FREE RANGE VHDL *10*

The no-frills guide to writing powerful code for your digital implementations

BRYAN MEALY
FABRIZIO TAPPERO

Update: Vitor Angelo

Free Range VHDL

Bryan Mealy, Fabrizio Tappero

Update: Vitor Angelo

Free Range VHDL

Final Release: 1.21
Updated by Vitor Angelo at: 23 March 2024
Updated Release: 2.23.1
Git Hash: d8a01d43f67af6d94b404808fe4024600758bcdf
Book size: 160 mm by 240 mm
Pages: 206

The electronic version of this book could originally be downloaded free of charge from: http://www.freerangefactory.org. At the time of this update, the original website was still unavailable. The improvements were based on the LaTeX source kindly released by the authors.

Feedback and Contribution

The author responsible for this update would be glad to receive corrections and improvement suggestions to this book.

Cover and Artwork by Robert Ash.

To everyone who helped

Table of Contents

Acknowledgments **v**

Purpose of this book **1**

1 Introduction To VHDL **5**
1.1 Golden Rules of VHDL 8
1.2 Tools Needed for VHDL Development 8

2 VHDL Invariants **11**
2.1 Case Sensitivity 11
2.2 White Space 11
2.3 Comments 12
2.4 Parentheses 13
2.5 VHDL Statements 13
2.6 `if`, `case` and `loop` Statements 14
2.7 Identifiers 14
2.8 Reserved Words 15
2.9 VHDL Coding Style 15

3 VHDL Design Units **17**
3.1 Entity 18
3.2 VHDL Standard Libraries 22

3.3 Architecture 22
3.4 Signal and Variable Assignments 24
3.5 Summary 26
3.6 Exercises 27

4 VHDL Concurrent Programming Paradigm 29
4.1 Concurrent Statements 30
4.2 Signal Assignment Operator "<=" 33
4.3 Concurrent Signal Assignment Statements 34
4.4 Conditional Signal Assignment `when` 39
4.5 Selected Signal Assignment `with select` 43
4.6 *Testbenches* 48
4.7 Summary 51
4.8 Exercises 52

5 Standard Models in VHDL Architectures 53
5.1 Data-flow Style Architecture 54
5.2 Behavioral Style Architecture 55
5.3 Process Statement 55
5.4 Sequential Statements 57
5.4.1 Assignment Statements 59
5.4.2 `if` Statement 60
5.4.3 `case` Statement 66
5.5 Caveats Regarding Sequential Statements 69
5.6 *Automatic Testbenches* 71
5.7 Summary 73
5.8 Exercises: Behavioral Modeling 73

6 VHDL Operators 75
6.1 Logical Operators 76
6.2 Relational Operators 76
6.3 Shift Operator 76
6.4 Other Operators 78
6.5 Concatenation Operator 78
6.6 Modulus and Remainder Operators 78

6.7 Review of Almost Everything Up to Now 80
6.8 *Writing less code* 81

7 Using VHDL for Sequential Circuits 83
7.1 Simple Storage Elements Using VHDL 83
7.2 Inducing Memory: Data-flow vs. Behavioral Modeling 91
7.3 Important Points 92
7.4 Testbenches with clock 93
7.5 Exercises: Basic Memory Elements 96

8 Finite State Machine Design Using VHDL 99
8.1 VHDL Behavioral Representation of FSMs 102
8.2 One-Hot Encoding for FSMs 111
8.3 Important Points 116
8.4 Exercises: Behavioral Modeling of FSMs 118

9 Structural Modeling In VHDL 129
9.1 VHDL Modularity with Components 131
9.1.1 Step 1: top level entity 134
9.1.2 Step 2: Components for lower-level design units 134
9.1.3 Step 3: Internal signals which connect the design units 135
9.1.4 Step 4: Instantiate all the design units 136
9.2 Generic Map 139
9.3 Structured Testbenches (Unit Testing) 141
9.4 Important Points 144
9.5 Exercises: Structural Modeling 146

10 Registers and Register Transfer Level 149
10.1 RTL-level Design 149
10.2 Text Based Testbenches 156
10.3 Important Points 159
10.4 Exercises: Register Transfer Level Circuits 159

11 Data Objects **163**
11.1 Types of Data Objects 163
11.2 Data Object Declarations 164
11.3 Variables and Assignment Operator "`:=`" 165
11.4 Signals vs. Variables 166
11.5 Standard Data Types 167
11.6 User-Defined Types 168
11.7 Integer Types 168
11.8 `signed` and `unsigned` Types 170
11.9 `std_logic` Types 171
11.10 Testbenches with Automatic Arithmetic 175
11.11 Important Points 176

12 Looping Constructs **179**
12.1 `for` and `while` Loops 179
12.1.1 `for` Loops 181
12.1.2 `while` Loops 182
12.1.3 Loop Control: `next` and `exit` Statements 183

13 Standard Digital Circuits in VHDL **185**
13.1 RET D Flip-flop - Behavioral Model 186
13.2 FET D Flip-flop with Active-low Asynchronous Preset - Behavioral Model 186
13.3 8-Bit Register with Load Enable - Behavioral Model 187
13.4 Synchronous Up/Down Counter - Behavioral Model 187
13.5 Shift Register with Synchronous Parallel Load - Behavioral Model 188
13.6 8-Bit Comparator - Behavioral Model 189
13.7 BCD to 7-Segment Decoder - Data-Flow Model 189
13.8 4:1 Multiplexer - Behavioral Model 190
13.9 4:1 Multiplexer - Data-Flow Model 191
13.10 Decoder 191

Appendix A Contributors to This Book **193**

Acknowledgments

The authors would like to thank Christina Jarron for her invaluable contribution to proofreading this book and for her useful suggestions. Special thanks also to Rob Ash for helping us make the cover of the book distinctive with his original artwork. A massive thank you goes to Keith Knowles for his time and effort in reviewing and editing the final draft of this book. Finally, the authors would like to thank all the people who have provided feedback and suggestions.

The main incentive for the update came from the students who liked the style of the book due to the quick results that it made possible. Corrections and improvements kept the spirit of allowing this dynamic and fun introduction to VHDL.

Purpose of this book

The purpose of this book is to provide students and young engineers with a guide to help them develop the skills necessary to be able to use VHDL for introductory and intermediate level digital design. These skills will also give you the ability and the confidence to continue on with VHDL-based digital design. In this way, you will also take steps toward developing the skills required to implement more advanced digital design systems. Although there are many books and on-line tutorials dealing with VHDL, these sources are often troublesome for several reasons. Firstly, 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 either 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 many topics and a lot of extraneous information too early. Most of this material would best appear later in the presentation. Material presented in this manner has a tendency to be confusing, is easily forgotten if misunderstood or simply is never applied. The approach taken by this book is to provide only what you need to know to quickly get up and running in VHDL. As with all learning, once you have 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 book is to present topics to someone familiar with digital logic design and with some skills in algorithmic programming languages such as Java or C. The information presented here is focused on giving a solid knowledge of the approach and function of VHDL. With a logical and intelligent introduction to basic VHDL concepts, you should be able to quickly and efficiently create useful VHDL code. In this way, you will see 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 an extremely powerful tool. The more you understand as you study and work with VHDL, the more it will enhance your learning experience independently of your particular area of interest. It is well worth noting that VHDL and other similar hardware design languages are used to create most of the digital integrated circuits found in the various electronic gizmos that overwhelm our modern lives. The concept of using software to design hardware that is controlled by software will surely provide you with endless hours of contemplation. VHDL is a very exciting language and mastering it will allow you to implement systems capable of handling and processing in parallel ns-level logic events in a comfortable software environment.

This book was written with the intention of being freely available to everybody. The formatted electronic version of this book is available from the Internet. Any part of this book can be copied, distributed and modified in accordance with the conditions of its license.

DISCLAIMER: This book quickly takes you down the path toward understanding VHDL and writing solid VHDL code. The ideas presented herein represent the core knowledge you will need to get up and running with VHDL. This book in no way presents a complete description of the VHDL language. In an effort to expedite the learning process, many of the finer details of VHDL have been omitted from this book. Anyone who has the time and inclination should feel free to further explore the true depth of the VHDL language. There are many on-line VHDL reference books and

free tutorials. If you find yourself becoming curious about what this book is not telling you about VHDL, take a look at some of these references. Another aspect of this quick approach is that it must be agnostic to the complete design flow for a specific hardware, which involves many crucial steps which depend on various tools and target technologies.

1

Introduction To VHDL

VHDL has a rich and interesting history[1]. But since knowing this history is probably not going to help you write better VHDL code, it will only be briefly mentioned here. Consulting other, lengthier texts or search engines will provide more information for those who are interested. Regarding the VHDL acronym, the V is short for yet another acronym: VHSIC or Very High-Speed Integrated Circuit. The HDL stands for Hardware Description Language. Clearly, the state of technical affairs these days has done away with the need for nested acronyms. VHDL is a true computer language with the accompanying set of syntax and usage rules. But, as opposed to higher-level computer languages, VHDL is primarily used to *describe hardware*. The tendency for most people familiar with a higher-level computer language such as C, Python or Java is to view VHDL as just another computer language. This is not altogether a bad approach if such a view facilitates the understanding and memorization of the language syntax and structure. The common mistake made by someone with this approach is to attempt to program in VHDL as they would program a higher-level computer language. Higher-level computer languages are sequential in nature; VHDL is not.

VHDL was invented to describe hardware and in fact VHDL is a *concurrent* language. What this means is that, normally, VHDL instructions are all executed at the same time (concurrently), regardless of the size of

[1]VHDL-Wikipedia: `http://en.wikipedia.org/wiki/VHDL`

your implementation. Another way of looking 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 should necessarily encourage you to re-think how you write your VHDL code. Attempts to write VHDL code with a high-level language style generally result in code that nobody understands. Moreover, the tools used to synthesize[2] this type of code have a tendency to generate circuits that generally do not 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 unnecessarily 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 it 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 it. 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. Second, 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[3] interpreted by software tools in such a way as to create actual digital circuits in a process known as *synthesis*.

There are other logic languages available to model the behavior of digital circuit designs that are easy to use because they provide a graphical method to model circuits. For them, the tendency 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

[2]Synthesis: the process of interpreting VHDL code and outputting a definition of the physical circuit implementation to be programmed on a device such as an FPGA.

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

by the ever-increasing complexity of digital circuits you are 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 able to describe exactly how your digital circuit works (in other words, modeling it) without having to worry about the details of connecting large quantities of signal lines. Having a working knowledge of VHDL will provide you with the tools to model digital circuits in a much more intelligent manner.

Finally, you will 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 will 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.

VHDL is a very exciting language that can allow the design and implementation of functions capable of processing an enormous amount of data by employing a relatively low-cost and low-power hardware. Moreover, what is really impressive is that, via simple VHDL modules, you can have direct access to basic ns-level logic events as well as communicate using a USB port or drive a VGA monitor to visualize graphics of modest complexity.

Modeling digital circuits with VHDL is a form of modern digital design distinct from schematic-based approaches. The programmer writes a loose description of what the final logic circuit should do and a language compiler, in this case called a synthesizer, attempts to "infer" what the actual final physical logic circuit should be. Novice programmers are not always able to convince the synthesizer to implement something that seems very clear in their minds. A somehow old-fashioned alternative to a descriptive language such as VHDL is one in which the programmer simply interconnects a finite number of digital blocks that he has pooled from a library in an attempt to reach the same objective. This approach is not only very time consuming but also inherently limiting and very error prone.

Modern digital design is more about appropriately modeling digital circuits and maintaining a quality description of the circuit. All that is left

now is to learn how to properly use VHDL to describe what you want to implement.

1.1 Golden Rules of VHDL

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

VHDL is a hardware-design language. Although most people have probably already been exposed 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. What does this mean? It means that unless you are inside certain constructs, your code lines will be executed almost all at once. If your VHDL code appears too similar to code of a higher-level computer language, it is 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 do not understand basic digital constructs, you will probably be unable 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 broken down into some simple programming constructs. 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. In other words, if you are not able to roughly envision the digital circuit you are trying to model in terms of basic digital circuits, you will probably misuse VHDL, thus angering the VHDL gods. VHDL is cool, but it is not as magical as it initially appears to be.

1.2 Tools Needed for VHDL Development

VHDL is a language used to implement hardware which will run other software (for example C). A Field Programmable Gate Array (FPGA) is probably the most common device that you can use for your VHDL

implementations. If you want to do VHDL coding for FPGAs you will have to play within the rules that current major FPGA manufacturers have drawn up to help you (rules which also ensure their continued existence in the market).

The successful implementation of a VHDL-based system roughly calls for the following steps: VHDL code writing, compiling, simulation and synthesis. All major FPGA manufacturers have a set of software and hardware tools that you can use to perform the mentioned steps. Most of these software tools are free of charge (occasionally with limitations) but are not open-source. Nevertheless, the same tools follow a license scheme, whereby paying a certain amount of money allows you to take advantage of sophisticated software features, access to high-end hardware targets or get your hands on proprietary libraries with lots of components (e.g. a 32-bit processor) that you can easily include in your own project.

If your have no interest in the advanced features offered by the paid tools, you can use open-source solutions (e.g. GHDL[4] and NVC[5] are quite actively maintained) which will allow you to compile and simulate your VHDL code using the open-source tool *gcc*[6]. At the time of writing, no open-source solution is available for the synthesis process with broad hardware support (although it seems to be worth following the F4PGA Project[7]). However synthesis can be accomplished using a free-license version of any major FPGA manufacturer's software tool (e.g. AMD/Xilinx Vivado or Intel/Altera Quartus).

Thanks to the open-source community, you can write, compile and simulate VHDL systems using excellent open-source solutions, even without installing software locally[8]. This book will show you how to get up and running with the VHDL language. For further tasks such as synthesis and upload of your code into an FPGA, the free of charge versions of the main manufacturers tools can be employed.

[4]VHDL simulator GHDL: `http://ghdl.free.fr`
[5]VHDL software tool NVC: `https://www.nickg.me.uk/nvc/`
[6]Multi-language open-source compiler GCC: `http://gcc.gnu.org`
[7]F4PGA Project: `https://f4pga.readthedocs.io`
[8]EDA Playground: `https://www.edaplayground.com/`

2

VHDL Invariants

There are several features of VHDL that you should know before moving forward. Although it is rarely a good idea for people to memorize anything, you should memorize the basic concepts presented in this section. This should help eliminate some of the drudgery involved in learning a new programming language and lay the foundation that will enable you to create visually pleasing and good VHDL source code. Don't worry about fully understanding the statements in the Listings presented in this Chapter.

2.1 Case Sensitivity

VHDL is not case sensitive. This means that the two statements shown in Listing 2.1 have the exact same meaning. Despite being valid, this is **not** an example of good VHDL coding style.

```
1 Dout <= A and B;
2 doUt <= a AnD b;
```

Listing 2.1: An example of VHDL case insensitivity

2.2 White Space

VHDL is not sensitive to white space and tabs in the source document. The two statements in Listing 2.2 have the same meaning. Keep in mind

that proper indenting using spaces can make the code a lot clearer.

```
1 nQ <= In_a or In_b;
2 nQ   <=in_a OR       in_b;
```

Listing 2.2: An example showing VHDL's indifference to white space

2.3 Comments

Comments in VHDL begin with the symbol "--" (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. Listing 2.3 shows two types of commenting styles. The block-style comments * ... *\ (that span multiple lines without requiring comment marks on every line) have been introduced in the 2008 version of VHDL. Keep in mind that every newer feature used increases the chance of problems when migrating the code from one tool to another.

```
1 -- This next section of code does this and that...
2 -- This type of comment is the best one for compatibility.
3 PS_reg <= NS_reg; -- Comments should explain WHY, not WHAT!
```

Listing 2.3: Two typical uses of comments

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 (like your "future self"). It is inappropriate to comment something that is patently obvious. It is hard to imagine code that has too few comments so don't be shy: use lots of comments. Research has shown that using lots of appropriate comments is a sign of high intelligence.

2.4 Parentheses

VHDL is relatively lax on its requirement for using parentheses. 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 are left scratching their heads, a better idea is to practice liberal use of parentheses to ensure the human reader of your source code understands the purpose of the code. Once again, the two statements appearing in Listing 2.4 have the same meaning. Note that extra white space has been added along with the parentheses to make the lower statement clearer.

```
1 if x = '0' and y = '0' or z = '1' then
2         blah; -- some useful statement
3         blah; -- some useful statement
4 end if;
5 if ( ((x = '0') and (y = '0')) or (z = '1') ) then
6         blah; -- some useful statement
7         blah; -- some useful statement
8 end if;
```

Listing 2.4: Example of parentheses that can improve clarity

2.5 VHDL Statements

Similar to other algorithmic computer languages, every VHDL statement is terminated with a semicolon. The VHDL synthesizer is not as forgiving as other languages when superfluous semicolons are placed in the source code. Missing semicolons can be even worse as the error message it generates may refer to other following lines. Advanced development environments may indicate this type of error much sooner than the compilation time.

2.6 `if`, `case` and `loop` Statements

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

In general, you should not worry too much about memorizing code syntax as chances are you will use an editor sophisticated enough to have code snippets (namely Gedit[1]). A good programmer distinguishes himself by other means than perfectly remembering code syntax.

2.7 Identifiers

An identifier refers to the name given to 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.

- Identifiers should be self-describing. The text you apply to identifiers should inform the use and purpose of the item it represents.
- Identifiers can contain as many characters as you want. Shorter names makes the code more readable, but longer names bring more information. It is up to the programmer to choose a reasonable compromise.

[1] Gedit, the official Linux GNOME text editor: `http://projects.gnome.org/gedit`

- Identifiers can only contain a combination of letters (A-Z and a-z), digits (0-9) and the underscore character ("_").
- Identifiers must start with an alphabetic character (A-Z or a-z).
- Identifiers must not end with an underscore and must never have two consecutive underscores.
- The best identifier for a function that calculates the position of the Earth is **CalcEarthPosition** or **calc_earth_position**. Try to be consistent.

Intelligent choices for identifiers make your VHDL code more readable, understandable and more impressive to coworkers, superiors, friends etc.

2.8 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, cannot be used as identifiers when writing VHDL code. A partial list of reserved words appears in Listing 2.5. Notably missing from Listing 2.5 are standard operators such as AND, OR, XOR and also others added by newer VHDL versions.

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

Listing 2.5: A short list of VHDL reserved words

2.9 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 parentheses creates a 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 where you have the ability to make the document more readable without changing the functioning of the code. This is primarily done by indenting certain portions of the program, using self-describing identifiers and providing proper comments when and where necessary.

Instead of stating here a bunch of rules to be followed or how the code should look like, the goal simply should be: make the source code readable. Listed below are a few thoughts on what makes readable source code.

- Chances are that if your VHDL source code is readable to you, it will be readable to others. 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. These are the people you want to please. These people are probably very busy and more than willing to make a superficial 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 will find that the VHDL compiler does a great job of knowing a document has an 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 when using proper indentation and sufficient comments.

3

VHDL Design Units

The "black-box" approach to any type of design implies a hierarchical structure in which 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 by 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 code.

Not surprisingly, VHDL descriptions of circuits are based on 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 can of course be other black boxes). In VHDL, the black box is referred to as **entity** and the stuff that goes inside it is referred to as the **architecture**. For this reason, the VHDL entity and architecture are closely related. As you can probably imagine, creating the entity is relatively simple while a good portion of the VHDL coding time is spent on properly writing the architecture. Our approach here is to present an introduction to writing VHDL code by describing the entity and then moving on to the details of writing the

architecture. Familiarity with the entity will hopefully aid in your learning of the techniques to describe the architecture.

3.1 Entity

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 describes digital circuits, the entity simply lists the various inputs and outputs of the underlying circuitry. In VHDL terms, the black box is described by an entity declaration. The syntax of the entity declaration is shown in Listing 3.1.

```
1 entity my_entity is
2 port(
3     port_name_1 : in    std_logic;
4     port_name_2 : out   std_logic;
5     port_name_3 : inout std_logic); --don't forget the semicolon
6 end my_entity; -- do not forget this semicolon either
```

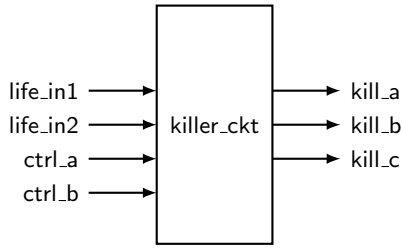
Listing 3.1: The entity declaration in VHDL

`my_entity` defines the name of the entity. The next section is nothing more than the list of signals from the underlying circuit that are available to the outside world, which is why it is often referred to as an interface specification. The `port_name_x` is an identifier used to differentiate the various signals. The next keyword (the keyword `in`) specifies the direction of the signal relative to the entity where signals can either enter, exit or do both. These input and output signals are associated with the keywords **in**, **out** and **inout**[1] respectively. The next keyword (the keyword `std_logic`) refers to the type of data that the port will handle. There are several data types available in VHDL but we will primarily deal with the `std_logic` type and derived versions. More information regarding the various VHDL data types will be discussed later.

[1]The `inout` data mode will be discussed later on in the book.

When you attempt to write fairly complicated VHDL code, you will need to split your code into different files, functions and packages constructors which will help you better deal with your code. In this scenario, the entity body will not only host the port definition statements but, most likely, other procedures as well. We will talk about this later in the book.

```
1  ------------------------------
2  --  interface description   --
3  --  of killer_ckt           --
4  ------------------------------
5  entity killer_ckt is
6  port (
7  life_in1        : in  std_logic;
8  life_in2        : in  std_logic;
9  ctrl_a, ctrl_b  : in  std_logic;
10 kill_a          : out std_logic;
11 kill_b, kill_c  : out std_logic);
12 end killer_ckt;
```



Listing 3.2: VHDL entity declaration

Listing 3.2 shows an example of a black box and the VHDL code used to describe it. Below are a few points about the code. Most of them deal with the readability and understandability of the VHDL code.

- Each port name is unique and has an associated mode (**in**/**out**) and data type (**std_logic**). This is a requirement.
- The VHDL compiler allows several port names to be included on a single line (separated by commas). Always strive for readability.
- Port names are somewhat lined up to increase readability. This is not a requirement but it is a good use of white spaces (which are ignored).
- A comment, which tells us what this entity does, is included.
- 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: do not be scared, draw a picture.

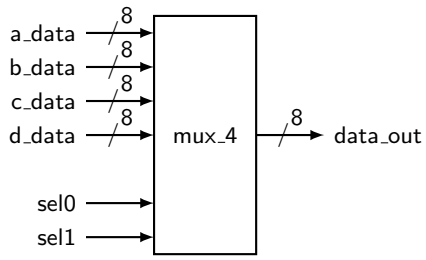
Hopefully, you are not finding these entity specifications too challenging. In fact, they are so straightforward, we will throw in one last twist before we leave the realm of VHDL entities. Most of the more meaningful circuits that you will be designing, analyzing and testing have many similar and closely 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, buses are often used in digital circuits. Unfortunately, the word bus also refers to established data transfer protocols. To disambiguate the word bus, we will be using the word "bundle" to refer to a set of similar signals and bus to refer to a protocol.

Bundles are easily described in the VHDL entity. All that is needed is a new data type and a special notation to indicate when a signal is a bundle or not. A few examples are shown in Listing 3.3. In these examples note that the mode remains the same but the type has changed. The `std_logic` data type has now been replaced by the word `std_logic_vector` to indicate that each signal name contains more than one signal. There are ways to reference individual members of each bundle, but we will get to those details later.

As you can see by examining Listing 3.3, there are two possible methods to describe the signals in a bundle. These two methods are shown in the argument lists that follow the data type declaration. The signals in the bundle can be listed in one of two orders which are specified by the **to** and **downto** keywords. If you want the most significant bit of your bundle to be the first bit on the left you use the **downto** keyword. Be sure not to forget the orientation of signals when you are using this notation in your VHDL model.

In the black box of Listing 3.3 you can see the formal notation for a bundle. Note that the black box uses a slash-and-number notation. 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 relative to Listing 3.3 is that the input lines `sel1` and `sel0` could have been made into one bundle containing the two signals.




```
1  ------------------------------------------------------------
2  -- Unlike the other examples, this is actually an interface
3  -- for a MUX that selects one of four bus lines for the output.
4  ------------------------------------------------------------
5  entity mux4 is
6  port (  a_data     : in     std_logic_vector(0 to 7);
7          b_data     : in     std_logic_vector(0 to 7);
8          c_data     : in     std_logic_vector(0 to 7);
9          d_data     : in     std_logic_vector(0 to 7);
10         sel1,sel0  : in     std_logic;
11         data_out   : out    std_logic_vector(7 downto 0));
12 end mux4;
```

Listing 3.3: Entity declaration with bundles

The data type `std_logic` and the data type `std_logic_vector` is what the IEEE has standardized for the representation of digital signals. Normally, you should consider that these data types assume the logic value `1` or the logic value `0`. However, as specified in the `std_logic_1164` package, the implementation of the `std_logic` type (and the `std_logic_vector` type) is a little more generous and includes 9 different values, specifically: `0,1,U,X,Z,W,L,H,-`.

The data type `std_logic` becomes available to you soon after the declaration `library IEEE; use IEEE.std_logic_1164.all;` at the beginning of your code.

