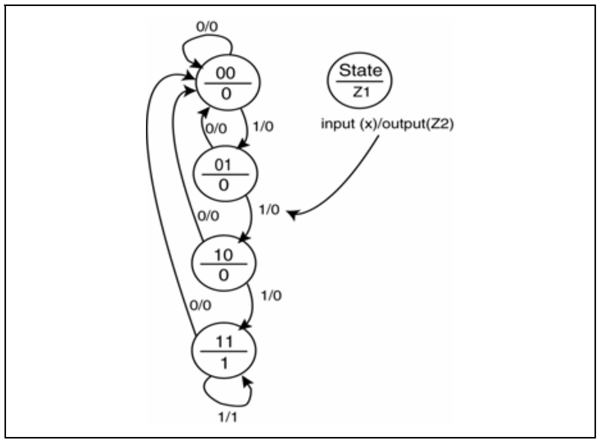


```
architecture my_fsm of fsm is
   type state_type is (sA,sB,sC,sD);
   attribute ENUM_ENCODING: STRING;
   attribute ENUM_ENCODING of state_type: type is "1000 0100 0010 0001";
   signal PS,NS : state_type;
begin
   sync_p: process (CLK,NS,CLR,SET)
   begin
      if (CLR = '1' and SET = '0') then
         PS <= sA;
      elsif (CLR = '0' and SET = '1') then
         PS <= sD;
      elsif (rising_edge(CLK)) then
         PS <= NS;
      end if;
   end process sync_p;

   comb_p: process (X1,X2,PS)
   begin
      case PS is
         when sA =>
            if ( X1 = '1')  then
               Z1 <= '0'; Z2 <= '0';
               NS <= sA;
            else
               Z1 <= '0'; Z2 <= '0';
               NS <= sB;
            end if;

         when sB =>
            if ( X2 = '1')  then
               Z1 <= '1'; Z2 <= '1';
               NS <= sC;
            else
               Z1 <= '1'; Z2 <= '0';
               NS <= sB;
            end if;

         when sC =>
            if ( X2 = '1')  then
               Z1 <= '0'; Z2 <= '0';
               NS <= sB;
            else
               Z1 <= '0'; Z2 <= '1';
               NS <= sC;
            end if;

         when sD =>
            if ( X1 = '1')  then
               Z1 <= '1'; Z2 <= '1';
               NS <= sD;
            else
               Z1 <= '1'; Z2 <= '1';
               NS <= sC;
            end if;
      end case;

   end process comb_p;
end my_fsm;
```

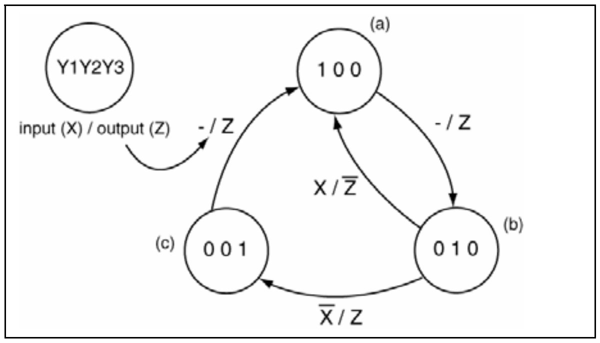
7. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



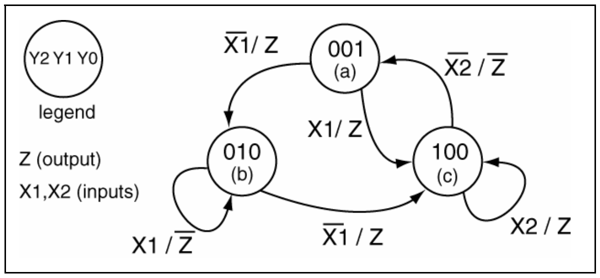
8. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



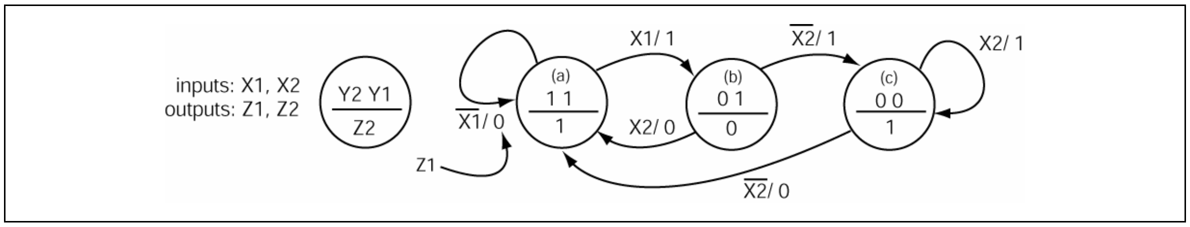
9. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



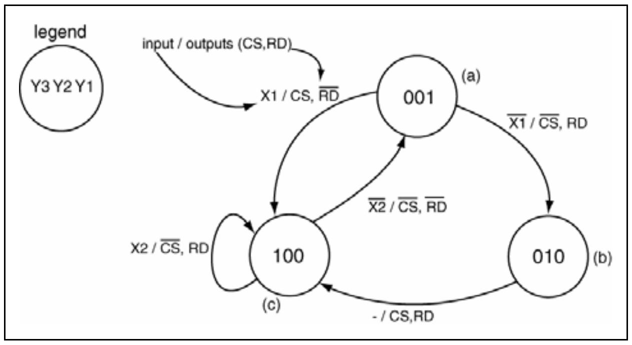
10. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



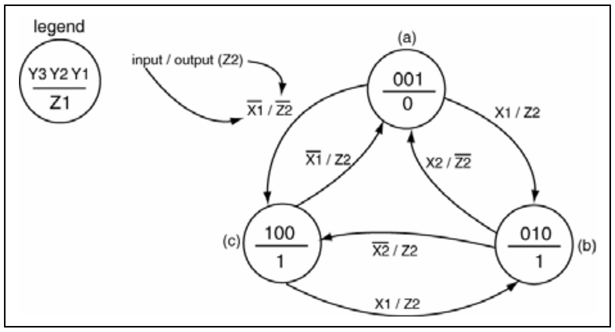
11. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



12. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



13. Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



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. Moreover, there are often libraries out there that contain useful modules that can only be accessed using a structural modeling approach. Having access to these libraries and being fluent in their use will serve to increase your perception of a VHDL guru.

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 in this section 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. Keep in mind that your first exposure to structural modeling may be somewhat rough. Although there is some new syntax to become familiar with, once you complete a few structural designs, this new syntax becomes ingrained in your brain and it becomes second nature to apply where required. The tendency at this juncture in your VHDL programming career is to use some type of schematic capture software instead of learning the structural modeling approach. The fact is that no one of consequence uses the schematic capture software except for tired old university professors who are more interested in selling books than they are teaching modern approach to VHDL modeling. The funny part about this entire process is that the schematic capture software is a tool that allows you to visually represent circuits but in the end generates VHDL code (the only thing the synthesizer understands is VHDL code).

11.1 VHDL and 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 10.

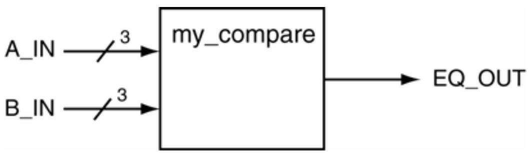
"C" programming language	VHDL
Describe function interface	the entity
Describe what the function does (coding)	the architecture
Provide a function prototype to main program	component declaration
Call the function from main program	component instantiation (mapping)

Table 10: 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 25

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 60. 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 60 contains some extra information that relates to the VHDL structural implementation. First, the dashed line represents the boundary of the top-level VHDL entity i.e., signals that cross this boundary must appear in the entity declaration for this implementation. Second, each of the internal signals is given a name. In this case, internal signals are defined to be signals that do not cross the dashed entity boundary. 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).

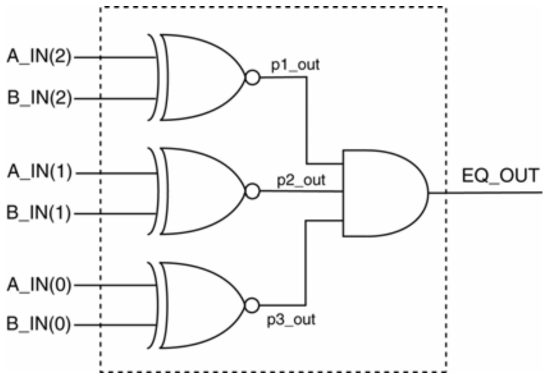


Figure 60: Discrete gate implementation of a 3-bit comparator.

The first part of the solution is to provide entity and architecture implementations for the individual gates shown in Figure 60. 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 61.

```
------------------------------------------------------------
-- 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 61: Entity and Architecture definitions for discrete gates.

The implementations shown in Figure 61 present no new VHDL details. The new information is contained in how the circuit elements listed in Figure 61 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 top-level entity declaration
2. Declare the lower-level design units used in design
3. Declare required internal signals used to connect the design units
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 60 and is shown in Figure 62. In other words, signals that intersect the dashed lines are signals that are known to the outside world and must be included in the entity declaration.

```
-----------------------------------------------------
-- 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 62: Entity declaration for 3-bit comparator.

<u>Step Two:</u> 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 for use 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. The design units are essentially modules that are used in the lower levels of the design. 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 11. Figure 63 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 11: A comparision of entity and component declarations.

<u>Step Three:</u> 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 60. 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. We've previously dealt with intermediate signals in other sections of this tutorial. The signal declarations are included as part of the final solution shown in Figure 63.

<u>Step Four</u>: 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 component 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 63.

One key thing to note in the instantiation process is the inclusion of labels for all the instantiated design units. Labels should always be used as part of design unit instantiation because the use of appropriate labels increases the understandability of your VHDL model. In other words, the proper choice of labels increases the self-commenting nature of your design and is considered good a VHDL programming approach.

```
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 63: 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 63 is not the only approach to use for the mapping process. The approach shown in Figure 63 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 only possible downside of this approach is that it uses up a lot of space in your VHDL source code.

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 design unit 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 64.