The reason for all these values is the desire for modeling three-state drivers, pull-up and pull-down outputs, high impedance state and a few others types of inputs/outputs. For more details refer to the IEEE 1164 Standard[2].

Alternatively to the `std_logic` data type, VHDL programmers sometimes use the much simpler data type `bit` which has only the logic values `1` and `0`.

3.2 VHDL Standard Libraries

The VHDL language, as many other computer languages, has gone through a long and intense evolution. Among the most important standardization steps we can mention are the release of the `IEEE Standard 1164` package as well as some child standards that further extended the functionality of the language. In order to take advantage of the main implementable feature of VHDL you just need to import the two main library packages as shown in lines 2∼4 of Listing 3.4.

Once these packages have been included, you will have access to a very large set of goodies: several data types, overloaded operators, various conversion functions, math functions and so on. For instance, the inclusion of the package `numeric_std.all` will give you the possibility of using the `unsigned` data type and the function `to_unsigned` shown in Listing 3.4. For a detailed description of what these libraries include, refer to the `Language Templates` of your favorite synthesis software tool.

3.3 Architecture

The VHDL **entity** declaration, introduced before, describes the interface or the external representation of the circuit. The **architecture** 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

[2]IEEE 1164 Standard: `http://en.wikipedia.org/wiki/IEEE_1164`

```vhdl
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all; -- basic IEEE library
use IEEE.numeric_std.all;    -- IEEE library for the unsigned
-- type and various arithmetic operators

-- WARNING: in general try NOT to use the following libraries
--          because they are not IEEE standard libraries
-- use IEEE.std_logic_arith.all;
-- use IEEE.std_logic_unsigned.all;
-- use IEEE.std_logic_signed

-- entity
entity my_ent is
    port ( A, B, C : in  std_logic;
           F       : out std_logic);
end my_ent;
-- architecture
architecture my_arch of my_ent is
    signal v1, v2 : std_logic_vector (3 downto 0);
    signal u1     : unsigned (3 downto 0);
    signal i1     : integer;
begin
    u1 <= "1101";
    i1 <= 13; -- decimal equivalent to "1101"
    v1 <= std_logic_vector(u1);
    v2 <= std_logic_vector(to_unsigned(i1, v2'length));

    -- "4" could be used instead of "v2'length", but the "length"
    -- attribute are better if you want to change the size of v2

    F <= NOT (A AND B AND C);
end my_arch;
```

Listing 3.4: Typical inclusions of IEEE standard libraries

statement becomes even more important as the circuits you are describing become more complex.

There can be any number of equivalent architectures describing a single

entity. As you will eventually discover, the VHDL coding style used inside the architecture body has a significant effect on the way the circuit is synthesized (how the circuit will be implemented inside an actual hardware). This gives the VHDL programmer the flexibility of designing systems with specific features such as particular physical size (measuring the number of needed basic digital elements) or operational speed.

For various reasons, such as facilitating code re-usability and connectibility, an architecture can be modeled in different ways. Understanding the various modeling techniques and understanding how to use them represent the first important steps in learning VHDL.

An architecture can be written by means of three modeling techniques plus any combination of these three. There is the **data-flow model**, the **behavioral model**, the **structural model** and the **hybrid models**. These models will be described throughout the book. Listing 3.5 gives a sneak preview of what a simple but complete VHDL code block looks like.

3.4 Signal and Variable Assignments

In VHDL there are several object types. Among the most frequently used we will mention the **signal** object type, the **variable** object type and the **constant** object type. The signal type is the software representation of a wire. The variable type, like in C or Java, is used to store local information. The constant is like a variable object type, the value of which cannot be changed. A signal object can be of different types; we saw before, for example, that a signal object can be of type `std_logic` or of other types like integer, custom types, etc. The same applies for variable objects.

Before using any signal or variable, it is mandatory to declare them. Signals are declared at the top of the architecture body, just before the keyword `begin`. Variables must be declared inside the *process construct* and can only be used within the scope of this process. Both examples are shown in line 14 and 17 of Listing 3.5.

As seen in line 19 and line 20 of Listing 3.5 when you want to assign a new value to an object of type signal you use the operator "<=". Alternatively, when you want to assign a new value to an object of type variable you will use the operator ":=", shown in line 21. Note that this book will be

```
1  ---------------------- FILE: my_sys.vhd ----------------------
2  -- library declaration
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  -- the ENTITY
7  entity circuit1 is
8      port (A, B, C : in  std_logic;
9            F, G    : out std_logic);
10 end circuit1;
11
12 -- the ARCHITECTURE
13 architecture circuit1_arc of circuit1 is
14     signal sig_1 : std_logic; -- signal definition
15 begin
16     process (a,b,c)
17         variable var_1 : integer; -- variable definition
18     begin
19         F <= not (A and B and C); -- signal assignment
20         sig_1 <= A;               -- another signal assignment
21         var_1 := 34;              -- variable assignment
22     end process;
23
24     G <= not (A and B);           -- concurrent assignment
25 end circuit1_arc;
```

Listing 3.5: Example of a simple VHDL block

consistent with the jargon used in the standards and related literature, hence the use of the words *assignment* and *execution*. Despite this, keep in mind that signals work as *"named wires"*, and the code will define how are they connected or have their state defined.

It is important to understand the difference between variables and signals, specifically when their value changes. A variable changes its value soon after the variable assignment is executed. Instead, a signal changes its value "some time" after the signal assignment expression is evaluated. This has important consequences for the updated values of variables and signals. This means that you should never assume that a signal assignment

can happen instantly and it also means that you can take advantage of variables every time you need to implement a counter or to store values when inside a process.

In order to be able to introduce the use of a variable we had to employ the *process construct*, a construct that you are not yet familiar with. We will see more in details later on in the book that any time we need a non-concurrent execution environment where code lines are executed one after the other (like in C or Java), we will be using the *process construct*. Inside a process, all instructions are executed consecutively from top to bottom. However the process itself will be executed concurrently with the rest of the code (e.g. the instruction at line 24).

Always remember that the assignment of Listing 3.5 at line 24 and the execution of the process, are not executed consecutively but instead concurrently (all at the same time). Any hope that the execution of line 24 will happen before or after the execution of the process will only result in great disappointment.

As a final note, let us to remind that the type `std_logic` only exists if you declare the library `ieee.std_logic_1164.all` as done in line 4 of Listing 3.5.

3.5 Summary

- The entity declaration describes the inputs and outputs of your circuit. This set of signals is often referred to as the interface to your circuit since these signals are what the circuitry, external to the entity, uses to interact with your 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* (or, as we will see later on, even an *inout*) 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 compared to scalar types such as `std_logic`.

- The architecture describes what your circuit actually does and what its behavior is. Several possible implementations (models) of the same behavior are possible in VHDL. These are the **data-flow** model, the **behavioral** model, the **structural** model as well as any combination of them, generally called **hybrid** model.

3.6 Exercises

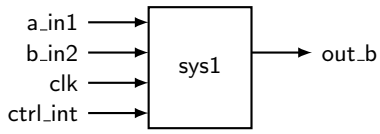
1. What is referred to by the word bundle?

2. What is a common method of representing bundles in black-box diagrams?

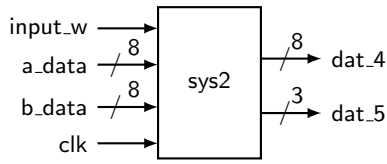
3. Why is it considered a good approach to always draw a black-box diagram when using VHDL to model digital circuits?

4. Write VHDL entity declarations that describe the following black-box diagrams:

   a)



   b)



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

   a)

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

b)

```
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_logic_vector(9 downto 0);
    RAM_DATA_OUT             : out std_logic_vector(7 downto 0));
end ckt_e;
```

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

a)

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

b)

```
entity ckt_b 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_b;
```

4

VHDL Concurrent Programming Paradigm

The previous chapter 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 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 for a moment and remember what it is we are trying to do with VHDL. We are, for one reason or another, *describing a digital circuit*. Realizing this is very important. The tendency for young VHDL programmers with computer programming backgrounds is to view VHDL as just another programming language they want or have to learn. Although many university students have used this approach to pass the basic digital classes, this is not a good idea.

When viewed correctly, VHDL represents a completely different approach to programming while still having 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. Most programming languages are used to implement functionalities in a sequential manner, one instruction at a time. VHDL however describes hardware and so instructions may be "executed" in a concurrent or in a sequential manner, meaning that several

instructions may be "executed" at once. Realizing this fact will help you to truly understand the VHDL programming paradigm and language.

4.1 Concurrent Statements

At the heart of most programming languages are the statements that form a majority of the associated source code. These statements represent finite quantities of actions to be taken. A statement in an algorithmic programming language such as C or Java represents an action to be taken by the processor. Once the processor finishes one action, it moves onto the next action specified somewhere in the associated source code. This makes sense and is comfortable to us as humans because just like the processor, we are generally only capable of doing one thing at a time. This description lays the foundation for an algorithmic method where the processor does a great job of 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 and in a concurrent manner (in other words, in parallel). Once again, the key thing to remember here is that we are designing hardware. Parallelism, or things happening concurrently, in the context of hardware is a much more straightforward concept than it is in the world of software. If you have had any introduction to basic digital hardware, you are most likely already both familiar and comfortable with the concept of parallelism, albeit not within a programming language.

Since most of us are human, we are only capable of reading one line of text at a time and in a sequential manner. We have the same limitation when we try to write some text, not to mention enter some text into a computer. So how then are we going to use text to describe a circuit that is inherently parallel? We did not 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 focus on at each given time.

Everything more or less follows up in a sequential manner, which fits nicely with our basic limitation as humans.

The VHDL programming paradigm is 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 4.1 shows a simple example of a circuit that operates in parallel. As you know, the output of the gates are a function of the gate inputs. Any time that any gate input changes, there is a possibility that, after an opportune delay, the gate output will change. This is true of all the gates in Figure 4.1 or in any digital circuit in general. Once changes to the gate inputs occur, the circuit status is re-evaluated and the gate outputs may change accordingly. Although the circuit in Figure 4.1 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.

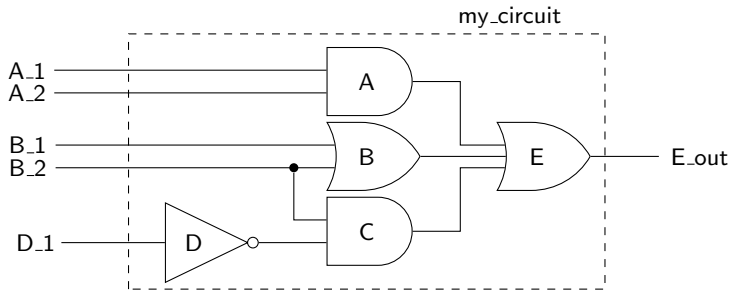


Figure 4.1: Some common circuit that is well known to `execute` parallel operations.

Listing 4.1 shows the code that implements the circuit shown in Figure 4.1. This code shows four concurrent signal assignment statements. As seen before, the "<=" construct refers to the signal assignment operator. It is true that we cannot write these four statements at the same time but we can interpret these statements as actions that occur concurrently. Remember to keep in mind that the concept of concurrency is a key concept in VHDL.

If you feel that the algorithmic style of thought is creeping into your soul, try to snap out of it quickly. If a different thought process is needed, think that these concurrent assignments are instructing the synthesis tool on how to connect components using "named wires" (see line 10). Once connected, the components work inherently in parallel. The concurrent signal assignment is discussed in greater detail in the next section.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity my_circuit is
5  port ( A_1,A_2,B_1,B_2,D_1 : in  std_logic;
6         E_out               : out std_logic);
7  end my_circuit;
8
9  architecture my_circuit_arc of my_circuit is
10     signal A_out, B_out, C_out : std_logic;
11 begin
12     A_out <= A_1 and A_2;
13     B_out <= B_1 or B_2;
14     C_out <= (not D_1) and B_2;
15     E_out <= A_out or B_out or C_out;
16 end my_circuit_arc;
```

Listing 4.1: VHDL code for the circuit of Figure 4.1

As a consequence of the concurrent nature of VHDL statements, the three chunks of code appearing in Listings 4.2 to 4.4 are 100% equivalent to the code shown in Listing 4.1. Once again, since the statements are interpreted as occurring concurrently: the order that these statements appear in your VHDL source code makes no difference. Generally speaking, it would be a better idea to describe the circuit as shown in Listing 4.1 since it somewhat reflects the natural organization of statements (from top to bottom and from the left to the right).

```
C_out <= (not D_1) and B_2;
A_out <= A_1 and A_2;
B_out <= B_1 or B_2;
E_out <= A_out or B_out or C_out;
```

Listing 4.2: Equivalent VHDL code for the circuit of Figure 4.1

```
A_out <= A_1 and A_2;
E_out <= A_out or B_out or C_out;
B_out <= B_1 or B_2;
C_out <= (not D_1) and B_2;
```

Listing 4.3: Equivalent VHDL code for the circuit of Figure 4.1

```
B_out <= B_1 or B_2;
A_out <= A_1 and A_2;
E_out <= A_out or B_out or C_out;
C_out <= (not D_1) and B_2;
```

Listing 4.4: Equivalent VHDL code for the circuit of Figure 4.1

4.2 Signal Assignment Operator "<="

Algorithmic programming languages always have some type of assignment operator. In C or Java, this is the well-known "=" sign. In these languages, the assignment operator signifies a transfer of data from the right-hand side of the operator to the left-hand 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-hand side of the signal assignment operator is dependent upon the signals on the right-hand side of the operator.

With these new insights into VHDL, you should be able to understand the

code of Listing 4.1 and its relationship to its schematic shown in Figure 4.1. The statement "`G <= A AND B;`" indicates that the value of the signal named `G` represents an AND logic operation between the signals `A` and `B`.

There are **four** types of concurrent statements that are examined in this chapter. We have already briefly discussed the **concurrent signal assignment** statement which we will soon examine further and put it into the context of an actual circuit. The three other types of concurrent statements that are of immediate interest to us are: the **process statement**, the **conditional signal assignment** and the **selected signal assignment**.

In essence, the four types of statements represent the tools that you will use to implement digital circuits in VHDL. You will soon be discovering the versatility of these statements. Unfortunately, this versatility effectively adds a fair amount of steepness to the learning curve. As you know from your experience in other programming languages, there are always multiple ways to do the same things. Stated differently, several seemingly different pieces of code can actually produce the same result. The same is true for VHDL code: several considerably different pieces of VHDL code can 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 in this tutorial are presented to show that something can be done a certain way, but that does not necessarily mean they can only be done in that way.

4.3 Concurrent Signal Assignment Statements

The general form of a concurrent signal assignment statement is shown in Listing 4.5. In this case, the target is a signal that receives the values of the expression. An expression is defined by a constant, by a signal, or by a set of operators that operate on other signals. Examples of expressions used in VHDL code are shown in the examples that follow.

EXAMPLE 1. Write the VHDL code that implements a three-input NAND gate. The three input signals are named A, B and C and the

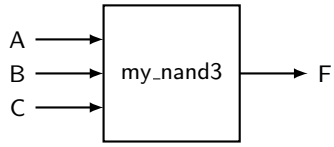
```
<target> <= <expression>;
```

Listing 4.5: Syntax for the concurrent signal assignment statement

output signal name is F.

SOLUTION. It is good practice to always draw a diagram of the circuit you are designing. Furthermore, although we could draw a diagram showing the familiar symbol for the NAND gate, we will 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. The solution to Example 1 is provided in Listing 4.6.

```
library IEEE;
use IEEE.std_logic_1164.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;
```



Listing 4.6: Solution of Example 1

This example contains new ideas that require further clarification.

- 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 Listing 4.6. The listed lines contain more than what is needed for this example but they will be required in later examples. **Note:** to save space, these lines will be omitted in some of the coming 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 not a logic operator but is also available. Moreover, these logic operators are considered to be binary operators in that they operate on the two values appearing on the left and right-hand side of the operator. The NOT operator is a unary operator and for that, it only operates on the value appearing to the right of the operator.
- In this solution, the entity only has one associated architecture. This is fairly common practice in most VHDL design. Conversely, different designs with the same interface could be implemented with focus on different optimizations: chip resource usage, throughput, latency etc.

Example 1 demonstrates the use of the concurrent signal assignment (CSA) statement in a working VHDL program (refer to line 11 of Listing 4.6). 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 any time one of the input signals changes. In other words, the output F is re-evaluated any time a signal on the input expression changes. This is a key concept in truly understanding the VHDL, so you may want to *read the previous sentence a few more times.* The idea of concurrency is more clearly demonstrated in Example 2.

EXAMPLE 2. Write the VHDL code to implement the function expressed by the following logic equation: $F3 = \overline{L} \cdot \overline{M} \cdot N + L \cdot M$

SOLUTION. The VHDL code in Listing 4.7 shows a one-line implementation (line 11) of the given logic equation.

An alternative solution to Example 2 is provided in Listing 4.8. This example represents an important concept in VHDL. The solution shown in Listing 4.8 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 equivalent to declaring extra variables in an algorithmic programming language to be used for storing intermediate results. The need for intermediate results is accompanied

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity my_ckt_f3 is
5  port ( L,M,N : in  std_logic;
6         F3    : out std_logic);
7  end my_ckt_f3;
8
9  architecture f3_2 of my_ckt_f3 is
10 begin
11     F3 <= ((NOT L) AND (NOT M) AND N) OR (L AND M);
12 end f3_2;
```

Listing 4.7: Solution of Example 2

by the declaration of extra signal values, which are often referred to as intermediate signals. Note in Listing 4.8 that the declaration of intermediate signals is similar to the port declarations appearing in the entity declaration, except that the mode specification (`in`, `out` or `inout`) is not used.

```
1  -- identical header and entity omitted
2  architecture f3_1 of my_ckt_f3 is
3      signal A1, A2 : std_logic; -- intermediate signals
4  begin
5      A1 <= ((NOT L) AND (NOT M) AND N);
6      A2 <= L AND M;
7      F3 <= A1 OR A2;
8  end f3_1;
```

Listing 4.8: Alternative solution of Example 2

The intermediate signals must be declared within the body of the architecture because they have no link to the outside world and thus do not appear in the entity declaration. Note that the intermediate signals are declared in the architecture body but appear before the `begin` statement.

Despite the fact that the architectures `f3_2` and `f3_1` of Listing 4.7 and

Listing 4.8 appear different, they are functionally equivalent. This is because all the statements are concurrent signal assignment statements. Even though 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 signals 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 Listing 4.8 due to the fact that the example was modeling a relatively simple circuit. As circuits become more complex, there are many occasions in which intermediate signals must be used. Secondly, intermediate signals are something of a tool that you will often need to use in your VHDL models. The idea here is that you are trying to *describe a digital circuit using a textual description language*: you will often need to use intermediate signals in order to accomplish your goal of modeling the circuit. The use of intermediate signals allows you to more easily model digital circuits 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, the circuit you are 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 way possible. Simple circuits have a higher probability of being understood and synthesized efficiently. But most importantly, a simple VHDL model is not related to the length of the actual VHDL code.

In Example 2, the conversion of the logic function to CSAs was relatively straightforward. The ease with which these functions can be implemented into VHDL code was almost trivial (the function was not complicated at all). As functions become more complex (more inputs and outputs), an equation entry approach becomes tedious and error prone. Luckily, there are other types of concurrent construct that can ease its implementation.

4.4 Conditional Signal Assignment when

Concurrent signal assignment statements, seen before, associate 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 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 Listing 4.9. 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.

```
1 <target> <= <expression> when <condition> else
2             <expression> when <condition> else
3             <expression>;
```

Listing 4.9: 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 us simply redo Example 2 using conditional signal assignment instead of concurrent signal assignment.

> **EXAMPLE 3.** Write the VHDL code to implement the function expressed in Example 2. Use only conditional signal assignment statements in your VHDL code.

SOLUTION. The entity declaration does not change from Example 2 so the solution only needs a new architecture description. By reconsidering the same logic equation of Example 2, $F3 = \overline{L} \cdot \overline{M} \cdot N + L \cdot M$, the solution to Example 3 is shown in Listing 4.10.

```
1 architecture f3_3 of my_ckt_f3 is
2 begin
3     F3 <= '1' when (L = '0' AND M = '0' AND N = '1') else
4           '1' when (L = '1' AND M = '1') else
5           '0';
6 end f3_3;
```

Listing 4.10: Solution of Example 3

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

- It is 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 code size.
- If you look carefully at this code you will 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 the "is not equal to" operators, respectively. Operators are discussed at greater length in further sections.

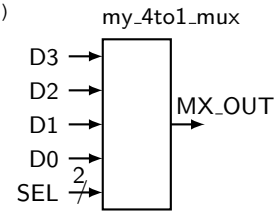
Note that this statement describes a fully combinational circuit with one output and 3 inputs, which can be fully specified by a table with 8 lines (one for each possible input combination). This means that this conditional statement, due to the specified logic, allowed a very concise (but complete) description of the table, with only 3 lines of code.

There are other interesting uses of the conditional signal assignment statement. One of the classic uses is for the implementation of a multiplexer (MUX), as detailed int the next example.

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 Listing 4.11.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity my_4t1_mux is
5      port(D3, D2, D1, D0 : in std_logic;
6           SEL : in std_logic_vector(1 downto 0)
7           MX_OUT : out std_logic);
8  end my_4t1_mux;
9
10 architecture mux4t1 of my_4t1_mux is
11 begin
12     MX_OUT <= D3 when (SEL = "11") else
13               D2 when (SEL = "10") else
14               D1 when (SEL = "01") else
15               D0 when (SEL = "00") else
16               '0';
17 end mux4t1;
```



Listing 4.11: Solution of Example 4

Here are some important notes about the VHDL code in Listing 4.11:

- The solution looks somewhat efficient compared to the amount of logic that would have been required if CSA statements were used. The VHDL code looks good and is pleasing to the eye, qualities required for readability.

- The "=" relational operator is used in conjunction with a bundle. In this case, the values on the bundle SEL are accessed using double quotes around the specified values. In other words, single quotes are used to describe values of individual signals while double quotes are used to describe values associated with multiple signals, or bundles.
- For the sake of completeness, we have 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 but would not be 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 in the conditional signals (the signals listed in the expressions on the right-hand 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-hand side of the signal assignment operator.

Though it is still early in the VHDL learning game, you have 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 however an alternative way to make sense of it. If you think about it, the conditional signal assignment statement is similar in function to the `if-else` constructs in algorithmic programming languages. We will touch more upon this relationship once we start talking about sequential statements.

Finally, this structure provided a very concise but complete description of a combinational circuit. This multiplexer has a total of 6 inputs, which means that it would require a table with 64 lines to be fully described. Despite this, not only the VHDL code provided a very short description of the logic (in only 4 lines of code) but it also allowed this description to be much more similar to what a functional specification of a MUX really is.

The concept of working with bundles is very important in VHDL. Generally speaking, if you can use a bundle as opposed to individual signals, you should. You will often need to access individual signals within a bundle. When this is the case, a special syntax is used (e.g. `SEL(1)`). Note that the code shown in Listing 4.12 is equivalent to but not as clear as the code shown in Listing 4.11 (check the similarities and differences).

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity my_4t1_mux is
5      port (D3, D2, D1, D0 : in  std_logic;
6            SEL            : in  std_logic_vector(1 downto 0);
7            MX_OUT         : out std_logic);
8  end my_4t1_mux;
9
10 architecture mux4t1 of my_4t1_mux is
11 begin
12     MX_OUT <= D3 when (SEL(1) = '1' and SEL(0) ='1') else
13               D2 when (SEL(1) = '1' and SEL(0) ='0') else
14               D1 when (SEL(1) = '0' and SEL(0) ='1') else
15               D0 when (SEL(1) = '0' and SEL(0) ='0') else
16              '0';
17 end mux4t1;
```

Listing 4.12: Alternative solution to Example 4 accessing individual signals

4.5 Selected Signal Assignment `with select`

Selected signal assignment statements are the third type of signal assignment that we will examine. As with conditional signal assignment statements, selected signal assignment statements only have one assignment operator (<=). They differ from conditional assignment statements in that assignments are based upon the evaluation of *only one expression* (preceeding the **select** keyword). The syntax for the selected signal assignment statement is shown in Listing 4.13.

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

Listing 4.13: Syntax for the selected signal assignment statement

> **EXAMPLE 5.** 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. The solution of Example 5 is shown in Listing 4.14. The black-box diagram is the same as before and is not repeated here.

```
library IEEE;
use IEEE.std_logic_1164.all;

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;
```

Listing 4.14: Solution of Example 5

One thing to notice about the solution shown in Listing 4.14 is the use of the `when others` clause as the final entry in the selected signal assignment statement. In reality, the last clause `D0 when "00"` could be removed from the solution without changing the meaning of the statement if the last line would be changed to `D0 when others;`. Despite this, it is considered good VHDL programming practice to include all the expected cases in the selected signal assignment statement, without relying on the `when others` clause.

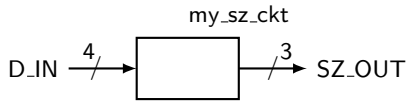
Once again, there are a few things of interest in the solution for Example 5 which are listed below.