```
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 64: Alternative architecture for *EXAMPLE 25* 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 65 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 65: An example of the same signal name crossing hierarchical boundaries.

11.2 Important Points

- Structural modeling in VHDL supports hierarchical design concepts. The ability to abstract digital circuits to higher levels is the key to understand and designing complex digital circuits.

- Digital design using schematic capture is an outdated approach: you should resist the inclination and/or directive at all costs.

- VHDL structural model supports the reuse of design units. This includes units you have previously designed as well as the ability to use pre-defined module libraries.

11.3 Exercises: Structural Modeling

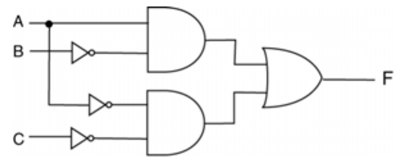
1. Draw a block diagram of the circuit represented by the VHDL code listed below. Be sure to completely label the final diagram.

```
entity quiz1_ckt is
    Port ( EN1, EN2 : in std_logic;
           CLK : in std_logic;
           Z : out std_logic);
end quiz1_ckt;

architecture quiz1_ckt of quiz1_ckt is

   component T_FF
      port ( T,CLK : in std_logic;
                 Q : out std_logic);
   end component;

   signal t_in, t1_s, t2_s : std_logic;
begin
   t1 : T_FF
   port map (   T => t_in,
              CLK => CLK,
                Q => t1_s );

   t2 : T_FF
   port map (   T => t1_s,
              CLK => CLK,
                Q => t2_s );

   Z <= t2_s OR t1_s;
   t_in <= EN1 AND EN2;

end quiz1_ckt;
```

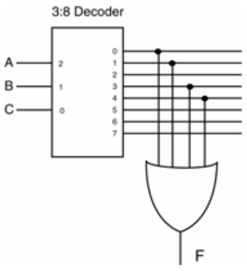
2. Draw a block diagram of the circuit represented by the VHDL code listed below. Be sure to completely label the final diagram.

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

architecture my_ckt of ckt is

   component bb1
      port (   D,E : in std_logic;
             F,G,H : out std_logic);
   end component;

   component bb2
      port ( L,M,N : in std_logic;
                 P : out std_logic);
   end component;

   signal x1,x2,x3 : std_logic;
begin

   b1: bb1
   port map ( D => A, E => B, F => x1, G => x2, H => x3);

   b2: bb2
   port map ( L => x1, M => x2, N => x3, P => C);

end my_ckt;
```

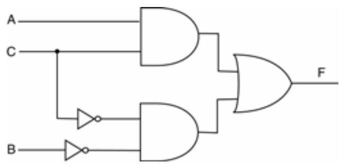
3. Provide VHDL structural models for the circuits listed below.



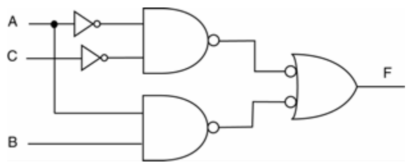
(a)



(b)



(c)



(d)

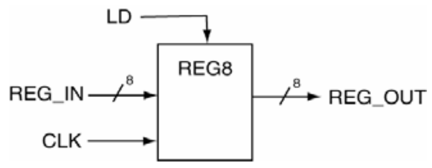
12. Registers and Register Transfer Level

The concept of a register in VHDL and its subsequent use in digital circuit design problems is probably one of the more straight-forward concepts in VHDL. A register is VHDL is simply a vector version of a D flip-flop in which all operations on the flip-flops occur simultaneously. The "register transfer level", or RTL, is a flavor of design that is primarily concern with how and when data is transferred between the various registers in a digital system. RTL-level design in often associated with "datapath" designs which requires the careful control and timing of the data that is being transferred between registers. The controls associated with even simple registers are sufficient to ensure that some outside entity has adequate control over the "sequencing" of data through the circuit associated with the datapath. In these cases, the proper sequencing of data transfers is controlled by a FSM.

The study of RTL-level design is best accomplished in the context of a datapath design. The design of datapaths is best accomplished in the context of a digital circuit that has some purpose such as an arithmetic logic unit design. Both of these topics are beyond what needs to be mentioned here. The good news is that the simplicity of the registers makes for a quick introduction to the matter; major circuit implementations are saved for some other time.

EXAMPLE 26

Use VHDL behavioral modeling to design the 8-bit register that has a synchronous active high parallel load signal LD. Consider the load of the register to be synchronized to rising edges of the clock.



Solution: The solution for the 8-bit register looks amazingly similar to a model of a D flip-flop. The full solution to *EXAMPLE 27* is shown in Figure 66. As usual, there are a couple of things worth noting in the solution shown in Figure 66.

- The body of the process statement describing the register looks massively similar to a D flip-flop. Note that there is an *if* statement that does not contain a corresponding else which is what generates the memory element. For this example, there are considered to be eight bit-sized memory elements (flip-flops). For this example, by the way the VHDL code is written, the flip flops are considered to be D-type flip-flops. The storage elements are associated with the REG_OUT bundle. The ease in using VHDL code to generate D flip-flops in this manner makes D flip-flops the most widely used type of flip-flop in digital design.

- The code uses a bundle signal for both the input and output. The assignment of the bundles to other bundles is straight-forward in VHDL as is shown in the code. In many cases, such as the one in this example, there is no need to use a bundle access operator in the VHDL model.

- The assignment of the input to the output is based on characteristics of both the clock edge and the state of the LD signal. The approach to taken in the VHDL model shown in Figure 66 is to provided a separate *if* clause for both the LD and CLK signals. Only one if statement could have been used by making the both conditions associated with the single if clause but this is not considered good VHDL programming practice when dealing with synchronized elements. In other words, you should always strive to keep special conditions associated with the clocking signal separate from all other conditions associated with the action in question. Clock signals are somewhat special in VHDL land; you should get into the habit of treating them gently.

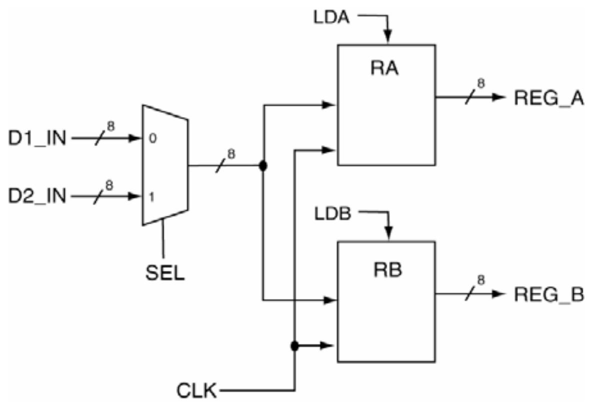
```
entity reg8 is
    Port (  REG_IN : in std_logic_vector(7 downto 0);
            LD,CLK : in std_logic;
           REG_OUT : out std_logic_vector(7 downto 0));
end reg8;