- The VHDL code has several similarities to the solutions of Example 4, but this solution is even more pleasing to the eye than the ones where the MUX was modeled referring to the `SEL` inputs repeatedly.
- A `when others` clause is used again. In the case of Example 5, 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. However, this is not a requirement. The only requirement here is that the line containing the `when others` keywords appears in the final line of the statement.

EXAMPLE 6. Write the VHDL code that implements the following circuit. The circuit contains an input bundle of four signals and an output bundle of three signals. The input bundle, D_IN, represents a 4-bit binary number. The output bundle, 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.

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

SOLUTION. This is an example of a generic decoder-type circuit. The solution to Example 6 is shown in Listing 4.15.




```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity my_sz_ckt is
5      port ( D_IN   : in  std_logic_vector(3 downto 0);
6             SZ_OUT : out std_logic_vector(2 downto 0));
7  end my_sz_ckt;
8
9  architecture spec_dec of my_sz_ckt is
10 begin
11     with D_IN select
12         SZ_OUT <= "100" when "0000"|"0001"|"0010"|"0011",
13                   "010" when "0100"|"0101"|"0110"|"0111"|"1000"|"1001",
14                   "001" when "1010"|"1011"|"1100"|"1101"|"1110"|"1111",
15                   "000" when others;
16 end spec_dec;
```

Listing 4.15: Solution of Example 6

The only comment for the solution of Example 6 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 do 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 selected signal assignment statement. Re-evaluation also occurs every time there is a change in a conditional signal on the right-hand side of the signal assignment operator.

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

EXAMPLE 7. Write VHDL code to implement the function expressed by the following logic equation: $F3 = \overline{L} \cdot \overline{M} \cdot N + L \cdot M$.

SOLUTION. This is the same problem examined before. The problem with the previous solutions to this example is that they required the user to somehow reduce the function before it was implemented. In this modern day of 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 having to use a *Karnaugh map*, or God forbid, *boolean algebra*, to reduce a function. The equivalent expression for $F3(L, M, N) = \overline{L} \cdot \overline{M} \cdot N + L \cdot M$ and its Karnaugh map is shown below. The solution of Example 7 is shown in Listing 4.16.

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

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

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity my_ckt_f3 is
5      port ( L, M, N : in  std_logic;
6             F3      : out std_logic);
7  end my_ckt_f3;
8
9  architecture f3_7 of my_ckt_f3 is
10     signal t_sig : std_logic_vector(2 downto 0); -- local bundle
11 begin
12     t_sig <= (L & M & N); -- concatenation operator
13
14     with (t_sig) select
15         F3 <= '1' when "001" | "110" | "111",
16               '0' when others;
17 end f3_7;
```

Listing 4.16: Solution of Example 7

4.6 Testbenches

Hardware description languages had their origin related to the need to test and integrate modules, as important parts of the development process. For this reason, it is expected that the languages included features to specify test routines for the hardware being described. Starting here with the VHDL features already presented, we will provide at the end of each chapter an example of using the same language to describe **test procedures** which are named **testbenches**.

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 is highlighted by the many different approaches that appear in VHDL books and by the many tutorials. This book approaches the learning process in a very informal way, and many concepts are introduced briefly so they don't get in the way of the main topic being presented. This is exactly the case here, where we need two topics which will be covered in detail in later chapters.

To understand the logic of our first testbench, we need only the following short descriptions for two important VHDL features:

component is the counterpart of the **entity** keyword, and it specifies the interface of a module, so it can be "connected" to another one.

process is a block in which the statements are evaluated sequentially, line by line, as they would be in procedural programming languages.

Our first module to be tested is a simple OR gate, which could be described as shown in Listing 4.17. We call this a DUT (Device/Design Under Test), which is the hardware description we need to verify.

```
1  -- library omitted
2  entity or_gate is
3  port(a, b: in std_logic;
4       q: out std_logic);
5  end or_gate;
6
7  architecture rtl of or_gate is
8  begin
9      q <= a or b;
10 end rtl;
```

Listing 4.17: VHDL description of an OR gate

If we connected this module to an Oscilloscope or any Logic Analyzer equipment, a waveform similar to the one presented in Figure 4.2, showing the inputs and the output, would be obtained.

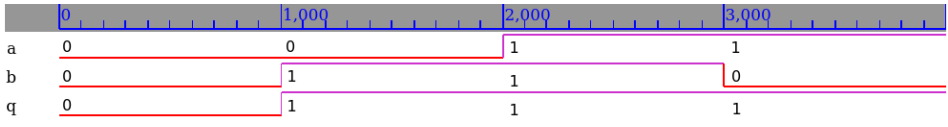


Figure 4.2: Waveform for the OR gate test generated by the EPWave tool.

Since this is a common interface for engineers, most VHDL simulation tools provide a similar graphical output. The only missing piece is the generation of the test inputs **a** and **b**, respectively connected to the signals **a_in** and **b_in**, provided by the testbench presented in Listing 4.18. Note that the same module interface defined by the **entity** keyword in the module is repeated within the architecture, with the **component** keyword.

```
1  -- library omitted
2  entity testbench is -- the testbench has no interfaces
3  end testbench;
4
5  architecture tb of testbench is
6  component or_gate is -- the same interface of the entity
7      port(a, b: in std_logic;
8           q: out std_logic);
9  end component;
10     -- these are the local signals conected to the DUT
11     signal a_in, b_in, q_out: std_logic;
12 begin
13
14     DUT: or_gate port map(a_in, b_in, q_out); -- DUT
15     process
16     begin
17         a_in <= '0';
18         b_in <= '0';
19         wait for 1 ns;
20         b_in <= '1'; -- b is 1, a remains '0'
21         wait for 1 ns;
22         a_in <= '1'; -- a is 1, b remains '1'
23         wait for 1 ns;
24         b_in <= '0'; -- b is 0, a remains '1'
25         wait for 1 ns;
26         report "Test done." severity note;
27         wait; -- STOP!
28     end process;
29 end tb;
```

Listing 4.18: Testbench for the OR gate design

A VHDL simulation tool analyzes the two codes (DUT and testbench) and evaluates the outputs provided by the design for each input stimulus. Additionally to the comments in the code, note that the keyword **wait**, evaluated sequentially, provides a simulated time for the DUT to react to the new signals at the inputs and update the correspondent output. The test signals defined at line 11 are "connected" to the DUT at line 14.

As the designs become more complex the testbenches will be adapted to cover more test procedures. The following chapters will gradually introduce other features to this very useful development resource.

4.7 Summary

- The entity/architecture pair is the interface description and behavior description of how a digital circuit operates.
- VHDL explores 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 a major difference between VHDL and higher-level computer programming languages.
- The major signal assignment types in VHDL are: concurrent, conditional and selected signal assignments. Concurrent statements are interpreted as acting in parallel (concurrently) to other concurrent statements.
- The architecture body can contain any or all of the mentioned concurrent statements.
- The outputs in the entity declaration cannot appear on the right-hand side of a signal assignment operator in *earlier versions* of VHDL. This characteristic is prevented from being a problem by the declaration and use of intermediate signals, which are declared **inside** the architecture body just **before** the begin statement.
- Generally speaking, there are multiple approaches in 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 and perform the optimizations.

4.8 Exercises

1. For the following function descriptions, write VHDL models that implement these functions using concurrent signal assignment.
   a) $F(A, B) = \overline{A}B + A + A\overline{B}$
   b) $F(A, B, C, D) = \overline{A}C\overline{D} + \overline{B}C + BC\overline{D}$
   c) $F(A, B, C, D) = (\overline{A} + B) \cdot (\overline{B} + C + \overline{D}) \cdot (\overline{A} + D)$
   d) $F(A, B, C, D) = \prod(3, 2)$
   e) $F(A, B, C) = \prod(5, 1, 4, 3)$
   f) $F(A, B, C, D) = \sum(1, 2)$
2. For the following function descriptions, write VHDL models that implement these functions using both conditional and selected signal assignment.
   a) $F(A, B, C, D) = \overline{A}C\overline{D} + \overline{B}C + BC\overline{D}$
   b) $F(A, B, C, D) = (\overline{A} + B) \cdot (\overline{B} + C + \overline{D}) \cdot (\overline{A} + D)$
   c) $F(A, B, C, D) = \prod(3, 2)$
   d) $F(A, B, C, D) = \sum(1, 2)$
3. Provide a VHDL model of an 8-input AND gate using concurrent, conditional and selected signal assignment.
4. Provide a VHDL model of an 8-input OR gate using concurrent, conditional and selected signal assignment.
5. Provide a VHDL model of an 8:1 MUX using conditional signal assignment and selected signal assignment.
6. Provide a VHDL model of a 3:8 decoder using conditional signal assignment and selected signal assignment; consider the decoder's outputs to be active-high.
7. Provide a VHDL model of a 3:8 decoder using conditional signal assignment and selected signal assignment; consider the decoder's outputs to be active-low.

5

Standard Models in VHDL Architectures

As you may remember, the VHDL architecture describes how your VHDL system will behave. The architecture body contains two parts: the declaration section and the `begin-end` section where a collection of (concurrent) signal assignments appear. In the previous Chapter, we have studied three types of signal assignment: **concurrent**, **conditional** and **selected** signal assignment. Now, before we describe another concurrent statement (the process statement), let us quickly introduce a new important topic.

There are three different approaches to writing VHDL architectures. These approaches are known as **data-flow** style, **behavioral** style and **structural** 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 is 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 data-flow 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 been implemented using the data-flow style. We are now at the point of talking about the behavioral style of architectures which is primarily centered around another concurrent statement known as the process statement. If it seems confusing, some of the confusion should go away once we start dealing with actual circuits and real VHDL code.

5.1 Data-flow Style Architecture

A data-flow style architecture specifies a circuit as a concurrent representation of the flow of data through the circuit. In the data-flow 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 have talked about up until now (concurrent, conditional and selected signal assignment) are all statements that are found in data-flow style architectures. In other words, *if you exclusively used concurrent, conditional and selected signal assignment statement in your VHDL models, you have used a data-flow model*. If you were to re-examine some of the examples we have done so far, you can in fact see how the data flows through the circuit. To put this in other words, if you have a working knowledge of digital logic, it is fairly straightforward 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-hand side of the assignment operator (the "<=") to the signal on the left-hand side of the operator.

The data-flow 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 data-flow models also allow you to make an intelligent guess as to how the actual logic will appear when you decide to synthesize the circuit. Data-flow 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.

5.2 Behavioral Style Architecture

In comparison to the data-flow style architecture, the behavioral style architecture provides no details as to how the design could be implemented in actual hardware. Instead, the behavioral style models how the circuit outputs will *react* to the circuit inputs. Whereas in data-flow modeling you somewhat need 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 leave the implementation details up to the synthesis tool. In other words, data-flow modeling describes how the circuit should look in terms of logic gates whereas behavioral modeling describes how the circuit should behave. For these reasons, behavioral modeling is considered higher up on the circuit abstraction level as compared to data-flow 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 the fourth type of concurrent statement that we will 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 when you deal with this new concurrent statement.

5.3 Process Statement

The process statement itself is a concurrent statement identified by its label, its sensitivity list, a declaration area and a begin-end area containing instructions executed sequentially. An example of the process statement is shown in Listing 5.1.

The main point to remember about the process statement is that its body is constituted of *sequential statements*. The main difference between concurrent signal assignment statements and process statements lies with these sequential statements. But once again, let us stick to the similarities before we dive into the differences. The process label (`my_label`), listed in Listing 5.1 is optional but should always be included to promote the self-description of your VHDL code.

```
-- this is my first process
my_label: process(sensitivity_list) is
    <item_declaration>
begin
    <sequential_statements>
end process my_label;
```

Listing 5.1: Syntax for the process statement

Listing 5.2 shows a data-flow architecture and a behavioral style architecture for the same XOR entity. The main difference between the two architectures is the presence of the process statement in the listed code.

```
library IEEE;
use IEEE.std_logic_1164.all;

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

architecture df of my_xor is
begin
    F <= A XOR B;
end df;
--
--
--
```

```
library IEEE;
use IEEE.std_logic_1164.all;

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

architecture bh of my_xor is
begin
  xor_proc: process(A, B) is
  begin
    F <= A XOR B;
  end process xor_proc;
end bh;
```

Listing 5.2: Data-flow (left) vs Behavioral (right) architectures

Let us remember that the concurrent signal assignment statement in the data-flow description operates as follows: since it is a concurrent statement, every time there is a change in any of the signals listed on the right-hand side of the signal assignment operator, the signal on the left-hand side of the operator is re-evaluated.

A later example with multiple lines will explain the sequential nature of the process declaration. But, since this behavioral example has only a single line with an assignment, let's focus on the important difference here: the sensitivity list `(A, B)`. Despite being a feature relevant only for pre-2008 VHDL and which affects only simulation (and not the hardware synthesis), this book will use it for two main reasons: it helps understanding how the process block is triggered and it is still quite possible that you will encounter VHDL code written with older standards (for older tools).

For the behavioral architecture description, when **simulating** the design, every time there is a change in signals in the process sensitivity list, all of the sequential statements in the process are re-evaluated. Evaluation of the process statement is controlled by the signals that are placed in the process sensitivity list. The exact same result described for the data-flow will be obtained **in synthesis only**: the tool evaluates the signals listed on the right-hand side.

If the inconsistent way the sensitivity list is used in simulation and synthesis looks strange, don't feel bad. This has historical roots on why HDLs were created decades ago and has been changed since to a better and less error prone alternative. For this reason, when writing your own code from scratch just use the "new" feature of automatic sensitivity list `process(all)` and be relieved that no simulation/synthesis mismatch will happen in your design due to errors in the sensitivity list.

5.4 Sequential Statements

The process statement should be considered a way the programmer has at his disposal to execute a series of sequential statements (i.e. in a behavioral manner); never forget that the process statement is itself a concurrent statement; therefore when you place two processes inside the architecture body, their execution will be concurrent. As opposed to the sequential nature inside each process statement, take a quick look back at the Listings 4.2 to 4.4 to remember how the order of assignments is irrelevant outside the process block.

In Listing 5.3, you can see what a complete process statement looks like. Remember that all variables defined inside the process body will only

be visible within the process body itself. Furthermore, notice that the statement at line 23 is placed inside the architecture body but outside the process body; therefore its execution happens concurrently with the process statement.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity my_system is
5      port ( A, B, C : in  std_logic;
6             F, Q    : out std_logic);
7  end my_system;
8
9  architecture behav of my_system is
10     signal A1 : std_logic;
11 begin
12     some_proc: process(A, B, C) is
13         variable x, y : integer;
14     begin
15         x := 74;
16         y := 67;
17         A1 <= A and B and C;
18         if x > y then
19             F <= A1 or B;
20         end if;
21     end process some_proc;
22     -- we are outside the process body
23     Q <= not A;
24 end behav;
```

Listing 5.3: Use of the process statement

Within a process, the execution of the sequential statements 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 made of sequential statements. This is actually a tough concept to

grasp. After years of contemplation, I am only starting to grasp the reality of this strange contradiction.

The key to understand 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 is to keep 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 intelligible (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 statements: 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 will be discussing. The first one is the assignment, which can be either to a **signal** (<=) or to a **variable** (:=). We have already been dealing with its analogue in the data-flow models so we will discuss what changes in the sequential interpretation. The other two types of statements are the **if statement** and the **case statement**. The nice part about all of these statements is that you have 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.

5.4.1 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 then it is a sequential statement; otherwise, it is a concurrent signal assignment statement. To drive the point home, the signal assignment statement

"$F <= A \ XOR \ B$;" in the Listing 5.2 data-flow style architecture is a concurrent signal assignment statement while the same statement in the behavioral style architecture would be a sequential one, if there were more statements. "Sequential", in this context, means: the last line within a process which assigns to a signal at a certain execution is the one that determines its externally visible value when the process ends.

Remember that a process is triggered by a change in the external signals at the right-hand side of the assignments or in the expressions used in the `if` or `case` statements (the signals or inputs we put on the sensitivity list, as explained earlier). This means that the same "new value" that triggered the process will be used throughout the process execution. Variables, on the other hand, are used only within the scope of a process for temporary storage. This means that variable assignments evaluated sequentially immediately affect the variable content when they are executed.

5.4.2 `if` Statement

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 when the `if` statement is processed. The general form of the `if` statement is shown in Listing 5.4.

```
1 if (condition) then
2     <statements>
3 elsif (condition) then
4     <statements>
5 else
6     <statements>
7 end if;
```

Listing 5.4: Syntax of 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 (with is different syntax). Secondly, the VHDL `if` statement is the sequential equivalent to the VHDL conditional signal assignment statement. These two statements essentially do the same thing but **the `if` statement is a sequential statement found inside a process body**.

Some important notes about the listed syntax of the `if` statement:

- The parentheses placed around the condition expressions are optional. They should be included to increase the readability of the VHDL code.
- Each `if`-type statement contains an associated `then` keyword. The final `else` clause does not have the `then` keyword associated with it.
- The `else` clause is a catch-all statement. If none of the previous conditions is evaluated as true, then the sequence of statements associated with the final `else` clause is executed. This guarantees that one of the listed sequence of statements 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 will discuss later.

Let us see now some examples that will help us to better understand how to use the `if` statement.

EXAMPLE 8. Write some VHDL code using an `if` statement that implements the following logic function: $F\_OUT(A, B, C) = A\overline{B}\,\overline{C} + BC$

SOLUTION. Although it is not directly stated in the problem description, the VHDL code for this solution must use a behavioral architecture (the problem states that an `if` statement should be used). The VHDL code for the solution is shown in Listing 5.5. We have opted to leave out the black-box diagram in this case since the problem is relatively simple and thus does not really demonstrate the power of behavioral modeling.

This is probably not the best way to implement a logic function but it does show an `if` statement in action. An alternate architecture for the solution of Example 8 is shown in Listing 5.6.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

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

architecture silly_example of my_ex 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 silly_example;
```

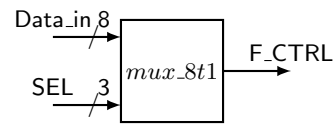
Listing 5.5: Solution to Example 8

```vhdl
architecture bad_example of my_ex_7 is
begin
    proc1: process(A,B,C)
    begin
        if (A='1' 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;
```

Listing 5.6: Alternative solution to Example 8

One final comment on process statements. Process statements can be preceded with an optional label. A label should always be included with process statements as a form of self-description. 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 is easier to provide a meaningful name to a process that is not trying to do too much. A more intelligent use of the `if` statement is demonstrated in the next example.

EXAMPLE 9. Write some VHDL code that implements the 8:1 MUX shown below. Use an `if` statement in your implementation.



SOLUTION. The solution to Example 9 is shown in Listing 5.7.

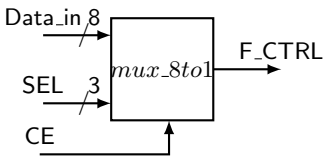
The solution to Example 9 shown in Listing 5.7 uses some new syntax. The entity uses bundle signals, but the associated architecture needs to access individual elements of these bundles. The solution is to use the bus index operator to access internal signals of the bus. This is done via the use of a number representing an index placed inside parentheses (for example `Data_in(7)`). Bus index operators are used extensively in VHDL and were previously mentioned. The solution to Example 9 shows a more typical use of the operator than was previously mentioned.

One other thing to notice about the solution in Example 9 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 Listing 5.7 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.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

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 mux_8t1_arc 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 mux_8t1_arc;
```

Listing 5.7: Solution to Example 9

EXAMPLE 10. Write a VHDL code that implements the 8:1 MUX shown here. Use as many `if` statements as you deem necessary to implement your design. In the black-box diagram, the `CE` input is a chip enable. When `CE = '1'`, the output acts like the MUX of Example 9, but when `CE` is `'0'`, the output of the MUX is `'0'`.



SOLUTION. The solution to Example 10 is similar to the solution of Example 9, since the MUX function is the same. Note that in this solution the `if` statements can be nested to attain various effects. The solution to Example 10 is shown in Listing 5.8.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity mux_8to1_ce is
5      port ( Data_in : in  std_logic_vector (7 downto 0);
6             SEL     : in  std_logic_vector (2 downto 0);
7             CE      : in  std_logic;
8             F_CTRL  : out std_logic);
9  end mux_8to1_ce;
10
11 architecture mux_8to1_ce_arch of mux_8to1_ce is
12 begin
13     my_mux: process (Data_in,SEL,CE)
14     begin
15         if (CE = '0') then
16             F_CTRL <= '0';
17         else
18             if    (SEL = "111") then F_CTRL <= Data_in(7);
19             elsif (SEL = "110") then F_CTRL <= Data_in(6);
20             elsif (SEL = "101") then F_CTRL <= Data_in(5);
21             elsif (SEL = "100") then F_CTRL <= Data_in(4);
22             elsif (SEL = "011") then F_CTRL <= Data_in(3);
23             elsif (SEL = "010") then F_CTRL <= Data_in(2);
24             elsif (SEL = "001") then F_CTRL <= Data_in(1);
25             elsif (SEL = "000") then F_CTRL <= Data_in(0);
26             else  F_CTRL <= '0';
27             end if;
28         end if;
29     end process my_mux;
30 end mux_8to1_ce_arch;
```

Listing 5.8: Solution to Example 10

5.4.3 case Statement

The case statement is somewhat similar to the if statement in that a sequence of statements is executed if an associated expression is true. The case statement differs from the if statement in that the resulting choice is made depending upon the value of the single control expression. Only one set of sequential statements is executed for each execution of the case statement. The statements executed are determined by the first when branch to evaluate as true. The syntax for the case statement is shown in Listing 5.9.

```
1 case (expression) is
2     when choices =>
3         <sequential statements>
4     when choices =>
5         <sequential statements>
6     when others =>
7         <sequential statements>
8 end case;
```

Listing 5.9: Syntax for the *case* statement

Once again, the concept of the case statement should be familiar to you in several regards. Firstly, it can generally be viewed as a 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 the case statement or the switch statement in other algorithmic programming languages. And finally, the VHDL case statement is the sequential equivalent of 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 as good programming practice.

EXAMPLE 11. Write some VHDL code that implements the following function using the `case` statement: $F\_OUT(A, B, C) = A\overline{B}\,\overline{C} + BC$

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 minterms. This is done by multiplying the non-minterm product term given in the example by 1. In this case, 1 is equivalent to $(A + \overline{A})$. This factoring operation is shown as:

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

The solution is shown in Listing 5.10. An interesting feature of this solution is the grouping of the three input signals which allows the use of a `case` statement in the solution. This approach requires the declaration of an intermediate signal which is appropriately labeled "ABC". Once again, this is 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.

Another similar approach to Example 11 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 if you had not taken a more clever approach. A different architecture for the solution of Example 11 is shown in Listing 5.11. One possible drawback of using a don't care feature in your VHDL code is that some synthesizers and some simulators do not handle it very well. I would avoid them at all costs and seek a more definitive method of modeling the circuits I am dealing with.

One of the main points that should be emphasized in any VHDL program is readability. In the next problem, we will redo Example 10 using a `case` statement instead of `if` statements.

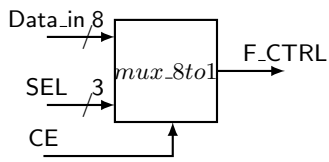
```
entity my_example is -- library declaration omitted
    port (A, B, C : in  std_logic;
          F_OUT   : out std_logic);
end my_example;

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;
```

Listing 5.10: Solution to Example 11

```
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;
```

Listing 5.11: Alternative solution to Example 11

EXAMPLE 12. Write some VHDL code that implements the 8:1 MUX shown. Use a case statement in your design. In the black-box diagram, 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 12 is shown in Listing 5.12. 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 have not noticed it yet, the number of possible solutions to a given problem increases as the circuits you are implementing become more complex.

One very important point in the solution to Example 12 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 data-flow modeling. The reality is that conditional and selective signal assignment statements cannot be nested.

5.5 Caveats Regarding Sequential Statements

As you begin to work with sequential statements, you tend to start getting the feeling that you are 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 this happens very often with people who are learning VHDL, it is appropriate to remind once again that **VHDL is not programming: VHDL is hardware design**. You are, generally speaking, not implementing algorithms in VHDL, you are **describing hardware**: this is a totally different paradigm.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity mux_8to1_ce is
5      port ( Data_in : in  std_logic_vector (7 downto 0);
6                 SEL : in  std_logic_vector (2 downto 0);
7                  CE : in  std_logic;
8             F_CTRL : out std_logic);
9  end mux_8to1_ce;
10
11 architecture my_case_ex of mux_8to1_ce is
12 begin
13     my_mux: process (SEL,Data_in,CE)
14     begin
15         if (CE = '1') then
16             case (SEL) is
17                 when "000"  =>  F_CTRL <= Data_in(0);
18                 when "001"  =>  F_CTRL <= Data_in(1);
19                 when "010"  =>  F_CTRL <= Data_in(2);
20                 when "011"  =>  F_CTRL <= Data_in(3);
21                 when "100"  =>  F_CTRL <= Data_in(4);
22                 when "101"  =>  F_CTRL <= Data_in(5);
23                 when "110"  =>  F_CTRL <= Data_in(6);
24                 when "111"  =>  F_CTRL <= Data_in(7);
25                 when others => F_CTRL <= '0';
26             end case;
27         else
28             F_CTRL <= '0';
29         end if;
30     end process my_mux;
31 end my_case_ex;
```

Listing 5.12: Solution to Example 12

It is not uncommon to see many, not so good, 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 are trusting the VHDL synthesizer to magically know what you are trying to describe. If you do not understand the ins and outs of VHDL at a low level, your circuit is not going to synthesize properly. Most likely you understand simple VHDL behavioral models. But once the models become complex, your understanding quickly fades away. The solution to this problem is really simple: keep your VHDL models simple, particularly your process statements.

In VHDL, the best approach is to keep your process statements centered around a single function and have several process statements that communicate with each other. The bad approach is to have one big process statement that does everything for you. The magic of VHDL is that if you provide simple code to the synthesizer, it is more than likely going to provide you with a circuit that works and with an implementation that is simple and eloquent. If you provide the synthesizer with complicated VHDL code, the final circuit may work and may even be efficient in both speed and real estate, 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.

5.6 Automatic Testbenches

In this chapter we introduce a simple but powerful testbench feature: automatic verification. The code presented in Listings 5.13 and 5.14 is very similar to the one presented in the last chapter, so some details were omitted for brevity. The verification is provided by the **assert** statement, which automatically checks if the output is the expected one, and prints an error message otherwise. As designs become more complex, it will be very difficult and error prone to find logic errors in waveform outputs.

```vhdl
-- library and entity omitted (see testbench component)
architecture rtl of xor_gate is
begin
    q <= a xor b;
end rtl;
```

Listing 5.13: VHDL description of a XOR gate

```vhdl
-- library and empty entity omitted
architecture tb of testbench is
    component xor_gate is
        port(a, b: in std_logic;
             q: out std_logic);
    end component;
    signal a_in, b_in, q_out: std_logic;
begin
    DUT: xor_gate port map(a_in, b_in, q_out);
    process
        begin
        a_in <= '0'; b_in <= '0';
        wait for 1 ns;
        assert(q_out='0') report "Fail 0/0" severity error;
        b_in <= '1'; -- b becomes 1, a remains '0'
        wait for 1 ns;
        assert(q_out='1') report "Fail 0/1" severity error;
        a_in <= '1'; -- a becomes 1, b remains '1'
        wait for 1 ns;
        assert(q_out='0') report "Fail 1/1" severity error;
        b_in <= '0'; -- b becomes 0, a remains '1'
        wait for 1 ns;
        assert(q_out='1') report "Fail 1/0" severity error;
        report "Test done." severity note;
        wait; -- STOP!
    end process;
end tb;
```

Listing 5.14: Automatic testbench for the XOR gate design

5.7 Summary

Let us now review some of the important concepts that have been introduced in this chapter.

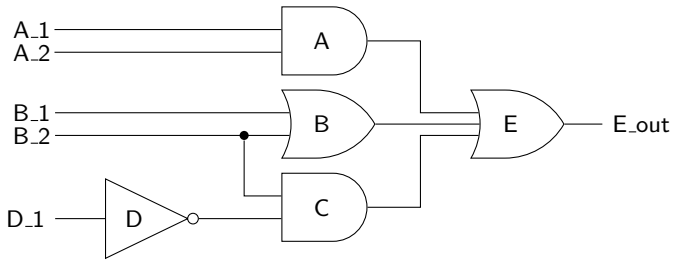
- The three main flavors of VHDL modeling styles include data-flow, behavioral and structural models.
- VHDL behavioral models, by definition, use process statements.
- VHDL data-flow models by definition use concurrent signal assignment, conditional signal assignment and/or selected signal assignment.
- 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 data-flow modeling.
- The `case` statement has a direct analogy to the selected signal assignment statement used in data-flow modeling.
- Both the `case` statement and the `if` statement can be nested. **Concurrent, conditional and selected signal assignment statements cannot be nested**.
- The simplest concurrent statement is the concurrent signal assignment statement (e.g. "*F* <= *A*;"). Its sequential equivalent is the sequential signal assignment statement and it looks identical.

5.8 Exercises: Behavioral Modeling

1. For the following function, write VHDL behavioral models that implement these functions using both `case` statements and `if` statements (two separate models for each function).
   a) $F(A, B) = \overline{A}B + A + A\overline{B}$
   b) $F(A, B, C, D) = \overline{A}C\overline{D} + \overline{B}C + BC\overline{D}$
   c) $F(A, B, C, D) = (\overline{A} + B) \cdot (\overline{B} + C + \overline{D}) \cdot (\overline{A} + D)$
   d) $F(A, B, C) = \prod(5, 1, 4, 3)$

e) $F(A, B, C, D) = \sum(1, 2)$

2. For the circuit below, write the VHDL behavioral model that implements it using both `case` statements and `if` statements (two separate models).



3. Model the previous circuit using concurrent, conditional, or selected signal assignment.

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 model that uses `if` statements and `case` statements (two separate models).

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

6 VHDL Operators

So far we have only implicitly mentioned the operators available in VHDL. 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 6.1. 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, adding, sign, multiplying, and miscellaneous. The ordering of this list is somewhat important because it presents the operators in order of increasing precedence. We said "somewhat" because your VHDL code should never rely on operator precedence to describe circuit behavior.

Operator type						
logical	and	or	nand	nor	xor	xnor
relational	=	/=	<	<=	>	>=
shift	sll	srl	sla	sra	rol	ror
adding	+	−	&			
sign	+	−				
multiplying	*	/	mod	rem		
miscellaneous	**	abs	not			

Table 6.1: VHDL operators.

Reliance on obscure precedence rules tends to make the VHDL code cryptic and hard to understand. A liberal use of parentheses is a better approach to VHDL coding.

The first column of Table 6.1 lists the operators from lowest to highest precedence, where logical operators have the lowest precedence. Although there is a precedence order for the types of operators, there is no 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.

6.1 Logical Operators

The logical operators are generally self-explanatory in nature. They have also been used throughout this book. The only thing worthy of note is that the `not` operator is not technically a logical operator and is grouped among the miscellaneous operators, which have the highest precedence. Two very useful enhancements in logical operators were presented in VHDL 2008: introduction of the very convenient *unary* version of the 6 logical operators, which operate on all bits of their single argument; the two arguments can have different lengths (e.g. all bits of one argument compared to the single bit or the other).

6.2 Relational Operators

The relational operators are generally self-explanatory in nature too. Many of them have been used in this book. A complete list of relational operators is provided in Table 6.2. Note that the equivalence and non-equivalence operators apply to any type but the "less/greater than" types require scalar or array types. Also important is the fact that VHDL is a strongly typed language, meaning that the two arguments being compared must be of the same type (or properly converted by casting).

6.3 Shift Operator

Available from `ieee.numeric_std` or `ieee.numeric_bit`, there are three types of shift operators: logical shift, arithmetic shift, and rotations.

Operator	Name	Explanation
`A = B`	equivalence	is A equivalent to B?
`A /= B`	non-equivalence	is A not equivalent to B?
`A < B`	less than	is A less than B?
`A <= B`	less than or equal	is A less than or equal to B?
`A > B`	greater than	is A greater than B?
`A >= B`	greater than or equal	is A greater than or equal to B?

Table 6.2: VHDL relational operators with brief explanations.

Although these operators basically shift bits either left-to-right or right-to-left, there are a few basic differences which are listed in Table 6.3.

Operator	Name		Example	Result
logical	sll	shift left logical	result <= "~~10~~*010101*" sll 2	"*01010100*"
	srl	shift right logical	result <= "*10010~~101~~*" srl 3	"*000**10010***"
arithmetic	sla	shift left arithmetic	result <= "~~100~~*10101*" sla 3	"*10101***111**"
	sra	shift right arithmetic	result <= "*1001010~~1~~*" sra 2	"**11***100101*"
rotate	rol	rotate left	result <= "**10***100011*" rol 2	"*10001110*"
	ror	rotate right	result <= "*10100011*" ror 2	"**11***101000*"

Table 6.3: VHDL shift operators with examples.

- Both logical shifts introduce 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 logical and arithmetic shifts is that in arithmetic shift, the sign-bit is never changed and the bit that is fed into one end can differ. Hence, for arithmetic shift lefts, the last right bit is stuffed to 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 independently of the value of the individual bits in the operand.

6.4 Other Operators

The other groups of operators are generally used with numeric types. Since this section does not present numerical operations in detail, the operators are briefly listed below in Table 6.4. Special attention is given to the `&`, `mod`, and `rem` operators. These operators are also limited to operating on specific types, which are also not listed here.

Operator		Name	Comment
adding	+	addition	
	−	subtraction	
	&	concatenation	can operate only on specific types
sign	+	identity	unary operator
	−	negation	unary operator
multiplying	*	multiplication	
	/	division	often limited to powers of two
	mod	modulus	can operate only on specific types
	rem	remainder	can operate only on specific types
miscellaneous	**	exponentiation	often limited to powers of two
	abs	absolute value	

Table 6.4: All the other VHDL operators not listed so far.

6.5 Concatenation Operator

The concatenation operator `&` is often a useful operator when dealing with digital circuits. There are many times when you will 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 Listing 6.1.

6.6 Modulus and Remainder Operators

Both the remainder operator `rem` and the modulus operator `mod` are applied to integers types and both give back an integer type. There is often confusion about the differences between the two operators and the difference in their operation on negative and positive numbers. The definitions that VHDL uses for these operators are shown in Table 6.5 while a few examples

```
signal A_val, B_val : std_logic_vector(3 downto 0);
signal C_val : std_logic_vector(5 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(4 downto 1);
D_val <= A_val & B_val;
```

Listing 6.1: Examples of the concatenation operator

of these operators are provided in Table 6.6. 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, answers are sometime counter-intuitive.

Operator	Name	Satisfies this Conditions
`rem`	remainder	1. sign of (X `rem` Y) is the same as X 2. abs (X `rem` Y) < `abs` (Y) 3. (X `rem` Y) = (X - (X / Y) * Y)
`mod`	modulus	1. sign of (X `mod` Y) is the same as Y 2. abs (X `mod` Y) < `abs` (Y) 3. (X `mod` Y) = (X * (Y - N)) for some integer N

Table 6.5: Definitions of `rem` and `mod` operators. (`abs` = absolute value)

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 6.6: Example of `rem` and `mod` operators.

6.7 Review of Almost Everything Up to Now

VHDL is a 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 architecture is made of statements that describe the behavior of the digital circuit. Because this is a hardware description language, 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 modeled as a set of logic gates that operate 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 constituted of sequential statements exclusively. 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.

Coding styles in VHDL fall under the category of data-flow, 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 data-flow model. VHDL code describing more complex digital circuits will generally contain both features of all of these types of modeling.

Since you should make no effort whatsoever to memorize VHDL syntax, it is recommended that a cheat sheet always be kept next to you as you perform VHDL modeling. *Developing a true understanding of VHDL is what is going to make you into a good hardware designer. The ability to memorize VHDL syntax proves almost nothing.*

6.8 Writing less code

Two VHDL features help in creating designs and testbenches without having to write repetitive code: loops and procedures. In this section, procedures are briefly presented in the design (Listing 6.2) and loops are shown in the testbench (Listing 6.3). This approach is taken for simplicity, but both features could be used in designs and testbenches.

The design **add3** implements a 3-bits adder with a 4-bits output ($s = a+b$) connecting 3 full-adders. To avoid writing the full-adder code 3 times, the **procedure** keyword is used. The architecture could be understood as if the procedure code was copied at each line where it is used. The contexts within the procedures (e.g. variables) are isolated from each other.

```
1  -- library omitted
2  entity add3 is
3      port(a, b : in std_ulogic_vector(2 downto 0);
4           s : out std_ulogic_vector(3 downto 0));
5  end add3;
6
7  architecture my_add3 of add3 is
8      procedure fulladd(signal a, b, cin : in std_ulogic;
9                        signal s, co : out std_ulogic) is
10         variable axorb : std_ulogic;
11     begin
12         axorb := a xor b;
13         s <= axorb xor cin;
14         co <= (a and b) or (axorb and cin);
15     end fulladd;
16     signal ci : std_ulogic_vector(2 downto 0);
17 begin
18     ci(0) <= '0'; -- each procedure has its own connections:
19     fulladd(a(0), b(0), ci(0), s(0), ci(1));
20     fulladd(a(1), b(1), ci(1), s(1), ci(2));
21     fulladd(a(2), b(2), ci(2), s(2), s(3));
22 end my_add3;
```

Listing 6.2: VHDL description of 3-bits adder

The **loop** keyword is used in this testbench to allow the generation of all possible input combinations for **a** and **b** without having to write them explicitly. Since this is a simulation routine, the nested loops can be analyzed like a sequence in which for every possible value of **a**, all values of **b** are tested, with a delay between each one. The `to_unsigned` keyword is a type conversion which can be ignored for now since Chapter 11 is dedicated to these types. When used in designs, "for loops" do not result in a sequence but still allow very concise code by avoiding repetitions.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all; -- for the 'to_unsigned' operation
4
5  architecture tb of tb_add is
6  component add3 is
7      port(a, b : in std_ulogic_vector(2 downto 0);
8           s : out std_ulogic_vector(3 downto 0));
9  end component;
10     signal a, b : std_logic_vector(2 downto 0);
11     signal s : std_logic_vector(3 downto 0);
12 begin
13     DUT: add3 port map (a=>a, b=>b, s=>s);
14
15     add3tb : process begin
16         for ai in 0 to 7 loop
17             a <= std_logic_vector((to_unsigned(ai, 3)));
18             for bi in 0 to 7 loop
19                 b <= std_logic_vector((to_unsigned(bi, 3)));
20                 wait for 10 ns;
21             end loop;
22         end loop;
23         wait; -- STOP
24     end process add3tb;
25 end tb;
```

Listing 6.3: VHDL testbench for the 3-bits adder

7 Using VHDL for Sequential Circuits

All the circuits we have examined up until now have been combinatorial logic circuits. In other words, none of the circuits we have examined so far are actually able to *store information*. This section shows some of the various methods used to describe sequential circuits, which can achieve that. 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 data-flow models to describe storage elements in VHDL, but it is much easier to use behavior models.

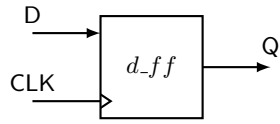
The few approaches for designing flip-flops shown in the this section cover just about all the possible functionality you could imagine when you make use of a D flip-flop. It will become clear that expanding on this concept to implement other types of flip-flops is a very trivial task.

7.1 Simple Storage Elements Using VHDL

The general approach for learning the implementation of storage elements in digital circuits is to expand on the idea of a basic cross-coupled cell. This circuit forms what is referred to as a **latch**, which has two stable states, hence being capable of storing a bit. The concept of a clocking signal is added to this device in order to enhance its controllability, by restricting its state change to a very short time at the clock level transition. This finally results in the **flip-flop**, which is nothing more than an *edge-sensitive bit-storage device* and will be the main building block of sequential designs.

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

EXAMPLE 13. Write the VHDL code that describes a D flip-flop shown on the right. Use a behavioral model in your description.



SOLUTION. The solution to Example 13 is shown in Listing 7.1. Listed below are a few interesting things to note about the solution.

- The statements within the process (behavioral model) are executed sequentially. The process is executed each time a change is detected in any of the signals in the process' sensitivity list. In this case, this happens each time there is a change in logic level of the `CLK` signal.
- Note that the `D` signal is on the right-hand side of an assignment but is *not* on the sensitivity list! This is because a change in this signal doesn't *trigger* a change in the `Q` output.
- The `rising_edge()` construct is used in the `if` statement to indicate that changes in the circuit output happen 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 has been written makes the whole circuit synchronous; in fact, 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 functionality of `rising_edge(CLK)` can be achieved using the "`event`" attribute via the construct: `(CLK'event and CLK='1')`. Please keep this in mind because the newer construction was only proposed on VHDL 2008.
- The process has the label `dff`. This is not required by the VHDL language but the addition of process labels promotes a self-describing nature of the code and increases its readability and understandability.

```
1  -- Model of a simple D Flip-Flop --
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4
5  entity d_ff is
6      port ( D, CLK : in  std_logic;
7             Q :      out std_logic);
8  end d_ff;
9
10 architecture my_d_ff of d_ff is
11 begin
12     dff:  process(CLK)
13     begin
14         if (rising_edge(CLK)) then
15 --          (CLK'event and CLK='1') this is the old way...
16              Q <= D;
17         end if;
18     end process dff;
19 end my_d_ff;
```

Listing 7.1: Solution to Example 13

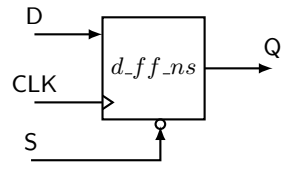
The D flip-flop is best known and loved for its ability to store (save, remember) a single bit. The way that the VHDL code in Listing 7.1 is able to store a bit is not however obvious. The bit-storage capability in VHDL is implied by the way the VHDL code is interpreted. The implied storage capability 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 value of `Q` and therefore it must remember the current value. The memory feature of the current value, or state, constitutes the famous bit storage quality of a flip-flop. If you have not specified what the output should be for every possible set of input conditions, the option taken by VHDL is to not change the current output. By definition, if the input changes to an unspecified state, the output remains unchanged. In this case, the output associated with the previous set of input 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 the VHDL code.

In terms of the D flip-flop shown in Example 13, 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 have induced a memory element is to look for the presence of an `else` clause associated with the `if` statement.

The previous two paragraphs are vastly important for understanding VHDL; the concept of inferring memory in VHDL is crucial to sequential circuits and their design dependent on this. 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 will only be using VHDL to design combinatorial circuits, you will most likely be faced with the comprehension of these concepts. One of the classic warnings generated by the VHDL synthesizer is the 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 to modify your VHDL code in such a way as to remove this warning. Assuming you did not intend to generate a latch, the cause of your problem is that you have 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 have not explicitly listed the current input condition.

EXAMPLE 14. 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 output when asserted.



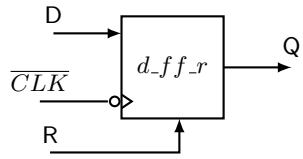
SOLUTION. The solution to Example 14 is shown in Listing 7.2. There are a few things of interest regarding this solution.

```
1  ----------------------------------------------------------------
2  -- D Flip-flop model with active-low synchronous set input. --
3  ----------------------------------------------------------------
4  library IEEE;
5  use IEEE.std_logic_1164.all;
6
7  entity d_ff_ns is
8      port (  D, S, CLK :  in  std_logic;
9              Q   :  out std_logic);
10 end d_ff_ns;
11
12 architecture my_d_ff_ns of d_ff_ns is
13 begin
14     dff:  process (CLK)
15     begin
16         if (rising_edge(CLK)) then
17             if (S = '0') then
18                 Q <= '1';
19             else
20                 Q <= D;
21             end if;
22         end if;
23     end process dff;
24 end my_d_ff_ns;
```

Listing 7.2: Solution to Example 14

- The `S` input 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 is checked. In other words, the `D` input is transferred to the output only on the rising edge of the clock and only if the `S` input is not asserted.

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 `R` input to be an active-high, **asynchronous** input that resets the `D` flip-flop outputs when asserted.

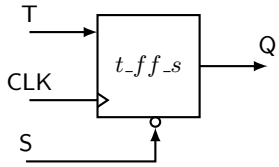


SOLUTION. The solution to Example 15 is shown in Listing 7.3. You can probably glean the most information about asynchronous input and synchronous inputs by comparing the solutions to Example 14 and Example 15. A couple of interesting points are listed below.

- The reset input is independent of the clock and takes priority over the clock. This prioritization 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.

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

EXAMPLE 16. 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.



```vhdl
-- D Flip-flop with active-high asynchronous reset input. --
library IEEE;
use IEEE.std_logic_1164.all;

entity d_ff_r is
    port (D, R, 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 (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;
```

Listing 7.3: Solution to Example 15

SOLUTION. The solution to Example 16 is shown in Listing 7.4. This example has some very important techniques associated with it that are well worth mentioning. A unique quality of the T flip-flop is demonstrated in this implementation. The output of a D flip-flop is only dependent upon the `D` input, but the output of a T flip-flop is dependent upon both the `T` input and the current output of the flip-flop.

- The dependency on the output adds a certain complexity to the T flip-flop model as compared to the D flip-flop as is shown in Listing 7.4. **In older versions of VHDL, signals that are declared as outputs can therefore not appear on the right-hand side of a signal assignment operator**. 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". For compatibility with older VHDL tools, this

```vhdl
-- T Flip-flop with active-low asynchronous set input. --
library IEEE;
use IEEE.std_logic_1164.all;

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 tsig : std_logic; -- intermediate signal declaration
begin
    tff:  process (S, CLK)
    begin
        if (S = '0') then
            tsig <= '1';
        elsif (rising_edge(CLK)) then
            tsig <= T XOR tsig; -- temp output assignment
        end if;
    end process tff;
    Q <= tsig; -- final output assignment
end my_t_ff_s;
```

Listing 7.4: Solution to Example 16 (before VHDL 2008)

T flip-flop model uses an internal signal in order to use the current state of the flip-flop (`tsig`) on both sides of the assignments.

- The process manipulates the intermediate signal and its value is assigned to the output `Q` in a concurrent statement. Note that the intermediate signal appears on both sides of assignments. This is a widely used approach in VHDL: please take time to understand and absorb it. Lastly, there are other mode specifications that would allow 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 im-

plementation. 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, you may not have been aware of it.

- There are certain advantages to using T flip-flops in some conditions, D flip-flops are generally the storage element of choice using VHDL. If you do not have a specific reason for using some type of flip-flop other than a D flip-flop, you probably should not.

Overcoming the limitation of the older VHDL versions, the Listing 7.5 presents a simpler and more intuitive implementation for the architecture of the T flip-flop (but be careful with the compatibility with older tools).

```
1  architecture my_t_ff_s of t_ff_s is
2  begin
3      tff:  process (S, CLK)
4      begin
5          if (S = '0') then
6              Q <= '1';
7          elsif (rising_edge(CLK)) then
8              Q <= T XOR Q; -- allowed in VHDL 2008
9          end if;
10     end process tff;
11 end my_t_ff_s;
```

Listing 7.5: Architecture for Example 16 (VHDL 2008)

7.2 Inducing Memory: Data-flow vs. Behavioral Modeling

A major portion of digital design deals with sequential circuits. Generally speaking, most sequential circuit design is about synchronizing events to a clock edge. In other words, output changes only occur on a 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 data-flow modeling as well: not explicitly specifying an output for every possible input condition generates a latch

(a storage element). 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 for 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 data-flow models, most knowledgeable people examining your VHDL code would not initially be clear as to what exactly you are 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 to be clever with.

7.3 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 data-flow and behavioral models, but the later is preferred and expected by VHDL developers and synthesis tools for certain targets (like FPGAs).
- If a signal declared in the entity declaration has a mode specifier of out, that signal cannot appear on the right-hand side of a signal assignment operator if the VHDL code is supposed to be compatible with older tools. 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 be avoided in favor of intermediate signals. The main synthesis tools discourage this mode as it spreads to other modules of the architecture, among other reasons.

7.4 Testbenches with clock

The inherent parallelism in VHDL allows independent specifications of different testbench stimulus, which are specially useful to test sequential hardware descriptions. In Figure 7.1, the expected behavior for a D Flip-Flop with synchronous reset is shown. It is clear in this waveform that the Q output is not determined until the reset becomes active at the rising clock edge. It can also be verified that a change in the D input doesn't appear at the output before this clock edge. An additional test, not included in this waveform, could check if the reset signal takes precedence over the D input. A possible VHDL description for this Flip-Flop is presented in Listing 7.6.

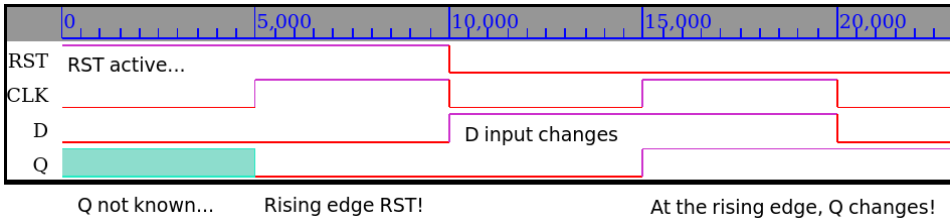


Figure 7.1: DFF with synchronous active high reset.

It should be noted that a proper test for such hardware description must activate different inputs at specific times, when compared to the clock edges, allowing the verification of the desired synchronous and asynchronous behavior. In summary: a synchronous input should not affect the outputs immediately when it changes, but only at the correct clock edge. Conversely, an asynchronous input must perform its function regardless of the moment it changes.

In Listing 7.7 a testbench which generates the described test sequence is presented. Note that the clock signal is concisely created in a single line of code (16). Using the simulation keyword **after**, the `CLK` signal is inverted

```
1  -- library and entity omitted (see testbench component)
2  architecture my_d_ff of d_ff is
3  begin
4      dff: process (CLK)
5      begin
6          if (rising_edge(CLK)) then
7              if (RST = '1') then
8                  Q <= '0';
9              else
10                 Q <= D;
11             end if;
12         end if;
13     end process dff;
14 end my_d_ff;
```

Listing 7.6: DFF with synchronous active high reset

at each half of the desired clock period (defined using a constant). These inversions proceed as long as the CLK_ENABLE signal remains high.

This scheme allows the clock to start its first period at the low level (defined at line 11), so the changes within the stimulus process, as long as separated by multiples of the clock period, always happen at the falling clock edge. As mentioned before, by simply adding different input combinations to this process with the proper delay, both synchronous and asynchronous behavior can be verified.

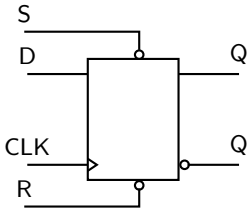
After all the input combinations are specified, the stimulus process can simply be terminated by setting the CLK_ENABLE signal to low. This forces the clock signal to stop and no further events will be registered by the simulator. It is clear how easy it is to create test signals independently, while controlling their changes reliably in the simulated time.