architecture reg8 of reg8 is
begin
   reg: process(LD,CLK,REG_IN)
   begin
      if (rising_edge(CLK)) then
         if (LD = '1') then
            REG_OUT <= REG_IN;
         end if;
      end if;
   end process;
end reg8;
```

Figure 66: The final solution to *EXAMPLE 28*.

The circuit in the following example is slightly more complex than most of the examples in this tutorial. This means that there are many different solutions to the same problem. This is a common occurrence in VHDL; there are many times no signal right or best method to implementing a circuit. The following examples are essentially the same problem solved using two different but functionally equivalent solutions.

EXAMPLE 29

Use VHDL behavioral modeling to design the circuit shown at the right. Consider both of the loading signals to be active high; consider the circuit to be synchronized to the rising edge of the clock signal.



Solution: The circuit shown in *EXAMPLE 30* includes two 8-bit registers and a 2:1 MUX. This is an example of a bus-based data transfer in the output of the MUX is connected to the inputs of the two registers. Each of the two registers has their own independent load control input. The solution to *EXAMPLE 31* is shown in Figure 67. And as we've grown to expect, there are a couple of things worth noting about this solution.

- There are three concurrent statements in this solution: two behavioral models and one dataflow model.

- There is a separate process for each of the two registers. Although it would have been possible to represent both registers using one process, it would have been somewhat complicated and somewhat hard to understand. The better approach in VHDL is always to break tasks down into their logically separate functions and use the various VHDL modeling techniques as tools to keep the tasks separate and simple. The reality is that the synthesizer becomes your friend if you provide it with simple models. The quantity of VHDL code describing a certain design is immaterial; the complexity of any given model is determined by the most complex piece of code in the model. Simple is always better in VHDL.

- All of signal shown in the *EXAMPLE 32* have external linkage except for the output of the MUX. The MUX output is connected to the inputs of both registers. The final approach taken in this solution is typical in VHDL: many processes that communicate with each other through shared signals. In this example, there is only one shared signal but this is a fairly simple program. The same inter-process communication model is used in more complicated circuits.

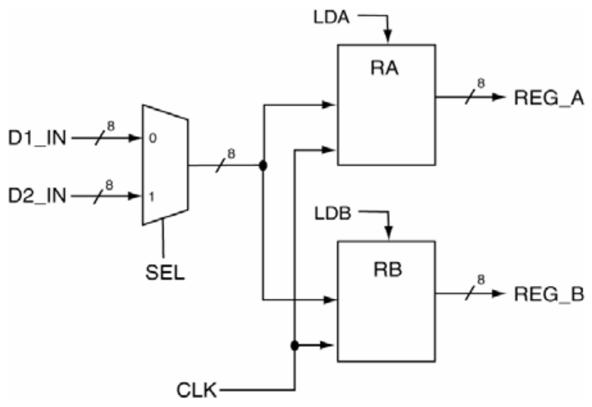
- The model for the 2:1 MUX uses the terminology "`(others => '0')`". This is a short-hand terminology for assigning all of the outputs to '0'. This real nice part about his code is that you do not need to know how many 0's you need to write. This is a nice feature in that if the width of the associated bundle were to change, this particular line of code would not need to be modified.