```
1  -- library and empty entity omitted
2  architecture teste of tb_ffd is
3  component d_ff is
4      port (  D, CLK, RST : in std_logic;
5              Q : out std_logic);
6  end component;
7      -- declarations used only by simulation:
8      constant PERIOD : time := 10 ns;
9      signal CLK_ENABLE: std_logic := '1';
10     -- signals connected to the DUT:
11     signal CLK : std_logic := '0';
12     signal RST : std_logic;
13     signal D : std_logic;
14     signal Q : std_logic;
15 begin
16     CLK <= CLK_ENABLE and not CLK after PERIOD / 2;
17
18     DUT : d_ff port map(CLK => CLK, RST => RST, D => D, Q => Q);
19
20     stimulus: process
21     begin
22         D <= '0';
23         RST <= '1';
24         wait for PERIOD;
25         RST <= '0';
26         D <= '1';
27         wait for PERIOD;
28         D <= '0';
29         wait for PERIOD;
30         CLK_ENABLE <= '0';
31         wait;
32     end process stimulus;
```

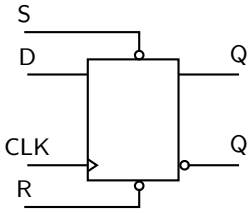
Listing 7.7: DFF with synchronous active high reset

7.5 Exercises: Basic Memory Elements

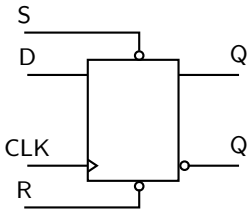
EXERCISE 1. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



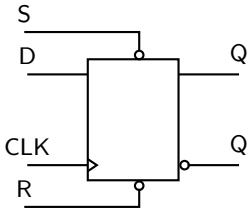
EXERCISE 2. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume the S input takes precedence over the R input in the case where both are asserted simultaneously.



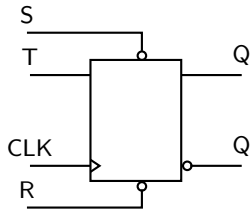
EXERCISE 3. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are synchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



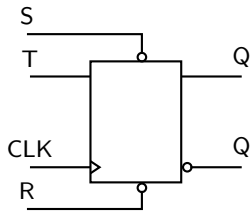
EXERCISE 4. Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. If both the S and R inputs are asserted simultaneously, the output of the flip-flop will toggle.



EXERCISE 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.



EXERCISE 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.



8

Finite State Machine Design Using VHDL

Finite state machines (FSMs) are mathematical abstractions that are used to solve a large variety of problems, among which are electronic design automation, communication protocol design, parsing and other engineering applications. At this point in your digital design career, you have probably designed several state machines on paper. You are now at the point where you can implement and test them using actual hardware. The first step in this process is to learn how to model FSMs using VHDL.

As you will see in this section, simple FSM designs are just a step beyond the sequential circuit design described in the previous section. The techniques presented here will allow you to quickly and easily design relatively complex FSMs which can be useful in a number of ways.

A block diagram for a standard Moore-type FSM is shown in Fig. 8.1. This diagram is fairly typical but different names are used for some of the blocks in the design. The `Next State Decoder` is a block of combinatorial

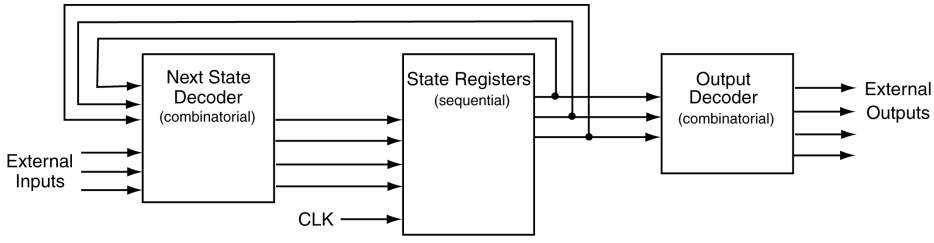


Figure 8.1: Block diagram for a Moore-type FSM.

logic that uses the current external inputs and the current state to decide upon the next state of the FSM. In other words, the inputs to the `Next State Decoder` block are decoded to produce an output that represents the next state of the FSM. The circuitry in the `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 a storage element that stores the present state of the machine. The inputs to the `Output Decoder` are used to generate the desired external outputs, which is accomplished via combinatorial logic. Because the external outputs are only dependent upon the current state of the machine, this FSM is classified as a Moore-type FSM.

The FSM model shown in Fig. 8.1 is the most common model for describing a Moore-type FSM. This is probably because students are often asked to generate the combinatorial logic required to implement the `Next State Decoder` and the `Output Decoder`; however here we want to think about FSMs in the context of VHDL. The true power of VHDL starts to emerge in dealing with FSMs. As you will see, the versatility of VHDL behavioral modeling removes the need for large paper designs of endless K-maps and endless combinatorial logic elements.

There are several different approaches used to model FSMs in VHDL. The various approaches are a result of the general versatility of VHDL as a language. What we will describe in this section is probably the clearest approach for FSM implementation. A block diagram of the approach we will use in the implementation of FSMs is shown in Fig. 8.2.

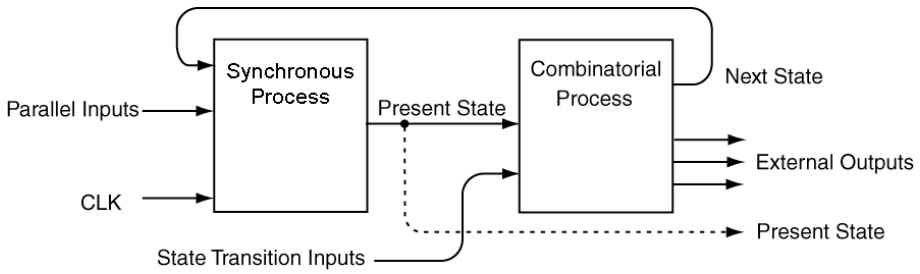


Figure 8.2: Model for VHDL implementations of FSMs.

Although it does not look that much clearer, you will soon find the FSM model shown in Fig. 8.2 to be a straightforward method for implementing 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 Fig. 8.1. Note that the two blocks in Fig. 8.1 are both made solely of combinatorial logic.

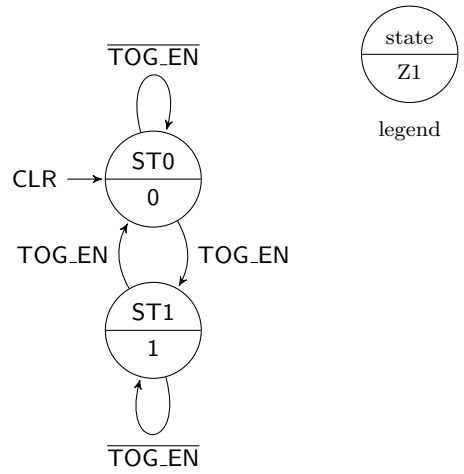
Some new lingo is used to describe the signals in Fig. 8.2, as outlined and described below:

- The inputs labelled `Parallel Inputs` are used to signify inputs that act in parallel on each of the storage elements. These inputs include enables, presets, clears, etc.
- The inputs labelled `State Transition Inputs` include external inputs that control the state transitions. These inputs also include external inputs used to decode Mealy-type external outputs (which are directly affected by inputs, differently from what happens with Moore-type FSMs).
- The `Present State` signals are used by the `Combinatorial Process` box for both next state decoding and output decoding. The diagram of Fig. 8.2 also shows that the `Present State` variables can also be provided as outputs to the FSM but they are not required.

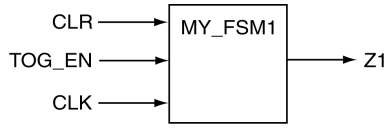
One final comment before we begin. There are many different methods that can be used to describe FSMs using VHDL. Once you understand the method presented here, learning any other style of FSM implementation is relatively painless. More information on the other FSM coding styles may be found in various VHDL texts or on the web. Regardless of the type of FSM your project demands, remember to keep the design simple and clear, using modularization as needed (avoid long and complicated processes).

8.1 VHDL Behavioral Representation of FSMs

EXAMPLE 17. Write the VHDL code that implements the FSM shown on the right.



SOLUTION. This problem represents a basic FSM implementation. It is instructive to show the black-box diagram which is an aid in writing the entity description. Starting FSM problems with the drawing of a black box diagram is a healthy approach. Oftentimes with FSM problems, it becomes challenging to discern the FSM inputs from the outputs. Drawing a diagram alleviates this problem. The black box diagram and the code for the solution of Example 17 is shown in Listing 8.1.



This solution has many things worth noting in it, as follows.

- We have declared a special VHDL type named `state_type` to represent the states in this FSM. This is an example of how enumeration types are used in VHDL. As with enumerated 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 symbolic equivalent. In this case, the type we have created has two "values": PS and NS. The key thing to note here is that a `state_type` is something we have created and is not a native VHDL type.

```
entity my_fsm1 is -- library declaration omitted
    port (  TOG_EN, 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, CLR) -- note: no NS here!
    begin -- state reset and transition
        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 statements
        end case;
    end process comb_proc;
end fsm1;
```

Listing 8.1: Solution to Example 17

- The synchronous process is equal in form and function to the simple D flip-flops we examined in the section about sequential circuits. The only difference is that we have substituted PS and NS for D and Q, respectively. Something 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 example is of 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 Moore output (independent of inputs) and the state transition calculation.

- 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 will have an effect of the state only at clock transitions. When the changes are actually instituted in the synchronous process on the next clock edge, the changes in the PS signal causes the combinatorial process to be evaluated.

- The case statement in the combinatorial process provides a `when` clause for each state of the FSM. A `when others` clause has also been used, but it is never reached with the current values. This statement represents good VHDL coding practice as it makes the code less prone to error upon future modifications.

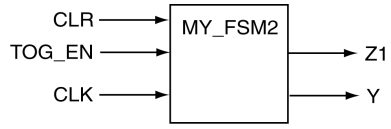
- The Moore output is a function of only the present state. This is expressed by the fact that the assignment of the Z1 output is unconditionally evaluated in each `when` clause of the case statement in the combinatorial process. In other words, the Z1 variable is inside the `when` clauses but outside of the `if` statements in the `when` clauses. This is of 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 which state the FSM transitions to (from any given state).

You will 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 simplify the source code (as it may replace the most frequently used assignment). The pre-assignment makes no difference to the VHDL code because if multiple assignments are made within the code, **only the final assignment takes effect**.

There is one final thing to note about Example 17. 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 of Listing 8.1 by the fact that the only output of the FSM is the signal Z1. This is reasonable if only one signal is required 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. This also provides more room for automatic optimization in the synthesis process.

In some FSM designs, the state variables are provided as outputs. To show this situation, we will provide a solution to Example 17 with the state variables as outputs. The black-box diagram and the VHDL code of this solution is shown in Listing 8.2.



Note that the VHDL code shown in Listing 8.2 differs in only two areas from the code shown in Listing 8.1 (identical processes have been omitted). 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

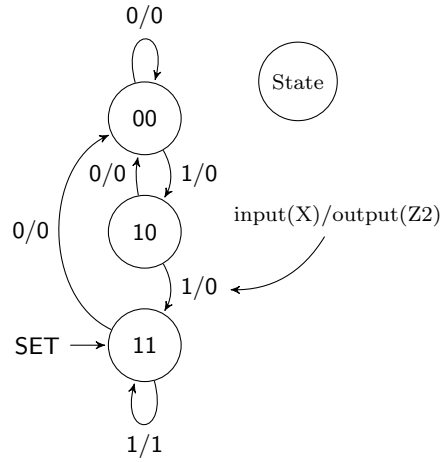
```
entity my_fsm2 is -- library declaration omitted
    port (  TOG_EN, CLK, CLR : in  std_logic;
            Y, Z1            : out std_logic);
end my_fsm2;

architecture fsm2 of my_fsm2 is
    type state_type is (ST0, ST1);
    signal PS, NS : state_type;
begin
    -- sync_proc: process(CLK, CLR)
    -- identical state transition process omitted

    -- comb_proc: process(PS, TOG_EN)
    -- identical combinatorial process omitted

    with PS select -- assign external state representation
        Y <= '0' when ST0,
             '1' when ST1,
             '0' when others;
end fsm2;
```

Listing 8.2: Solution to Example 17 that include state variable as output

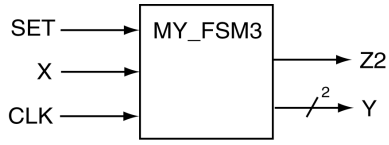
output Y based on the condition of the state variable. The selective signal assignment statement is evaluated each time a change in signal PS is detected. Once again, since we have declared an enumeration type for the state variables, we have no way of knowing exactly how the synthesizer will decide to represent the state variable. The selective signal assignment statement in the code of Listing 8.2 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 will deal with those soon.

Lastly, it is important to note that in Listing 8.2 both processes (omitted, identical to Listing 8.1) and the selective signal assignment are all concurrent statements. Each of them react independently to their respective input signals.

EXAMPLE 18. Write the VHDL code that implements the FSM shown on the right. Consider the state variables as outputs of the FSM.



SOLUTION. The state diagram shown in this problem 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 Listing 8.3. Note that the two state variables are handled as a bundled signal.



As usual, there are a couple of fun things to point out about the solution for Example 18. Most importantly, you should note the similarities between this solution and the previous one.

- The FSM has one Mealy-type output but its value is the default one in the first two states. In the final `when` clause, the Z2 output appears with a different value within `if` statement. The fact that the Z2 output is different in the context of state ST2 makes it a Mealy-type output.
- When providing the state variable outputs (from internal enumeration types), two signals are required since the state diagram contains more than 2^1 and less than 2^2 states. The solution opted is to represent these

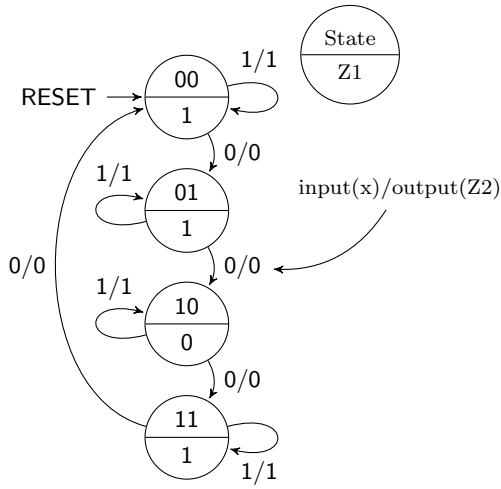
```
entity my_fsm3 is -- library declaration omitted
    port ( X, CLK, SET : in  std_logic;
           Y           : out std_logic_vector(1 downto 0);
           Z2          : out std_logic);
end my_fsm3;

architecture fsm3 of my_fsm3 is
    type state_type is (ST0, ST1, ST2);
    signal PS, NS : state_type;
begin
    sync_proc: process(CLK, 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
        Z2 <= '0'; NS <= ST0; -- default FSM output and state
        case PS is
        when ST0 => -- condition to leave state ST0
            if (X = '1') then NS <= ST1;
            end if;
        when ST1 => -- condition to leave state ST1
            if (X = '1') then NS <= ST2;
            end if;
        when ST2 =>        -- items regarding state ST2
            if (X = '1') then
                NS <= ST2;
                Z2 <= '1'; -- Z2 is a Mealy output!
            end if;
        end case; -- the catch all condition omitted
    end process comb_proc;

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

Listing 8.3: Solution to Example 18

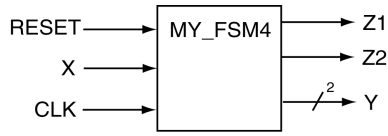
outputs as a bundle which has the effect of slightly changing the form of the selected signal assignment statement appearing at the end of the architecture description (`"00", "10"` etc).

- Note that the default values at line 18 considerably simplified the code, as no `else` statements were required.

EXAMPLE 19. Write the VHDL code that implements the FSM shown on the right. Consider the listed state variables as output.



SOLUTION. The state diagram indicates that its implementation will contain four states, one external input and two external outputs. This is a hybrid FSM since it contains both a Mealy and Moore-type output. The black-box diagram is shown below and the VHDL code is in Listing 8.4.



If you haven't noticed by now, implementing FSMs using the VHDL behavioral model is remarkably straightforward. In reality, I rarely code a FSM from scratch; I usually opt to grab some previous FSM I have coded and start from there. For FSM problems, the engineering is in the testing and creation of the state diagram. Do not get too comfortable with behavioral modeling of FSMs; the real fun is actually generating a FSM that solves a given problem.

```vhdl
entity my_fsm4 is -- library declaration omitted
    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
    signal PS, NS : std_logic_vector(1 downto 0);
begin
    sync_proc: process(CLK, RESET) begin
        if (RESET = '1') then PS <= "00";
        elsif (rising_edge(CLK)) then PS <= NS;
        end if;
    end process sync_proc;

    comb_proc: process(PS, X) begin
        Z1 <= '1'; Z2 <= '0'; -- Moore (Z1) and Mealy (Z2)
        case PS is
        when "00" => if (X = '0') then NS <= "01"; Z2 <= '0';
                     else  NS <= "00"; Z2 <= '1';
                     end if;
        when "01" => if (X = '0') then NS <= "10"; Z2 <= '0';
                     else  NS <= "01"; Z2 <= '1';
                     end if;
        when "10" => Z1 <= '0';  -- Moore output
                     if (X = '0') then NS <= "11"; Z2 <= '0';
                     else  NS <= "10"; Z2 <= '1';
                     end if;
        when "11" => if (X = '0') then NS <= "00"; Z2 <= '0';
                     else  NS <= "11";  Z2 <= '1';
                     end if;
        end case;
    end process comb_proc;
    Y <= PS;
end fsm4;
```

Listing 8.4: Solution to Example 19

Note that, specifically in this example, an explicit representation of the states was used, since it is directly available also as an output.

8.2 One-Hot Encoding for FSMs

Truth be told, there are many different methods that can be used to encode state variables[1]. If the exact form of the representation used is important, then you will need to take the necessary steps in order to control how the state variables are represented by the synthesizer. There are two approaches to control state variable representation. The first approach is to allow the synthesizing tool to handle the details. The first FSMs we have seen used enumeration types to represent the state variables, the synthesizer could choose to actually represent them with an encoding according to some tool configuration or required optization. Most tools have an option to select the desired encoding scheme (they are able to infer FSMs the same way they infer flip-flops from descriptive code). The downside of this approach is that you are 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. One simple and numeric approach for specifying the state variables in the VHDL code was presented in the last FSM and will be expanded in this section.

The approach taken in the previous FSM example was to use full encoding for the state variables of the FSM. This 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 8.1. The bracket-like symbols used in equation 8.1 indicate a ceiling function[2]. The binary nature expressed by this equation is so apparent that this encoding is often referred to as binary encoding.

$$\#(flip\_flops) = \lceil \log_2(\#states) \rceil \tag{8.1}$$

One-hot encoding uses one bit in the state register for each state of the

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

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

FSM. For a one-hot encoding FSM with 16 states, 16 flip flops are required. However only four flip flops are required if the same FSM is implemented using a binary encoding. One-hot encoding simplifies the logic and the interconnections between overall logic. Despite looking quite wasteful in terms of employed logic, one-hot encoding often results in smaller and faster FSMs.

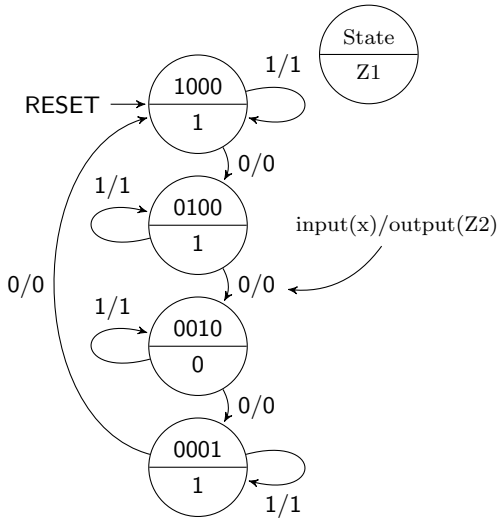
For one-hot encoded FSMs, only one flip-flop is asserted at any given time. This requires that each distinct state be represented by one flip-flop. In one-hot encoding, the number of flip-flops required to implement a FSM is therefore equal to the number of states in the FSM. The closed form of this relationship is shown in equation 8.2.

$$\#(flip\_flops) = \#(states) \tag{8.2}$$

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 will need to grab control away from the synthesizer. And since we are 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. The changes 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 fully encoded approach are thus limited to the entity declaration (you will need more variables to represent the states), the declaration of the state variables (you will 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 are actually not faking it like we were in other examples).

EXAMPLE 20. Write the VHDL code that implements the FSM shown on the right. Consider the listed state variables as output. Use one-hot encoding for the state variables. This problem is Example 19 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 will approach the implementation of this FSM one piece at the time.

Listing 8.5 shows the modifications to the entity declaration required to convert the full encoding used in Example 19 to one-hot encoding. It is clear that the only change is in the `Y` output bit-width.

Listing 8.6 shows the required modifications to the state variable 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 Listing 8.6. 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 that this merges the two best features of the previous examples: the `state_type` enumeration allows the convenient use of the symbolic names (`ST0` etc) and the attribute specifies the encoding without actually forcing the signal representation (like in Example 19).

```
entity my_fsm4 is -- full encoded approach (Y with 2 bits)
    port ( X, CLK, RESET : in  std_logic;
           Y : out std_logic_vector(1 downto 0);
           Z1, Z2 : out std_logic);
end my_fsm4;

entity my_fsm4_oh is -- one-hot encoding approach (Y with 4 bits)
    port ( X, CLK, RESET : in  std_logic;
           Y : out std_logic_vector(3 downto 0);
           Z1, Z2 : out std_logic);
end my_fsm4_oh;
```

Listing 8.5: Modifications to convert Example 19 to one-hot encoding

```
-- the approach for enumeration types (Examples 17 and 18)
type state_type is (ST0, ST1, ST2, ST3);
signal PS, NS : state_type;

-- the approach used with explicit encoding (Example 19)
signal PS, NS : std_logic_vector(1 downto 0);

-- the new approach (merging the best of the two previous ones)
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;
```

Listing 8.6: Modifications to convert state variables to use one-hot encoding

Listing 8.7 presents the final architecture for Example 20. Note that the default case is assigned an invalid one-hot state (all bits are zero). This decision depends strongly on the actual problem being solved.

```
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, 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 <= '1'; Z2 <= '0'; -- outputs: Z1 (Moore)  Z2 (Mealy)
        case PS is
        when ST0 => if (X = '0') then NS <= ST1; Z2 <= '0';
                    else  NS <= ST0; Z2 <= '1';
                    end if;
        when ST1 => if (X = '0') then NS <= ST2; Z2 <= '0';
                    else  NS <= ST1; Z2 <= '1';
                    end if;
        when ST2 => Z1 <= '0';  -- Moore output
                    if (X = '0') then NS <= ST3; Z2 <= '0';
                    else  NS <= ST2; Z2 <= '1';
                    end if;
        when ST3 => if (X = '0') then NS <= ST0; Z2 <= '0';
                    else  NS <= ST3;  Z2 <= '1';
                    end if;
        when others => NS <= ST0; -- Important for One Hot Enc.
        end case;
    end process comb_proc;

    with PS select Y <= "1000" when ST0, "0100" when ST1,
       "0010" when ST2, "0001" when ST3, "0000" when others;
end fsm4_oh;
```

Listing 8.7: The architecture of the full solution to Example 20

8.3 Important Points

No testbench example is provided for this Chapter as it would have the same structure as the one presented to test sequential circuit descriptions. To fully test an FSM may become a complex task that grows exponentially with the number of inputs. The same applies to the number states as every possible transition, including the ones to the same state, should be tested.

The technique presented in the previous testbench, which changes inputs concurrently to the clock generation, is also relevant for FSMs. By changing the inputs at the opposite clock edge which activates the FSM, two advantages are obtained:

1. the asynchronous behavior of Mealy outputs are checked against the synchronous ones (Moore), since this difference becomes clearly distinguishable at the simulated outputs;

2. changing the inputs at the same clock edge which activates the FSM, besides being a probable violation in the final hardware, may result in erroneous simulated behavior, with delayed state transitions;

- Modeling FSMs from a state diagram is a straightforward process using VHDL behavioral modeling. The process is so straightforward 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 here details only one of those styles but is generally considered the most straightforward of all styles.

- The actual encoding of the FSM's state variables when enumeration types are used may be left up to the synthesis tool or informed as a synthesis option. If a preferred method of variable encoding is to be specified in the VHDL code, using the attribute approach detailed in this section is a simple but viable alternative. Another approach, presented here, is to force the signal type, like in Example 19, and use

`constant` definitions for this specific type, providing the symbolic names for the states (see Listing 8.8). This is a little more "verbose", but it follows the strong-typed nature of VHDL, the definitions are not repeated and may result in simpler combinational logic.

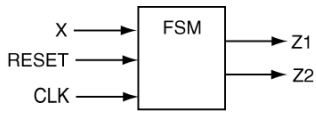
```
1 architecture fsm4_oh of my_fsm4_oh is
2     constant ST0: std_logic_vector(3 downto 0) := "1000";
3     constant ST1: std_logic_vector(3 downto 0) := "0100";
4     constant ST2: std_logic_vector(3 downto 0) := "0010";
5     constant ST3: std_logic_vector(3 downto 0) := "0001";
6     signal PS, NS : std_logic_vector(3 downto 0);
7 begin -- no change follows, except for the final assignment:
8     Y <= PS; -- no need for selective assignment
9 end fsm4_oh;
```

Listing 8.8: Alternative state representation for Example 20

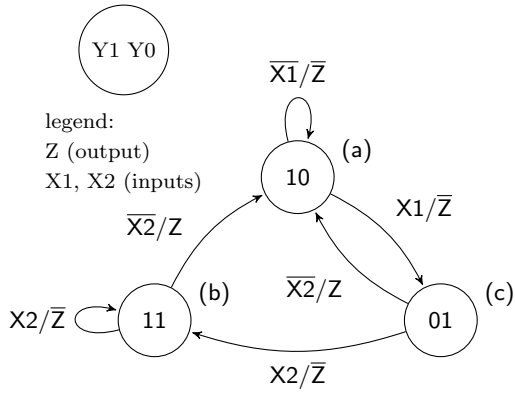
8.4 Exercises: Behavioral Modeling of FSMs

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

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsm is
   port (  X,CLK : in  std_logic;
           RESET : in  std_logic;
           Z1,Z2 : out std_logic;
end fsm;
-- architecture
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;
```

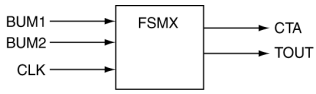


EXERCISE 2. Write a VHDL behavioral model that could be used to implement the state diagram as shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.

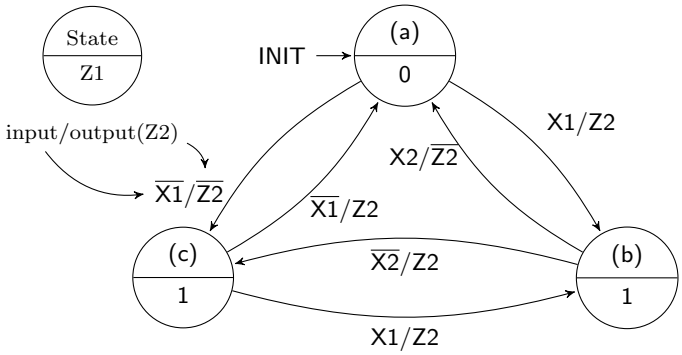


EXERCISE 3. Draw the state diagram associated with the following VHDL code. Be sure to provide a legend and remember to label everything.

```vhdl
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsmx is
    Port ( BUM1,BUM2 : in  std_logic;
                 CLK : in   std_logic;
           TOUT,CTA : out std_logic);
end fsmx;
-- architecture
architecture my_fsmx of fsmx is
   type state_type is (S1,S2,S3);
    signal PS,NS : state_type;
begin
   sync_p: process (CLK)
   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;
```



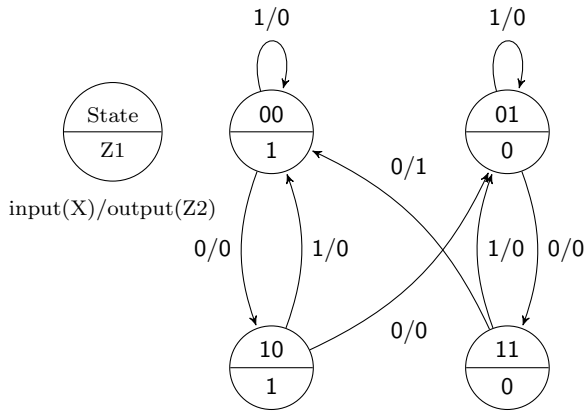
EXERCISE 4. Write the VHDL behavioral model code that could be used to implement the state diagram shown on the right.



EXERCISE 5. 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 state variables and their symbolic equivalents.

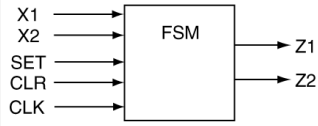
```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsm is
port (      X,CLK : in  std_logic;
            RESET : in  std_logic;
            Z1,Z2 : out std_logic;
                Y : out std_logic_vector(2 downto 0));
end fsm;
-- architecture
architecture my_fsm of 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, RESET) -- process
   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) -- process
   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;
```

EXERCISE 6. Write a VHDL behavioral model code that can be used to implement the state diagram shown on the right. All state variables should be encoded as listed and also provided as outputs of the FSM.

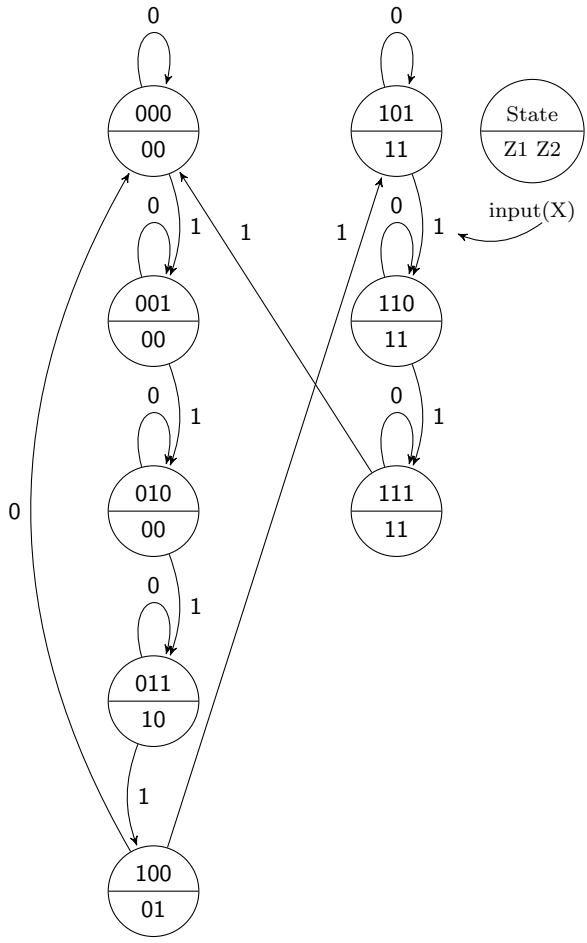


EXERCISE 7. Draw the state diagram that corresponds to the following VHDL model and state whether the FSM is a Mealy machine or a Moore machine. Be sure to label everything.

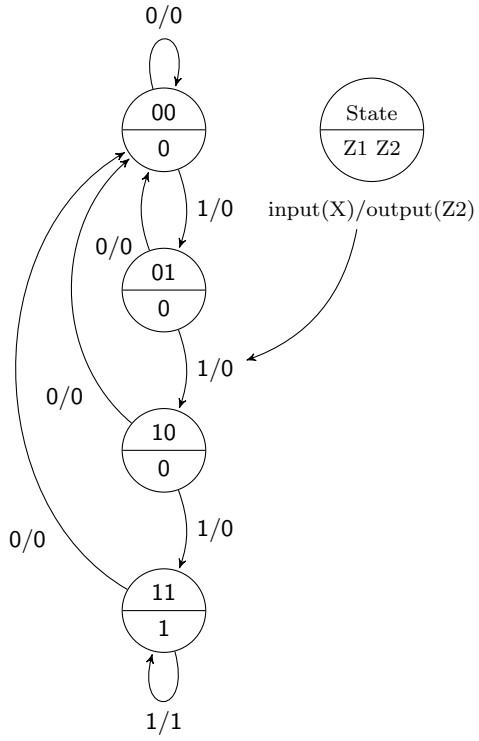
```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsm is
    Port ( CLK,CLR,SET,X1,X2 : in  std_logic;
                        Z1,Z2 : out std_logic);
end fsm;
-- architecture
architecture my_fsm of fsm is
   type state_type is (sA,sB,sC,sD);
   attribute ENUM_ENCODING: STRING;
   attribute ENUM_ENCODING of state_type: type
       is "1000 0100 0010 0001";
   signal PS,NS : state_type;
begin
   sync_p: process (CLK, CLR, SET) -- process
   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) -- process
   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;
```



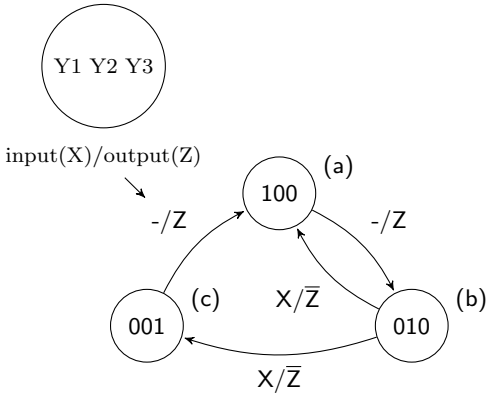
EXERCISE 8. Write the VHDL behavioral model code that can be used to implement the state diagram shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



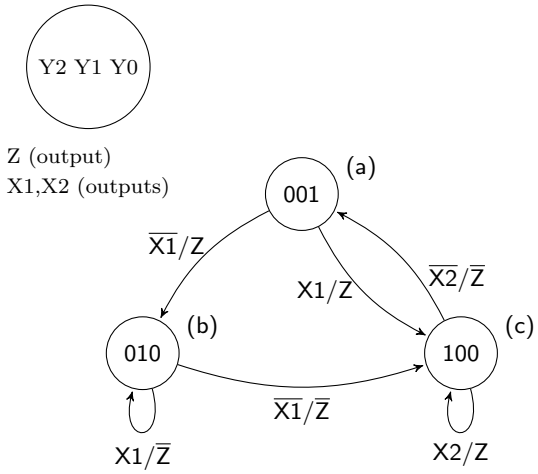
EXERCISE 9. Write the VHDL behavioral model code that can be used to implement the state diagram shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



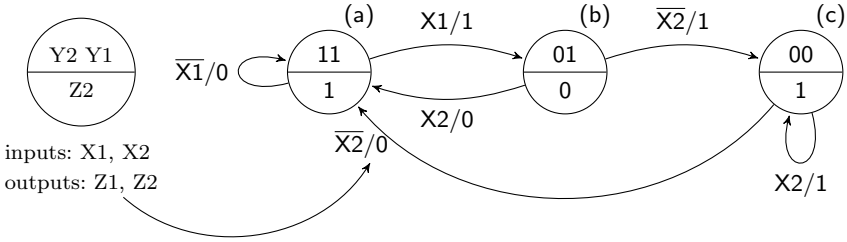
EXERCISE 10. Write the VHDL behavioral model code that can be used to implement the state diagram shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



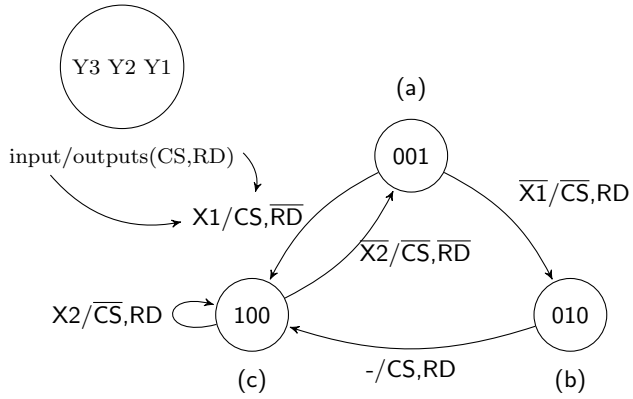
EXERCISE 11. Write the VHDL behavioral model code that can be used to implement the state diagram shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



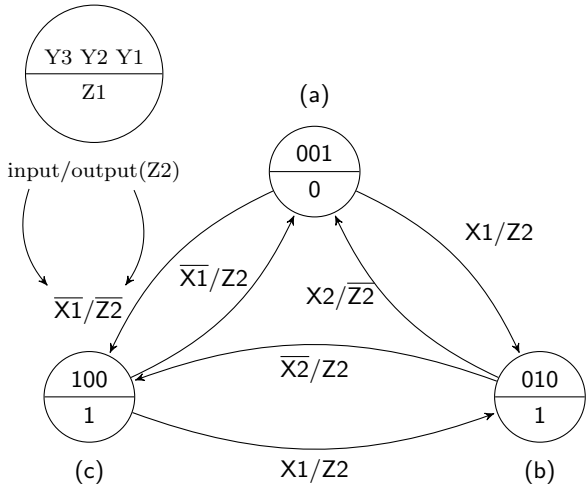
EXERCISE 12. Write the VHDL behavioral model code that can be used to implement the state diagram shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



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



EXERCISE 14. Write the VHDL behavioral model code that can be used to implement the state diagram shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



9

Structural Modeling In VHDL

There are generally three approaches to writing VHDL code: data-flow modeling, behavioral modeling and structural modeling. Up to this point, this book has only dealt with data-flow and behavioral models, since all examples were implemented with a single module. This section presents an introduction to structural modeling.

As projects become more complex, it becomes less likely that any useful design can be implemented with a single module (using one or two of the types of VHDL models we have seen so far). We have already seen this property in dealings with FSMs where process statements (behavioral) and selective signal assignment statements (data-flow) were mixed. The result was a hybrid VHDL model. This will also be the case for structural modeling because even if all modules follow the behavioral approach, their connection intrinsically follows the data-flow approach.

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. 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 modular design. The VHDL modular design approach directly supports hierarchical design which is essentially employed 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 re-test the wheel. The hierarchical approach extends beyond code written on 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 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 as a VHDL guru.

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 (for didactical purposes) 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 these days even though it is taught in many university textbooks. 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 a *Hardware Description Language* like VHDL).

9.1 VHDL Modularity with Components

The main tool for modularity in higher-level languages such as C is the function. In other computer languages, similar modularity is accomplished through the use of 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 prototype) and 4) call the function from the main portion of the code. In VDHL modularity is achieved via the use of **packages**, **components** and **functions**. In the following sections we are going to see how to use `components`.

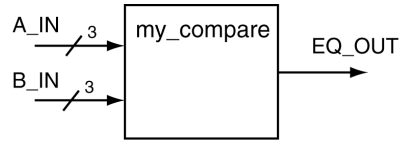
The approach to use a `component` in VHDL is: 1) name the module you plan to describe (the entity), 2) describe what the module will do (the architecture), 3) let the program know the module exists and can be used (component declaration) and 4) use the module in your code (component instantiation, or mapping). The similarities between these two approaches are listed in Table 9.1.

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 or mapping

Table 9.1: Similarities between modules in C and VHDL.

Let us now use these principles in a practical example. Our approach is to describe a template-type approach to VHDL structural design using a simple and well-known combinational circuit. Keep in mind that this could easily be designed in one module but we use this simplicity to introduce VHDL modularity with a small code.

EXAMPLE 21. 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 engineer must derive at some point in his career. The solution presented here implements the discrete gate version of the circuit which is shown in Fig. 9.1. 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 blocks. In this case, 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 Fig. 9.1 contains some extra information that relates to its VHDL structural implementation. First, the **outer** rectangular dashed line represents the boundary of the top-level VHDL entity; therefore 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 in the interior of the design (somewhere in the architecture). Finally, the **inner** rounded dashed lines represent the boundaries of the internal components. Two internal components will be **implemented**, and one of them will be **used** three times, as follows.

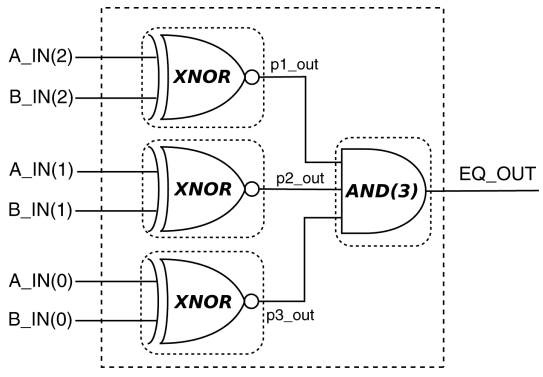


Figure 9.1: Discrete gate implementation of a 3-bit comparator.

The first part of the solution is to provide entity and architecture implementations for the individual gates shown in Fig. 9.1. We need to provide one implementation of an XNOR gate and one of a 3-input AND gate. We only need to provide one definition of the XNOR gate despite the fact that actually 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 Listing 9.1.

```
1  ---------------------------------
2  -- Description of XNOR function --
3  ---------------------------------
4  entity big_xnor is
5      port ( A, B : in  std_logic;
6             F : out std_logic);
7  end big_xnor;
8
9  architecture gate1 of big_xnor is
10 begin
11     F <= not ( (A and (not B)) or ( (not A) and B) );
12 end gate1;
13 ----------------------------------------
14 -- Description of 3-input AND function --
15 ----------------------------------------
16 entity big_and3 is
17     port ( A, B, C : in std_logic;
18            F : out std_logic);
19 end big_and3;
20
21 architecture gate2 of big_and3 is
22 begin
23     F <= ( A and B and C );
24 end gate2;
```

Listing 9.1: Entity and Architecture definitions for discrete gates

The implementations shown in Listing 9.1 present no new VHDL details. The new information is contained in how these circuit elements are used as components in a larger circuit. The procedures for implementing a struc-

tural VHDL design can be summarized in the following steps (assuming that the entity declarations for the interior modules already exist).

Step 1. Generate the top-level entity declaration.

Step 2. Declare the lower-level design units used in design.

Step 3. Declare required internal signals used to connect the design units.

Step 4. Instantiate the design units.

9.1.1 Step 1: top level entity

The first step in a structural implementation is identical to the standard approach we have used for the implementing other VHDL circuits: the `entity`. The entity declaration is derived directly from **outer** dashed box in Fig. 9.1 and is shown in Listing 9.2. 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.

```
1 ------------------------------------------------
2 -- Interface description of 3-bit comparator --
3 ------------------------------------------------
4 entity my_compare is
5 port ( A_IN   : in  std_logic_vector(2 downto 0);
6        B_IN   : in  std_logic_vector(2 downto 0);
7        EQ_OUT : out std_logic);
8 end my_compare;
```

Listing 9.2: Entity declaration for 3-bit comparator

9.1.2 Step 2: Components for lower-level design units

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 another 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. Design units are essentially modules that reside in the lower levels of the design. For our design, we need to declare two separate design units: the XNOR 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 be followed by the word `end component` to terminate the declaration. The best way to do this is by copying, 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 the Listing 9.3 (compared to the original `entity` code, copied from Listing 9.1).

```
1 component big_xnor is -- was "entity big_xnor is"
2     port ( A, B : in  std_logic;
3            F : out std_logic);
4 end component; -- was "end big_xnor;"
5
6 component big_and3 is -- was "entity big_and3 is"
7     port ( A, B, C : in  std_logic;
8            F : out std_logic);
9 end component; -- was "end big_and3;"
```

Listing 9.3

9.1.3 Step 3: Internal signals which connect the design units

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 **outer** dashed line shown in Fig. 9.1. 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. A more appropriate

in this context would be that these signals represent wires which connect components inside a closed box, and are visible from the outside.

The internal signals effectively provide an interface between the various design units that are instantiated in the final design. For this example, three signals are required to connect the outputs of the XNOR gates to the 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 contained in the entity body. The only difference is that the intermediate signal declaration does not contain the mode specifier. We have previously dealt with intermediate signals in other sections of this book. Signal declarations are included as part of the final solution shown in Listing 9.4.

9.1.4 Step 4: Instantiate all the design units

The final step is to create instances of the required modules and map the instances of the various components in the architecture body. Technically speaking, as the word instantiation 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 have referred to as component declaration but we have opted not to do this here. The approach presented here is to have the 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 Listing 9.4.

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

architecture comp1 of my_compare is
   component big_xnor is -- XNOR gate
       port ( A, B : in  std_logic;
              F : out std_logic);
   end component;

   component big_and3 is -- 3-input AND gates
       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 comp1;
```

Listing 9.4: VHDL code for the design hierarchy for the 3-bit comparator

Note that the instantiation process includes labels for all instantiated design units (`x1`, `x2`, `x3` and `a1`). Labels should always be used as part of design unit instantiation because they increase the understandability

of your VHDL model and messages (warning or errors) shown by the development, simulation and synthesis tool.

It is worth noting that the solution shown in Listing 9.4 is not the only approach to use for the mapping process. This code shows what is referred to as 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 signals to be listed in any order. The only minor downside of this approach is that it uses up more space in your VHDL source code (which is irrelevant compared to the advantages).

To successfully simulate and synthesize the design shown in Listing 9.4, the code of Listing 9.1 needs to be included in your VHDL project as well. It is normal practice to keep the two listings in two distinctive files.

The alternative approach to mapping is to use `implied mapping`: 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 and entity declarations. This approach uses less space in the source code but requires signals to be placed in the proper order. An alternative but equivalent architecture for the previous example using implied mapping is shown in Listing 9.5. This is shown here just for completeness but this approach to mapping is very error prone and should be avoided.

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

Listing 9.5: Alternative (and dangerous) way for implied mapping.

Due to the fact that this design was relatively simple, it was possible 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 Listing 9.6 is both valid and commonly seen in designs containing a system clock. Name collision does not occur because the signal name on the left-hand side of the => operator is understood to be internal to the component while the signal on the right-hand side is understood to reside in the next level up in the hierarchy.

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

Listing 9.6: Example of the same signal name crossing hierarchical boundaries

9.2 Generic Map

As we have seen in the previous section, the use of the keyword `component` allows us to declare a VHDL module for further instantiation.

Often it is desirable to write code that is generic. For instance a routine that performs a certain task on an input array of any size, specified only at the instantiation. Let us suppose that we want to implement a piece of code for parity calculation that returns `'0'` when the input N-size array has an even number of `'1'`s and returns `'1'` otherwise. The Listing 9.7 shows such an implementation. Note that N can be left undefined or set to a default value (`:= 2`, in this example), which will be ignored if specified differently at instantiation. This feature is questionable as the top module must know the width to implement its signals.

```
1  entity gen_parity_check is  -- library declaration omitted
2      generic ( N: positive := 2 ); -- the default N is optional
3      port    ( x: in  std_logic_vector(N-1 downto 0);
4                y: out std_logic);
5  end gen_parity_check;
6
7  architecture arch of gen_parity_check is
8  begin
9      process(x)
10         variable temp: std_logic;
11     begin
12         temp := '0';
13         for i in x'range loop
14             temp := temp XOR x(i);
15         end loop;
16         y <= temp;
17     end process;
18 end arch;
```

Listing 9.7: Parity calculation implementation with generic input array size

Listing 9.8 shows how the generic code can be declared and instantiated in your own code via the already seen `component` method. Specifically, in Listing 9.8 the above generic parity check module is used to create a 4-bit parity check module.

To achieve the mentioned modularity the keyword `generic` was used inside the `entity` field in the code above and again inside the `component` field during its declaration in the code below. The `generic` field is used to allow you to control all generic parameters.

Notice how during instantiation (see Listing 9.8, lines 13 and 14) the keyword `generic map` was used in conjunction with the keyword `port map` to define the generic variables.

Once again, to successfully simulate and synthesize the design shown in Listing 9.8, the code of Listing 9.7 needs to be included in your VHDL project as well.

```
1  entity my_parity_chk is -- library declaration omitted
2      port ( input   : in  std_logic_vector(3 downto 0);
3             output  : out std_logic);
4  end my_parity_chk;
5
6  architecture arch of my_parity_chk is
7      component gen_parity_check is
8          generic ( N  : positive);
9          port    ( x  : in  std_logic_vector(N-1 downto 0);
10                   y  : out std_logic);
11     end component;
12 begin
13     cp1: gen_parity_check generic map (N => 4)
14          port map (x => input, y => output);
15 end arch;
```

Listing 9.8: Use of `generic` for the construct of a generic parity check code

9.3 Structured Testbenches (Unit Testing)

Without mentioning, structured designs are being presented since the first simple testbench was shown in Chapter 4. When simulating, the testbench code is nothing more than the top level design, which includes (instantiates) the DUT as a sub-module. If the designs themselves follow the structural model, testbenches can also be adapted to test each module separately. This is very important since it is possible to guarantee that simpler modules are working before integrating them in more complex designs.

Listings 9.9 and 9.10 present respectively a full adder circuit description and its testbench. It is quite simple to fully test this design since there are only 8 possible input combinations. It is interesting to note how the "aggregation" performed at line 9 simplifies the test code. This operation is similar to the concatenation presented in Section 6.5, but can also appear on the left hand side of the attribution.