```
entity ckt_rtl is
   port (D1_IN,D2_IN : in std_logic_vector(7 downto 0);
             CLK,SEL : in std_logic;
             LDA,LDB : in std_logic;
         REG_A,REG_B : out std_logic_vector(7 downto 0));
end ckt_rtl;

architecture rtl_behavioral of ckt_rtl is
   -- intermediate signal declaration ---------------
   signal s_mux_result : std_logic_vector(7 downto 0);
begin

   ra: process(LDA,CLK,s_mux_result)
   begin
      if (rising_edge(CLK)) then
         if (LDA = '1') then
            REG_A <= s_mux_result;
         end if;
      end if;
   end process;

   rb: process(LDB,CLK,s_mux_result)
   begin
      if (rising_edge(CLK)) then
         if (LDB = '1') then
            REG_B <= s_mux_result;
         end if;
      end if;
   end process;

   with SEL select
   s_mux_result <= D1_IN when '1',
                   D2_IN when '0',
                   (others => '0') when others;

end rtl_behavioral;
```

Figure 67: The final solution of example *EXAMPLE 33*.

EXAMPLE 34

Use VHDL structural modeling to design the circuit shown at the right. Consider both of the loading signals to be active high; consider the circuit to be synchronized to the rising edge of the clock signal.



Solution: The final solution to *EXAMPLE 35* is shown in Figure 68. The solution to *EXAMPLE 36* is somewhat longer than the solution for *EXAMPLE 37*. This is because the original entity declarations are included in the solution. Other than that, there is not too much interesting to note here. This is a more realistic example of a structural model compared to the example presented in the section on structural modeling. There are only a few new and wonderful things to note about this solution.

- The massively important thing to note about the solution in Figure 68 is to not be intimidated by the shear quantity of code listed. The code is well structured; if you are able to recognize this structure, you will be more apt to understand the solution. And better yet, you'll be more on your way of being able to write you own amazing chunks of VHDL code.

- The VHDL source code shown in Figure 68 is very nicely formatted (if I do say so myself). In particular, the code is nicely indented. Properly indented code is highly desirable in that it nicely presents information based on the indented. No surprise here but, properly formatted code is easier to understand. Better yet, good looking code leads people who may or may not know otherwise into thinking your code is as actually as good as it looks. In this busy world of ours, a quick glance is just about all the time people (bosses and teachers) have to dedicate to perusing your VHDL source code. Welcome to the world.

```
entity mux2t1 is
   port (  A,B : in std_logic_vector(7 downto 0);
           SEL : in  std_logic;
         M_OUT : out std_logic_vector(7 downto 0));
end mux2t1;

architecture my_mux of mux2t1 is
begin
   with SEL select
   M_OUT <= A when ‘1’,
            B when ‘0’,
            (others => ‘0’) when others;
end my_mux;

entity reg8 is
    Port (  REG_IN : in std_logic_vector(7 downto 0);
            LD,CLK : in std_logic;
           REG_OUT : out std_logic_vector(7 downto 0));
end reg8;

architecture reg8 of reg8 is
begin
   reg: process(LD,CLK,REG_IN)
   begin
      if (rising_edge(CLK)) then
         if (LD = '1') then
            REG_OUT <= REG_IN;
         end if;
      end if;
   end process;
end reg8;

entity ckt_rtl is
   port (D1_IN,D2_IN : in std_logic_vector(7 downto 0);
             CLK,SEL : in std_logic;
             LDA,LDB : in std_logic;
         REG_A,REG_B : out std_logic_vector(7 downto 0));
end ckt_rtl;

architecture rtl_structural of ckt_rtl is

   -- component declaration -----------------------
   component mux2t1
      port (  A,B : in std_logic_vector(7 downto 0);
              SEL : in std_logic;
            M_OUT : out std_logic_vector(7 downto 0));
   end component;

   component reg8
       Port (  REG_IN : in std_logic_vector(7 downto 0);
               LD,CLK : in std_logic;
              REG_OUT : out std_logic_vector(7 downto 0));
   end component;

   -- intermediate signal declaration ---------------
   signal s_mux_result : std_logic_vector(7 downto 0);

begin
   ra: reg8
   port map ( REG_IN => s_mux_result,
                  LD => LDA,
                 CLK => CLK,
             REG_OUT => REG_A );

   rb: reg8
   port map ( REG_IN => s_mux_result,
                  LD => LDB,
                 CLK => CLK,
             REG_OUT => REG_B );

   m1: mux2t1
   port map (   A => D1_IN,
                B => D2_IN,
              SEL => SEL,
            M_OUT => s_mux_result);