With the simpler design fully tested, it is possible to include it in a larger one which creates several instances of the first. If any problem is found with the testbench written for the more complex design, it is possible to start the search for errors on the top-level code.

```vhdl
-- library omitted
entity fulladd is
    port(a, b, cin : in  std_ulogic;
         sum, cout : out std_ulogic);
end fulladd;

architecture arch of fulladd is
begin
    sum <= a xor b xor cin;
    cout <= (a and b) or (a and cin) or (b and cin);
end arch;
```

Listing 9.9: VHDL description of a full adder

```vhdl
-- library, entity and component omitted (see previous Listing)
architecture arch of fulladd_tb is
    signal a, b, cin, sum, cout : std_ulogic;
begin
    DUT: fulladd port map (cin => cin, a => a, b => b,
                           sum => sum, cout => cout);
    tb : process begin
        for h in 0 to 7 loop
            (a, b, cin) <= to_unsigned(h, 3);
            wait for 10 ns;
        end loop;
        wait for 10 ns;
        wait;
    end process tb;
end arch;
```

Listing 9.10: Testbench for the full adder

The described approach is detailed in Listings 9.11 and 9.12 which implement and test a parameterizable adder. In Section 6.8, procedures were used to avoid the writing of repetitive code. The testbench in that

example used "for loops" to simplify the test code. The **generate** loop, which is used outside de scope of a process, not only allows the code to be more compact but also permits the use of the **generic** feature. Take some time to understand how the "fulladd" instances are interconnected. Start by unrolling the loop and pay attention to how the `cout(i)` and `cout(i+1)` signals are used to connect the several carries of the operation.

```
1  -- library omitted
2  entity add is
3      generic (W: positive := 3);
4         port (a, b : in std_logic_vector(W-1 downto 0);
5               sum : out std_logic_vector(W downto 0));
6  end add;
7
8  architecture arch of add is
9      component fulladd is
10         port (a, b, cin : in  std_ulogic;
11               sum, cout : out std_ulogic);
12     end component;
13     signal cout : std_ulogic_vector(W downto 0);
14 begin
15     cout(0) <= '0';
16     add_gen : for i in 0 to W - 1 generate
17         FA: fulladd port map (cin => cout(i),
18                               a => a(i), b => b(i),
19                               sum => sum(i), cout => cout(i+1));
20     end generate;
21     sum(W) <= cout(W);
22 end arch;
```

Listing 9.11: VHDL structural description of a generic adder

The testbench for the generic adder generates all the possible values for addition and also adapts the code to whatever bit witdth is defined as a constant at line 8. It would also be trivial to add automatic verification to this example but this will be postponed to Chapter 11 where data objects the conversion are covered in detail.

```
-- library and entity omitted
architecture arch of tb_add is
    component add is
        generic (W: positive := 3);
        port (a, b : in std_logic_vector(W-1 downto 0);
              sum : out std_logic_vector(W downto 0));
    end component;
    constant BITS : positive := 4;
    signal a, b : std_logic_vector(BITS-1 downto 0);
    signal sum : std_logic_vector(BITS downto 0);
begin
    DUT: add generic map (W => BITS)
             port map (a => a, b => b, sum => sum);
    tb: process begin
        for ai in 0 to (2**BITS)-1 loop
            a <= std_logic_vector(to_unsigned(ai, BITS));
            for bi in 0 to (2**BITS)-1 loop
                b <= std_logic_vector(to_unsigned(bi, BITS));
                wait for 10 ns;
            end loop;
        end loop;
        wait;
    end process tb;
end arch;
```

Listing 9.12: Testbench for the generic adder

9.4 Important Points

- Structural modeling in VHDL supports hierarchical design concepts. The ability to abstract digital circuits to higher levels is the key to understanding and designing complex digital circuits.
- Digital design using schematic capture is an outdated approach for complex projects: you should resist the inclination and/or directive at all costs.
- The VHDL structural model supports the reuse of design units. This

includes units you have previously designed as well as the ability to use predefined module libraries.

- If you use one FPGA software development tool from one of the major FPGA players in the market, you will be able to use digital blocks already developed once you declare them. In this case the entity declaration is not the one of Listing 9.2 but instead a simple library inclusion in your VHDL code that looks like:

```
library UNISIM;
use UNISIM.VComponents.all;
```

All digital blocks available from this library package are described in the documentation of the FPGA software development tool (e.g. Xilinx Vivado).

9.5 Exercises: Structural Modeling

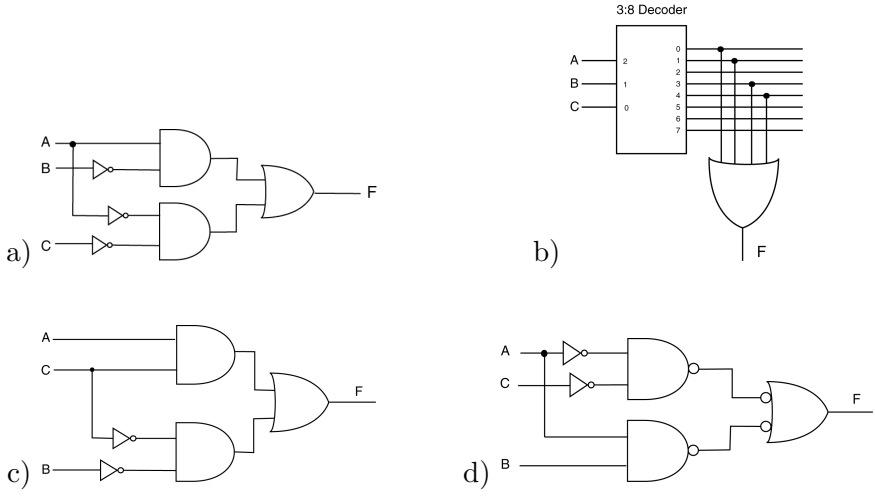
EXERCISE 1. Draw a block diagram of the circuit represented by the VHDL code listed below. Be sure to completely label the final diagram.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity ckt1 is
    Port ( EN1, EN2 : in  std_logic;
                CLK : in  std_logic;
                  Z : out std_logic);
end ckt1;
-- architecture
architecture arch of ckt1 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 arch;
```

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

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity ckt is
    port ( A,B : in  std_logic;
             C : out std_logic);
end ckt;
-- architecture
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;
```

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



10

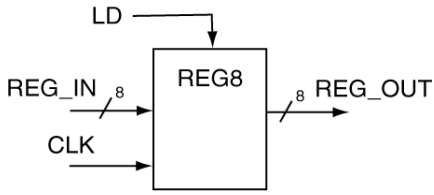
Registers and Register Transfer Level

The concept of a register in VHDL and its subsequent use in digital circuit design is probably one of the more straightforward concepts in VHDL. A register in 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 concerned with how and when data is transferred between the various registers in a digital system. RTL-level design in often associated with "data-path" designs which require 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 data-path. In these cases, the proper sequencing of data transfers is controlled by an FSM.

10.1 RTL-level Design

The study of RTL-level design is best accomplished in the context of a data-path design. The design of data-paths 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 22. 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 22 is shown in Listing 10.1. There are a couple of things worth noting in this solution:

- Note that there is an `if` statement that does not contain a corresponding `else` which is what generates the memory element. For this example, eight bit-sized memory elements (D-flip-flops) are inferred from the `in` and `out` signals widths. The storage elements are associated with the `REG_OUT` bundle. The ease in using VHDL code to generate D flip-flops in this manner is related to D flip-flops being 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 straightforward 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.

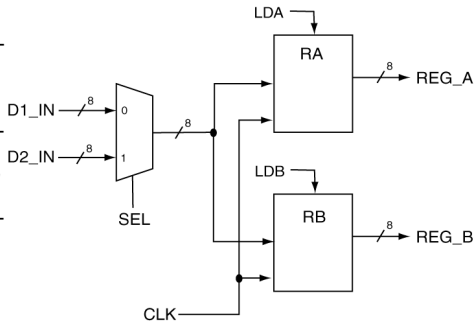
- The assignment of the input to the output is dependent of both the clock edge and the state of the LD signal. The approach taken in the VHDL model shown in Listing 10.1 is to provide a separate `if` clause for both the LD and CLK signals. Only one `if` statement could have been used by making both 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 the digital circuits design; you should get into the habit of treating them gently.

- Since signals REG_IN and LD are required to be synchronous they do not appear inside the process sensitivity list. As seen before, this means the are *read* inside the process, but their change doesn't *trigger* it. Their value is only evaluated at the specified clock edge.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity reg8 is
5      port ( REG_IN  :  in std_logic_vector(7 downto 0);
6             LD, CLK :  in std_logic;
7             REG_OUT : out std_logic_vector(7 downto 0));
8  end reg8;
9
10 architecture reg8 of reg8 is
11 begin
12     reg: process(CLK)
13     begin
14         if (rising_edge(CLK)) then
15             if (LD = '1') then
16                 REG_OUT <= REG_IN;
17             end if;
18         end if;
19     end process;
20 end reg8;
```

Listing 10.1: Solution to Example 22

EXAMPLE 23. Use VHDL behavioral modeling to design the circuit shown on the right. Consider both 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 23 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 that is connected to the inputs of the two registers. Each of the two registers has its own independent load control input. The solution to Example 23 is shown in Listing 10.2. As expected, there are a couple of things worth noting about this solution.

- There are three concurrent statements in this solution: two behavioral models and one data-flow 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 more complicated and harder 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 the signals shown in the Example 23 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 project. 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'. The real nice part about this instruction 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. This syntax together with concatenation also allows, for example, that a few specific bits are set to '1' while all the "others" are set to '0' (regardless of how many)

```
entity ckt_rtl is -- library declaration omitted
    port (D0_IN, D1_IN : in  std_logic_vector(7 downto 0);
          CLK, SEL, 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
    signal s_mux_result : std_logic_vector(7 downto 0);
begin
    ra: process(CLK) begin
        if (rising_edge(CLK)) then
            if (LDA = '1') then
                REG_A <= s_mux_result;
            end if;
        end if;
    end process;

    rb: process(CLK) 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', D0_IN when '0',
                        (others => '0') when others;
end rtl_behavioral;
```

Listing 10.2: Solution to Example 23

The circuit in this example is slightly more complex than most of the examples seen so far. Additionally, remember that there are many different solutions to the same problem. This is a common occurrence in VHDL; in fact, many times there is no best method for implementing a given circuit but different approaches may provide better results in one aspect or another (speed, resource usage, maintainability etc).

EXAMPLE 24. Redesign the last circuit using a structural approach. Use the 8-bit register from Example 22, create a new module for the MUX and integrate both modules in the final design.

SOLUTION. The first part of the solution to Example 24 is shown in Listing 10.3. There is not too much interesting to note here. This is a more realistic example of a structural model compared to the didactical example presented in the section on structural modeling.

```
1  entity mux2t1 is
2      port (  A, B : in  std_logic_vector(7 downto 0);
3              SEL : in  std_logic;
4              M_OUT : out std_logic_vector(7 downto 0));
5  end mux2t1;
6
7  architecture my_mux of mux2t1 is
8  begin
9      with SEL select M_OUT <= B when '1', A when '0',
10                     (others => '0') when others;
11 end my_mux;
```

Listing 10.3: MUX module to be used in the solution to Example 24

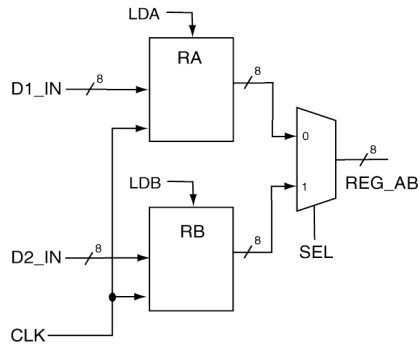
The very important thing to note about the top-level solution in Listing 10.4 is to not be intimidated by the fact the the code doesn't have any logic besides the simple interconnection of modules. The code is well structured; if you are able to recognize this structure, you will be more apt to understand the solution.

The VHDL source code shown in Listing 10.4 is nicely formatted. In particular, the code is nicely indented. Properly indented code is highly desirable in that it nicely presents information based on the indentation. 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 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.

```
entity ckt_rtl is
    port (D0_IN, D1_IN : in  std_logic_vector(7 downto 0);
          CLK, SEL, 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 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;
    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 => D0_IN, B => D1_IN,
                          SEL => SEL, M_OUT => s_mux_result);
end rtl_structural;
```

Listing 10.4: Top-level for a structural modeling approach to Example 24

FINAL TIP. Note how easy it would be to design the different circuit to the right from the structural approach above. A different internal connection with two signals and a single output would form a totally different circuit, capable of storing two different values.



10.2 Text Based Testbenches

One option to implement complex tests is to move the automatic verification from VHDL to text files, which could be generated and verified by a software written in your favorite programming language. As an example, this section presents a testbench which loads its inputs from a text file and saves the outputs to another text file. The test works for both examples presented in this Chapter, since they share the same interface. Listing 10.5 presents the part of the testbench without the text input and output, with the same structure presented for sequential circuits.

```
1  -- library and empty entity omitted
2  architecture tb of tb_rtl is
3      component ckt_rtl is
4          port (D0_IN, D1_IN : in  std_logic_vector(7 downto 0);
5                CLK, SEL, LDA, LDB : in  std_logic;
6                REG_A, REG_B : out std_logic_vector(7 downto 0));
7  end component;
8      constant PERIOD : time := 10 ns;
9      signal CLK_ENABLE: std_logic := '1';
10     signal CLK: std_logic := '0';
11     signal D0_IN, D1_IN : std_logic_vector(7 downto 0);
12     signal SEL, LDA, LDB : std_logic;
13     signal REG_A, REG_B : std_logic_vector(7 downto 0);
14 begin
15     DUT: ckt_rtl port map (D0_IN => D0_IN, D1_IN => D1_IN,
16          CLK => CLK, SEL => SEL, LDA => LDA, LDB => LDB,
17          REG_A => REG_A, REG_B => REG_B);
18
19     CLK <= CLK_ENABLE and not CLK after PERIOD / 2;
20 proc_input : process -- the specific text part is inserted here
21     -- check the code in the next Listing
22     CLK_ENABLE <= '0';
23     wait;
24 end process;
25 end tb;
```

Listing 10.5: Common part of the text based testbench

Listing 10.6 introduces the process for text processing, using some features present only in the 2008 version of the VHDL standard.

```
file text_file : text open read_mode is "input.txt";
file test_output : text open write_mode is "output.txt";
variable tline, output_line : line;
variable ok : boolean;
variable char : character;
variable slv : D0_IN'subtype;
variable sl : SEL'subtype;
begin -- from the process, see previous Listing
    while not endfile(text_file) loop
        readline(text_file, tline);
        if tline.all'length = 0 or tline.all(1) = '#' then
            next; -- skip comments and empty lines
        end if;
        hread(tline, slv, ok);
        assert ok report "D0_IN " & tline.all severity failure;
        D0_IN <= slv;
        hread(tline, slv, ok);
        assert ok report "D1_IN " & tline.all severity failure;
        D1_IN <= slv;
        read(tline, sl, ok);
        assert ok report "SEL " & tline.all severity failure;
        SEL <= sl;
        read(tline, sl, ok);
        assert ok report "LDA " & tline.all severity failure;
        LDA <= sl;
        read(tline, sl, ok);
        assert ok report "LDB " & tline.all severity failure;
        LDB <= sl;
        wait for PERIOD; -- each line is applied to a clock cycle
        hwrite(output_line, REG_A, right, REG_A'length);
        hwrite(output_line, REG_B, right, REG_B'length);
        writeline(test_output, output_line);
    end loop;
    file_close(text_file); file_close(test_output);
```

Listing 10.6: Specific part of the text based testbench

At line 10, a single line from the input text file is read to a local variable. Each following call to `hread` (hexadecimal format) and `read` (binary format) parses a value from the `line` variable and stores in an appropriate variable, which transfers the value to a signal. If any of the parsing fails, an error is indicated in the `ok` variable, which will result in an error message. After the clock cycle has passed (line 29) the outputs of the circuit being tested are formatted in a text line, which is saved into an output file.

Listing 10.7 presents a possible text input file which applies alternating enable signals and values to the inputs, testing both registers. The sequence also verifies if input values are ignored when no enable is active. The first line is discarded as a comment.

```
# D0_IN D1_IN SEL LDA LDB
00 00 0 1 1
01 01 0 0 0
00 02 1 1 0
03 00 0 0 1
04 04 0 0 0
```

Listing 10.7: Text input for the testbench

Listing 10.8 is the resulting output for the specified test, which is consistent with the waveform output, presented in Figure 10.1, confirming not only the storage but also the ignored input values (without enable signals).

```
00      00
00      00
02      00
02      03
02      03
```

Listing 10.8: Text output for the testbench

The read and write functions may seem to provide less features than the ones in a good programming language library, but are more than enough considering that the text files may be generated and verified automatically.

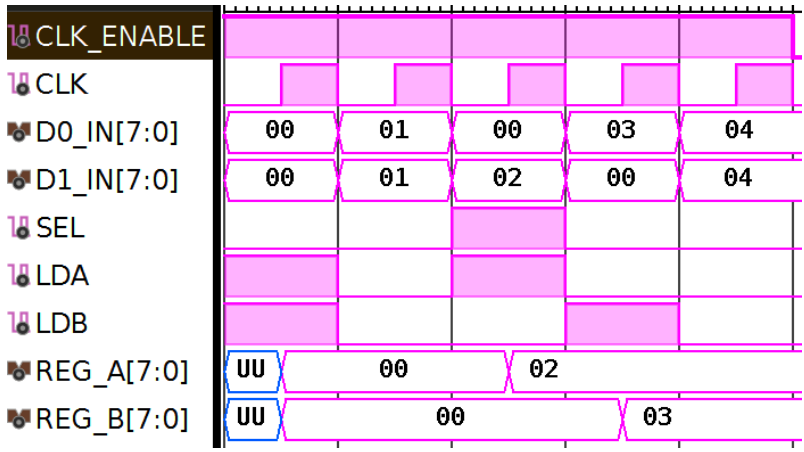


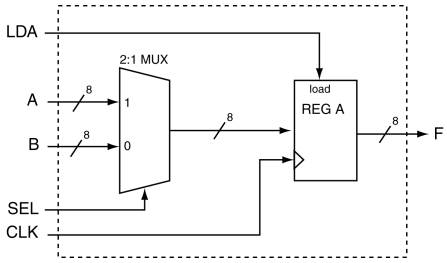
Figure 10.1: Waveform output for the testbench.

10.3 Important Points

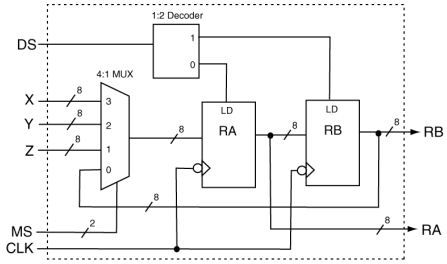
- VHDL can be used to easily implement circuits at the register transfer level. The corresponding VHDL models can be implemented in either structural or full behavioral format.
- RTL-level VHDL models should strive for simplicity in their designs. If the complexity of the models is not well handled, the chances that your circuit works correctly greatly diminish.

10.4 Exercises: Register Transfer Level Circuits

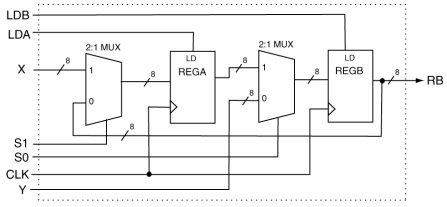
EXERCISE 1. Provide a VHDL model that can be used to implement the following circuit.



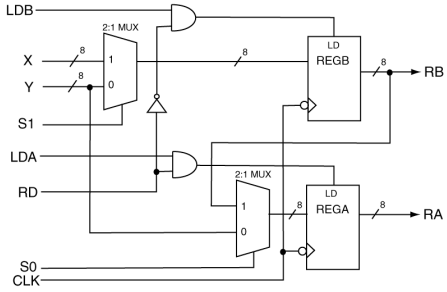
EXERCISE 2. Provide a VHDL model that can be used to implement the following circuit.



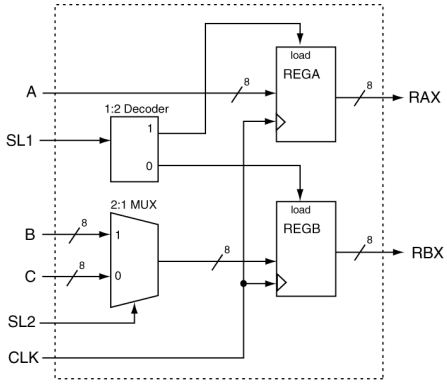
EXERCISE 3. Provide a VHDL model that can be used to implement the following circuit.



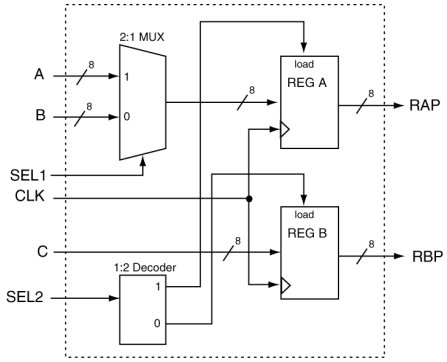
EXERCISE 4. Provide a VHDL model that can be used to implement the following circuit.



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



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



11

Data Objects

Many of the concepts presented so far have been implicitly presented in the context of example problems. In this way, you have probably been able to generate quality VHDL code but were constrained to use the VHDL style presented in these examples. In this section, we will present some of the underlying details and theories that surround VHDL as a backdoor approach for presenting tools that will allow you to use VHDL for describing the behavior of more complex digital circuits.

In order to move into more sophisticated VHDL, a good place to start is with the definition of VHDL objects (e.g. data types). 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 have only used `signal` data objects and `std_logic` data types. Two new data objects and several new data types are introduced and discussed in this section.

11.1 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 straightforward to understand and use once the concepts of signals and variables are understood. File data objects, exclusively used in

simulations, are not discussed in this chapter.

Mind that VHDL is a vast language that goes well beyond the VHDL code that is used to program an FPGA or a CPLD. In fact the actual VHDL that can be translated into an FPGA/CPLD bit-stream is called RTL VHDL and represents only a small subset of what is included in the current VHDL standard. The file data objects are an example of a data object that cannot be implemented in a silicon device.

Just as side note, it is interesting to point out that it is also possible to compile VHDL code into an executable file that can be executed, generally for simulation purposes, with any general purpose Intel PC. For more details refer to the open-source work of Tristan Gingold available at: `http://ghdl.free.fr`.

11.2 Data Object Declarations

The first thing to note about data objects is the similarity in their declarations. The forms for the three data objects we will be discussing are listed in Table 11.1. 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 have used it extensively up to this point.

VHDL data object	Declaration form
Signal	**signal** `sig_name : sig_type:=initial_value;`
Variable	**variable** `var_name : var_type:=initial_value;`
Constant	**constant** `const_name : const_type:=initial_value;`

Table 11.1: Data object declaration forms.

Each of the data objects can optionally be assigned initial values. Signal declarations do not usually include initial values as opposed to constants which obviously require one. Initial values for signals are not implementable on silicon by the synthesizing tools but are taken into consideration by VHDL simulation tools. When simulating a project, the module that drives the simulation (called "testbench") may have initialized signals. Even if an FPGA tool supports this configuration, the value would only be applied at power-up. Since signal initial values are only relevant to storage elements, proper initialization of should be performed explicitly by a reset signal.

Example declarations for the three flavors of data objects are provided in Table 11.2. These examples include several new data types which will be discussed in the next sections.

Data object	Declaration form
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 11.2: Example declarations for signal, variable and constant data objects.

11.3 Variables and 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 (functions and procedures will not be discussed here). 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. Many important differences between the assignments will be detailed in the next section.

The signal assignment operator, <=, was used to transfer the value of one signal to another while 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 11.2, the assignment operator is overloaded which allows it to be used to assign initial values to the three listed forms of data objects.

11.4 Signals vs. Variables

The use of signals and variables can be somewhat confusing because of their similarities. Generally speaking, a signal 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 include connections to the outside world. 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 here 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.

With relatively simple circuits, signal objects are generally sufficient. As your digital designs become more complex, there is a greater chance that you will 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 inside a process are actually only scheduled when the same process is completed. 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 signal assignments inside the process, 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 that you might face is that the new result (the new value assigned to the signal) is not available to use inside the process before it finishes.

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 very important ramifications in both the circuit simulation and synthesis realm.

Variables will not always result in wires or registers in a circuit. Both possibilities exist, but the dynamic nature of their assignments makes them more similar to the role variables perform in software. If you wonder what is 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.

Even though instructions inside a process are "executed" consecutively, this should not fool you in thinking that a process environment is similar to a segment of C code. Remember that while lines of C code require some tens of clock cycles each to be executed, VHDL instructions are **synthesized to circuits** that should reach their final outputs within the same clock signal which triggered them. For purely combinational circuits, the trigger is the input signal which changed.

The important differences between signals and variables are yet another reason to keep your processes (for combinational or sequential circuits) are short and simple, so you don't lose track of their behavior.

11.5 Standard Data Types

Not only does VHDL has many defined data types but it also allows you to define your own types. Among the most popular VHDL data types we would like to mention the following data types:

`std_logic` : It is the fundamental type for digital signals

`boolean` : It is a two-value enumerated type (true or false).

`natural` : It is a subtype of `integer` because it is a non-negative integer.

`positive` : It is a subtype of `integer` because it is a positive integer.

`integer_vector` : It is the vector form of an `integer` type.

`integer` : Refer to the Integer Types section.

`character` : A 256-symbol enumerated type.

`string` : It is the vector form of a `character` type.

The `std_logic` type is quite powerful and, besides the expected '0' and '1', can assume values like 'U' (uninitialized), 'X' (unknown), '-' (don't care), and many others related to the actual electrical implementation. It is not uncommon to find older implementations which use the simpler type **`bit`** (and its respective vector **`bit_vector`**), which is restricted to the values '0' and '1'.

11.6 User-Defined Types

VHDL allows you to define your own data type. A typical example of a custom integer type is:

```
type my_type is range 0 to 100;
constant my_const : my_type := 31;
```

Obviously it is possible to define more complex data structures. For instance it is a common practice to use a custom data type when you want to implement a ROM (read-only memory) in VHDL.