end rtl_structural;
```

Figure 68: The final solution to *EXAMPLE 38* using a structural modeling approach.

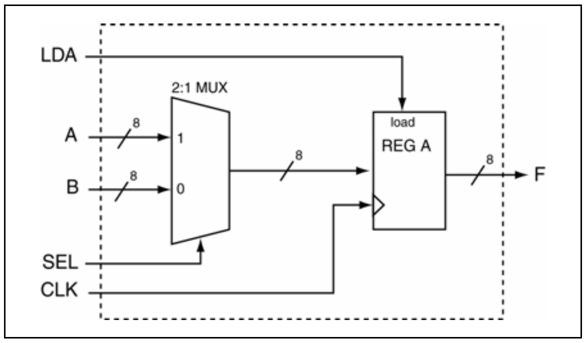
12.1 Important Points

- VHDL can be used to easily implement circuits at the register transfer level. The corresponding VHDL models can be implemented in either structural of full behavioral format.

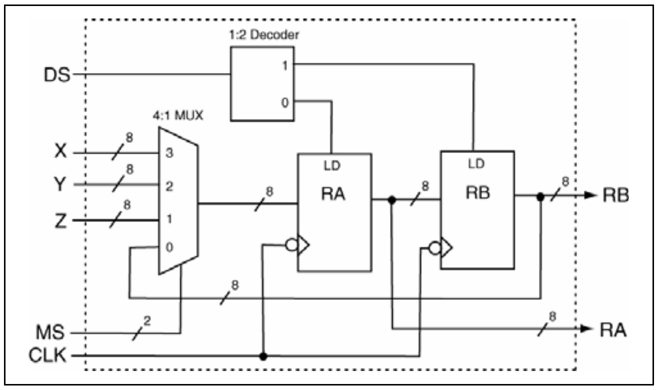
- RTL level VHDL models should strive for simplicity in their designs. If the behavioral models in the RTL design become complicated, the chances that your circuit works correctly greatly diminish due to the synthesis of the complicated circuit.

12.2 Exercises: Register Transfer Level Circuits

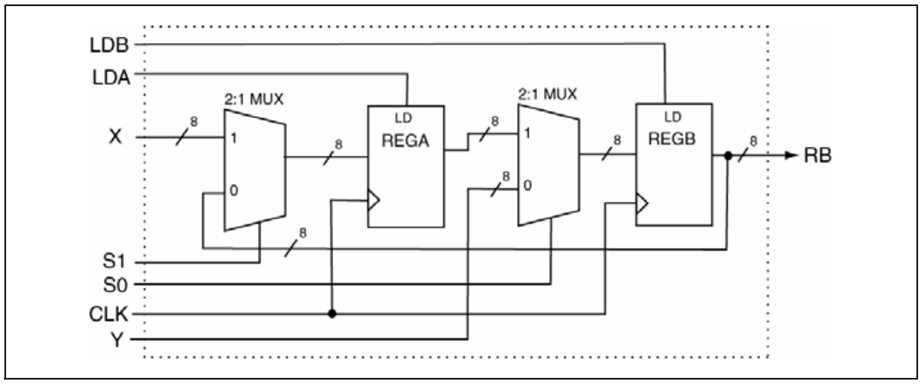
1. Provide a VHDL model that can be used to implement the following circuit.



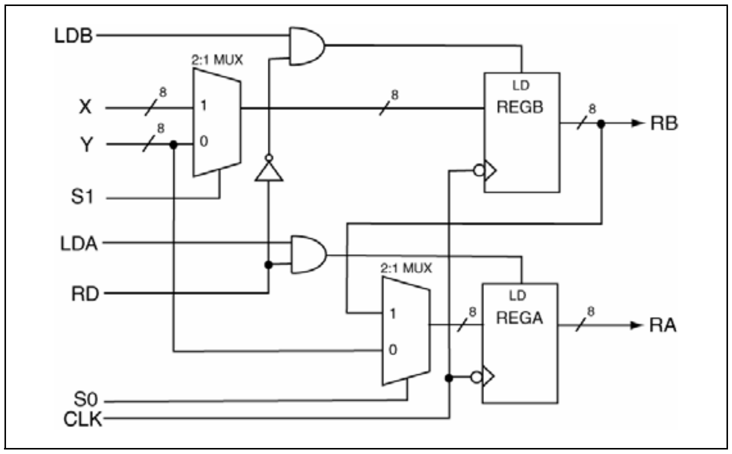
2. Provide a VHDL model that can be used to implement the following circuit.



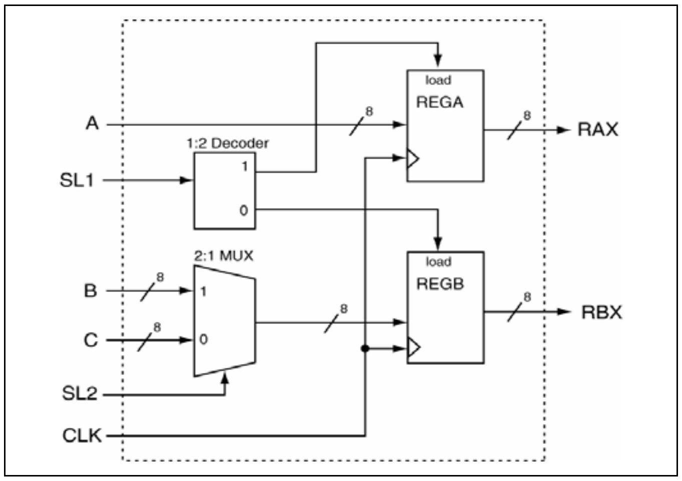
3. Provide a VHDL model that can be used to implement the following circuit.



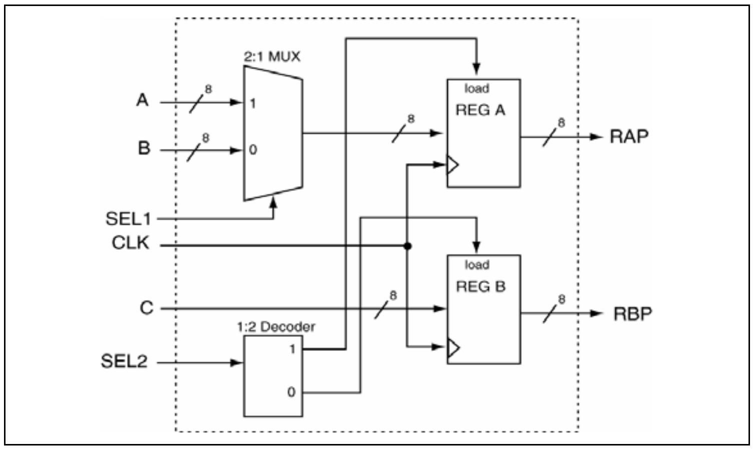
4. Provide a VHDL model that can be used to implement the following circuit.



5. Provide a VHDL model that can be used to implement the following circuit.



6. Provide a VHDL model that can be used to implement the following circuit.



13. 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.

13.1 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.

13.2 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 12. 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 13. These examples include several new data types which will be discussed in Section 13.5.

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 12: 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 13: Example declarations for signal, variable, and constant data objects.

13.3 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 13, the assignment operator is overloaded which allows it to be used to assign initial values to the three listed forms of data objects.

13.4 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.

Since this tutorial has primarily been focusing on relatively simple circuits, using exclusively signal objects has been sufficient. As your digital designs become more complicated, there is a greater chance that you’ll need more control of your models than signals alone can provide. The main characteristic of signals that leave them somewhat limited in complex designs is when and how they are scheduled. More specifically, assignments made to signals during in a process are actually only “scheduled” when the signal assignment statement appears in the process. The actual assignment is not made until after the process terminates. This is why multiple signal assignments can be made to the same signal during the execution of a process without generating any type of synthesis error. In the case of multiple assignments, only the most recent assignment to the signal during process execution is assigned. The important thing here is that the signal assignment is not made until after the process terminates. The potential problem here is that the “new” result (the new value assigned to the signal) is not available to use inside the process. The mechanism here is that the process uses all the values of the signals that were present when the process was started regardless of if the signals were assigned inside of the process. Once again, signal assignment within a process only schedules the assignment to occur once the process has terminated.

Variable assignment within processes is different. When a variable is assigned a value inside of a process, the assignment is immediate and the newly assigned value can be used immediately inside of the process. In other words, the variable assignment is not scheduled as it was for the signal. This is a giant difference and has massive ramifications in both the circuit simulation and synthesis realm.

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.

13.5 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.

13.6 Commonly Used Types

The types used thus far in this tutorial as well as two new types are listed in Table 14. 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 13.8. The enumerated type was used during the discussion of Finite States Machines in Section 10. The integer type was cryptically mentioned in Section 13.2 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 22
boolean	`variable my_test : boolean := false;`	
integer	`signal iter_cnt : integer := 0;`	EXAMPLE 39

Table 14: Data types used in this tutorial.

13.7 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 69.

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

Figure 69: 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 70.

```
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 70: Examples of integer type declarations.

Although each of the types listed in Figure 70 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 70, each of the statements in Figure 71 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 71: Examples of illegal assignment statements.

13.8 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 72. 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.

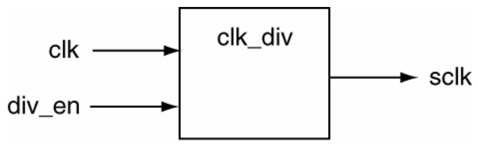
```
type std_ulogic is ( ‘U’, -- uninitialized
                     ‘X’, -- forcing unknown
                     ‘0’, -- forcing 0
                     ‘1’, -- forcing 1
                     ‘Z’, -- high impedance
                     ‘W’, -- weak unknown
                     ‘L’, -- weak 0
        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

Figure 72: 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 72, 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 39

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 73. 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 : out std_logic);
end clk_div;


architecture my_clk_div of clk_div is

   type my_count is range 0 to 100;        -- user-defined type
   constant max_count : my_count := 63;    -- user-defined constant

   signal tmp_sclk : std_logic;            -- intermediate signal for clock

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
               tmp_sclk <= not tmp_sclk;    -- toggle output
               div_count := 0;      -- reset count
            else
               div_count := div_count + 1;
            end if;
         else                       -- divider disabled
            div_count := 0;         -- reset count
            tmp_sclk <= ‘0’;        -- turn off output
         end if;
      end if;
   end process my_div;

   s_clk <= tmp_sclk;   -- assign to output

end my_clk_div;
```

Figure 73: Solution for *EXAMPLE 39*

14. 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.