```
1  -- typical custom data type for a 20-byte ROM
2  type memory is array (0 to 19) of std_logic_vector(7 downto 0);
3  constant my_rom : memory := (
4           1 => "11111111"
5           2 => "11110111"
6           5 => "11001111"
7          12 => "10110101"
8          18 => "10001101"
9     others => "00000000");
```

11.7 Integer Types

The `integer` type was cryptically mentioned before in one of the examples, but it will be discussed further here. The use of integer types aids in the design of algorithmic-type VHDL code. This type of coding allows VHDL to describe the behaviour of complex digital circuits. As you progress in your digital studies, you will soon find yourself in need of more complex descriptive VHDL tools. Data types such as integers partially fill that desire. This section briefly looks at integer types as well as the definition of user-specified integer types.

The range of the default `integer` type is $(-2,147,483,648$ to $2,147,483,647)$. These numbers should seem familiar since they represent the standard 32-bit range for a signed number: from $-(2^{31})$ to $+(2^{31}-1)$. Other types similar to integers include `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 31-bit range, respectively. Examples of integer declarations are shown in the following listing.

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

Although it could be possible to use only basic integer declarations in your code, as we have seen before, VHDL allows you to define your own data types with their own personalized range constraints. These special types should be used wherever possible to make your code more readable. The custom integer-type definition uses the type `range` construct and the `to` or the `downto` keywords for the definition. Some examples of integer-type declarations are provided in the following listing.

Although each of the types listed in the previous listing 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 previously presented, each of the assignments in the following listing is illegal.

```
-- Integer type declarations and valid initializations
type years   is range -3000 to 3000;
type apples  is range 0 to 15;
type oranges is range 0 to 15;

signal my_year   : years   := 2022;
signal my_apple  : apples  := 0;
signal my_orange : oranges := 0;

-- These assignments are invalid:
my_year   <= -10000;     -- out of range
my_apple  <= my_orange;  -- different types
```

11.8 `signed` and `unsigned` Types

`signed` and `unsigned` data types are available once you declare the standard IEEE `ieee.numeric_std` package. Mind that these two data types are also defined in the **non-standard** `std_logic_arith` package. The use of non-standard libraries is however highly discouraged.

A `signed` value ranges from -2^{N-1} to $2^{N-1}-1$ and an `unsigned` value ranges from 0 to 2^N-1 where N is the number of bits.

`signed` and `unsigned` types can be conveniently used for internal variables as well as for entity ports. Additionally the `ieee.numeric_signed` library and the `ieee.numeric_unsigned` library offer arithmetic and type conversion for both types.

`signed` and `unsigned` types, in a way, look like `std_logic_vector` types, especially in how they are declared and so the question that you might have is:

```
Why would I need to use a signed or unsigned
type in place of a std_logic_vector type?
```

The answer to this question is in Listing 11.1. The `std_logic_vector` type should not be used to define a numerically meaningful signal or variable. These types are reserved for vectors that are interpreted and behave like a number. The `std_logic_vector` type should be only employed for defining "bags of bits".

The use of `signed/unsigned` types is desirable any time your bags of bits (signals, variables or constants) stop being "bags" and become numbers of type signed, unsigned or even integers.

As final note, we should mention that the inclusion of the non-standard `std_logic_arith` library could have given us the possibility of doing `out1 <= in1 + 1;` in line 16 of Listing 11.1, making things much simpler. However, once again, the use of non-standard library is highly discouraged. Avoiding this, allows the VHDL code to become more portable between different tools and hardware targets. The process of converting non-standard code to a compatible one may result in subtle and difficult to find bugs.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all; -- defines std_logic_vector type
3  use IEEE.numeric_std.all;    -- defines signed and unsigned types
4
5  entity double_sum is
6      port (
7          in1, in2  : in  std_logic_vector (7 downto 0);
8          out1      : out std_logic_vector (7 downto 0));
9          unsig_in  : in  unsigned(7 downto 0);
10         unsig_out : out unsigned(7 downto 0));
11 end double_sum;
12
13 architecture arch of sum is
14 begin
15     -- ILLEGAL OPERATIONS:
16     out1 <= in1 + 1;    -- 1 is an integer
17     out1 <= in1 + in2; -- addition is not defined
18
19     -- legal operations:
20     unsig_out <= unsig_in + 1;
21     unsig_out <= unsigned(in1) + 1;
22     out1 <= std_logic_vector(unsigned(in1) + 1);
23 end arch;
```

Listing 11.1: Use of unsigned types in your code

11.9 std_logic Types

For the representation of digital signals so far in this book, we have used the std_logic type. However, one of the data types, similar to std_logic, neither used nor endorsed in this book is the bit type. This type can take on only the values of '1' or '0'. While this set of values seems appropriate for designing digital circuits, it is actually somewhat limited. Due to its versatility and a more complete range of possible values, the std_logic type is preferred over bit types. The std_logic type is defined in the VHDL package ieee.std_logic_1164 and provides a common standard that can be used by all VHDL programmers.

The std_logic type is officially defined as enumerated. Two of the

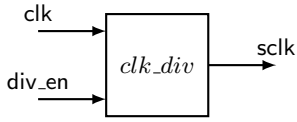
possible enumerations of course include '1' and '0'. The actual definition is shown in the Listing 11.2. The std_logic type is a **resolved version** of the std_ulogic type. Resolved means that unlike for std_ulogic types, when you use std_logic type signals, you can assign multiple drivers to the same signal without having the compiler complain about it.

```
type std_logic is  ( 'U', -- uninitialised
                     'X', -- forcing unknown
                     '0', -- forcing 0
                     '1', -- forcing 1
                     'Z', -- high impedance
                     'W', -- weak unknown
                     'L', -- weak 0
                     'H', -- weak 1
                     '-'  -- unspecified (do not care)
                   );
```

Listing 11.2: Declaration of the std_logic enumerated type

The std_logic type uses the VHDL character type in its definition. Although there are nine values in the definition shown in Listing 11.2, this book only deals with '0', '1', 'Z' and '-'. The 'Z' is 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. Finally, since the characters used in the std_logic type are part of the definition, they must be used as listed. Mind the use of lower-case letters will generate an error.

EXAMPLE 25. 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 Listing 11.3. 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.
- Variable declarations occur in the process body before the begin line.

The VHDL implementation of frequency divider that takes a certain clock signal and generates a second clock signal of higher or lower frequency is quite common practice in VHDL. Such an implementation is normally done using clock management blocks built in the FPGA fabric specifically for this purpose. Digital Clock Managers (DCM), Mixed Mode Clock Managers (MMCM) or Phase Locked Loops (PLL) are just some examples.

The use of clock management blocks will guarantee your design meets timing requirements or clock phase noise constraints that will make your job a lot easier in the long run. Try to remember this.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

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 := 31; -- user-defined constant
    signal tmp_sclk : std_logic;          -- intermediate signal
begin
    my_div: process (clk, div_en)
        variable div_count : my_count := 0;
    begin
        if (div_en = '0') then
            -- asynchronous disable
            div_count := 0;
            tmp_sclk <= '0';
        elsif (rising_edge(clk)) 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; -- count
            end if;
        end if;
    end process my_div;
    sclk <= tmp_sclk; -- final/concurrent assignment
end my_clk_div;
```

Listing 11.3: Solution to Example 25

11.10 Testbenches with Automatic Arithmetic

Now that a more detailed view of the data types is given, we will revisit the testbench presented in Listing 9.12.

```
1  -- library and entity omitted
2  architecture arch of tb_add is
3      component add is
4              generic (W: positive := 3);
5              port (a, b : in std_logic_vector(W-1 downto 0);
6                sum : out std_logic_vector(W downto 0));
7      end component;
8      constant BITS : positive := 4;
9      signal a, b : std_logic_vector(BITS-1 downto 0);
10     signal sum : std_logic_vector(BITS downto 0);
11 begin
12     DUT: add generic map (W => BITS)
13               port map (a => a, b => b, sum => sum);
14     tb: process
15         variable isum, check : integer;
16     begin
17         for ai in 0 to (2**BITS)-1 loop
18             a <= std_logic_vector(to_unsigned(ai, BITS));
19             for bi in 0 to (2**BITS)-1 loop
20                 b <= std_logic_vector(to_unsigned(bi, BITS));
21                 wait for 10 ns;
22                 isum := to_integer(unsigned(sum));
23                 check := ai + bi;
24                 assert(isum = check) report "Fail at " &
25                     to_string(ai) & " + " & to_string(bi) &
26                     " = " & to_string(isum) severity failure;
27             end loop;
28         end loop;
29         report "Test done." severity note;
30         wait;
31     end process tb;
32 end arch;
```

Listing 11.4: Automatic testbench for the generic adder

With all the conversions explained and the differences between variables and signals detailed (Section 11.4), the automatic verification is added, as detailed in Listing 11.4.

The delay in line 21 is crucial to allow the arithmetic at line 23 to reflect the value `sum = a + b`. Since these three vectors are assigned to signals, their values would be the ones from the previous loop, up until de `wait`. After that, the conversion to `isum` already takes the result of the addition in the current loop, as well as the sum performed directly with the two implicit variables (`ai` and `bi`).

The `assert` directive will compare the values which are expected to be the same and print an error message otherwise. Note that, by using other conversions, a full report of the test values and the wrong addition is informed. The keyword `failure` can be replaced by `error` at line 26, if the desired behavior of the test is not to be interrupted at the first error found. This may be used it is sometimes easier to spot a fault is a pattern emerges in the errors.

11.11 Important Points

- The use of `signed/unsigned` types is desirable any time your "bags of bits" (signals, variables or constants) stop being "bags" and become numbers of type signed, unsigned or even integers. A typical example is the variable used for a counter for which there is really no reason to use a `std_logic_vector` type for. Note the increment in line 30 of Listing 11.3.

- The standard IEEE library `numeric_std` is needed when you want to use `signed` and/or `unsigned` types. The standard IEEE library `numeric_std` is almost always preferred over the non-standard `std_logic_arith` library.

- Any use of the non-standard Synopsys libraries: `std_logic_signed`, `std_logic_unsigned` and `std_logic_arith` is highly discouraged.

- You cannot increment a `std_logic_vector` type signal, you need to first convert it into an unsigned, a signed or an integer:

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use ieee.numeric_std.all
4
5 signal val1, val2 : std_logic_vector( 31 downto 0 );
6 val2 <= val1 + 1;      -- ILLEGAL OPERATION
7 val2 <= std_logic_vector( unsigned(val1) + 1 );
```

Figure 11.1 presents a nice summary of data types conversions and casts. It becomes clear that there is a boundary between number and vector types, as well as between vectors with and without a value interpretation. This is viewed by many as one of the strong points of VHDL: there is no margin left for errors due to implicit behavior.

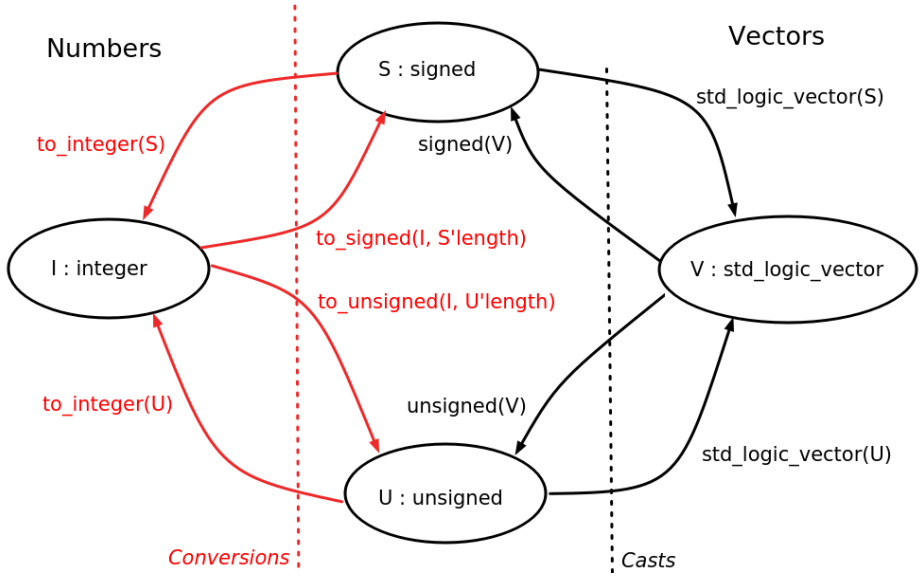


Figure 11.1: Direct casts and type conversions with explicit lengths.

12

Looping Constructs

As the circuits you are required to design become more and more complex, you will find yourself searching for more functionality and versatility from VHDL. You will probably find what you are looking for in various looping constructs which are yet another form of VHDL statement. This chapter provides descriptions of several types of looping constructs and some 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, they can only appear inside processes. You will also be able to apply to the circuits you will be describing using VHDL the algorithmic thinking and designing skills you developed in coding with higher-level computer languages. The syntax is slightly different but the basic structured programming concepts are the same.

12.1 `for` and `while` Loops

The purpose of a loop construct is to allow some coding instructions to happen 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 some 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 is to make the code readable and understandable. Keeping this concept in mind lets us see the functional differences between `for` and `while` loops. This basic difference can be best highlighted by examining the code provided in Listing 12.1.

```
1 -- for loop
2 my_label: for index in a_range loop
3     -- sequential statements...
4 end loop my_label;
5
6 -- while loop
7 my_label: while (condition) loop
8     -- sequential statements...
9 end loop my_label;
```

Listing 12.1: The basic structure of the `for` and `while` loops.

The major difference between these two loops lies in the number of iterations the loops 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 will see in the examples that follow, the `for` loop allows you to explicitly state the number of iterations that the loop performs.

The `while` loop should be used when you do not know the number of iterations the loop needs to perform. In this case, the loop stops iterating when the terms stated in the condition clause are not met. Using these loops in this manner constitutes a good programming practice. The loop labels are optional but should be always 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.

12.1.1 for Loops

The basic form of the for loop was shown in Listing 12.1. 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. Hereafter you find two examples.

```
1 for cnt_val in 0 to 9 loop
2     -- sequential_statements
3 end loop;
4 -- is the equivalent to:
5 type my_range is range 0 to 9;
6 for cnt_val in my_range loop
7     -- sequential_statements
8 end loop;
```

```
1 for cnt_val in 9 downto 0 loop
2     -- sequential_statements
3 end loop;
4 -- is the equivalent to:
5 type my_range is range 9 downto 0;
6 for cnt_val in my_range loop
7     -- sequential_statements
8 end loop
```

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 will 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 a good thing.

- The index variable (cnt_val in the examples) does not need to be declared, it is in fact 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 of any enumerated type.

12.1.2 `while` Loops

`while` loops are somewhat simpler 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 the following listing.

```
1  constant max_fib : integer := 2000;
2  variable fib_sum : integer := 1;
3  variable tmp_sum : integer := 0;
4
5  while (fib_sum < max_fib) loop
6      fib_sum := fib_sum + tmp_sum;
7      tmp_sum := fib_sum;
8  end loop;
9
10 -- A 'for' should be used for this one:
11 constant max_num : integer := 10;
12 variable fib_sum : integer := 1;
13 variable tmp_sum : integer := 0;
14 variable int_cnt : integer := 0;
15 -- number of iterations is actually known!
16 while (int_cnt < max_num) loop
17     fib_sum := fib_sum + tmp_sum;
18     tmp_sum := fib_sum;
19     int_cnt := int_cnt + 1;
20 end loop;
```

12.1.3 Loop Control: next and exit Statements

Similarly 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 higher-level languages in the control they can exert over loops. Both constructs are available for use in either the for or the while loop.

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 next listing. These are two examples that use the next statement and do not necessarily represent a good programming practice nor really contain meaningful code.

```
1  variable my_sum : integer := 0;
2
3  -- using 'if':
4  for cnt_val in 0 to 50 loop
5      if (my_sum = 20) then
6          next;
7      end if;
8      my_sum := my_sum + 1;
9  end loop;
10
11 -- using 'when':
12 variable my_sum : integer := 0;
13
14 while (my_sum < 300) loop
15     next when (my_sum = 20);
16     my_sum := my_sum + 1;
17 end loop;
```

`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 the 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 the next listing. Note that these codes are merely presenting the syntax for the `exit` statement. It is obviously a bad practice to prematurely abort loops with known lengths like these.

```
variable my_sum : integer := 0;

-- using 'if':
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;

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

13

Standard Digital Circuits in VHDL

As you know or as you will be finding out soon, even the most complex digital circuit is composed 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 around 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 and efficiently design complex digital circuits. The following listings show a set of standard digital devices and the VHDL code used to describe them. The following circuits are 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 for your own design needs.

13.1 RET D Flip-flop - Behavioral Model

```
1  -----------------------------------------------------
2  --  D flip-flop: RET D flip-flop with single output
3  --
4  --  Required signals:
5  -----------------------------------------------------
6  --  CLK,D: in  std_logic;
7  --  Q:     out std_logic;
8  -----------------------------------------------------
9  process (CLK)
10 begin
11    if (rising_edge(CLK)) then
12       Q <= D;
13    end if;
14 end process;
```

13.2 FET D Flip-flop with Active-low Asynchronous Preset - Behavioral Model

```
1  -------------------------------------------------------------------
2  --  D flip-flop: FET D flip-flop with asynchronous preset. The
3  --  preset input takes precedence over the synchronous input.
4  --
5  --  Required signals:
6  -------------------------------------------------------------------
7  --  CLK,D,S: in  std_logic;
8  --  Q:       out std_logic;
9  -------------------------------------------------------------------
10 process (CLK,S)
11 begin
12    if (S = '0') then
13       Q <= '1';
14    elsif (falling_edge(CLK)) then
15       Q <= D;
16    end if;
17 end process;
```

13.3 8-Bit Register with Load Enable - Behavioral Model

```
1  ------------------------------------------------
2  --  Register: 8-bit Register with load enable.
3  --
4  --  Required signals:
5  ------------------------------------------------
6  --  CLK,LD: in  std_logic;
7  --  D_IN:   in  std_logic_vector(7 downto 0);
8  --  D_OUT:  out std_logic_vector(7 downto 0);
9  ------------------------------------------------
10 process (CLK)
11 begin
12    if (rising_edge(CLK)) then
13       if (LD = '1') then    -- positive logic for LD
14          D_OUT <= D_IN;
15       end if;
16    end if;
17 end process;
```

13.4 Synchronous Up/Down Counter - Behavioral Model

```
1  ------------------------------------------------------------
2  -- Counter: synchronous up/down counter with asynchronous
3  -- reset and synchronous parallel load.
4  ------------------------------------------------------------
5  -- library declaration
6  library IEEE;
7  use IEEE.std_logic_1164.all;
8  use IEEE.numeric_std.all;
9
10 entity COUNT_8B is
11    port ( RESET,CLK,LD,UP : in  std_logic;
12                      DIN : in  std_logic_vector (7 downto 0);
13                    COUNT : out std_logic_vector (7 downto 0));
14 end COUNT_8B;
15 architecture my_count of COUNT_8B is
16    signal  t_cnt : unsigned(7 downto 0); -- internal counter signal
17 begin
```

```
18    process (CLK, RESET)
19    begin
20       if (RESET = '1') then
21          t_cnt <= (others => '0');  -- clear
22       elsif (rising_edge(CLK)) then
23          if (LD = '1') then     t_cnt <= unsigned(DIN);  -- load
24          else
25             if (UP = '1') then  t_cnt <= t_cnt + 1; -- incr
26             else                t_cnt <= t_cnt - 1; -- decr
27             end if;
28          end if;
29       end if;
30    end process;
31    COUNT <= std_logic_vector(t_cnt);
32 end my_count;
```

13.5 Shift Register with Synchronous Parallel Load - Behavioral Model

```
1  ----------------------------------------------------------------------
2  -- Shift Register: unidirectional shift register with synchronous
3  -- parallel load.
4  --
5  -- Required signals:
6  ----------------------------------------------------------------------
7  -- CLK, D_IN:   in  std_logic;
8  -- P_LOAD:      in  std_logic;
9  -- P_LOAD_DATA: in  std_logic_vector(7 downto 0);
10 -- D_OUT:       out std_logic;
11 --
12 -- Required intermediate signals:
13 signal REG_TMP: std_logic_vector(7 downto 0);
14 ----------------------------------------------------------------------
15 process (CLK)
16 begin
17    if (rising_edge(CLK)) then
18       if (P_LOAD = '1') then
19          REG_TMP <= P_LOAD_DATA;
20       else
21          REG_TMP <= REG_TMP(6 downto 0) & D_IN;
```

```
22        end if;
23     end if;
24     D_OUT <= REG_TMP(7);
25  end process;
```

13.6 8-Bit Comparator - Behavioral Model

```
1  ----------------------------------------------------------------------
2  -- Comparator: Implemented as a behavioral model. The outputs
3  -- include equals, less than and greater than status.
4  --
5  -- Required signals:
6  ----------------------------------------------------------------------
7  -- CLK:           in  std_logic;
8  -- A_IN, B_IN:    in  std_logic_vector(7 downto 0);
9  -- ALB, AGB, AEB: out std_logic
10 ----------------------------------------------------------------------
11 process(CLK)
12 begin
13    if ( A_IN < B_IN ) then ALB <= '1';
14    else ALB <= '0';
15    end if;
16
17    if ( A_IN > B_IN ) then AGB <= '1';
18    else AGB <= '0';
19    end if;
20
21    if ( A_IN = B_IN ) then AEB <= '1';
22    else AEB <= '0';
23    end if;
24 end process;
```

13.7 BCD to 7-Segment Decoder - Data-Flow Model

```
1  ----------------------------------------------------------------------
2  -- BCD to 7-Segment Decoder: Implemented as combinatorial circuit.
3  -- Outputs are active low; Hex outputs are included. The SSEG format
4  -- is ABCDEFG (segA, segB etc.)
5  --
```

```
-- 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
```

13.8 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;
```

```
16        when "10" => MUX_OUT <= C;
17        when "11" => MUX_OUT <= D;
18        when others => (others => '0');
19     end case;
20  end process;
```

13.9 4:1 Multiplexer - Data-Flow Model

```
1  ----------------------------------------------------------------
2  -- A 4:1 multiplexer implemented as data-flow model using a
3  -- selective signal assignment statement.
4  --
5  -- Required signals:
6  ----------------------------------------------------------------
7  -- SEL:        in  std_logic_vector(1 downto 0);
8  -- A, B, C, D: in  std_logic;
9  -- MUX_OUT:    out std_logic;
10 ----------------------------------------------------------------
11 with SEL select
12    MUX_OUT <= A when "00",
13               B when "01",
14               C when "10",
15               D when "11",
16               (others => '0') when others;
```

13.10 Decoder

```
1  ----------------------------------------------------------------------
2  -- Decoder: 3:8 decoder with active high outputs implemented as
3  -- combinatorial circuit with selective signal assignment statement
4  --
5  -- Required signals:
6  ----------------------------------------------------------------------
7  -- D_IN: in  std_logic_vector(2 downto 0);
8  -- FOUT: out std_logic_vector(7 downto 0);
9  ----------------------------------------------------------------------
10 with D_IN select
11    F_OUT <= "00000001" when "000",
12             "00000010" when "001",
```

```
13              "00000100" when "010",
14              "00001000" when "011",
15              "00010000" when "100",
16              "00100000" when "101",
17              "01000000" when "110",
18              "10000000" when "111",
19              "00000000" when others;
```

A

Contributors to This Book

Bryan Mealy is an associate professor at the California Polytechnic State University, San Luis Obispo, USA. His technical interests include designing new courses, embedded systems and digital hardware. His real interests are developing his luthier skills and making noise on bass guitar and piano.

Fabrizio Tappero is an embedded system developer with experience in academic research on satellite-based navigation systems and GNSS receiver design. Among other things, he enjoys very much coding in VHDL and Python.

Christina Jarron is an Aussie technical writer and editor. When she is not busy running after her customers, she spends her time wandering around the beautiful city of Barcelona.

Rob Ash lives on the South Coast of the UK. Originally a sign painter he now spends his time completing art, design and illustration work. When not at his easel or digital tablet he enjoys photography, music and playing bass guitar.

Vitor Angelo is a professor at the Electronics Department of the Federal University of Minas Gerais (UFMG) with a career in embedded systems and strict real-time development at several companies.