14.1 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 74.

```
label: for index in a_range loop
   sequential statements...
end loop label;
```
	```
	label: while (condition) loop
	   sequential statements...
	end loop label;
	```

Figure 74: 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.

14.2 for Loops

The basic form of the **`for`** loop was shown in Figure 74. 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.

```vhdl
for cnt_val in 0 to 24 loop
   -- sequential_statements
end loop;
```

```vhdl
type my_range is range 0 to 24;

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

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

```vhdl
type my_range is range 24 downto 0;

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

(a) (b)

Figure 75: 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 76, `for` loops can also apply the `downto` option. This option makes more sense when the range is specified in the `for` loop declaration.

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

```vhdl
type my_range is range 24 downto 0;

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

(a) (b)

Figure 76: `for` loops using the downto approach.

14.3 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 77. Needless to say, the VHDL code appearing in Figure 77(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 77: Two examples of while loops calculating a Fibonacci sum.

14.4 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.

14.5 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 78. 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 78: Examples of the two forms of next statements.

14.6 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 79.

```
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 79: Example of the two forms of exit statements.

15. 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.

15.1 RET D Flip-flop (Behavioral Model)

```
-------------------------------------------------------------------
--  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 80: VHDL code for D flip-flop.

15.2 FET D Flip-flop with Active-low Asynchronous Preset (Behavioral Model)

```
-------------------------------------------------------------------
--  D flip-flop: FET D flip-flop with asynchronous preset. The
--  preset input takes precedence over the synchronous input.
--
--  Required signals:
---------------------------------------------------
--  CLK,D,S: in STD_LOGIC;
--  Q: out STD_LOGIC;
-------------------------------------------------------------------
process (CLK,D,S)
begin
   if (S = ‘0’) then
      Q <= ‘1’;
   elsif (falling_edge(CLK)) then
      Q <= D;
   end if;
end process;
```

15.3 8-Bit Register with Load Enable (Behavioral Model)

```
-------------------------------------------------------------------
--  Register: 8-bit Register with load enable.
--
--  Required signals:
---------------------------------------------------
--  CLK,LD: 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 (rising_edge(CLK)) then
      if (LD = ‘1’) then   -- positive logic for CS
         D_OUT <= D_IN;
      end if;
   end if;
end process;
```

Figure 81: VHDL code for 8-bit register.

15.4 Synchronous Up/Down Counter (Behavioral Model)

```
-------------------------------------------------------------------
-- Counter: synchronous up/down counter with asynchronous
-- reset and synchronous parallel load.
---------------------------------------------------
entity COUNT_8B is
   port ( RESET,CLK,LD,UP : in std_logic;
                      DIN : in std_logic_vector (7 downto 0);
                    COUNT : out std_logic_vector (7 downto 0));
end COUNT_8B;

architecture my_count of COUNT_8B is
   signal  t_cnt : std_logic_vector(7 downto 0);
begin
   process (CLK, RESET)
   begin
      if (RESET = '1') then
         t_cnt <= (others => ‘0’);  -- clear
      elsif (rising_edge(CLK)) then
         if (LD = '1') then     t_cnt <= DIN;  -- load
         else
            if (UP = '1') then  t_cnt <= t_cnt + 1; -- incr
            else                t_cnt <= t_cnt - 1; -- decr
            end if;
         end if;
      end if;
   end process;

   COUNT <= t_cnt;

end my_count;
```

Figure 82: VHDL code for Up/Down counter.

15.5 Shift Register with Synchronous Parallel Load (Behavioral Model)

```
-------------------------------------------------------------------
-- 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 83: VHDL code for shift register.

15.6 8-Bit Comparator (Behavioral Model)

```
-------------------------------------------------------------------
-- Comparator: Implemented as a behavioral model. The outputs
-- include equals, less than, and greater than status.
--
-- 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 ( 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 process;
```

Figure 84: VHDL code for Comparator.

15.7 BCD to 7-Segment Decoder (Dataflow Model)

```
-------------------------------------------------------------------
-- 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 85: VHDL code for BCD to 7-Segment Decoder.

15.8 4:1 Multiplexer (Behavioral Model)

```
-------------------------------------------------------------------
-- 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 => (others => ‘0’);
   end case;
end process;
```

Figure 86: VHDL code for 4:1 Multiplexor.

15.9 4:1 Multiplexer (Dataflow Model)

```
-------------------------------------------------------------------
-- A 4:1 multiplexer implemented as dataflow model using a
-- selective signal assignment statement.
--
-- Required signals:
---------------------------------------------------
-- SEL: in STD_LOGIC_VECTOR(1 downto 0);
-- A, B, C, D:in STD_LOGIC;
-- MUX_OUT: out STD_LOGIC;
----------------------------------------------------------------------
with SEL select
   MUX_OUT <= A when “00”,
              B when “01”,
              C when “10”,
              D when “11”,
              (others => ‘0’) when others;
```

Figure 87: VHDL code for 4:1 Multiplexor.

15.10 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 88: VHDL code for 3:8 Decoder.

A *Appendix*: VHDL Reserved Words

Table 15 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 15: A complete list of VHDL reserved words.